The Multistrand Simulator: Stochastic Simulation of the Kinetics of Multiple Interacting DNA Strands

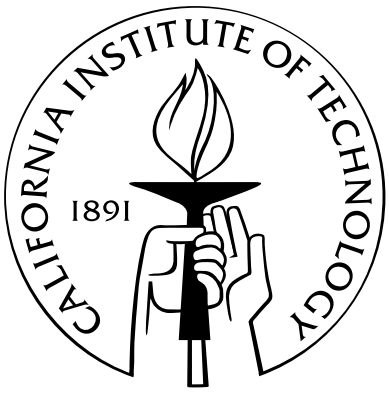
Thesis by

Joseph Malcolm Schaeffer

In Partial Fulfillment of the Requirements

for the Degree of

Master of Science



California Institute of Technology

Pasadena, California

2012

(Submitted February 4, 2012)

Acknowledgements

Thanks to my advisor Erik Winfree, for his enthusiasm, expertise and encouragement. The models presented here are due in a large part to helpful discussions with Niles Pierce, Robert Dirks, Justin Bois and Victor Beck. Two undergraduates, Chris Berlind and Jashua Loving, did summer research projects based on Multistrand and in the process helped build and shape the simulator. There are many people who have used Multistrand and provided very helpful feedback for improving the simulator, especially Josh Bishop, Nadine Dabby, Jonathan Othmer, and Niranjan Srinivas. Thanks also to the many past and current members of the DNA and Natural Algorithms group for providing a stimulating environment in which to work.

There are many medical professionals to which I owe my good health while writing this thesis, especially Dr. Jeanette Butler, Dr. Mariel Tourani, Cathy Evaristo, and the staff of the Caltech Health Center, especially Alice, Divina and Jeannie.

I want to acknowledge all my family and friends for their support. A journey is made all the richer for having good company, and I would not have made it nearly as far without all the encouragement.

Finally, I must thank my wife Lorian, who has been with me every step of this journey and has shared all the high points and low points with her endless love and support.

Abstract

DNA nanotechnology is an emerging field which utilizes the unique structural properties of nucleic acids in order to build nanoscale devices, such as logic gates, motors, walkers, and algorithmic structures. These devices are built out of DNA strands whose sequences have been carefully designed in order to control their secondary structure - the hydrogen bonding state of the bases within the strand (called "base-pairing"). This base-pairing is used to not only control the physical structure of the device, but also to enable specific interactions between different components of the system, such as allowing a DNA walker to take steps along a prefabricated track. Predicting the structure and interactions of a DNA device requires good modeling of both the thermodynamics and the kinetics of the DNA strands within the system. Thermodynamic models can be used to make equilibrium predictions for these systems, allowing us to look at questions like "Is the walker-track interaction a well-formed and stable molecular structure?", while kinetics models allow us to predict the non-equilibrium dynamics, such as "How quickly will the walker take a step?". While the thermodynamics of multiple interacting DNA strands is a well-studied model which allows for both analysis and design of DNA devices, previous work on the kinetics models only explored the kinetics of how a single strand folds on itself.

The kinetics of a set of DNA strands can be modeled as a continuous time Markov process through the state space of all secondary structures. Due to the exponential size of this state space it is computationally intractable to obtain an analytic solution for most problem sizes of interest. Thus the primary means of exploring the kinetics of a DNA system is by simulating trajectories through the state space and aggregating data over many such trajectories. We developed the the **Multistrand** kinetics simulator, which extends the previous work by including the multiple strand version of the thermodynamics model (a core component for calculating parameters of the kinetics model), extending the thermodynamics model to include terms accounting for the fixed volume simulations, and

by adding new kinetic moves that allow interactions between distinct strands. Furthermore, we prove that our modified thermodynamic and kinetic models are exactly equivalent to the canonical thermodynamics model when the simulation is run for sufficiently long time to reach equilibrium.

The kinetic simulator was implemented in C++ for time critical components and in Python for input and output routines as well as post-processing of trajectory data. A key contribution of this work was the development of data structures and algorithms that take advantage of local properties of secondary structures. These algorithms enable the efficient reuse of the basic objects that form the system, such that only a very small part of the state's neighborhood information needs to be recalculated with every step. Another key addition was the implementation of algorithms to handle the new kinetic steps that occur between different DNA strands, without increasing the time complexity of the overall simulation. These improvements led to a reduction in worst case time complexity of a single step being just quadratic in the input size (the number of bases in the simulated system), rather than cubic, and also led to additional improvements in the average case time complexity.

What data does the simulation produce? At the very simplest, the simulation produces a full kinetic trajectory through the state space - the exact states it passed through, and the time at which it reached them. A small system might produce trajectories that pass through hundreds of thousands of states, and that number increases rapidly as the system gets larger! Going back to our original question, the type of information a researcher hopes to get out of the data could be very simple: "How quickly does the walker take a step?", with the implied question of whether it's worth it to actually order the particular DNA strands composing the walker, or go back to the drawing board and redesign the device. One way to acquire that type of information is to look at the first time in the trajectory where we reached the "walker took a step" state, and record that information for a large number of simulated trajectories in order to obtain a useful answer. We designed and implemented new simulation modes that allow the full trajectory data to be condensed as it's generated into only the pieces the user cares about for their particular question. This analysis tool also required the development of flexible ways to talk about states that occur in trajectory data; if someone wants data on when the walker took a step, we have to be able to express that in terms of the Markov process states which meet that condition.

Contents

List of Figures

Chapter 1

Introduction

DNA nanotechnology is an emerging field which utilizes the unique structural properties of nucleic acids in order to build nanoscale devices, such as logic gates [18], motors [4, 1], walkers [19, 1, 20], and algorithmic structures [14, 23]. These devices are built out of DNA strands whose sequences have been carefully designed in order to control their secondary structure - the hydrogen bonding state of the bases within the strand (called "base-pairing"). This base-pairing is used to not only control the physical structure of the device, but also to enable specific interactions between different components of the system, such as allowing a DNA walker to take steps along a prefabricated track. Predicting the structure and interactions of a DNA device requires good modeling of both the thermodynamics and the kinetics of the DNA strands within the system. Thermodynamic models can be used to make equilibrium predictions for these systems, allowing us to look at questions like "Is the walker-track interaction a well-formed and stable molecular structure?", while kinetics models allow us to predict the non-equilibrium dynamics, such as "How quickly will the walker take a step?". While the thermodynamics of multiple interacting DNA strands is a well-studied model [5] which allows for both analysis and design of DNA devices [24, 6], previous work on secondary structure kinetics models only explored the kinetics of how a single strand folds on itself [7].

The kinetics of a set of DNA strands can be modeled as a continuous time Markov process through the state space of all secondary structures. Due to the exponential size of this state space it is computationally intractable to obtain an analytic solution for most problem sizes of interest. Thus the primary means of exploring the kinetics of a DNA system is by simulating trajectories through the state space and aggregating data over many such trajectories. We present here the **Multistrand** kinetics simulator, which extends the previous

work [7] by using the multiple strand thermodynamics model [5] (a core component for calculating transition rates in the kinetics model), adding new terms to the thermodynamics model to account for stochastic modeling considerations, and by adding new kinetic moves that allow bimolecular interactions between strands. Furthermore, we prove that this new kinetics and thermodynamics model is consistent with the prior work on multiple strand thermodynamics models [5].

The **Multistrand** simulator is based on the Gillespie algorithm [8] for generating statistically correct trajectories of a stochastic Markov process. We developed data structures and algorithms that take advantage of local properties of secondary structures. These algorithms enable the efficient reuse of the basic objects that form the system, such that only a very small part of the state's neighborhood information needs to be recalculated with every step. A key addition was the implementation of algorithms to handle the new kinetic steps that occur between different DNA strands, without increasing the time complexity of the overall simulation. These improvements lead to a reduction in worst case time complexity of a single step and also led to additional improvements in the average case time complexity.

What data does the simulation produce? At the very simplest, the simulation produces a full kinetic trajectory through the state space - the exact states it passed through, and the time at which it reached them. A small system might produce trajectories that pass through hundreds of thousands of states, and that number increases rapidly as the system gets larger. Going back to our original question, the type of information a researcher hopes to get out of the data could be very simple: "How quickly does the walker take a step?", with the implied question of whether it's worth it to actually purchase the particular DNA strands composing the walker to perform an experiment, or go back to the drawing board and redesign the device. One way to acquire that type of information is to look at the first time in the trajectory where we reached the "walker took a step" state, and record that information for a large number of simulated trajectories in order to obtain a useful answer. We designed and implemented new simulation modes that allow the full trajectory data to be condensed as it's generated into only the pieces the user cares about for their particular question. This analysis tool also required the development of flexible ways to talk about states that occur in trajectory data; if someone wants data on when the walker took a step, we have to be able to express that in terms of the Markov process states which meet that condition.

Chapter 2

System

We are interested in simulating nucleic acid molecules (DNA or RNA) in a stochastic regime; that is to say that we have a discrete number of molecules in a fixed volume. This regime is found in experimental systems that have a small volume with a fixed count of each molecule present, such as the interior of a cell. We can also apply this to experimental systems with a larger volume (such as a test tube) when the system is well mixed, as we can pick a fixed (small) volume and deal with the expected counts of each molecule within it, rather than the whole test tube.

To discuss the modeling and simulation of the system, we need to be very careful to define the components of the system, and what comprises a state of the system within the simulation.

2.1 Strands

Each DNA molecule to be simulated is represented by a *strand*. Our system then contains a set of strands Ψ^*, where each strand $s \in \Psi^*$ is defined by $s = (id, label, sequence)$. A strand's *id* uniquely identifies the strand within the system, while the *sequence* is the ordered list of nucleotides that compose the strand.

Two strands could be considered *identical* if they have the same sequence. However, in some cases it is convenient to make a distinction between strands with identical sequences. For example, if one strand were to be labelled with a fluorophore, it would no longer be physically identical to another with the same sequence but no fluorophore. Thus, the *label* is used to designate whether two strands are identical. We define two strands as being *identical* if they have the same labels and sequences. In most cases this distinction between

the label and the sequence is not used, so it will be explicitly noted when it is important.

2.2 Complex Microstate

A *complex* is a set of strands connected by base pairing (secondary structure). We define the state of a complex by $c = (ST, \pi^*, BP)$, called the "complex microstate". The components are a nonempty set of strands $ST \subseteq \Psi^*$, an ordering π^* on the strands ST, and a list of base pairings $BP = \{(i_j \cdot k_l) \mid$ base i on strand j is paired to base k on strand l, and $j \leq l$, with $i < k$ if $j = l\}$, where we note that "strand l" refers to the strand occuring in position l in the ordering π^*. Note that we require a complex to be "connected": there is no proper subset of strands in the complex for which the base pairings involving those strands do not involve at least one base outside that subset. Given a complex microstate c, we will use $ST(c), \pi^*(c), BP(c)$ to refer to the individual components.

While this definition defines the full space of complex microstates, it is common to disallow some secondary structures due to physical or computational constraints. For example, we disallow the pairing of a base with any other within three bases on the same strand, as this would correspond to an impossible physical configuration. Another class of disallowed structures are called the *pseudoknotted* secondary structures, which require computationally difficult energy model calculations, and are fully defined and discussed further in Section D.

2.3 System Microstate

A system microstate represents the configuration of the strands in the volume we are simulating (the "box"). Since we allow complexes to be formed of one or more strands, every unique strand in the system must be present in a single complex and thus we can represent the system microstate by a set of those complexes.

We define a *system microstate* i as a set of complex microstates, such that each strand in the system is in exactly one complex within the system. This is formally stated in the following equation:

$$\bigcup_{c \in i} ST(c) = \Psi^* \text{ and } \forall c, c' \in i \text{ with } c \neq c', ST(c) \cap ST(c') = \emptyset \tag{2.1}$$

This definition leads to the natural use of $|i|$ to indicate the number of complexes present in system microstate i, and $i \setminus j$ to indicate the complex microstates present in system microstate i that are not in j.

Chapter 3

Energy

The conformation of a nucleic acid strand at equilibrium can be predicted by a well studied model, called the nearest neighbor energy model [16, 15, 17]. Recent work has extended this model to cover systems with multiple interacting nucleic acid strands [5].

The distribution of system microstates at equilibrium is a Boltzmann distribution, where the probability of observing a microstate i is given by

$$Pr(i) = \frac{1}{Q} * e^{-\Delta G_{box}(i)/RT} \tag{3.1}$$

where $\Delta G_{box}(i)$ is the free energy of the system microstate i, and is the key quantity determined by these energy models. $Q = \sum_i e^{-\Delta G_{box}(i)/RT}$ is the partition function of the system, R is the gas constant, and T is the temperature of the system.

3.1 Energy of a System Microstate

We now consider the energy of the system microstate i, and break it down into components. The system consists of many complex microstates c, each with their own energy. We also must account for the entropy of the system (the number of configurations of the complexes spatially within the "box") in the energy, and thus must define these two terms.

Let us first consider the entropy term. We consider the "zero" energy system microstate to be the one in which all strands are in separate complexes, thus our entropy term is in terms of the reduction of available states caused by having strands join together. We assume

that the number of complexes in the system, N, is much smaller than the number of solvent molecules within our box, M_s. We can then approximate the standard statistical entropy of the system as $N * RT \log M_s$. Letting K be the total number of strands in the system, our zero state is then $K * RT \log M_s$. Defining $\Delta G_{volume} = RT \log M_s$, the contribution to the energy of the system microstate i from the entropy of the box is then:

$$(K - N) * \Delta G_{volume}$$

And thus in terms of $N, K, \Delta G_{volume}$ and $\overline{\Delta G}(c)$ (the energy of complex microstate c, defined in the next section), we define $\Delta G_{box}(i)$, the energy of the system microstate i, as follows:

$$\Delta G_{box}(i) = (K - N) * \Delta G_{volume} + \sum_{c \in i} \overline{\Delta G}(c)$$

Before we turn our attention to the energy of a complex microstate, let us examine a situation where we are modeling a fixed volume of a larger solution, and how that relates to the quantity M_s. If we assume that the volume studied is V (units of liters), we can easily compute the number of solvent molecules in this volume using the density d of water (the solvent, in g/L) , the molar mass M of water (in grams per mol), and Avogadro's number, as follows: $M_s = \frac{V*d}{M} * N_A$. In practice, we may wish to choose a simulation volume V based on other physical quantities, such as the concentration of a single molecule within the volume[1].

The energy formulas derived here, suitable for our stochastic model, differ from those in [5] in two main ways: the lack of symmetry terms, and the addition of the ΔG_{volume} term. We compare this stochastic model to the mass action model in much more detail in Section C.

[1]Calculating M_s from a concentration u in mol/L of a single molecule in the volume is straightforward. We assume that our concentration u implies the volume V is chosen corresponding to exactly one molecule being present in that volume, as follows: $V = \frac{1}{u*N_A}$ and thus $M_s = \frac{d}{M*u} = \frac{\rho_{H_2O}}{u}$ where ρ_{H_2O} is the molarity of water (55.14 mol/L at 37 °C) and the other quantities are as defined above.

3.2 Energy of a Complex Microstate

We previously defined a complex microstate in terms of the list of base pairings present within it. However, the well studied models are based upon nearest neighbor interactions between the nucleic acid bases. These interactions divide the secondary structure of the system into local components which we refer to as *loops*, shown in figure 3.1.

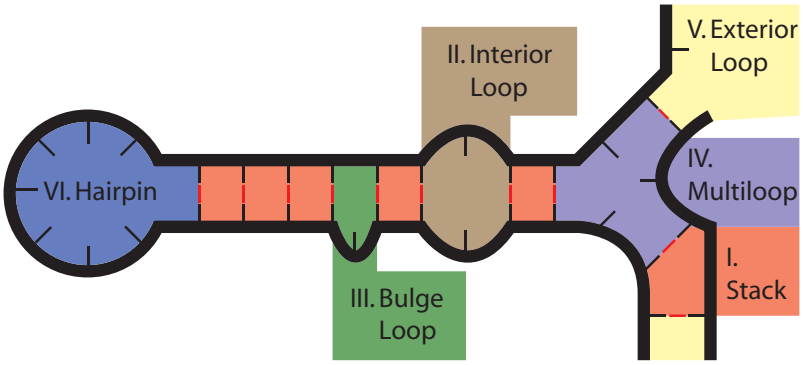


Figure 3.1: Secondary structure divided into loops.

These loops can be broken down into different categories, and parameter tables for each category have been determined from experimental data [17]. Each loop l has an energy, $\Delta G(l)$ which can be retrieved from the appropriate parameter table for its category, which is discussed in more detail in section B.3. Each complex also has an energy contribution associated with the entropic initiation cost [3] (e.g. rotational) of bringing two strands together, ΔG_{assoc}, and the total contribution is proportional to the number of strands L within the complex, as follows [2]: $(L-1) * \Delta G_{assoc}$.

[2]The free energy ΔG° for a reaction $A + B \rightleftharpoons C$ is usually expressed in terms of the equilibrium constant K_{eq} and the concentrations $[A], [B], [C]$ (in mol/L) of the molecules involved, as follows: $e^{\Delta G^\circ/RT} = K_{eq} = \frac{[A][B]}{[C]}$. We can also express the free energy $\Delta G'$ in terms of the dimensionless mole fractions x_A, x_B, x_C, where $x_i = [i]/\rho_{H_2O}$ (for dilute solutions), and ρ_{H_2O} is the molarity of water. In this case, we have $e^{\Delta G'/RT} = K'_{eq} = \frac{x_A * x_B}{x_C}$, and relating it to the previous equation, we see that $e^{\Delta G'/RT} = \frac{([A]/\rho_{H_2O}) * ([B]/\rho_{H_2O})}{[C] * \rho_{H_2O}} = \frac{[A][B]}{[C]} * \frac{1}{\rho_{H_2O}} = e^{\Delta G^\circ/RT} * e^{-\log \rho_{H_2O}}$. Thus if we have an energy ΔG° which was for concentration units and we wish to use mole fraction units, we must adjust it by $-RT \log \rho_{H_2O}$ to obtain the correct quantity. In general, if we have a complex of N molecules, the conversion to mole fractions will require an adjustment of $-(N-1) * RT \log \rho_{H_2O}$. To be consistent with [5], we wish to always use free energies which are based on the mole fraction units, and thus must include this factor since the reference free energies are for concentration units. In [5], the factor is included in the ΔG_{assoc} term, and thus we include it in the same place, as follows: $\Delta G_{assoc} = \Delta G^{pub}_{assoc} - RT \log \rho_{H_2O}$, where ΔG^{pub}_{assoc} is found in [3]. Thus our ΔG_{assoc} is the same as the ΔG^{assoc} found in [5] (footnote 13).

The energy of a complex microstate c is then the sum of these two types of contributions. We can also divide any free energy ΔG into the enthalpic and entropic components, ΔH and ΔS related by $\Delta G = \Delta H + T * \Delta S$, where T is the temperature of the system. For a complex microstate, each loop can have both enthalpic and entropic components, but ΔG_{assoc} is usually assumed to be purely entropic [16]. This becomes important when determining the kinetic rates, in section 4.

We use $\overline{\Delta G}(c)$ to refer to the energy of a complex microstate to be consistent with the nomenclature in [5], where $\overline{\Delta G}(c)$ refers to the energy of a complex when all strands within it are consider unique (as is the case in our system), and $\Delta G(c)$ is the energy of the complex, without assuming that all strands are unique (and thus it must account for rotational symmetries). This is discussed more in Section C.

In summary, the standard free energy of a complex microstate c, containing $L = |ST(c)|$ strands:

$$\overline{\Delta G}(c) = \left(\sum_{\text{loop } l \in c} \Delta G(l) \right) + (L-1)\Delta G_{assoc}$$

3.3 Computational Considerations

While the simulator could use the system microstate energy in the form given in the previous sections, it is convenient for us to group terms such that the computation need only take place per complex. Thus we wish to include the $(K - N)\Delta G_{volume}$ term in the energy computation for the complex microstates. Recall that K is the number of strands in the system, and N is the number of complexes in the system microstate. Assuming that we are computing the energy ΔG_{box} of system microstate i, we can rewrite K and N as follows:

$$\begin{aligned} K &= \sum_{c \in i} |ST(c)| \\ N &= \sum_{c \in i} 1 \end{aligned}$$

And thus arrive at:

$$\Delta G_{box}(i) = \sum_{c \in i} \left(\overline{\Delta G}(c) + (|ST(c)| - 1) * \Delta G_{volume} \right)$$

We then define $\Delta G^*(c) = \overline{\Delta G}(c) + (|ST(c)| - 1) * \Delta G_{volume}$, and $L_c = |ST(c)|$ and thus have the following forms for the energy of a system microstate and the energy of a complex microstate:

$$\begin{aligned} \Delta G_{box}(i) &= \sum_{c \in i} \Delta G^*(c) \\ \Delta G^*(c) &= \left(\sum_{\text{loop } l \in c} \Delta G(l) \right) + (L_c - 1) * (\Delta G_{assoc} + \Delta G_{volume}) \end{aligned}$$

Since we expect the probability of observing a particular complex microstate to remain the same no matter what reference units we use for the free energy (see footnote 2), this implies that if we wanted to express our $\Delta G^*(c)$ for concentration units, we would use $\Delta G_{assoc} = \Delta G^{pub}_{assoc}$ and $\Delta G_{volume} = RT \log \frac{M_s}{\rho_{H_2O}} = RT \log \frac{1}{u} = RT \log \frac{V}{V_0}$, where u is the molar concentration of a single molecule in the box volume V, and V_0 is the volume for 1 molecule at the standard concentration of 1 M.

Chapter 4

Kinetics

4.1 Basics

Thermodynamic predictions have only limited use for some systems of interest, if the key information to be gathered is the reaction rates and not the equilibrium states. Many systems have well defined ending states that can be found by thermodynamic prediction, but predicting whether it will reach the end state in a reasonable amount of time requires modeling the kinetics. Kinetic analysis can also help uncover poor sequence designs, such as those with alternate reactions leading to the same states, or kinetic traps which prevent an intended reaction from occurring quickly.

The kinetics are modeled as a continuous time Markov process over secondary structure space. System microstates i, j are considered adjacent if they differ by a single base pair (Figure 4.1), and we choose the transition rates k_{ij} (the transition from state i to state j) and k_{ji} such that they obey detailed balance:

$$\frac{k_{ij}}{k_{ji}} = e^{-\frac{\Delta G_{box}(j) - \Delta G_{box}(i)}{RT}} \tag{4.1}$$

This property ensures that given sufficient time we will arrive at the same equilibrium state distribution as the thermodynamic prediction, (i.e. the Boltzmann distribution on system microstates, equation 3.1) but it does not fully define the kinetics as this only constrains the ratio $\frac{k_{ij}}{k_{ji}}$. We discuss how to choose these transition rates in the following sections, but regardless of this choice, we can still determine how the next state is chosen and the time at which that transition occurs.

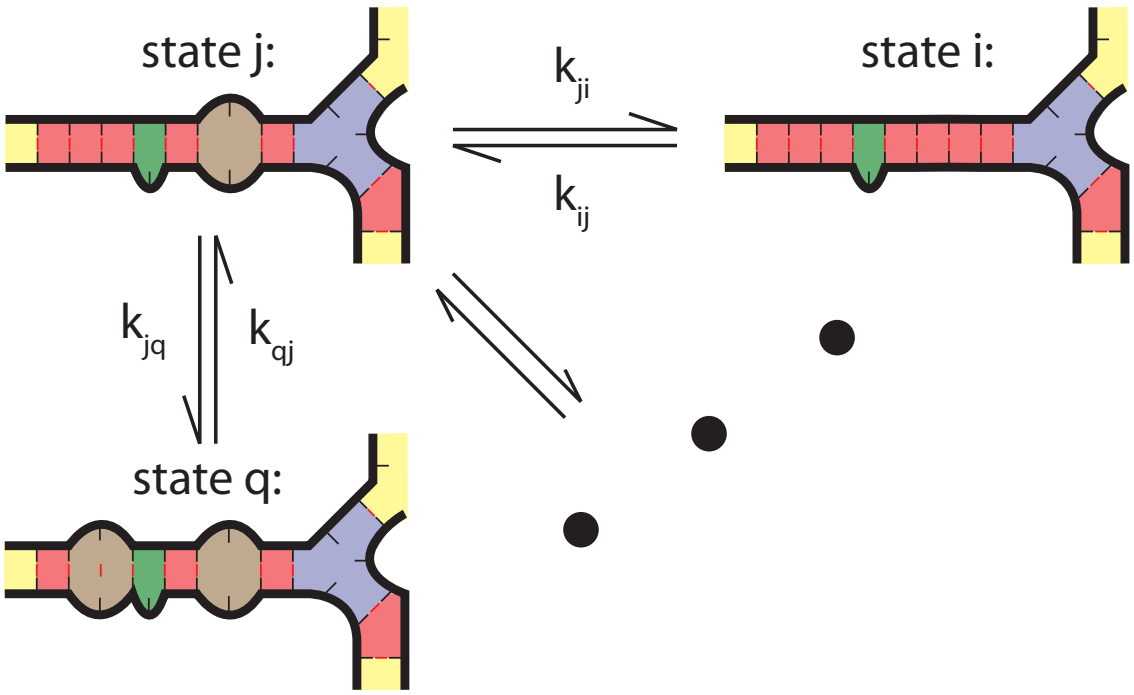


Figure 4.1: System microstates i, q adjacent to current state j, with many others not shown.

Given that we are currently in state i, the next state m in a simulated trajectory is chosen randomly among the adjacent states j, weighted by the rate of transition to each.

$$Pr(m) = \frac{k_{im}}{\Sigma_j k_{ij}} \tag{4.2}$$

Similarly, the time taken to transition to the next state is chosen randomly from an exponential distribution with rate parameter λ, where λ is the total rate out of the current state, $\Sigma_j k_{ij}$.

$$Pr(\Delta t) = \lambda \exp(-\lambda \Delta t) \tag{4.3}$$

We will now classify transitions into two exclusive types: those that change the number of complexes present in the system, called *bimolecular transitions*, and those where changes are within a single complex, called *unimolecular transitions*.

4.2 Unimolecular Transitions

Because unimolecular transitions involve only a single complex, it is natural to define these transitions in terms of the complex microstate which changed, rather than the full system microstate. Like Figure 4.1 implies, we define a complex microstate d as being adjacent to a complex microstate c if it differs by exactly one base pair. We call a transition from c to d that adds a base pair a *creation* move, and a transition from c to d that removes a base pair a *deletion* move. The exclusion of pseudoknotted structures is not inherent in this definition of adjacent states, but rather arises from our disallowing pseudoknotted complex microstates.

In more formal terms we now define the adjacent states to a system microstate, rather than those adjacent to a complex microstate as in the simple definition above. Recall from section 2.3 that $|i|$ is the number of complexes present in system microstate i, and $i \setminus j$ is the set of complex microstates in i that are not also in system microstate j.

Two system microstates i, j are adjacent by a unimolecular transition iff $\exists c \in i, d \in j$ such that:

$$|i| = |j| \text{ and } i \setminus j = \{c\} \text{ and } j \setminus i = \{d\} \tag{4.4}$$

and one of these two holds:

$$BP(c) \subset BP(d) \text{ and } |BP(d)| = |BP(c)| + 1 \tag{4.5}$$

$$BP(d) \subset BP(c) \text{ and } |BP(c)| = |BP(d)| + 1 \tag{4.6}$$

In other words, the only differences between i and j are in c and d, and they differ by exactly one base pair. If equation 4.5 is true, we call the transition from i to j a *base pair creation move*, and if equation 4.6 is true, we call the transition from i to j a *base pair deletion move*. Note that if i to j is a creation move, j to i must be a deletion move, and vice versa. Similarly, if there is no transition from i to j, there cannot be a transition from j to i, which implies that every unimolecular move in this system is reversible.

4.3 Bimolecular Transitions

A bimolecular transition from system microstate i to system microstate j is one where the single base pair difference between them leads to a differing number of complexes within each system microstate. This differing number of complexes could be due to a base pair joining two complexes in i to form a single complex in j, which we will call a *join move*. Conversely, the removal of this base pair from i could cause one complex in i to break into two complexes within j, which we will call a *break move*. Note that if i to j is a join move, then j to i must be a break move, and vice versa. As we saw before, this also implies that every bimolecular move is reversible.

Formally, a transition from system microstate i to system microstate j is a join move if $|i| = |j| + 1$ and we can find complex microstates $c, c' \in i$ and $d \in j$, with $c \neq c'$ such that the following equations hold:

$$i \setminus \{c, c'\} = j \setminus \{d\} \tag{4.7}$$

$$\exists x \in BP(d) \text{ s.t. } BP(d) \setminus \{x\} = BP(c) \cup BP(c') \tag{4.8}$$

Similarly, a transition from system microstate i to system microstate j is a break move if $|i| + 1 = |j|$ and we can find complex microstates $c \in i$ and $d, d' \in j$ with $d \neq d'$ such that the following equations hold:

$$i \setminus \{c\} = j \setminus \{d, d'\} \tag{4.9}$$

$$\exists x \in BP(c) \text{ s.t. } BP(c) \setminus \{x\} = BP(d) \cup BP(d') \tag{4.10}$$

While arbitrary bimolecular transitions are not inherently prevented from forming pseudoknots in this model, we again implicitly prevent them by using only complex microstates that are not pseudoknotted.

4.4 Transition Rates

Now that we have defined all of the possible transitions between system microstates, we must decide how to assign rates to each transition. We know that if there is a transition from system microstate i to system microstate j with rate k_{ij} there must be a transition from j to i with rate k_{ji} which are related by:

$$\frac{k_{ij}}{k_{ji}} = e^{-\frac{\Delta G_{box}(j) - \Delta G_{box}(i)}{RT}} \tag{4.11}$$

This condition is known as *detailed balance*, and does not completely define the rates k_{ij}, k_{ji}. Thus a key part of our model is the choice of *rate method*, the way we set the rates of a pair of reactions so that they obey detailed balance.

While our simulator can use any arbitrary rate method we can describe, we would like our choice to be physically realistic (i.e. accurate and predictive for experimental systems). There are several rate methods found in the literature [10, 11, 26] which have been used for kinetics models for single-stranded nucleic acids [7, 26] with various energy models. As a start, we have implemented three of these simple rate methods which were previously used in single base pair elementary step kinetics models for single stranded systems. In addition we present a rate method for use in bimolecular transitions that is physically consistent with both mass action and stochastic chemical kinetics. We verify that the kinetics model (and thus our choice of rate method) have been correctly implemented by verifying that the detailed balance condition holds (Section 6.1.2).

In order to maintain consistency with known thermodynamic models, each pair of k_{ij} and k_{ji} must satisfy detailed balance and thus their ratio is determined by the thermodynamic model, but in principle each pair could be independently scaled by some arbitrary prefactor, perhaps chosen to optimize agreement with experimental results on nucleic acid kinetics. However, since the number of microstates is exponential, this leads to far more model parameters (the prefactors) than is warranted by available experimental data. For the time being, we limit ourselves to using only two scaling factors: k_{uni} for use with unimolecular transitions, and k_{bi} for bimolecular transitions.

4.5 Unimolecular Rate Models

The first rate model we will examine is the Kawasaki method [10]. This model has the property that both "downhill" (energetically favorable) and uphill transitions scale directly with the steepness of their slopes.

$$k_{ij} = k_{uni} * e^{-\frac{\Delta G_{box}(j) - \Delta G_{box}(i)}{2RT}} \tag{4.12}$$

$$k_{ji} = k_{uni} * e^{-\frac{\Delta G_{box}(i) - \Delta G_{box}(j)}{2RT}} \tag{4.13}$$

The second rate model under consideration is the Metropolis method [11]. In this model, all downhill moves occur at the same fixed rate, and only the uphill moves scale with the slope. This means that the maximum rate for any move is bounded, and in fact all downhill moves occur at this rate. This is in direct contrast to the Kawasaki method, where there is no bound on the maximum rate.

$$\text{if } \Delta G_{box}(i) > \Delta G_{box}(j) \text{ then} \quad k_{ij} = 1 * k_{uni} \tag{4.14}$$

$$k_{ji} = k_{uni} * e^{-\frac{\Delta G_{box}(i) - \Delta G_{box}(j)}{RT}} \tag{4.15}$$

$$\text{otherwise,} \quad k_{ij} = k_{uni} * e^{-\frac{\Delta G_{box}(j) - \Delta G_{box}(i)}{RT}} \tag{4.16}$$

$$k_{ji} = 1 * k_{uni} \tag{4.17}$$

Finally, the entropy/enthalpy method [26] uses the division of free energies into entropic and enthalpic components to assign the transition rates in an intuitive manner: base pair creation moves must overcome the entropic energy barrier to bring the bases into contact, and base pair deletion moves must overcome the enthalpic energy barrier in order to break them apart.

$$\text{if } i \text{ to } j \text{ is a creation:} \quad k_{ij} = k_{uni} * e^{\frac{\Delta S_{box}(j) - \Delta S_{box}(i)}{R}} \tag{4.18}$$

$$k_{ji} = k_{uni} * e^{-\frac{\Delta H_{box}(i) - \Delta H_{box}(j)}{RT}} \tag{4.19}$$

$$\text{otherwise,} \quad k_{ij} = k_{uni} * e^{-\frac{\Delta H_{box}(j) - \Delta H_{box}(i)}{RT}} \tag{4.20}$$

$$k_{ji} = k_{uni} * e^{\frac{\Delta S_{box}(i) - \Delta S_{box}(j)}{R}} \tag{4.21}$$

We note that the value of k_{uni} that best fits experimental data is likely to be different for all three models.

4.6 Bimolecular Rate Model

When dealing with moves that join or break complexes, we must consider the choice of how to assign rates for each transition in a new light. In the particular situation of the join move, where two molecules in a stochastic regime collide and form a base pair, this rate is expected to be modeled by stochastic chemical kinetics.

Stochastic chemical kinetics theory [8] tells us that there should be a rate constant k such that the propensity of a particular bimolecular reaction between two species X and Y should be $k * \#X * \#Y/V$, where $\#X$ and $\#Y$ are the number of copies of X and Y in the volume V. Since our simulation considers each strand to be unique, $\#X = \#Y = 1$, and thus we see the propensity should scale as $1/V$. Recalling that $\Delta G_{volume} = RT \log(V * y)$, where y is a collection of constant terms (discussed in Section 3.1) and V is the simulated volume, we see that we can obtain the $1/V$ scaling by letting the join rate be proportional to $e^{-\Delta G_{volume}/RT}$.

Thus, we arrive at the following rate method, and note that the choice of k (above) or our scalar term k_{bi} can be found by comparison to experiments measuring the hybridization rate of oligonucleotides [21].

$$\text{if } i \text{ to } j \text{ is a complex join move:} \quad k_{ij} = k_{bi} * e^{\frac{-\Delta G_{volume}}{RT}} \tag{4.22}$$

$$k_{ji} = k_{bi} * e^{-\frac{\Delta G_{box}(i) - \Delta G_{box}(j) + \Delta G_{volume}}{RT}} \tag{4.23}$$

$$\text{otherwise,} \quad k_{ij} = k_{bi} * e^{-\frac{\Delta G_{box}(j) - \Delta G_{box}(i) + \Delta G_{volume}}{RT}} \tag{4.24}$$

$$k_{ji} = k_{bi} * e^{-\frac{\Delta G_{volume}}{RT}} \tag{4.25}$$

This formulation is convenient for simulation, as the join rates are then independent of the resulting secondary structure. We could use the other choices for assigning rates from 4.4, but they would require much more computation time. While the above model is of course an approximation to the physical reality (albeit one which we believe at least intuitively agrees with what we expect from stochastic chemical kinetics), if we later determine there is a better approximation we could use that instead, even if it cost us a bit in computation time. One issue in the above model that we wish to revisit in the future is that due to the rate being determined for **every** possible first base pair between two complexes, the overall rate for two complexes to bind (by a single base pair) is proportional roughly to the square of the number of exposed nucleotides, in addition to the $\frac{1}{V}$ dependence noted earlier.

Chapter 5

The Simulator : Multistrand

Energy and kinetics models similar to these can been solved analytically; however, the standard master equation methods [22] scale with the size of the system's state space. For our DNA secondary structure state space, the size gets exponentially large as the strand length increases, so these methods become computationally prohibitive. One alternate method we can use is stochastic simulation [8], which has previously been done for single-stranded DNA and RNA folding (the **Kinfold** simulator [7]). Our stochastic simulation refines these methods for our particular energetics and kinetics models, which extends the simulator to handle systems with multiple strands and takes advantage of the localized energy model for DNA and RNA.

5.1 Data Structures

There are two main pieces that go into this new stochastic simulator. The first piece is the multiple data structures needed for the simulation: the *loop graph* which represents the complex microstates contained within a system microstate (Section 5.1.2), the *moves* which represent transitions in our kinetics model – the single base pair changes in our structure that are the basic step in the Markov process, and the *move tree* the container for moves that lets us efficiently store and organize them (Section 5.1.3).

5.1.1 Energy Model

Since the basic step for calculating the rate of a move involves the computation of a state's energy, we must be able to handle the energy model parameter set in a manner that simplifies this computation. Previous kinetic simulations (Kinfold) rely on the energy model we have

described, though without the extension to multiple strand systems. While the format of the parameter set that is used remains the same, we must implement an interface to this data which allows us to quickly compute the energy for particular loop structures (local components of the secondary structure, described in 3.2). This allows us to do the energy computations needed to compute the kinetic rates for individual components of the system microstate, allowing us to use more efficient algorithms for recomputing the energy and moves available to a state after each Markov step.

The energy model parameter set and calculations are implemented in a simple modular data structure that allows for both the energy computations at a local scale as we have previously mentioned, but also as a flexible subunit that can be extended to handle energy model parameter sets from different sources. In particular, we have implemented two particular parameter set sources: the NUPACK parameter set [24] and the Vienna RNA parameters [9] (which does not include multistranded parameters, so defaults for those are used). Adding new parameter set sources (such as the mfold parameters [27]) is a simple extension of the existing source code. Additionally, the energy model interface allows for easy extension of existing models to handle new situations, e.g. adding a sequence dependent term for long hairpins. We hope this energy model interface will be useful for future research where authors may wish to simulate systems with a unique energy model and kinetics model.

5.1.2 The Current State: Loop Structure

A system microstate can be stored in many different ways, as shown in figure 5.1. Each of these has different advantages: the flat ("dot-paren") representation (Figure 5.1C) can be used for both the input and output of non-pseudoknotted structures, but the information contained in the representation needs additional processing to be used in an energy computation (we must break it into loops). Base pair list representation (Figure 5.1B) allows the definition of secondary structures which include pseudoknots, but also requires processing for energy computation. Loop representation (Figure 5.1D) allows the energy to be computed and stored in local components, but requires processing to obtain the global structure, used in input and output. While the loop graph cannot represent pseudoknotted structures without introducing a loop type for pseudoknots (for which we may not know how to calculate the energy), and making the loop graph cyclic, since this work is primar-

ily concerned with non-pseudoknotted structures this is only a minor point. In the future when we have excellent pseudoknot energy models, we will have to revisit this choice and hopefully find a good representation that still allows us similar computational efficiency.

We use the loop graph representation for each complex within a system microstate, and organize those with a simple list. This gives us the advantage that the energy can be computed for each individual node in the graph, and since each move only affects a small portion of the graph (Figure 5.3), we will only have to compute the energy for the affected nodes. While providing useful output of the current state then requires processing of the graph, it turns out to be a constant time operation if we store a flat representation which

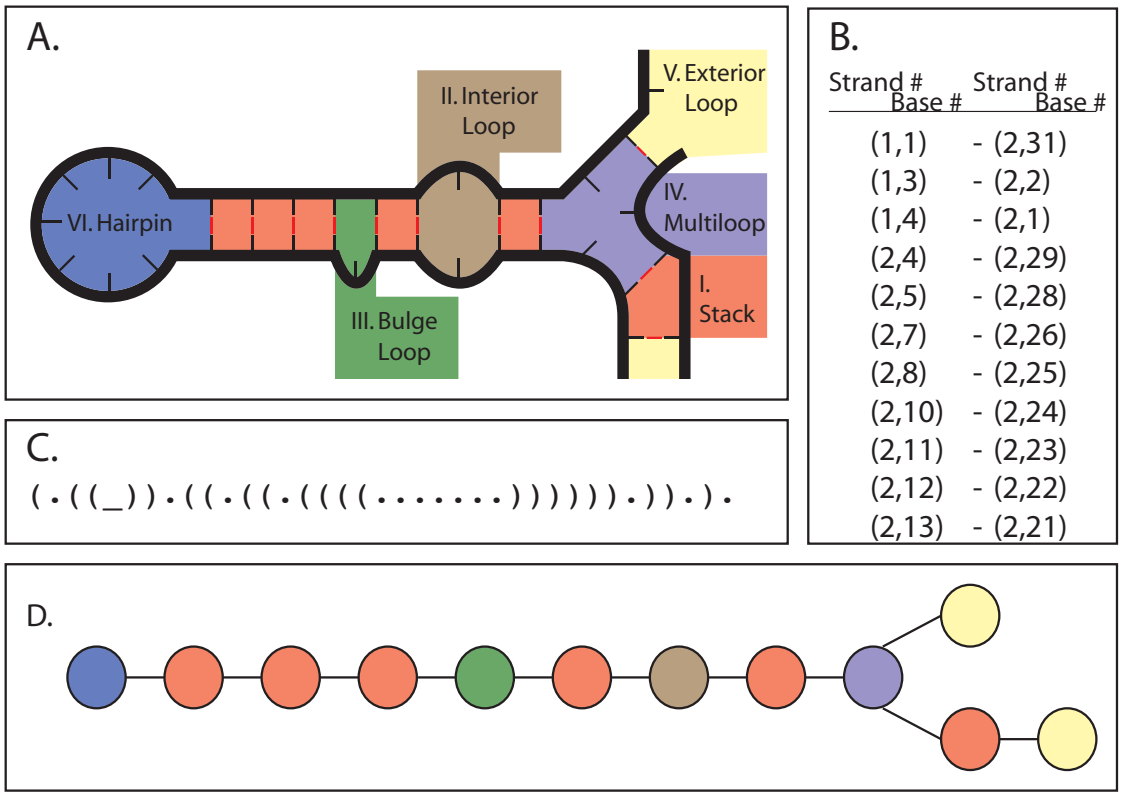


Figure 5.1: Example secondary structure, with different representations: (A) Original loop diagram representation. (B) Base pair list representation. Each base pairing is represented by the indices of the bases involved. (C) Dot-paren representation, also called the flat representation. Each base is represented by either a period, representing an unpaired base, or by a parenthesis, representing a pairing with the base that has the (balanced) matching parenthesis. An underscore represents a break between multiple strands. (D) Loop graph representation. Each loop in the secondary structure is a single node in the graph, which contains the sequence information within the loop.

gets updated incrementally as each move is performed by the simulator.

We contrast this approach with that in the original Kinfold, which uses a flat representation augmented by the base pairing list computed from it. Since we use a loop graph augmented by a flat representation, our space requirements are clearly greater, but only in a linear fashion: for each base pair in the list, we have exactly two loop nodes which must include the same information and the sequence data in that region.

5.1.3 Reachable States: Moves

When dealing with a flat representation or base pair list for a current state, we can simply store an available move as the indices of the bases involved in the move, as well as the rate at which the transition should occur. This approach is very straightforward to implement (as was done in the original Kinfold), and we can store all of the moves for the current state in a single global structure such as a list. However, when our current state is represented as a loop graph this simple representation can work, but does not contain enough information to efficiently identify the loops affected by the move. Thus we elect to add enough complexity to how we store the moves so that we can quickly identify the affected nodes in our loop graph, which allows us to quickly identify the loops for which we need to recalculate the available moves.

We let each move contain a reference to the loop(s) it affects (Figure 5.2A), as well as an index to the bases within the loop, such that we can uniquely identify the structural change that should be performed if this move is chosen. This reference allows us to quickly find the affected loop(s) once a move is chosen. We then collect all the moves which affect a particular loop and store them in a container associated with the loop (Figure 5.2B). This allows us to quickly access all the moves associated with a loop whose structure is being modified by the current move. We should note that since deletion moves by nature affect the two loops adjacent to the base pair being deletion, they must necessarily show up in the available moves for either loop. This is handled by including a copy of the deletion move in each loop's moves, and halving the rate at which each occurs.

Finally, since this method of move storage is not a global structure, we add a final layer of complexity on top, so that we can easily access all the moves available from the current state without needing to traverse the loop graph. This is as simple as storing each loop's move container in a larger structure such as a list or a tree, which represents the entire

complex's available moves as shown in figure 5.2C.

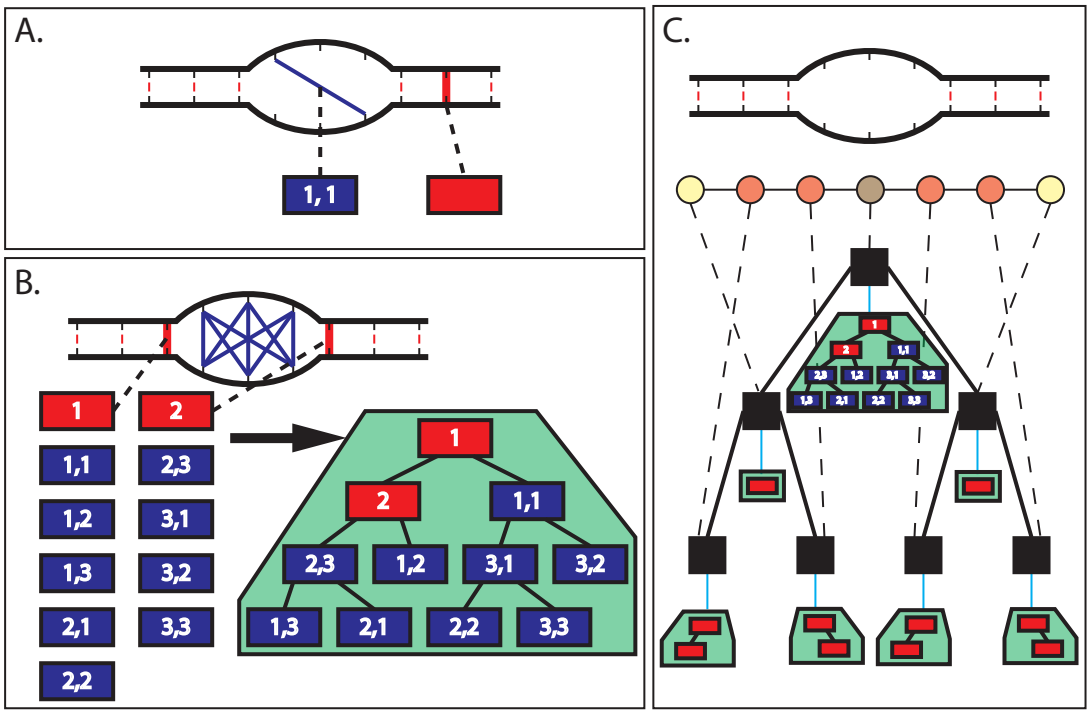


Figure 5.2: (A) Creation moves (blue line) and deletion moves (red highlight) are represented here by rectangles. Either type of move is associated with a particular loop, and has indices to designate which bases within the loop are affected. (B) All possible moves which affect the interior loop in the center of the structure. These are then arranged into a tree (green area), which can be used to quickly choose a move. (C) Each loop in the loop graph then has a tree of moves that affect it, and we can arrange these into another tree (black boxes), each node of which is associated with a particular loop (dashed line) and thus a tree of moves (blue line). This resulting tree then contains all the moves available in the complex.

5.2 Algorithms

The second main piece of the simulator is the algorithms that control the individual steps of the simulator. The algorithm implementing the Markov process simulation closely follows the Gillespie algorithm[8] in structure:

1. Initialization: Generate the initial loop graph representing the input state, and compute the possible transitions.

2. Stochastic Step: Generate random numbers to determine the next transition (5.2.1), as well as the time interval elapsed before the transition occurs.

3. Update: Change the current loop graph to reflect the chosen move (5.2.2). Recompute the available transitions from the new state (5.2.3). Update the current time using the time interval in the previous step.

4. Check Stopping Conditions: check if we are at some predetermined stopping condition (such as a maximum amount of simulated time) and stop if it is met. Otherwise, go back to step 2. These stopping conditions and other considerations relating to providing output are discussed further in section 6.

The striking difference between this structure and the Gillespie algorithm is the necessity of recomputing the possible transitions from the current state at every step, and the complexity of that recalculation. Since we are dealing with an exponential state space we have no hope of storing all possible transitions between any possible pair of states, and instead must look at the transitions that occur only around the current state. Our examination of the key algorithms must include an analysis of their efficiency, so we define the key terms here:

1. N, the total length of the input's sequence(s).

2. T, the total amount of simulation time for each Monte Carlo trajectory.

3. J, the number of nodes in the loop graph of the current state. At worst case, this is $O(N)$, which occurs in highly structured configurations, like a full duplex.

4. K, the largest number of unpaired bases occuring in any loop in the current state. At worst case, this is exactly N, but on average it is typically much smaller.

5. L, the current number of complexes in the system. At worst this could be $O(N)$, but in practice the number of complexes is much fewer.

5.2.1 Move Selection

First let's look at the unimolecular moves in the system. The tree-based data structure containing the unimolecular moves leads to a simple choice algorithm that uses the generated random number to make a decision at each level of the tree based on the relative rates of the moves in each branch. We have two levels of tree to perform the choice on, the first having J nodes (one for every loop graph node) which each hold the local move containers

for a particular loop, and the second having at most $O(K^2)$ nodes (the worst case number of moves possible within a single loop). Thus our selection algorithm for unimolecular moves takes $O(\log(J) + \log(K))$ time to arrive at a final decision.

What about the moves that take place between two different complexes? With our method of assigning rates for these moves, we know that regardless of the resulting structure, all possible moves of this type must occur at the same rate. Thus the main problem is knowing how many such moves exist and then efficiently selecting one.

How many such moves exist? This is a straightforward calculation: for each complex microstate in the system, we count the number of A, G, C and T bases present in open loops within the complex. For the sake of example, let's call these quantities c_A, c_G, c_C, c_T for a complex microstate c. Let's also define the total number of each base in the system as follows: $A_{total} = \sum_{c \in s} c_A$, etc, where s in the system microstate we are computing the moves for. We can now compute how many moves there are where (for example) an A base in complex c becomes paired with a T base in any other complex: $c_A * (T_{total} - c_T)$, that is, the number of A bases within c multiplied by the number of T bases present in all other complexes in the system. So the number of moves between c and any other complex in the system is then $c_A * (T_{total} - c_T) + c_G * (C_{total} - c_C) + c_C * (G_{total} - c_G) + c_T * (A_{total} - c_A)$ and if we allow GT pairs, there are two additional terms with the same form. Summing over this quantity for each $c \in s$ we then get 2 times the total number of bimolecular moves (and in fact we can eliminate the redundency by using the total open loop bases in complexes "after" c in our data structure, rather than the total open loop bases in all complexes other than c). Since we do this in an algorithmic manner, it is straightforward to uniquely identify any particular move we need by simply following this counting process in reverse.

What is the time complexity for this bimolecular move choice? It is straightforward to see that calculating the total bimolecular move rate is $O(L)$ (recall L is the number of complexes within the system). Slightly more complex is choosing the bimolecular move, which must also be $O(L)$, as it takes 2 traversals through the list of complexes to determine the pair of reacting complexes in the bimolecular step. We note that for typical L and bimolecular reaction rates (e.g. typical strand concentrations which set our ΔG_{volume}) this quantity is quite small relative to that for the unimolecular reactions.

Our move choice algorithm can now be summed up as follows: given our random choice, decide whether the next move is bimolecular (using the total number of such moves as a

first step) or unimolecular (thus one of the ones stored in the tree). If it's bimolecular, reverse the counting process using the random number to pick the unique combination of open loops and bases involved in the bimolecular step. If it's a unimolecular step, pick a move out of the trees of moves for each complex in the system as discussed above.

5.2.2 Move Update

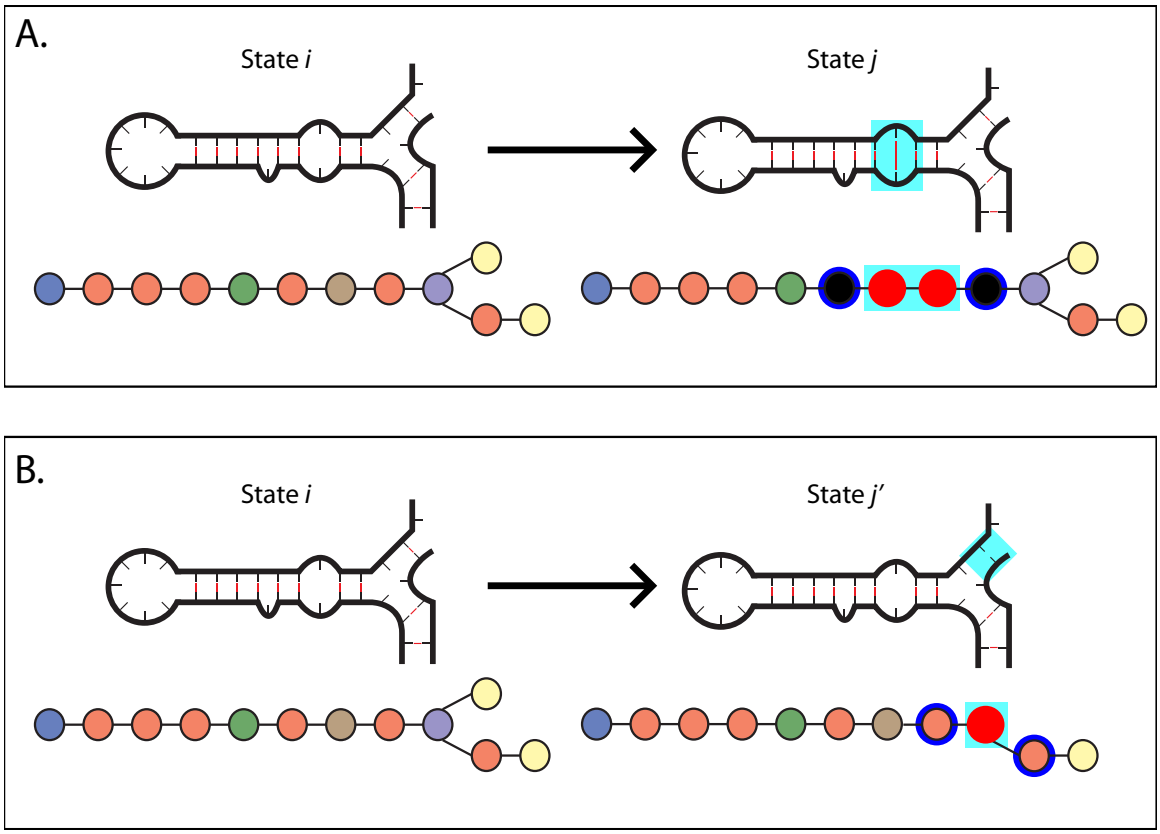


Figure 5.3: Moves of varying types which take current state i to state j. The changed region is highlighted in cyan. Loops that are in j but not i are highlighted in red (in the loop graph) and must be created and have their moves generated. Loops shown highlighted in blue have had an adjacent loop change, and thus must have their deletion moves recalculated. (A) A creation move. (B) A deletion move.

Once a move has been chosen, we must update the loop graph to reflect the new state. This is a straightforward substitution: for a creation move, which affects a single loop, we must create the resulting pair of loops which replace the affected loop and update the graph connections appropriately (Figure 5.3A). Similarly, for a deletion move, which affects two loops, we must create the single loop that results from joining the two affected loops, and update the graph connections appropriately (Figure 5.3B).

The computationally intensive part for this algorithm lies in the updating of the tree structure containing all the moves. We must remove the moves which involved the affected loops from the container, a process that takes $O(\log(J))$ time (assuming we implement tree deletions efficiently), generate the moves which correspond to the new loops (Section 5.2.3), and add these moves into the global move structure, which also takes $O(\log(J))$ time.

5.2.3 Move Generation

The creation and deletion moves must be generated for each new loop created by the move update algorithm, and we must update the deletion moves for each loop adjacent to an affected loop in the move update algorithm. The number of deletion moves which must be recalculated is fixed, though at worst case is linear in N, and so we will concern ourselves with the (typically) greater quantity of creation moves which need to be generated for the new loops.

For all types of loops, we can generate the creation moves by testing all possible combinations of two unpaired bases within the loop, tossing out those combinations which are too close together (and thus could not form even a hairpin, which requires at least three bases within the hairpin), and those for which the bases could not actually pair (for example, a T–T pairing). An example of this is shown for a simple interior loop, in figure 5.4. The remaining combinations are all valid, and we must compute the energy of the loops which would result if the base pair were formed, in order to compute the rate using one of the kinetic rate methods (Section 4.5). This means we need to check $O(L^2)$ possible moves and do two loop energy computations for each. At worst case, that is $O(N^2)$ energy computations in this step, and so the efficiency of performing an energy computation becomes vitally important.

Once we have generated these moves we must collect them into a tree which represents the new loop's available moves. This can be handled in a linear fashion in the number of moves with a simple tree creation algorithm, and thus it is in the same order as the number of energy computations.

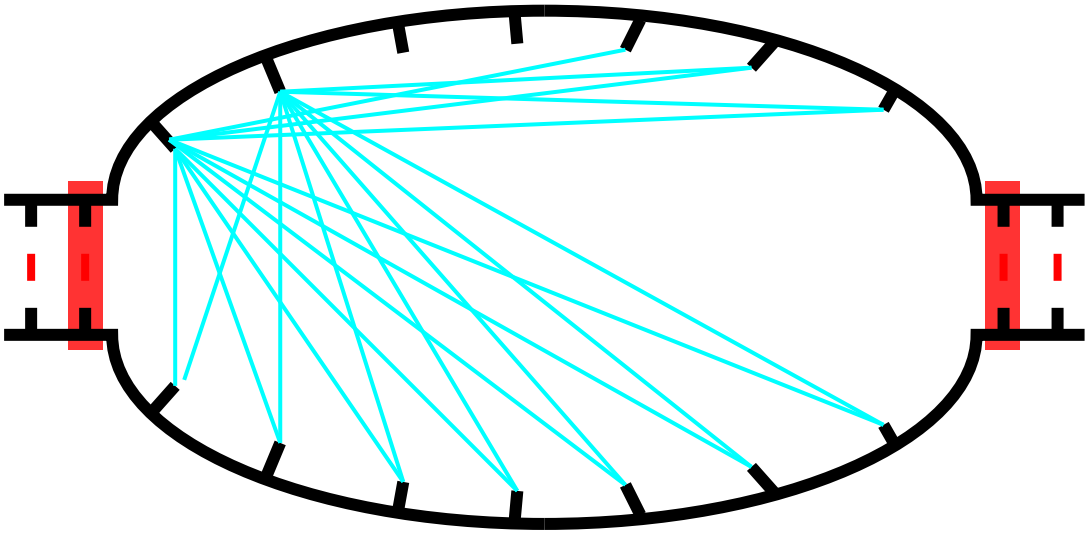


Figure 5.4: A interior loop, with all theoretically possible creation moves for the first two bases on the top strand shown as cyan lines, and all possible deletion moves shown as red boxes. Note that for each creation move shown here, we must check whether the bases could actually pair, and only then continue with the energy computations to determine the rate of the move.

5.2.4 Energy Computation

It is in the energy computation that our loop graph representation of the complex microstate shines, as the basic operation required for each possible move in the move generation step is computing the difference of energies between the affected loop(s) that would be present after the move and those present beforehand.

For all loop types except open loops and multiloops, computing the loop energy is as simple as looking up the sequence information (base pairs and adjacent bases) and loop sizes in a table [16, 27], and is a constant-time lookup. For open loops and multiloops, this computation is linear in the number of adjacent helices (e.g. adjacent nodes in the loop graph) if we are using an optional configuration of the energy model which adds energies that come from bases adjacent to the base pairs within the loop (called the "dangles" option). Theoretically we could have an open loop or multiloop that is adjacent to $O(N)$ other nodes in the loop graph, but this is an extraordinarily unlikely situation and present only with particular energy model options, so we will consider the energy computation step to be $O(1)$.

5.3 Analysis

Now that we have examined each algorithm needed to perform a single step of the stochastic simulation, we can derive the worst-case time. First, recall that J is the number of nodes in the loop graph, K is the largest number of unpaired bases in a loop, L is the number of complexes in the system and N is the total length of strands in the system. The move selection algorithm is $O(\log(J) + \log(K) + L) = O(N)$, move update is $O(\log(J)) = O(\log(N))$, and move generation is $O(N^2 * O(1))$, where energy computation is the $O(1)$ term. These algorithms are done in sequence and thus their times are additive: $O(N) + O(\log(N)) + O(N^2) = O(N^2)$. Thus our worst case time for a **single** Markov step is quadratic in the number of bases in our structure.

However, one step does not a kinetic trajectory make. We are attempting to simulate for a fixed amount of time T, as mentioned before, and so we must compute the expected number of steps needed to reach this time. Since the distribution on the time interval Δt between steps is an exponential distribution with rate parameter R, which is the total rate of all moves in the current state, we know that the expected $\Delta t = 1/R$. However, this still leaves us needing to approximate R in order to compute the needed amount of time for an entire trajectory. To make a worst case estimate, we must use the largest R that occurs in any given trajectory, as this provides the lower bound on the mean of the smallest time step Δt in that trajectory. However, the relative rates of favorable moves tends to be highly dependent on the rate method used: the Kawasaki method can have very large rates for a single move, while the Metropolis method has a maximum rate for any move, and the Entropy/Enthalpy method is also bounded in this manner as all moves have to overcome an energy barrier.

We thus make an average case estimate for the the total rate R, based on the number of "favorable" moves that typically have the largest rates. While a "favorable" move is merely one where the ΔG is negative (thus it results in an energetically more favorable state) or one which uses the maximum rate (for non-Kawasaki methods), the actual rate for these moves depends on the model chosen. The key question is whether we can come up with an average situation where there are $O(N^2)$ favorable moves or if $O(N)$ is more likely. What types of secondary structures give rise to quadratic numbers of moves? They are all situations where there are long unpaired sequences, whether in an interior loop, multiloop

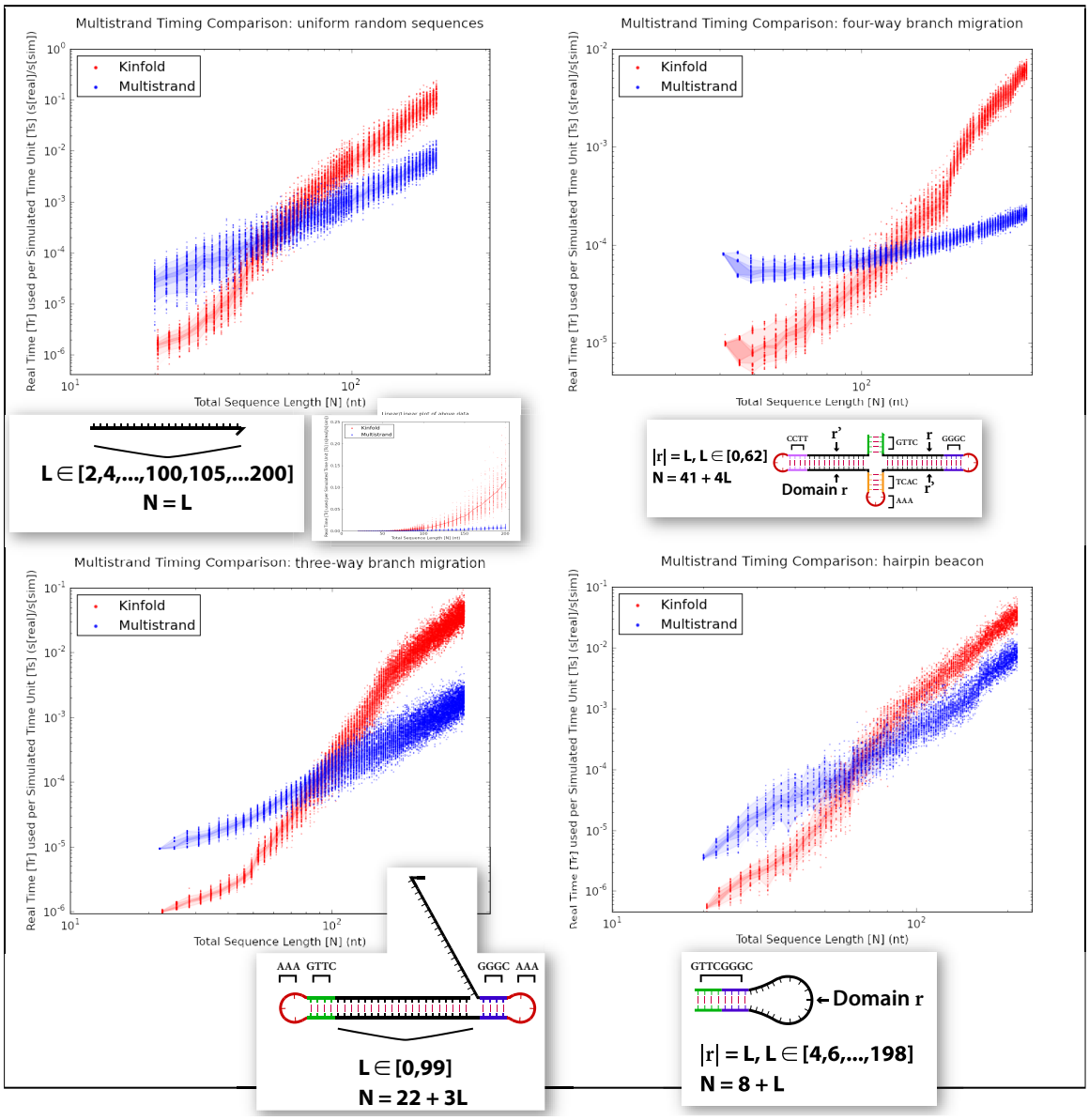


Figure 5.5: Comparison of real time used per simulated time unit between Multistrand and Kinfold 1.0, for four different single stranded systems with varying total length. All plots are log/log except for the inset, which is a linear/linear plot of the same data is in the uniform random sequence plot. The density of test cases is shown using overlaid regions of varying intensity. From lightest to darkest, these correspond to 80%, 40% and 10% of the test cases being within the region.

or open loop. These creation moves are generally unfavorable, except for the small number that lead to new stack loops. Thus we do not expect there to be a quadratic number of favorable moves. A linear number is much more likely: a long duplex region could reach a linear number (if say, $\frac{N}{4}$ bases were unpaired but could be formed easily into stacks). Thus,

we make the (weak) argument that a good average case is that the average rate is at worst $O(N)$.

From this estimate for the average rate, we conclude that each step would have an expected (lower bound) $\Delta t = 1/N$, and thus to simulate for time T we would need $T/(1/N) = T * N$ steps, and thus $O(T * N^3)$ time to simulate a single trajectory. Since this is the worst case behavior, it is fairly difficult to show this with actual simulation data, so instead we present a comprehensive comparison with the original Kinfold for a variety of test systems (Figure 5.5), noting that the resulting slopes on the log/log plots lie easily within the $O(N^3)$ bound for the time taken to simulate 1 time unit.

Chapter 6

Multistrand : Output and Analysis

We have now presented the models and algorithms that form the continuous time Markov process simulator. Now we move on to discuss the most important part of the simulator from a user's perspective: the huge volume of data produced by the simulation, and methods for processing that data into useful information for analyzing the simulated system.

How much data are we talking about here? Following the discussion in the previous chapter, we expect an average of $O(N)$ moves per time unit simulated. This doesn't tell us much about the actual amount of data, only that we expect it to not change drastically for different size input systems. In practice this amount can be quite large, even for simple systems: for a simple 25 base hairpin sequence (similar to Fig 5.5D), it takes 4,000,000 Markov steps to simulate 1s of real time. For an even larger system, such as a 4-way branch migration system (Fig 5.5C) with 108 total bases, simulating 1s of real time takes 14,000,000 Markov steps.

What can we do with all the data produced by the simulator? In the following sections we discuss several different processing methods.

6.1 Trajectory Mode

This full trajectory information can be useful to the user in several ways: finding kinetic traps in the system, visualizing a kinetic pathway, or as raw data to be passed to another analysis tool.

Trajectory mode is Multistrand's simplest output mode. The data produced by this mode is a trajectory through the secondary structure state space. While many trajectories could be produced for a given system, for most analysis purposes discussed in this section

we are only concerned with a single trajectory. Similarly, these trajectories are infinite but unfortunately our computers have only a finite amount of storage so we must cut the trajectory off at some point.

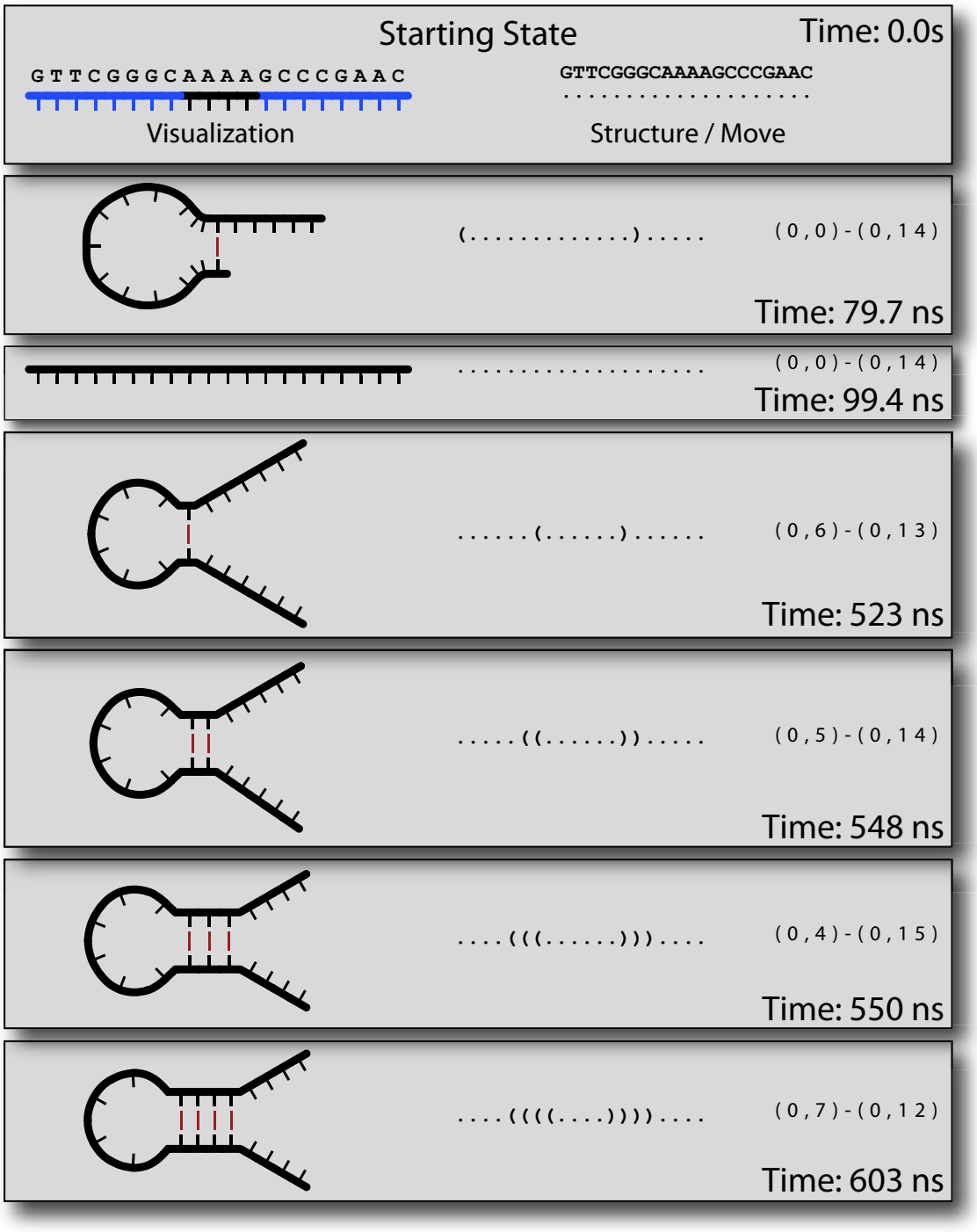


Figure 6.1: Trajectory Data

A trajectory is represented by a finite ordered list of (s, t) pairs, where s is a system

microstate, and t is the time in the simulation at which that state is reached. We call this time the *simulation time*, as opposed to the *wall clock time*, the real world time it has taken to simulate the trajectory up to that point. There are many different ways to represent a trajectory, as shown in Figure 6.1.

For practical reasons, we set up conditions to stop the simulation so that our trajectories are finite. There are two basic stop conditions that can be used, and the system stops when any condition is met:

1. Maximum simulation time. We set a maximum simulation time t' for a trajectory, and stop when the current simulation state (s, t) has $t > t'$. Note that the state (s, t) which caused the stopping condition to be met is not included in the trajectory, as it is different from the state at time t'.

2. Stop state. Given a system microstate s', we stop the trajectory when the current simulation state (s, t) has $s = s'$. This type of stopping condition can be specified multiple times, with a new system microstate s' each time; the simulation will stop when any of the provided microstates is reached.

We will now use an example to show how trajectory mode can be used to compare two different sequence designs for a particular system. The system is a straightforward three-way branch migration with three strands, with a six base toehold region and twenty base branch migration region, shown below (Fig 6.2).

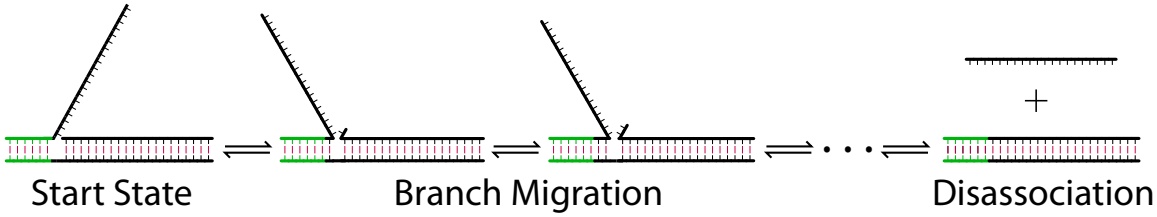


Figure 6.2: Three way branch migration system. The toehold region is in green, and the branch migration region is black. A few intermediate states along a sample trajectory are shown, with transition arrows indicating not a single base-pair step but a pair of steps that break one base-pair then form another. Many possible side reactions also exist, such as breathing of duplex regions and sequence depedent hairpin formation within the single-stranded region.

The simulation is started in the shown **Start State** using a toehold sequence of `GTGGGT` and a differing branch migration region for which we use the designs in Table 6.1. We then start trajectory mode for each design, with a stop condition of 0.05 s of simulation time, and save the resulting trajectories.

	Branch Migration Region
Design A	`ACCGCACGTCCACGGTGTCG`
Design B	`ACCGCACCACGTGGGTGTCG`

Table 6.1: Two different branch migration sequences

Rather than spam the interested reader with several thousand pages of trajectory printouts, since there are $5*10^6$ states in a 0.05 s trajectory for this system, we instead highlight one revealing section in each design's trajectory. Let us look at the state the trajectory is in after 0.01 s of simulation time, shown below in Figure 6.3 using a visual representation.

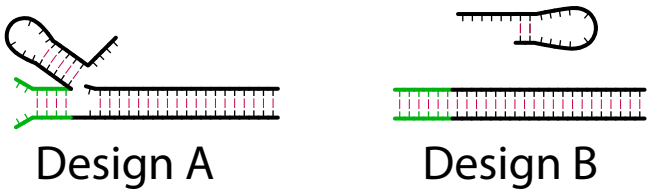


Figure 6.3: Structure after 0.01 s simulation time for two different sequence designs.

What happened? It appears that sequence design A has a structure that can form before the branch migration process initiates, that contains a hairpin in the single stranded branch migration region. Does this structure prevent the branch migration from completing? In the long run it shouldn't, as the equilibrium structure remains unchanged, but if we look at the final state in each trajectory (Figure 6.4), we see that design B has completed the process in 0.05 s of simulation time and indeed was complete at 0.01 s, where A is still stuck in that offending structure after the same amount of time. So for these specific trajectories, it's certainly slowing down the branch migration process.

Did this structure only appear because we were unlucky in the trajectory for design A? We could try running several more trajectories and seeing whether it appears in all or most of them, but a more complete answer is better handled using a different simulation mode, such as the first passage time mode discussed in Section 6.4.

A better type of question for trajectory mode is "How did this kinetic trap form?". In this example, we can examine the trajectory for design A and find the sequence of system

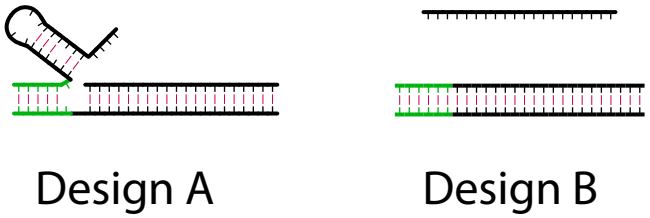


Figure 6.4: Final structure (0.05 s simulation time) for the two different sequence designs from Table 6.1. Branch migration regions: Design A: `ACCGCACGTCCACGGTGTCG`, Design B: `ACCGCACCACGTGGGTGTCG`.

microstates that lead to the first time the hairpin structure forms. This example has a straightforward answer: the competing structure forms before the branch migration starts, and is therefore in direct competition with the correct kinetic pathway.

We expect that the most common usage for trajectory mode is in providing the raw trajectory data for a separate tool to perform processing on. For example, taking the raw trajectory data and producing a movie of the structure's conformational changes can be very helpful in visualizing a system, and also is quite helpful for examining kinetic traps. A quick movie of the 3-way branch migration system could identify how the kinetic trap forms, rather than our examination of thousands of states by hand to locate that point.

6.1.1 Testing: Energy Model

We have also used the trajectory mode to aid in verifying that the kinetics model and energy model was implemented correctly. For the energy model, we can use an augmented output that includes the Multistrand-calculated energy for a given state, and compare that to the energy predicted by NUPACK [24] (or whichever tool / source we are using for our energy parameter dataset). This can be done using trajectory mode, with a cutoff time of 0 s, so the initial state is the only one in each trajectory. Multistrand's energy model was verified to be consistent with NUPACK for every sequence and structure in a comprehensive test set of secondary structures (part of the NUPACK package) that covers all possible loop configurations.

6.1.2 Testing: Kinetics Model

Testing the kinetics model can be done by testing that the *detailed balance* condition in fact holds: We know that at equilibrium, if our kinetics model obeys detailed balance,

the distribution of states seen by our simulator (after sufficient time to reach equilibrium) should agree with the Boltzmann distribution on each system microstate's energy. There are several ways we could extract this information from trajectory mode, such as recording all microstates seen in the trajectory (perhaps after some minimum time) and the amount of time spent in each one.

For our testing of the detailed balance condition we use a different method that is simpler to implement: we run many trajectories with a fixed maximum simulation time t and record only the final state in the trajectory (note that this is the state at time t in the trajectory, **not** the state which caused the stopping condition to be met). Assuming that the time t is large enough for us to reach equilibrium, we can compare the probability distribution over the final states seen by the simulation to that predicted using the NUPACK partition function and energy calculation utilities. In particular, for each final state observed in a trajectory we count the number of times it occured as a final state in our simulation, and use that to compute the simulation probability for that state. We then calculate the thermodynamic probability of observing that state using the NUPACK tools. Finally, we take the absolute value of the difference between the thermodynamic probability and the simulation probability for each final state observed and sum those quantities to obtain the total probability difference between our simulator and the thermodynamic predictions.

For our test cases we found this probability difference to be less than 1% when running a sufficient number of trajectories (approximately 10^5). This measure steadily decreases with increased trajectory count, and does not change when the simulation time is exponentially increased, indicating that our chosen t was enough to reach an equilibrium state and the probability difference is due to the stochastic nature of the simulation. The states which we observed accounted for 99.95% of the partition function, and that percentage also increases with increased number of trajectories.

6.2 Macrostates

In section 2.3 we defined a *system microstate*, which represents the configuration (primary and secondary structure) of the strands in the simulation volume. In this section, we will define a *macrostate* of the system and show how these objects can help us analyze a system by providing better stop states, as well as allowing new avenues of analysis, as discussed in

section 6.3. To make things simpler in this section, when we refer to a *microstate* we always mean a system microstate unless stated otherwise.

Formally, we define a *macrostate* m as a non-empty set of microstates: $m = \{s_1, s_2, ..., s_n\}$, where each s_i is a microstate of the system. Now we wish to derive the free energy of a macrostate, $\Delta G(m)$ in such a way that the probability of observing the macrostate m at equilibrium is consistent with probability of observing any of the contained microstates.

$$\begin{aligned} Pr(m) = & \quad Pr(s_1) + Pr(s_2) + \ldots + Pr(s_n) \\ = & \quad \sum_{1 \leq i \leq n} Pr(s_i) \\ = & \quad \sum_{1 \leq i \leq n} \frac{1}{Q} * e^{-\Delta G_{box}(s_i)/RT} \\ = & \quad \frac{1}{Q} * \sum_{1 \leq i \leq n} e^{-\Delta G_{box}(s_i)/RT} \end{aligned} \tag{6.1}$$

Now, letting $Q_m = \sum_{1 \leq i \leq n} e^{-\Delta G_{box}(s_i)/RT}$, the partition function of the macrostate m, we have $Pr(m) = \frac{Q_m}{Q}$. Similarly, in terms of the energy of the macrostate, we can express $Pr(m)$ as $\frac{1}{Q} * e^{-\Delta G(m)/RT}$, and plugging into (6.1) and solving for $\Delta G(m)$, we get:

$$\begin{aligned} \frac{1}{Q} * e^{-\Delta G(m)/RT} = & \quad \frac{1}{Q} * Q_m \\ e^{-\Delta G(m)/RT} = & \quad Q_m \\ -\Delta G(m)/RT = & \quad \log Q_m \\ \Delta G(m) = & \quad -RT * \log Q_m \end{aligned} \tag{6.2}$$

Now that we have the formal definition out of the way, let's look at an example macrostate using the same three-way branch migration system as in the previous section, figure 6.2.

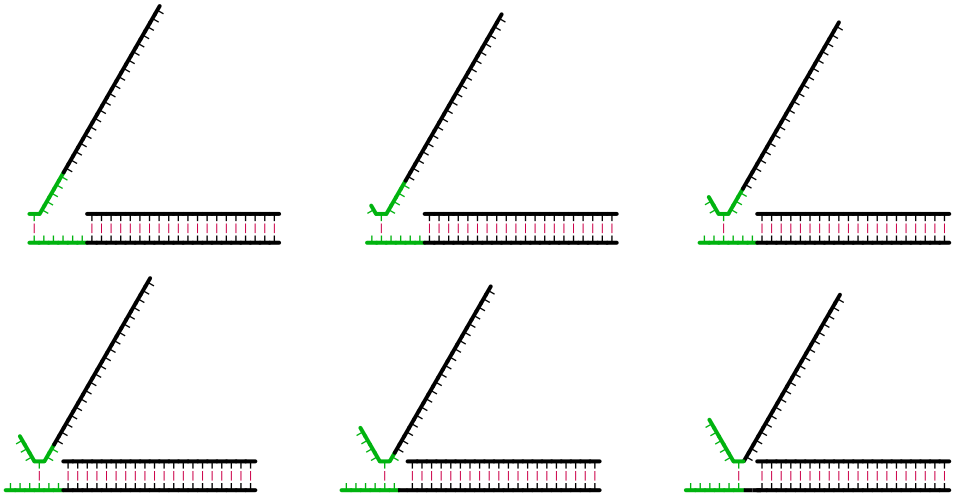
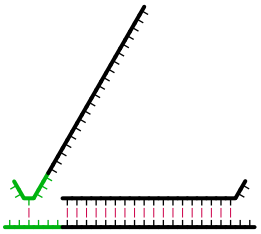


Figure 6.5: Example Macrostate

What does this macrostate represent? It's a set of microstates that has exactly one basepair formed in the toehold region, but it's not every such microstate – every microstate shown has the entire branch migration region fully formed. Thus the following microstate isn't included in the macrostate, but it does have exactly one basepair formed in the toehold region:



Why are these general macrostates interesting? Previously, we defined stop states as being microstates of the system, and we can use any number of them as part of the simulator's stop conditions. From that, it's easy to see that any given macrostate m could be used as a stop state of the system by simply expanding it out into the list of microstates contained within and using those as individual stop states.

Of particular interest to us are several classes of macrostates which can be described in very simple terms and also checked efficiently by the simulator *without* having to individually check for each microstate within those macrostates. The ability to check for a macrostate efficiently is very important: if we allowed the branch migration region in the previous example to have any structure, the macrostate would contain over 2^{22} microstates, and

even if we allowed only a limited number of bases in the branch migration region to be breathing (such as 3 base pairs, e.g. 6 bases) this is still 1140 microstates.

One useful tool in defining these classes of macrostates is a distance metric for comparing two complex microstates c_i, c_j. The distance $d(c_i, c_j)$ is defined as ∞ if c_i and c_j do not have the same set of strands and strand ordering, and otherwise as the number of bases which are paired differently between the two structures: e.g. if base x is paired with base y in c_i, but base x isn't paired with y in c_j, or if base x is unpaired in c_i, but base x is paired in c_j. This distance metric has been used in other work, using a slightly different but equivalent formulation for example [5, 12] and references therein. Some examples are shown below, in table 6.2.

c_i	Structure	Distance
c_0	....((((_))))....	
c_1	(((((((((_)))))))))	$d(c_0, c_1) = 8$
c_2	....((((_)).....))	$d(c_0, c_2) = 6$
c_3	.....((_)).....	$d(c_0, c_3) = 4$
c_4	....((((_)))).....)	$d(c_0, c_4) = 3$
c_5	.(....)(_).....	$d(c_0, c_5) = 7$

Table 6.2: Distance metric examples, for complex microstates on the two strand complex with sequences AGCTAGCT,AGCTAGCT. Bases that differ from the structure c_0 are shown in red.

Now that we have a distance metric, we define several common macrostates that can be used in the simulator as stopping conditions.

6.2.1 Common Macrostates

Disassoc: Given a set of strands ST and an ordering π^* on those strands, we define the **Disassoc** macrostate m as the set of all system microstates s which contain a complex microstate c with exactly the strands ST and ordering π^*. Recall that a complex microstate (Section 2.2) is defined by three quantities, the strands contained in the connected complex, the ordering on those strands, and the base pairs present; thus this definition implies no particular set of base pairs are present, though it does require that the complex be connected. Note that this macrostate can only be reached by either a association or disassociation step, allowing it to be efficiently checked as we only need to do so when encountering a bimolecular move. It's called **Disassoc** in

light of its most common usage, but it could also be used to stop after an association event.

Bound: Given a single strand S, we define the **Bound** macrostate m as the set of all system microstates s which contain a complex microstate c with set of strands ST that has $S \in ST$ and $|ST| > 1$.

Count: Given a complex microstate c and an integer count k, we define the **Count** macrostate m as the set of all system microstates s which contain a complex microstate c' for which $d(c, c') \leq k$. Note that c' which meet this criteria must have the same strands and strand ordering, as $d(c, c') = \infty$ if they do not. For convenience, instead of using the integer count k we allow passing a percentage p which represents a percentage of the total number of bases N in the complex c. If this is done, we use a cutoff $k = \lceil p * N \rceil$.

Loose: Given a complex microstate c, a integer count k and a set of bases B that is a subset of all the bases in c, we define the **Loose** macrostate m as the set of all system microstates s which contain a complex microstate c' for which $d_B(c, c') \leq k$, where we define d_B as the distance metric d over only the set of bases B in c. Similar to the **Count** macrostate, we allow a percentage p instead of k, for which we set $k = \lceil p * |B| \rceil$. This macrostate allows us to specify a specific region of interest in a microstate, such as just a toehold region we wish to be bound without caring about other areas in the complex microstate.

Note that each of these macrostates is based on the properties of a single complex microstate occuring within a system microstate; thus if desired we could make a stopping condition which uses several of these in conjunction. For example, we might make a stopping conditions that has **Disassoc** for strand A and **Disassoc** for strand B, thus creating a macrostate which can be described in words as "strand A is in a complex by itself, and strand B is in a complex by itself, and we don't care about any other parts of the system". Similarly we can implement disjunction simply by using multiple independent stopping conditions. Though the **NOT** operation is not currently implemented for these stop conditions, it may be added in the future, allowing us to have the full range of boolean operations on these common macrostates. As it is, we can easily implement the original example macrostate

simply by using an **OR** of the six exact system microstates. Or we could use **Loose** macrostates to implement the one we might have intended, where we didn't care very much about the branch migration region (and thus allowed it to have some breathing base pairs), only that a single base of the toehold had been formed.

6.3 Transition Mode

What is transition mode? The basic idea is that instead of **every** system microstate being an interesting piece of the trajectory, we provide (as part of the input) a list T of *transition states* of the system, the states which we think are interesting, and the output is then the times when we enter or leave any transition state in the list T. These transition states can be exact states of the system (e.g. system microstates), or macrostates of the system (e.g. a combinition of common macrostates such as **Dissasoc** or **Loose** macrostates), and we note that they are not required to be technical "transition states" as in chemical reaction theory – we are interested in how trajectories move (i.e. transition) from macrostate to macrostate, no matter how those macrostates are defined. One way to look at this form of output is as a trajectory across transition state membership vectors. We note that since these transition states are defined in exactly the same way as stop states, we generally lump them both together in the list of transition states that get reported (after all, you'd like to know what state caused the simulation to finish, right?), with a special labelling for which transition states are also stop states.

What is transition mode good for? The simplest answer is that it allows us to ask questions about specific kinetic pathways. Here's an example of this: Given a simple sequence that forms a hairpin, does it form starting from the bases closest to the 5'/3' ends (Fig 6.6B), or starting from the bases closest to the hairpin region (Fig 6.6C)?

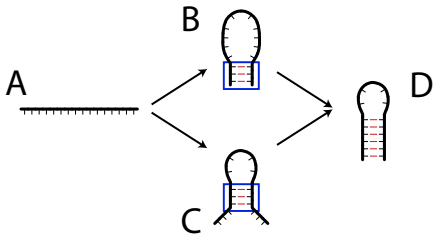


Figure 6.6: Hairpin Folding Pathways. Blue boxes indicate regions of interest used in loose structure definitions (Table 6.3). A) Starting State. B) Bases near the 5'/3' ends form first. C) Bases near the hairpin region form first. D) Final hairpin structure.

How do we represent these pathways in terms of transition states? Here we take advantage of the common macrostate definitions (Section 6.2.1) to define the intermediate structures B and C, using loose macrostates with a distance of 2, while A and D are defined with exact microstates.

Transition State Label	Sequence / Structure	State Type
	GCATGCAAAAGCATGC	
A (start)	.....	Exact
B	(((**********)))	Loose, $d \leq 2$
C	***(((****)))***	Loose, $d \leq 2$
D (stop)	((((((....))))))	Exact

Table 6.3: Transition states for hairpin pathway example. State type of **Exact** is exactly the given structure as a system microstate, and **Loose** is a loose macrostate covering only the bases in blue (or alternately, the bases not marked with "*").

Why is using a loose macrostate for these transition states useful? First, we note that we produce output any time the transition state membership changes, hence each step of the pathway is the set of all transition states which match the system microstate. Let's look at a possible pathway to the stop state where the first bases that form are near the 5' and 3' ends and the base pairs are added sequentially without ever being broken. With exact states this would result in the following transition pathway: $\{A\} \rightarrow \emptyset \rightarrow \{B\} \rightarrow \emptyset \rightarrow \{D\}$ and with the loose macrostates it would be this transition pathway: $\{A\} \rightarrow \emptyset \rightarrow \{B\} \rightarrow \{B, C\} \rightarrow \{B, C, D\}$. So far, so good. What about if we form two bases of B, then all of C, then the last base of B? For loose states, this is the exact same transition pathway - recall that we use a distance of 2, and two base pairs formed in B is exactly that distance away from the given structure. But for exact states, this is now the (very boring) pathway $\{A\} \rightarrow \emptyset \rightarrow \{D\}$, which doesn't answer our question about which part of the helix formed first!

Two possible transition pathways, using either the loose structures for B and C, or exact structures:

Time	Transition States
0.00	A
$3.63 * 10^{-7}$	$\emptyset$
$1.03 * 10^{-6}$	A
$1.40 * 10^{-6}$	$\emptyset$
$1.78 * 10^{-6}$	B
$1.92 * 10^{-6}$	B, C
$2.15 * 10^{-6}$	B, C, D

(a) Sample Transition Pathway (Loose)

Time	Transition States
0.00	A
$9.02 * 10^{-7}$	$\emptyset$
$1.31 * 10^{-6}$	A
$2.26 * 10^{-6}$	$\emptyset$
$2.72 * 10^{-6}$	D

(b) Sample Transition Pathway (Exact)

Table 6.4: Two different transition pathways via transition mode simulation, using either the given B and C states with the loose macrostate definitions from Table 6.3, or exact system microstates using the states from the same table with all "*" replaced by "." (unpaired) and distance set to 0, effectively. Note that the times listed are the times of first entering the given state.

Does this mean every simulated trajectory takes these transition pathways? Definitely not! The stochastic nature of the simulator means we're likely to see many different transition pathways if we run many trajectories. So, let's now answer the original question: which transition pathway is more likely? We do this by accumulating statistics over many kinetic trajectories as follows: For each transition path trajectory (such as those in Table 6.4) we break down the trajectory into pieces which have non-empty sets of transition states, separated only by zero or one empty set of transition states. So for example, the path shown in Table 6.4a breaks down into four separate reactions: $\{A\} \to \emptyset \to \{A\}$, $\{A\} \to \emptyset \to \{B\}$, $\{B\} \to \{B, C\}$, and $\{B, C\} \to \{B, C, D\}$. For our statistics, we'll group reactions of the form $x \to \emptyset \to y$ with those of the form $x \to y$, and for every possible reaction, we record the number of times it occurred and the average time it took to occur. So for the single pathway in Table 6.4a we get the following statistics:

Reaction	Average Time	Number of Occurences
$A \to A$	$1.03 * 10^{-6}$	1
$A \to B$	$7.43 * 10^{-7}$	1
$B \to B, C$	$1.47 * 10^{-7}$	1
$B, C \to B, C, D$	$2.29 * 10^{-7}$	1

Table 6.5: Statistics for the single transition pathway shown in Table 6.4a.

Now that we've seen an example of these statistics for a single kinetic trajectory, let's look at the same statistics over a hundred kinetic trajectories, again using the system with loose macrostates.

Reaction	Average Time	Number of Occurences
$A \to A$	$2.48 * 10^{-6}$	829
$A \to B$	$2.17 * 10^{-7}$	37
$A \to C$	$2.53 * 10^{-7}$	73
$B \to A$	$1.09 * 10^{-6}$	5
$B \to B$	$1.46 * 10^{-7}$	2
$B \to B, C$	$3.78 * 10^{-7}$	33
$C \to A$	$5.63 * 10^{-7}$	5
$C \to C$	$2.48 * 10^{-7}$	7
$C \to B, C$	$5.84 * 10^{-7}$	77
$B, C \to B$	$4.32 * 10^{-7}$	1
$B, C \to C$	$1.21 * 10^{-7}$	9
$B, C \to B, C, D$	$2.10 * 10^{-7}$	100

Table 6.6: Statistics for 100 simulated trajectories using the transition states from Table 6.3.

What can we conclude from these statistics? Both pathways do occur, but it is much more likely that the first bases formed are those closest to the hairpin region. The average times for each pathway are roughly within an order of magnitude of each other, and our selection of transition states was good: we didn't see any unexpected pathways, such as $\{A\} \to \{D\}$.

We could use these "reactions" as to create a coarse-grained representation of the original system as a chemical reaction network, using $\frac{1}{\text{avg time}}$ as the reaction rate constants. Whether this will be an accurate representation or not depends on the choice of transition states and the structure of the energy landscape. For example, if we were to try this using the average times for this system, we would end up with a formal CRN in which the $A \to A$ reaction is taken far less frequently than shown in Table 6.6. Finding appropriate coarse-grained representations is a deep and subtle topic [13].

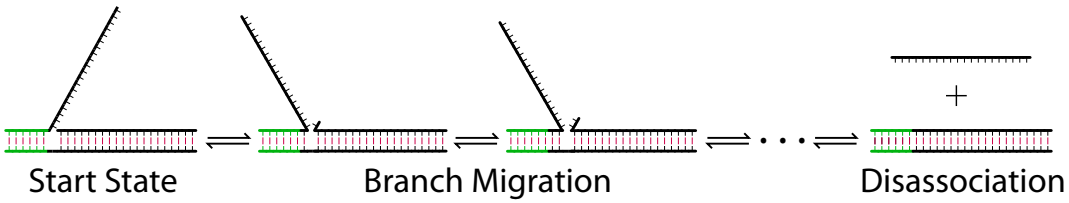
6.4 First Passage Time Mode

First passage time mode is the most basic simulation mode in Multistrand. It produces a single piece of data for each trajectory simulated: the first passage time for reaching any stop state in the system, and which stop state was reached. This is a rather striking difference from our previous simulation modes in the amount of data produced for each individual trajectory, but it is still quite powerful!

This first passage time data could be produced via trajectory mode: we can just discard

all the output until a stop state is reached. There is a distinct efficiency advantage to making it a separate simulation mode: we don't have to pay the overhead of reporting every piece of trajectory data only for it to be discarded. Similarly, we could generate the same data using transition mode by only using stop states in our list of transition states. We implement this as a distinct simulation mode in order to better separate the reasons for using each simulation mode: for transition mode, we are interested in the pathway our system takes to reach a stop state, and for first passage time mode we are interested in how quickly the system reaches the stop state(s).

What does first passage time data look like? Let's revisit our example system from section 6.1 (Figure 6.2):



We start the system as shown, and use two different stop states: the **complete** stop condition where the incumbent strand has disassociated (as shown in the figure), and the **failed** stop condition where the invading strand has disassociated without completing the branch migration. Both of these are done using *Disassoc* macrostates, which makes it very efficient to check the stop states. Note that we include the invading strand disassociating as a stop state so that if it occurs (which should be very rarely), we can find out easily without waiting until the maximum simulation time or until the strands reassociate and complete the branch migration.

The following table (Table 6.7) shows a five trajectories worth of data from first passage time mode on the example system, using sequence design B (Table 6.1) for the branch migration region.

Note that we have included a third piece of data for each trajectory, which is the pseudorandom number generator seed used to simulate that trajectory. This allows us to produce the exact same trajectory again using a different simulation mode, stop states or other output conditions. For example, we might wish to run the fifth trajectory in the table again using trajectory mode, to see why it took longer than the others, or run the first

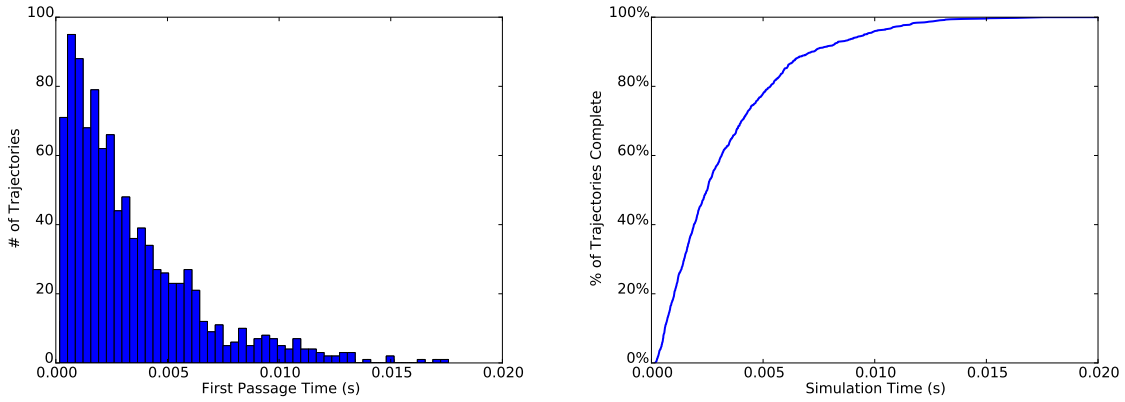
Random Number Seed	Completion Time	Stop Condition
`0x790e400d`	$3.7 * 10^{-3}$	failed
`0x38188213`	$3.8 * 10^{-3}$	complete
`0x47607ebf`	$2.1 * 10^{-3}$	complete
`0x02efe7fa`	$2.8 * 10^{-3}$	complete
`0x7c590233`	$6.7 * 10^{-3}$	complete

Table 6.7: First passage time data for the example three way branch migration system. Stop conditions are either "complete", indicating the branch migration completed successfully, or "failed", indicating the strands fell apart before the branch migration could complete.

trajectory to see what kinetic pathway it took to reach the **failed** stop condition.

Let's now look at a much larger data set for first passage time mode. Here we again use the three way branch migration system with sequence design B for the branch migration region and increase the toehold region to be ten bases, to minimize the number of trajectories that reach the **failed** stop condition. We run 1000 trajectories, using a maximum simulation time of 1 s, though no trajectory actually used that much as we shall shortly see.

Instead of listing all the trajectories in a table, we graph the first passage time data for the **complete** stop condition in two different ways: first (Figure 6.7a) we make a histogram of the distribution of first passage times for the data set, and second (Figure 6.7b) we graph the percentage of trajectories in our sample that have reached the **complete** stop condition as a function of the simulation time.



(a) Histogram of first passage times

(b) Percent completion by simulation time

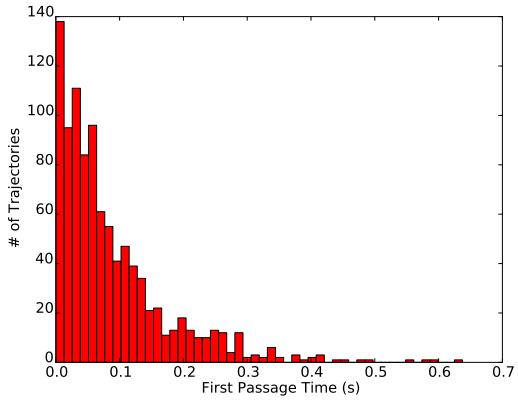
Figure 6.7: First passage time data for the three way branch migration system, using sequence design B (Table 6.1) and with a ten base toehold sequence. 1000 trajectories were simulated and all of them ended with the **complete** stop condition.

While there are many ways to analyze these figures, we note two particular observations. Firstly, the histogram of the first passage time distribution looks suspiciously like an exponential distribution, possibly with a short delay. This is not always typical (as we shall shortly see), but the shape of this histogram can be very helpful in inferring how we might wish to model our system based on the simulation data; e.g. for this system, we might decide that the three way branch migration process is roughly exponential (with some fitted rate parameter) and so we could model it as a one step unimolecular process.

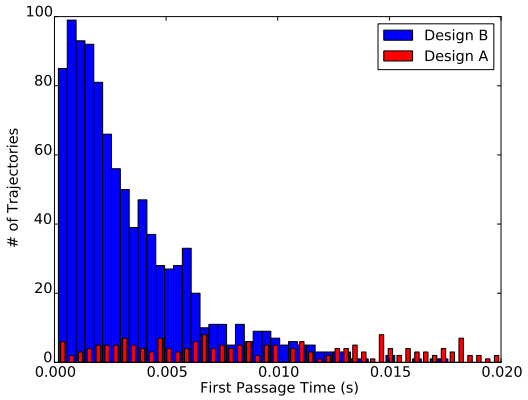
The second observation is that while the percentage completion graph looks very similar to an experimental fluourescence microscopy curve, they should **NOT** be assumed to be directly comparable. The main pitfall to watch out for is when comparing fluourescence curves from systems where the reactions are bimolecular: in these the concentration of the relevant molecules are changing over time, but in our stochastic simulation the bimolecular steps are at a fixed volume/concentration (reflected in the ΔG_{volume} energy term) and data is aggregated over many trajectories.

6.4.1 Comparing Sequence Designs

A common usage of first step mode is in the comparison of sequence designs, as we previously brought up in Section 6.1. We now run another 1000 trajectories on the same three way branch migration system as in the previous section, including the increased toehold length, but using the sequence design A (Table 6.1) for the branch migration region. Note the change in x-axis scale; this design is indeed much slower than design B!



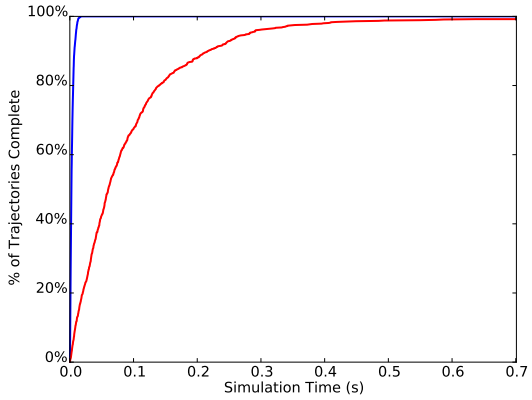
(a) Histogram of first passage times, design A



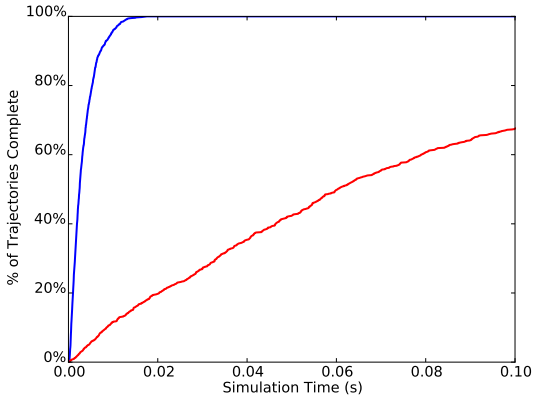
(b) Histogram of first passage times, both designs

Figure 6.8: First passage time data for the three way branch migration system, comparing sequence designs using histograms. For figure (b), we compare the two designs on the range of times from 0 s to 0.02 s. The buckets for sequence design A have been reduced in visual size to show overlapping regions, but overall bucket sizes are consistent between the two designs (though they are slightly different from those in Figure 6.7a).

Let's also look at the same data but using the percentage completion as a function of simulated time graphs:



(a) Design comparison, percent completion graph



(b) Design comparison, zoomed

Figure 6.9: First passage time data for the three way branch migration system, comparing sequence designs using percent completion graphs. Figure (b) is a zoom in on the range of times from 0 s to 0.1 s from Figure (a).

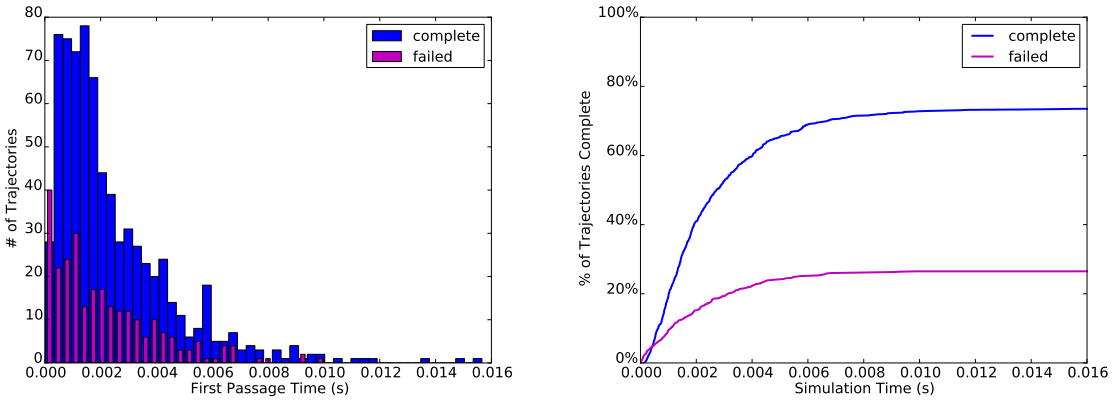
We can now clearly see just how different these two sequence designs are! Our results from the trajectory mode section were clearly not unusual: the majority of sequence design B's trajectories finish in under 0.01 s, whereas the same amount for sequence design A

is over ten times longer. This is highlighted in the combined graph (Figure 6.8b), which shows how vastly different the timescales are for each process. Looking at the percentage completion graphs, we note that sequence design A did not actually reach 100% – it actually had 8 trajectories reach the **failed** stop condition!

While both types of graphs are presenting the same information, they are frequently useful in different cases: the first passage time histograms are helpful for gaining an intuition into the actual distribution of times for the process we are simulating, while the percent completion graphs are better for looking at the relative rates of different designs, especially when working with more than two sequence designs.

6.4.2 Systems with Multiple Stop Conditions

What about our original three way branch migration system, which had toeholds of length six? Let's now look at this situation, where we have a system that has more than one competing stop state. We again run the system with 1000 trajectories and a 1 s maximum simulation time (though it's never reached). Instead of plotting only the first passage times for the **complete** stop condition, we overlay those with the first passage times for the **failed** stop condition.



(a) Histogram of first passage times

(b) Percent completion by simulation time

Figure 6.10: First passage time data for the three way branch migration system with 6 base toeholds, comparing sequence designs.

Competing stop conditions certainly make the data more interesting! We can pick out pieces of the kinetic pathways a lot easier using these graphs. For example, the competing pathway leading to the **failed** stop condition frequently occurs faster than the **complete**

stop condition pathway; this should be unsurprising, as one pathway involves a long random walk while the other is very unlikely to include one.

6.5 Fitting Chemical Reaction Equations

We are frequently interested in DNA systems which can be represented (with some choice of the appropriate internal parameters/sequences) by chemical reaction equations of the following form:

$$A + B \xrightarrow{k_{eff}} C + D \tag{6.3}$$

These systems usually involve an intermediate step, so typically the concentration is low enough for the above equation to actually be a good fit. Experimental observation of these systems tend to be in the range of concentrations where the above equation is an accurate characterization of the system.

Let us now associate the species in the equation above with actual DNA complexes (ordered collections of connected strands). We designate strands by unique letters, and indicate complementary strands with an asterisk. For the toehold-mediated 3-way branch migration example, the chemical equation then becomes:

$$x^*y + x \xrightarrow{k_{eff}} xx^* + y \tag{6.4}$$

Let us examine one possible DNA configuration for the state of the entire system that could follow this equation's implied dynamics. The left hand side could be the configuration given in figure 6.11, and the right hand side could be the configuration given in figure 6.12. Note that while we show particular exact secondary structures for each of these figures, there are actually many such secondary structures that represent the appropriate parts of the equation, which might be specified using **Loose** macrostates.

Finally, what would the equations look like if we did not expect them to fit equation 6.3? One possibility is as follows (as in Zhang, Winfree 2009 [25]):

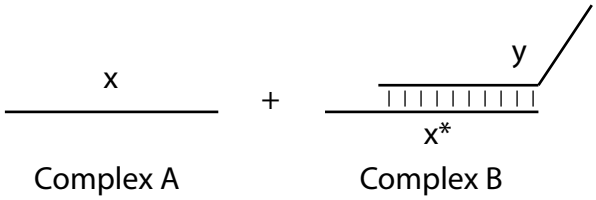


Figure 6.11: Starting Complexes and Strand labels

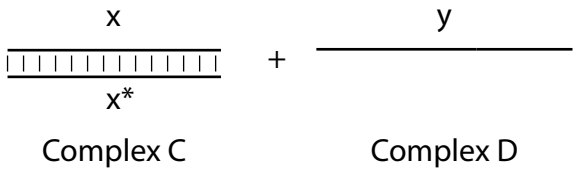


Figure 6.12: Final Complexes and Strand labels

$$A + B \underset{k_r}{\overset{k_f}{\rightleftharpoons}} AB \underset{k_{bm}}{\overset{k_{bm}}{\rightleftharpoons}} AB' \xrightarrow{k_r} C + D \tag{6.5}$$

Though this model is used in several experimental papers, it is difficult for us to define simulation macrostates in order to determine the various rates present in the equation. Instead, we look at a model where it is easy to separate out the steps into discrete components which can be individually simulated. Specifically, we look at a system where the molecules A and B can collide and form either a reactive molecule which will go to C and D, or a nonreactive molecule which will fall apart after some time[1]. We call this the **first step model**, and it is described by the equations below:

$$A + B \xrightarrow{k_1} AB \xrightarrow{k_2} C + D \tag{6.6}$$

$$A + B \underset{k_2'}{\overset{k_1'}{\rightleftharpoons}} AB' \tag{6.7}$$

We will use this model extensively to analyze the results of the **first step mode** simu-

[1]Thanks to Niles Pierce and Victor Beck for suggesting this approach.

lations discussed in section 6.6. Note that for low concentrations, the controlling step will be the bimolecular reactions and thus we should be able to also fit to a k_{eff} type model in those cases.

We discuss fitting first passage time data to the simple k_{eff} model (Equation 6.3, as well as using **first step mode** to generate data which can easily be fit to the first step model.

6.5.1 Fitting Full Simulation Data to the k_{eff} model

For this simulation mode, we start the simulation in the exact state shown in figure 6.11, and measure the first passage time to reach any state with the same complexes (but not exact secondary structure, i.e. we use a **Disassoc** stop state) as that shown in figure 6.12. This gives us a data set of first passage times Δt_i, where $1 \leq i \leq n$, and n is the total number of trajectories simulated. The simulation is done at a particular concentration z. Note that this simulation will require time inversely proportional to the simulation concentration, since we are simulating elementary steps. Thus we would prefer to simulate at higher concentrations and step downwards in concentration until we are in the region where the bimolecular reaction dominates, and equation 6.3 holds.

If we assume equation 6.3 holds and determines our distribution of first reaction times, we can fit our data to an exponential distribution in order to determine k_{eff}: Recall that (via Gillespie [8]) in a formal chemical reaction network, a state with total outgoing propensity a_0 will have a passage time (τ) distribution according to the probability density function $P(\tau) = a_0 * \exp(-a_0\tau)$. Since the propensity a_0 for a bimolecular reaction is the reaction rate times the concentration, we solve for k_{eff} as follows, where expfit is the function that takes a data set of first passage times (Δt_i) and returns the parameter a_0 of the exponential distribution given by those Δt_i, and z is the simulated concentration:

$$k_{eff} = \text{expfit}(\Delta t_i) * 1/z \tag{6.8}$$

If we are in the regime where equation 6.3 holds, we expect this to give us a consistent value for k_{eff}. If we are outside of that regime (and thus the unimolecular reactions from

equation 6.5 or equation 6.6 dominates), this will err by exactly the factor of $1/z$, thus the graph of k_{eff} versus concentration z should appear linear in this regime.

6.6 First Step Mode

This simulation mode makes a simple assumption: we always start the Markov simulation by making a "join" step, that is, one where a pair of molecules A and B come together and form a single base pair. The choice of secondary structure states for the molecules A and B before collision can be done either by using particular exact complex microstates, or by Boltzmann sampling the secondary structure space of the molecules. This sampling is valid when the bimolecular reaction rates are slow enough that the initial complexes reach equilibrium. In either case, we have a valid system microstate once we know the structure for the A and B molecules, and then choose a "join" step from those present in the system microstate and use the resulting system microstate (after making the join) as our start state for a trajectory. Since this mode runs many trajectories, we must make many such random choices for the join step and for the Boltzmann sampling of the initial molecules (if we are using the sampling rather than exact states).

The simulation then starts from this configuration, and we track two distinct end states: the molecules falling apart back into one of the $A + B$ configurations, or the molecules reacting into one of the $C + D$ configurations. Our data then consists of first passage times where we can separate each trajectory into one that reacted or one that failed. The advantage to this mode of simulation is that we no longer are directly simulating the "join" bimolecular steps, whose rates are proportional to the simulated concentration and thus are going to be very slow relative to the normal unimolecular steps. This allows us to use the simulator to concentrate on the trajectories where we have a collision, rather than spending (at low concentrations) most of the time simulating unimolecular reactions while waiting for the very rare bimolecular reaction.

6.6.1 Fitting the First Step Model

Our first step mode simulation produces the following pieces of data: Δt^i_{react}, the first passage times for reactive trajectories, Δt^i_{fail} the first passage times for trajectories did not react, the number of trajectories that reacted N_{react} and failed N_{fail}, and the rate constant

k_{coll}, the simulation's estimate of the rate of collision (in $/M/s$) of the A and B molecules. This k_{coll} is calculated based on *all* the join moves possible from the initial configuration of the A and B molecules, and thus is very likely to include many such moves which do not lead to very stable structures and thus disassociate quickly.

We then fit to the model given in equations 6.6 and 6.7, as follows:

$$k_1 = \frac{N_{react}}{N_{react} + N_{fail}} * k_{coll} \tag{6.9}$$

$$k_1' = \frac{N_{fail}}{N_{react} + N_{fail}} * k_{coll} \tag{6.10}$$

$$k_2 = \text{expfit}(\Delta t^i_{react}) \tag{6.11}$$

$$k_2' = \text{expfit}(\Delta t^i_{fail}) \tag{6.12}$$

Thus we can directly find each of the model parameters from the collected simulation data, in a natural way. We then use this model to predict the k_{eff} parameter we would observe if we assume that the simple chemical reaction model (eqn 6.3) is valid.

6.6.2 Analysis of First Step Model Parameters

We first show a natural (but inexact) way to calculate k_{eff} from the first step model parameters, using the assumption that the time used in "failed" collisions is negligible compared to that needed for a successful reaction. In this situation, we can estimate k_{eff} for a particular concentration z by calculating the expected average time for a reaction. We use the fact that the expected value of an exponential distribution with rate parameter λ will be $\frac{1}{\lambda}$ in order to derive k_{eff}:

$$k_{eff} = \frac{1}{z} * \frac{1}{\frac{1}{k_1 * z} + \frac{1}{k_2}} \tag{6.13}$$

Again, this makes the assumption that equation 6.3 holds and thus that the reaction dominated by the bimolecular step. The observation from the full simulation mode still holds: if we are not in this regime, we will err by a factor of $\frac{1}{z}$ and thus the graph should be linear. Note that since this simulation does not require a set concentration, we can run

one simulation (with a large number of trajectories) and use the extracted data to produce the same type of graph as the full simulation mode.

We now would like to remove the assumption that the "failed" collision time is negligible: though that assumption makes k_{eff} straightforward to calculate, many systems of interest will not satisfy that condition.

We now need to calculate the expected time for a "successful" reaction to occur based on both the "failed" and "reactive" collision parameters. We do this by summing over all possible paths through the reactions in equations 6.6 and 6.7, weighted by the probability of those reactions. Let $\Delta t_{coll} = \frac{1}{(k_1+k_1')*z}$ (the expected time for any collision to occur), $\Delta t_{fail} = \Delta t_{coll} + \frac{1}{k_2'}$ (the expected time needed for a failed collision to return to the initial state), $\Delta t_{react} = \Delta t_{coll} + \frac{1}{k_2}$ (the expected time for a reactive collision to reach the final state), $p(path)$ is the probability of a particular path occuring, and $\Delta t_{path} = n\Delta t_{fail} + \Delta t_{react}$ is the expected time for a path which has n failed collisions and then a successful collision. Finally, the quantity which we want to solve for is $\Delta t_{correct}$, the expected time it takes for a successful reaction to occur.

$$\Delta t_{correct} = \sum_{path} \Delta t_{path} * p(path) \tag{6.14}$$

$$= \sum_{n=0}^{\infty} (n\Delta t_{fail} + \Delta t_{react}) * (\frac{k_1'}{k_1 + k_1'})^n * \frac{k_1}{k_1 + k_1'} \tag{6.15}$$

To simplify the next step, let $\alpha = \frac{k_1}{k_1+k_1'}$ and $\alpha' = \frac{k_1'}{k_1+k_1'}$, and recall that for $\beta > 0$, $\sum_{n=0}^{\infty} \beta^n = \frac{1}{1-\beta}$ and $\sum_{n=0}^{\infty} n * \beta^n = \beta * \frac{1}{(1-\beta)^2}$, and we get:

$$\Delta t_{correct}) = \Delta t_{fail} * \frac{\alpha'}{(1-\alpha')^2} * \alpha + \Delta t_{react} * \frac{1}{1-\alpha'} * \alpha \tag{6.16}$$

Now we note that $\frac{\alpha}{1-\alpha'} = 1$, and $\frac{\alpha'}{1-\alpha'} = \frac{k_1'}{k_1}$, and simplify:

$$\Delta t_{correct}) = \Delta t_{fail} * \frac{k_1'}{k_1} + \Delta t_{react} \tag{6.17}$$

And thus we arrive at the (full) form for k_{eff}:

$$k_{eff} = \frac{1}{\Delta t_{correct}} * \frac{1}{z} \tag{6.18}$$

This requires only a single assumption: that the reaction is dominated by the bimolecular step, and thus can be described by equation 6.3. We note that this derivation can be generalized for multiple possible stop states [2].

Appendix A

Data Structures

A.1 Overview

The simulator is composed of many objects, which have very strong dependencies and are one of the key components in allowing us to use efficient algorithms for recomputing the set of adjacent moves. Our data structures contain a lot more information than those in previous works, but this is what allows us to use the more efficient algorithms for move selection, update and energy computation.

The following is a list of the primary objects which we will elaborate on in this section:

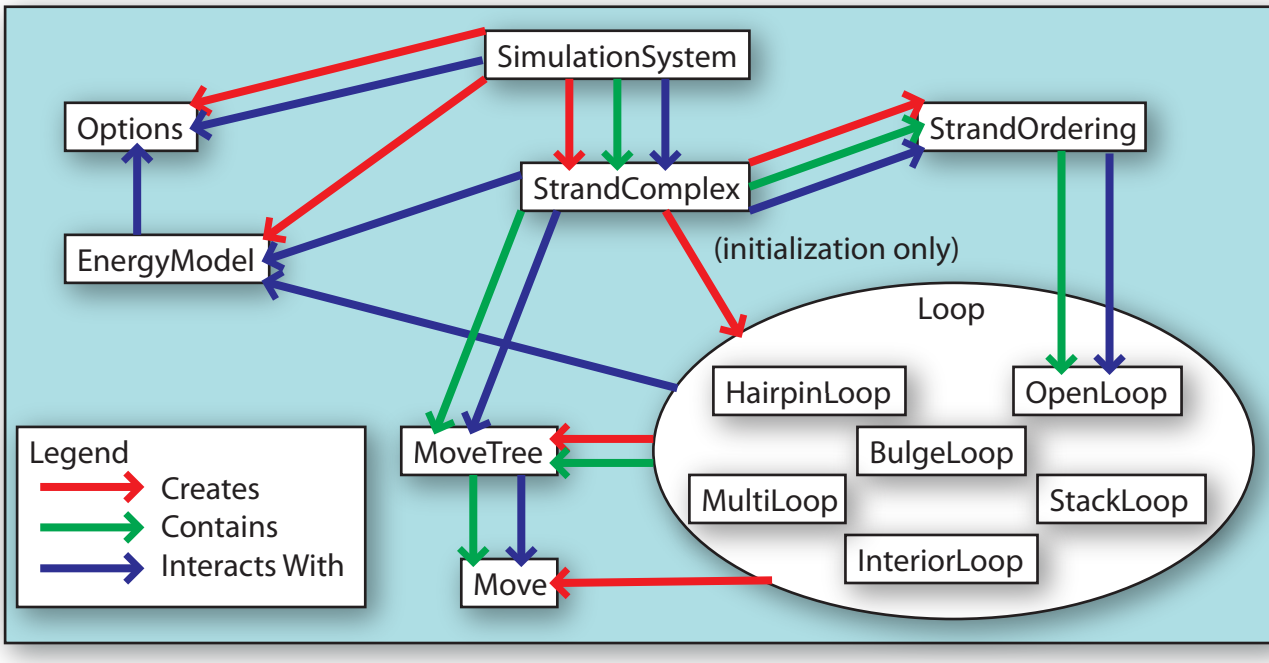


Figure A.1: Relationships between data structure components

Object Name	Basic Function
SimulationSystem	Controls the simulation, handles trajectory parameters and simulation setup.
StrandComplex	Summarizes the information for a particular complex, including the ordering, intra complex moves, and loop structure.
SComplexList	Stores all the complexes in the system and handles inter-complex moves.
Loop	Base class for all types of local loop objects. Contains information about the energy, adjacent loops, and has all the generic loop accessors which are implemented by derived classes.
Move	Contains information about a single move, including the affected loops, type of move and rate at which it should occur.
MoveTree	Container type for organizing the Move and MoveTree nodes for a particular unit: a loop, a complex or the system.
EnergyModel	Base class for all types of energy models. Defines all the accessors for finding the energy of particular loop structures. Derived classes implement these, and contain all the necessary parameter data.
StrandOrdering	Auxiliary class used by a Complex to handle strand ordering information and functions for checking stop states, handling output of the complex and other related tasks.
Options	Auxiliary class which contains all the simulation options. This is the object which contains all the model and simulation parameters necessary for the simulator, as well as the input DNA system. It is also used to handle the results coming from the simulator (e.g. the output). This is the only Python object included in this section.

A.2 SimulationSystem

Purpose: This object handles the main control loop of the program in addition to performing the setup and initialization duties.

Functionality:

1. Initializes the EnergyModel object based on parameters from the Options object.
2. Initializes the system state for each trajectory.
3. Handles random number generation for each step of the markov process.
4. Performs the main loop of the simulator, performing a move, updating the time and handling output duties.

Data Members:

1. EnergyModel: contains a reference to the energy model being used for the system. This data is also used to initialize a static member of the Loop class, so that all loops have a direct reference to the correct energy model, but we need to store it here so that it can be properly freed later.

2. SComplexList: the current secondary structure state of the system is stored in this object, which needs to be reinitialized to the starting state for each trajectory.

Discussion: The system object, while containing the main control loop for the program, is very limited in the scope of what it does, handling only the main initialization duties and the random number generation component of each step of the markov process. This lets us focus on handling all of the different simulation modes (and thus output options) at this point without having to know about any of the secondary structure information in the system.

This is one of two C++ objects that are closely connected to the Python side of the simulator, as we need to both access the parameters and inputs for a particular simulation system as well as provide output back across that interface.

A.3 StrandComplex

Purpose: Represents a single connected complex of strands. Things that take place at that level are handled here if they don't involve the specific ordering of the strands. This includes the initial generation of the loop objects representing the complex as well as handling joining moves between two complexes.

Functionality:

1. Contains code that generates the entire Loop-based structure from the sequence and dot-paren structure input. Only used when the trajectory is initialized.

2. Performing base pair creation moves between two complexes is handled here, joining the appropriate data together.

3. Performing base pair deletion moves which cause a complex to break into two connected structures is done here.

Data Members:

1. StrandOrdering: contains information about the strands/sequences used in the complex, the cyclic permutation corresponding to the current structure and links to the actual structure.

2. MoveTree: tree structure collecting all the moves corresponding to loops within the complex.

Discussion: Originally this object also contained all the information about the current sequence and dot paren representation of the structure, but it made more sense to split those into a seperate object when it became apparent that all the functions which dealt with those used a very similar linked list approach. Because of this, there are really only three main functions here which don't involve the ordering in some way: joining a complex, breaking a complex, and initializing a complex.

The initialization of loop structure for a particular complex is easily the most complicated algorithm here, and is discussed in Section B.2. Why not handle the loop creation in another location? The simple answer is that at some point, there needs to be a function which can translate from the sequence and structure for a complex into the loop structure, and since those are only relevent at the complex level, it can't be any lower. Loop structure only knows very local bits of the sequence and has no concept of the dot-paren structure, and while the strand ordering does involve both the sequence and structure, the main functions contained there deal with modifications to that ordering and subsequent changes to the output representation, and not really anything to do with the actual structures underneath.

A.4 SComplexList

Purpose: Contains all the complexes in the current state of the system. Handles computation of join move rates between complexes, and controls move choice selection between join moves and moves internal to each complex.

Functionality:

1. Computes the total join rate between complexes, which requires information about the external bases (unpaired bases appearing in an open loop) present in each complex.

2. Handles control flow for picking which complex the next move takes place within, or computing which join move took place.

3. Handles high level code for checking a stop state, specifically when a stop state involves multiple complexes being present, we need to check for the simultaneous presence of all of them, without overlapping.

Data Members:

1. StrandComplex: linked list of complexes, along with the current energy and total rate for each complex, so that they do not have to be recomputed when a move doesn't involve that complex.

2. Join Rate: total flux for all join moves between complexes. If a move is performed within a complex, this total only needs to be recalculated when that move involves an OpenLoop.

Discussion: Exists mainly as a way of seperating the simulation system from the actual strand complexes. This layer handles summarizing the rates from each complex, as well as everything about the join moves. The join moves are calculated very simply by figuring out the total number of combinatorial choices for pairing bases. While this is done under the assumption that a single base pair is sufficient to join two complexes, it could easily be extended to require two adjacent bases to pair, by changing the definitions of exterior bases and the resulting combinatorial choice.

Note our bimolecular rate method's effects are focussed mainly here: the join move rate only involves the numble of possible pairs plus the energy model's base join move rate. If we changed the rate method this may have to be handled at a much lower level so that we can include energy computations on the resulting open loops. Currently we don't have to do that, as the energy contributions due to those loops are in the reverse rate (when breaking a complex apart), as at that point we will have detailed information about the secondary structure. If we tried to use those energies in computing the join move rates, we would have to have *much* more information about the secondary structure of each complex at this level, which would be a lot more computationally intensive. However, for some types of energies, it's conceivable that a similar redefinition of the exterior bases would work, like what we mentioned above for forming two bases at a time.

The other key thing handled here is checking for the existence of a stop state. A stop state can involve the secondary structure of multiple complexes, so we have to start handling it at this level, and it's important that we try to optimize the computation as much as possible, as the simulator goes through a huge number of states, and only a small portion of those are going to be stop states. At this level several shortcuts are implemented, such as only checking for a stop state when the number of complexes currently present would allow it, and halting the checking for a particular stop state when there aren't enough unchecked complexes in the current state to match the remaining ones in the stop state.

A.5 Loop

Purpose: Base class for one of the two (ad hoc) polymorphic types used in the simulator. Secondary structure can be broken down into a tree consisting of connected loops. The particular type of loop does not matter in this structure, but does matter for calculating the energy and possible moves available to that loop. Thus this class defines all the methods which can be used to interact with generic loops, and the specific implementations handle the details.

This class, along with the six derived classes (StackLoop, HairpinLoop, BulgeLoop, InteriorLoop, MultiLoop and OpenLoop) composes the largest percentage of the code in the program, currently around 40%. This is due to the need for each derived class needs to handle move generation and energy calculation in a specific way, as well as delete moves which need to be able to handle each of the possible combinations of two derived types.

Base Class Functionality:

1. Compute delete move rates for arbitrary pairs of derived loop types.
2. Perform delete moves for arbitrary pairs of derived loop types.
3. Perform deletion moves which cause a complex to split (OpenLoop-OpenLoop delete moves).

Derived Class Functionality:

1. Energy Computation: passes appropriate data members to the energy model for calculation.

2. Move Generation: computes rates for all internal base pair creation moves, adds generated moves to move tree data structure for that loop.

3. Move Evaluation: builds the resulting loop structure after performing a base pair creation move.

4. (OpenLoop only) Exterior bases: computes the counts of each base type accessible in that open loop for a complex join.

5. (OpenLoop only) Complex joins: create the resulting pair of connected open loops from the pair of open loops involved in a complex join move.

Data Members (general for most derived classes):

1. Loop: list of adjacent loops, in a fixed 5' to 3' ordering such that base pairs can be associated with the correct adjacent loop.

2. sequence: base sequence for each strand participating in the loop. Usually just two, but can be any number for Multi and Open loop types.

3. lengths: lengths of each base sequence present in the loop.

4. energy: stored value from calling the energy model functions, to avoid recomputation.

5. MoveContainer: container for all creation and deletion moves involving this loop. Used to make deleting this set of moves from a larger structure a simple task.

Discussion: Algorithms: The per-step move generation algorithm (Section B.4) and the move update algorithm (Section B.5) are implemented in a distributed manner in the derived Loop objects. The advantage for these algorithms are in the local properties. We only need to know for creation moves the information about the particular loop involved, and for deletion moves the two loops adjacent to the base pair being deleted. This means that creation moves can be handled within each loop type cleanly, using only the internal data to that loop to generate the two new loops which are to replace it. Deletion moves are handled in a similar way by creating a friend function of the base Loop class which handles

each case of a pair of loop types.

Output Reconstruction: Though each loop object only stores a very small portion of the total sequence, it is straightforward (and indeed, must be possible) to use only those pieces to reconstruct the actual strand sequences and dot paren structures. In practice, this reconstruction, especially that of the dot paren structure for use in output, would need to be done each time we wanted to output a state, which for full trajectory output would be every step. Thus there are functions to perform this reconstruction within the class, but instead of using those at each step, the information about which base(s) were affected in a move are passed back by the relevant move update function so that the StrandOrdering object can keep the output structure up to date for use both in trajectory mode and for stop state checking.

Deletion Move Duplication: Note that a particular deletion move occurs twice, once in each loop adjacent to the base pair involved. The resulting rates are then halved, such that the total rate is correct, and the deletion move is performed identically no matter which loop's copy of the move is chosen. This means that each loop needs to be able to reset its deletion moves independently of the creation moves within that loop, and handling this cleanly is one difficulty with how we build the overall set of moves for the complex. The alternative, however, would be to store a deletion move relative to only one of the two loops adjacent, which would make the recomputation of deletion moves caused by performing creation moves to be excessively complicated. Since the total number of deletion moves per loop is small (usually two), calculating and storing the deletion moves twice is a small effect on the complexity of move generation, and is much easier to implement and more efficient for move update.

Complex Join/Break: The complex join and break moves are handled at the lowest level within this object. Complex breaks are implemented in the base class, as they need to rebuild the two open loops adjacent to the breaking base pair into disconnected open loops. Joins are handled within the OpenLoop object, forming the resulting connected open loops. While it would be convenient to handle these all within a particular object, they affect the data structure at many different levels and so there is no convenient place to handle all the

appropriate changes to the strand ordering, the underlying open loops, the new resulting complex(es) and the moves available to each.

A.6 Move

Purpose: Represents a single creation or deletion move. Contains the necessary information for the appropriate Loop to perform the move.

Functionality:

1. Directs control to the correct loop to handle the actual creation/deletion necessary to perform the move.

Data Members:

1. affected loops: links to the loops affected by this move (i.e. the ones that created it).
2. move rate: rate at which the move should occur, for use by the move choice algorithm.
3. type information: used by the affected loops to perform this move

Discussion: This is the simplest data structure that's important enough to actually mention. It's really just used as data storage by the loop's move generation algorithms, so that the move choice algorithms can traverse the moves directly without having to go through the local loop structure, and once the move is chosen transfer control into the affected loop to perform the choice.

This is the basic unit that's being allocated and deallocated in large quantities at each step, and so one possibility for improving speed is to keep a pool of pre-allocated Move objects that are passed out rather than constructed from memory, and returned to the pool instead of deallocated. The allocation/deallocation hasn't appeared to be a major factor at this point though, so it's likely that other methods are a better choice for improving the simulation speed.

A.7 MoveTree

Purpose: Holds all the Node objects for a particular logical unit, such as a Loop, StrandComplex, or SimulationSystem. Implements the addition of nodes and deletion of nodes in

an efficient way. Can itself be used as a base type to be organized into higher level move trees.

Note that this object is no longer fully implemented and has been deprecated in favor of moving to C++ Standard Template Library containers for handling moves. Specifically, we currently use an array data structure as our move container for each logical unit, with a move selection algorithm that is linear in the number of moves; as discussed in section B.7, move selection is not typically the rate limiting step in simulations, so the non optimal data structure does not result in a significant performance drop.

We present the following information as a look into the original construction of these data structures and algorithms.

Functionality:

1. Contains a tree structure of node objects (implemented as either Moves or MoveTrees).
2. Can singly add a node efficiently.
3. Can be constructed with serial addition of Move objects efficiently.
4. Can delete a single node efficiently while leaving the tree well balanced.
5. Performs the move choice algorithm, taking the random number and efficiently selects a node. This node then is either the Move we wish to perform, or a nested MoveTree from which we must continue the algorithm.

Data Members:

1. Tree root: pointer to the root node in the tree.
2. Tree total rate: total rate of all nodes in the contained tree.

Discussion: While we could implement a container for the Move type objects in many other ways, it is convenient to use a tree for efficiency reasons. While the worst case complexity of our simulator doesn't change, the average case can be affected be the choice of data storage for these objects, as we will discuss further in section B.7.

The ideal structure for us to use would actually be a Huffman encoding tree based on the relative rates of each move/container stored in the tree. However, maintaining this encoding for a rapidly updating selection of moves is just as inefficient as using a list structure, so while it would be ideal for our choice operations, we would have just moved the time complexity into the update steps.

Finally, we have many different choices of methods to implement our trees in a balanced fashion. Since we do not need a search operation as the elements are not sorted, and are instead accessed by the move choice algorithm using their relative rate, we can implement any simple balanced tree, as long as the insert and delete operations are logarithmic with the total size of the tree. We do need a special algorithm for constructing a tree from a loop's set of moves, as performing these insertions on an empty tree would add an extra logarithmic factor to the requirement. This is easy to implement by maintaining a static list as the tree gets built, and make a single pass to connect the structure and sum the totals once we have all the moves required.

A.8 EnergyModel

Purpose: Computes the energy of particular loop types. Computes rates for moves, based on energy model parameters and rate options. This is a base class, used to implement specific instances of energy models, such as the NUPACK energy model.

Functionality:

1. Calculates the energy for each of the six specific types of loop, based on the energy model parameters used and other energy model options.
2. Calculates the move rate based on the starting and ending state's energy, type of move and other energy model options.
3. Reads in the energy model's parameter set from a file.
4. Transforms the energy model parameter set for non-standard temperatures.

Data Members:

1. Energy model parameters: stores all the parameters for the energy model. This includes dG and dH parameters so that we can calculate dG's for different temperatures.

Discussion: Data Size: Though we list the energy model parameters as a single data member, it should be noted that this is the largest object in terms of data size. There are a huge number of parameters involved in calculating the loop energies, and extra data needs to be stored in order to calculate the energy at different temperatures. Collecting these parameters in a single place is the main reason why the energy is not calculated directly within each loop type, as the storage and extra functionality are better placed in a single external object which can be quickly called.

Scope: Note that this object is used to calculate the energy of specific loops in a secondary structure. Previous libraries for computing the energy only deal with (at the input level) the energy of an entire structure, and as such were not suitable for use here, as the computation of energy for a single loop is the basic operation which we use a large number of times for every step of the simulator.

Different Energy Models: The EnergyModel object is actually a base class, providing a standard interface for calculating the energy of a specific loop type. We implement two different derived classes, ViennaEnergyModel and NupackEnergyModel, based on the Vienna and Nupack parameter sets, respectively. These are particular implementations of the general nearest neighbor energy model and use subtly different parameter files. Other variations on the standard energy model can be used here as necessary, as long as they have the same basic function of being able to calculate the energy of a loop component.

A.9 StrandOrdering

Purpose: Handles the functions within a complex that involve the ordering of the strands within the complex. These include most output functions, reordering of strands that happens when a complex is joined to another, and returning the exterior bases in all the OpenLoops in the structure.

Functionality:

1. Maintains a linked list of the strands within the complex and the implied cyclic permutation needed.

2. Performs circular permutations on the strand ordering, necessary when joining two complexes.

3. Maintains the current sequence and dot-paren structures on demand for output use.

4. Performs stop structure checks to see if the strands in the ordering could match the given set of strand ids and their ordering.

5. Assists SComplex in building the initial loop structure, and maintains links to all the OpenLoops contained within the complex.

Data Members:

1. Strand List: doubly linked list of strands, each containing the following:

   (a) Strand ID.

   (b) Strand Sequence (output version - ASCII).

   (c) Strand Sequence (binary encoded version, used internally).

   (d) Current Dot-paren structure.

   (e) OpenLoop pointer associated with the strand.

2. Exterior base count (updated when open loops change).

Discussion: When we join two complexes together, we need to ensure that the permutation of the strands used for output corresponds to the cyclic permutation which does not contain any crossed chords (this is guaranteed by the Representation Theorem, Section D.1). Maintaining the ordering such that this is guaranteed is very straightforward, we just need to take the cyclic permutation of each set of strands such that the strands being joined together are on an 'edge'. This type of operation happens often when dealing with the structure output and multi-complex moves (joining or breaking).

This object also maintains links into the actual loop structure, used for retreiving the exterior base counts. These get updated automatically when moves are used that affect

OpenLoops. This is also used when we need to actually perform a join move, to help choose which pair of bases is actually being joined, since we only compute the total rate of all join moves combinatorially and don't actually enumerate all the options until one is chosen.

A.10 Options

Purpose: Collects all the information about input, including simulation options, energy model options, associated files and output information.

NOTE: this is a Python object and included here due to the relative dependence on the data contained within at the highest simulation levels (SimulationSystem and EnergyModel). Full documentation on the input and output routines is in the Multistrand documentation, generated automatically for the Python side of the simulator.

Functionality:

1. Maintains simulation options and variables for use by SimulationSystem.
2. Returns energy model options for use by the EnergyModel.
3. Provides interface methods for reporting output to the user.

Discussion: Collects all the options used in the simulator in one data structure. Accessed by EnergyModel and SimulationSystem to retreive the appropriate options when neeeded. Contains the default settings for all options which are not required to be set in the input.

Appendix B

Algorithms

In this section we will examine in greater detail the algorithms used to perform the simulation, both those which take place in the main simulation loop and the initial generation of the loop structure.

B.1 Main Simulation Loop

The main loop of the simulation involves several basic steps needed for most continuous time markov processes as well as the updating of the current structure, from which we can update the currently available moves.

Algorithm (Main Simulation Loop)

For each trajectory:

1. Initialize starting state of the system. (Section B.2)
2. Initialize current time $t = 0$.
3. Calculate total rate of all moves in the current state, R.
4. Generate choice random variable r from the uniform distribution on $[0, R)$.
5. Generate the Δt random variable using u, a random number from the unit interval uniform distribution: $\Delta t = \log(1/u)/R$.
6. Using r, choose the next move to take place, M. (Section B.6)

7. Perform move M, updating the loop structure of the system. (Section B.5)

   (a) Update the set of all possible moves, and thus also R. This is handled within the above step, using the algorithm in Section B.4.

8. Update current time $t' = t + \Delta t$

9. Perform output if needed for current trajectory options.

10. Check for stopping conditions (stop states, maximum simulation time), return to 4 if no stop conditions are met, otherwise continue to the final step.

11. Perform final output and clean up memory for the current trajectory.

Efficiency

The primary loop of the algorithm consists of steps 4 through 10. Each of these steps, except 7a occurs in a linear fashion and thus their time complexities are additive. Step 7a, as discussed in section B.5, occurs at most a constant amount for each cycle. Thus the overall algorithm efficiency is straightforward to compute. We do this in section B.7 after going through each component's complexity.

B.2 Initial Loop Generation

While the initial generation of our data structures is not a significant component of the overall time complexity of this algorithm, it is helpful to understand this translation from the dot-paren representation to the loop representation of the secondary structure. It is the local nature of the loop representation which allows the algorithms of sections B.3, B.4, and B.5 to operate efficiently. Note that we deal with the loop generation on a per complex basis. The parsing of the starting structure into the sequences and dot paren structures for each complex in the initial system is a trivial task and not something we wish expand upon in this section.

Input

1. List L of strands, in the circular permutation corresponding to the non-pseudoknotted structure.

2. Sequence S corresponding to the strands L, with some seperator representing strand breaks.

3. Dot-paren structure T corresponding to the initial secondary structure of the strands L, with some seperator representing strand breaks.

Note that both S and T are essentially character arrays: S being an array of base identifiers and some strand break character, and T being periods (unpaired base), parens (paired to corresponding matching paren), and the strand break character. We also note that in the algorithm below, the 0 index in these arrays is a strand break.

Algorithm Outline

The basic idea here is we convert a flat representation to the loop graph by sequentially traversing the sequence of a single unexplored loop (starting with an open loop), adding the adjacent base pairs to a list of base pairs which need exploration (and are connected to our current loop). This list is then used to retreive another unexplored loop, and we continue the process of sequence traversal to find new base pairs that need exploration. Note that this algorithm requires the input structure to be non-pseudoknotted or it will not terminate (as noted below). Since our input format is implicitly non-pseudoknotted due to being a flat representation, this is not a practical worry.

Algorithm (Initial Loop Generation)

1. Initialize the current location l to 0.

2. Initialize the queues g, g' to be empty.

3. Initialize predecessor pointer p to $NULL$.

4. Starting at position l, let $l' = l + 1$, $seqlen = 0$ and repeat the following until $l' = pair(l)$, or if $l = 0$, until $l' = size(L) + 1$:

   (a) If $T(l')$ is unpaired, let $seqlen = seqlen + 1$.

   (b) Otherwise, add the tuple $(l', seqlen, NULL)$ to g, and let $l' = pair(l')$, $seqlen = 0$.

(c) Let $l' = l' + 1$

5. Using the information in g about sequence lengths, build the current loop J, with adjacent loop p. If J is an open loop, add it to our list of open loops for use in the StrandOrdering.

6. Set p to have the loop J adjacent, across the corresponding base pair.

7. Update all items in g to have j as the predecessor (3rd entry).

8. Add all items in g to g', empty g.

9. If g' is empty, stop - the algorithm is finished.

10. Pop an item i in g', setting $l = i_1$, $p = i_3$, and $seqlen = 0$.

11. Continue from step 4.

Note: Doesn't include the details which connect the StrandOrdering to the correct open loops at each point. This is essentially a single extra operation during step 5, when J is an open loop.

Correctness

To show that this algorithm correctly generates our loop graph, we must argue that it reaches each loop exactly once. We do this by proving that l traverses every base in the complex exactly once: Assume l did not reach some base k. k must then be a part of a loop which was never generated, and whose adjacent base pairs must never have been added to g'. This then implies that this loop containing k must not be connected by any path through a (correct) loop graph to our initial loop, and thus that the structure given was not a completely connected complex. Similarly, if we assume that l reached the base k more than once, it would imply (by corresponding reasoning) that there was a closed cycle in the loop graph, which would mean that the structure was pseudo-knotted.

Discussion

Once each loop has been generated, we can then generate the adjacent moves by a simple traversal through every loop in the structure, using the algorithm in B.4 to create the moves for that loop and adding the resulting MoveContainer to the complex's list of moves.

B.3 Energy Computation

Computing the energy of a particular loop is one of the key components to the overall efficiency of the simulator. This is a straightforward computation with our data structure, as all of the necessary components are directly stored in the Loop object.

Input

1. Type of Loop (Stack, Hairpin, Bulge, Interior, Open, Multi)
2. Sequences for each side of the loop.

Note that while the full sequence of each side of the loop is sufficient for computing the energy, for many of the types we only need a much smaller set of information, usually the bases directly adjacent to each basepair the loop involves, and the side lengths. The steps in the algorithm below are specific to the energy models we implemented. Other forms of energy model may require additional steps for some loop types, or fewer steps.

Algorithm

1. (StackLoop) Look up energy in parameter table.
2. (BulgeLoop) Look up bulge energy in parameter table. Add stacking energy from parameter table if it applies, and terminal AT penalties if they apply.
3. (InteriorLoop) If both sides are ≤ 2, look up the energy in the special case interior loop parameter tables. Otherwise, get interior energy from parameter tables, and add Ninio and mismatch terms via parameter table lookups.

4. (HairpinLoop) Look up hairpin energy via hairpin length in parameter table. If hairpin length is less than 5, add triloop or tetraloop penalties via lookup. Otherwise, add hairpin mismatch energies via lookup table.

5. (MultiLoop) Compute multiloop energy via parameter tables based on number of bases and basepairs in the loop. Add AT penalties for each base pair that applies. If dangles option is set, add dangles energy (lookup for each base pair based on the pair and adjacent bases).

6. (OpenLoop) Default energy is 0, add AT penalties for each base pair if they apply, if dangles option is set, add dangles energy (lookup for each adjacent base pair as in the Multiloop case).

Correctness

This is the energy model as discussed in references [9, 5, 27]. We validate the energy model by comparing our computed energies for arbitrary structures with those of published programs, discussed in Section 6.1.1.

Discussion

While this algorithm essentially amounts to being straight table lookup for each loop type, we note that both Multi and Open loops can require slightly more than constant time, as they may require lookups on the order of the number of base pairs in the loop. This can be easily constrained to be less than the number of bases present and is typically a very small number, but it is a possible concern for simulation efficiency when using dangles terms. In any case, the key improvement here is that previous algorithms had to compute the energy of the entire secondary structure (a $O(N)$ computation) for each generated move, where we only need to compute the energy of one or two loops as our basic step.

B.4 Move Generation

Whenever we have performed a move we need to recalculate all the possible moves for all the new and adjacent loops involved in that move, using this algorithm for each new loop, and just the deletion loop components for the adjacent loops. Each new loop needs to calculate

the set of base pair creation moves that take place within it, as well as the deletion moves which involve that loop (one for each adjacent loop).

Algorithm Outline

Deletion moves are straightforward to generate: given the base pair to be deleted, compute the energy of the loop that would result if it were deleted (generally a constant time computation to form that loop and get its energy), and use that to compute the rate given the energy of the two original loops.

For creation moves, the algorithm is based on finding all the possible internal bases that could pair. This is a two step process: for each side of a loop (a single stranded region adjacent to a base pair, or connecting two base pairs), consider every pair of bases which are three bases away or further, a creation move is possible if they are complementary bases. For each pair of sides of the loop, every pair of complementary bases with one chosen from each side leads to a possible creation move. Each of these possible creation moves needs to have the energy of the resulting pair of loops calculated, and setting up these energy calls and computing the energy is generally a constant time operation. This is then used to generate the move rate using the chosen rate method from the model, since the only energy difference between the new structure and the old will be the difference between the energy of the loop where the move occured, and the energy of the resulting pair of loops.

Note: we only indent the relevant looping constructs when necessary to indicate several different looping constructs for the algorithm; if not indented, it implies we repeat the remainder of the steps at that level while the loop condition holds.

Algorithm: StackLoop Creation Moves

StackLoops have no free internal bases, and thus can't have creation moves.

Algorithm: HairpinLoop Creation Moves

1. For each base i in the hairpin:

2. For each base $j > i + 3$ in the hairpin:

3. If i,j are complementary, compute energy of the resulting loops and add this creation move to the hairpin's MoveContainer.

Algorithm: BulgeLoop Creation Moves

1. For each base i on the bulge side:
2. For each base $j > i + 3$ on the bulge side:
3. If i,j are complementary, compute energy of the resulting loops and add this creation move to the bulge's MoveContainer.

Algorithm: InteriorLoop Creation Moves

1. For each interior side i:
   (a) For each base j on side i:
   (b) For each base $k > j + 3$ on side i:
   (c) If j,k are complementary, compute energy of the resulting loops and add this creation move to the interior loop's MoveContainer.
2. For the pair of interior sides i, l:
3. For each base j on side i and each base k on side l:
4. If j,k are complementary, compute energy of the resulting loops and add this creation move to the interior loop's MoveContainer.

Algorithm: MultiLoop Creation Moves

Same as Interior, with each pair of sides $i \neq l$ for the second part.

Algorithm: OpenLoop Creation Moves

Same as Interior, with each pair of sides $i \neq l$ for the second part.

Correctness

Creation moves can only occur within a loop, otherwise we would get a pseudoknotted structure. Since we consider every possible pairing of bases within a loop, we cover all possible creation moves. Deletion moves are covered in a similar manner: each deletion move is added twice, once to each loop adjacent to that base pair, and the overall rate from each is halved to account for this. Since we always add a deletion move for each loop's adjacent base pairs, we include all possible deletion moves.

Discussion

The complexity of any loop's move generation is relatively straightforward: if that loop contains N unpaired bases and k basepairs, we have $O(N^2)$ possible resulting creation moves, and exactly k deletion moves. To see why $O(N^2)$ is the worst case for creation moves, let us use the example of an open loop with l sides, each having N/l unpaired bases. If we assume these bases can always pair, there are then $l * (l-1)/2$ possible choices of two sides, each combination having $(N/l)^2$ possible pairings, for a total of $O(N^2)$ pairings. (Single side creation moves in this case lead to a similar l sides times N^2/l^2 possible moves, for the same magnitude).

It is interesting to note that these large loops with high numbers of unpaired bases are actually very rare for typical strand sequences and secondary structures. If we break down the average number of possible moves, weighted by the boltzmann distribution for energy of the loop, all loop types (except open loops) tend towards a very small number of moves possible on average.

B.5 Move Update

After a move is chosen, we must actually perform that move and update the appropriate parts of the data structure. Specifically, the loop graph will undergo the changes illustrated previously in Figure 5.3, and those changed loops will need to have their moves calculated (Section B.4) and the complex must have the total rate updated. This algorithm covers the loop structure update as well as the updates to the appropriate move containers to remove those moves associated with the deleted loops.

Algorithm Outline

This algorithm is given a move that needs to be performed. Each move has an associated loop (if it is a creation move) or pair of loops (if it is a deletion move). Performing the move is then straightforward: using a similar algorithm to the move generation algorithm, we can compute the resulting loop types from our move. These are created, and the loop structure updated to include the new loops and remove the old ones.

Input

1. A Move object m, provided by the Move Selection algorithm (B.6), which contains:.
   (a) Pointers to the loops l affected by the move (up to two).
   (b) Indices i which uniquely identify which move it is within the affected loop (up to four).

Algorithm

1. If m affects two loops l_1, l_2, we have a deletion move:
   (a) Using l_1, l_2, compute the loops l_3, l_4 which result from deleting the base pair joining l_1 and l_2 (note that we get two resulting loops only if the loops involved in the deletion are both open loops). The loop computation is equivalent to the same component in the initial loop generation, and is linear in the sequence length contained in the loops.
   (b) Rebuild connections between loops adjacent to l_1, l_2 to now be adjacent to l_3, l_4 with the same base pair connections as before.
   (c) If l_1 and l_2 are both open loops, we will have a disconnected complex after the deletion, so we seperate one of the connected regions into a new complex and add it to the list of all complexes.
   (d) Delete l_1, l_2.
   (e) For each of the newly generated loops, generate the associated move set. For each loop adjacent to the newly generated loops, recalculate the deletion moves.
2. Otherwise, we have a creation move within the single affected loop, l:

(a) Using the indices i within l, compute the resulting pair of loops m, n.

(b) Reconnect the loops adjacent to l with the equivalent base pairs of m, n.

(c) Delete l.

(d) For each of m, n, generate their move sets. For each loop adjacent to m or n, recalculate the associated deletion moves.

Correctness

In order for this update to be correct, it must match the same conditions as those in the initial loop generation: each unpaired base must occur in exactly one loop, and each paired base only in two loops (that are adjacent).

We handle creation and deletion moves seperately:

Correctness - Deletion

Step (a) generates one or two resulting loops l_3, l_4 using all of the bases contained within l_1,l_2. Each unpaired base from l_1, l_2 occurs in exactly one of l_3, l_4. Similarly, each paired base from l_1, l_2 must either still be present in one of l_3, l_4 if it was from a basepair to an adjacent loop, or it is now an unpaired base (if it is from the basepair connecting l_1 and l_2. Thus this step maintains the correct usage of each base.

Step (b) is necessary to maintain the overall structure, otherwise a loop m adjacent to l_1 will still be connected to l_1 rather than the correct new loop l_3 or l_4. Thus this step ensures that no loop in the main structure is now connected to l_1 or l_2, and l_3, l_4 are now correctly connected.

Step (c) maintains the requirement that each complex be connected, by adding a new complex when a deletion move would result in two seperate connected components.

Step (d) is standard memory management, but doesn't affect the overall structure now as there are no further references to l_1 or l_2 due to step (b).

Step (e) rebuilds all the moves which are in l_3, l_4 and the deletion moves in the loops now adjacent to them.

Correctness - Creation

Step (a) generates two new loops m, n using all of the bases contained within l. The two unpaired bases in l which were joined in the creation move are now in each of m, n as paired bases, and every other unpaired base from l occurs in exactly one of m, n. Similarly, each paired base from l must either still be present in exactly one of m, n. Thus this step maintains the correct usage of each base.

Step (b) is necessary to maintain the overall structure, otherwise a loop k adjacent to l will still be connected to l rather than the correct new loop m or n. Thus this step ensures that no loop in the main structure is now connected to l, and m, n are now correctly connected.

Step (c) is standard memory management, but doesn't affect the overall structure as there are no further references to l.

Step (d) rebuilds all the moves which are in m, n and those deletion moves in the loops adjacent to l.

Discussion

This algorithm is straightforward to implement, as the hard part is the classification of the new loops, and the stored indices from the move generation stage contain all of the information needed to classify the loops, without an additional look at the structure. In terms of complexity, all of the steps here are either constant time or linear in the sequence length of the affected loops, and they are all additive, so this algorithm is not a large component of the overall time. Note that the move generation algorithm B.4 is called here implicitly, but the full algorithm is only called on either one or two loops, while the deletion moves are recomputed for all adjacent loops (but are a constant time recomputation).

B.6 Move Choice

The process of choosing a move out of all of the options has the potential to be a significant factor in the overall time complexity. Hence, the choice of selection algorithm and the underlying data structures to support the algorithm is an important one.

Algorithm Outline

We use a Gillespie-style approach to choosing the next move to take in the markov process, with a few differences at the underlying structure level. The key difference is that the selection of moves available to the markov process is constantly changing at every step, since we cannot just enumerate all the moves available for any stage of process. For a single complex, we arrange the moves into two layers of trees: The first layer's nodes are always the root of a second layer tree whose nodes are those of a single Loop's available moves. Thus when choosing which move to make from a complex, we have a logarithmic number of steps to make using our single random number to arrive at a choice. Similarly, for multiple complexes, we just add an extra layer whose nodes are the two-layer structures for a single complex.

These details are handled mostly within the data structure implementation in a manner than makes our algorithm straightforward: Each layer is a MoveTree, thus its nodes have a data component that is either a MoveTree (if the layer represents the collection of a complex's moves, or the collection of multiple complexes' moves), or a Move (if the layer represents all the moves available to a single Loop). Thus our algorithm proceeds through these tree structures as usual for a Gillespie-style markov choice step, except that on arriving at a chosen node, it may itself be a MoveTree which requires additional traversal in the same manner before finally arriving at a Move node which can then be performed. We note that while the description given is for MoveTree containers, in general this could be for any type of container holding the Move objects, such as the default array container currently being used.

Input

1. A MoveTree M, which contains all the moves for all complexes in the system.

2. A random number r, chosen from the $[0, T)$ uniform distribution, where T is the total flux over all moves in all complexes.

Algorithm

Recall that both the MoveTree and Move objects are usable as a node, and thus contain pointers to a left child l, right child r, the total rate through each child node $t(l), t(r)$, and the rate contained within the MoveTree or Move itself, R. Our input is then not part of any tree itself (and so the Node fields are empty), but the MoveTree object contains a pointer to the root Node N, which we then operate on as follows:

1. Set current node C to N.
2. If $r < C.R$:
   (a) If C is of type Move, we return C as our selection.
   (b) Otherwise, C is of type MoveTree and thus contains a root node N, so we set our current node C to N and go to 2.
3. Let $r = r - C.R$
4. If $r < C.t(l)$ then let $C = C.l$ and go to 2.
5. Otherwise, let $r = r - C.t(l)$, $C = C.r$ and go to 2.

Correctness

To show that this algorithm is correct, we need to prove that each Move is chosen according to the stated distribution in the model (reference), that is to say, a Move with rate R is chosen by this algorithm exactly R/T of the time, where T is the total rate of all moves in the system's current state.

This is indeed the case, as we can easily prove via induction:

Base case: Assume that we are choosing a Move from tree which consists of a single Move node. The total rate in the tree is T, our random number r is in the range $[0, T)$, and the rate of the single node is $R = T$, so we always pick the single move in the tree.

Inductive Step: Assume that our algorithm works for a tree with a fixed size, we need to prove that it works for two cases: a tree with one extra layer of depth, and for a single MoveTree node C (which thus contains a tree N). The second case is straightforward, as our r is then in the range $[0, C.R)$ and can thus be used correctly to choose a node from

the subtree N, which must have total rate $C.R$. The first case is similar: If our current node is C, we choose the node's underlying MoveTree or Move with rate $C.R/T$ correctly. The left child is chosen with rate $C.t(l)/T$ and r is then in the range $[0, C.t(l))$, and the right child is chosen with rate $C.t(r)/T$ and r is in the correct range $[0, C.t(r))$. (Recall that $T = C.R + C.t(l) + C.t(r)$) Thus with our inductive assumption, if we picked the left child, the eventual move chosen M with rate $M.R$ is picked $M.R/C.t(l)$ of the time, and thus overall is picked $M.R/T$ of the time relative to this tree. The same holds true for the right child, and thus the inductive assumption holds.

Discussion

The main complexity in this algorithm lies in the actual data structures used, and not the algorithm itself, as it is straightforward, even with the multiple layers. Since we have at most 3 layers of nested trees, and each could have in worst case N^2 nodes, where N is the total sequence length, we have a running time of approximately $3 * 2 \log N$ or $O(\log N)$ in the worst case to perform a choice, and only a single random number is needed.

B.7 Efficiency

Summarizing from section B.1, we have the following steps within the simulation's main loop, which must be considered in order to determine the overall algorithm efficiency:

Main Loop Steps

1. Move Choice Algorithm (Section B.6)
2. Move Update Algorithm (Section B.5), which requires:
   (a) Move Generation Algorithm (Section B.4).
3. Output Step.
4. Stop Condition Check.

Controlling Quantities

There are three main input quantities that control the algorithm complexity:

1. The total sequence length N.

2. The maximum simulation time, T.

3. The number of trajectories to simulation, V.

Analysis

Each of the first three steps have been analyzed in the corresponding section, so let us examine the running time for the final two steps. The output step is only used in trajectory mode and transition state mode (Sections 6.1 and 6.3) and is $O(N)$. However, this does require passing information back to the Python side (which might then require file I/O to store the data), it can run much slower if the file system gets overloaded.

Stop structure checking is also worst-case linear in the sequence length, per stop structure in the input. Thus if we have S total stop structures in the input, this step is $O(S*N)$. In practice we have two mitigating factors, one is that S is typically much smaller than N, and the other is that there are several straightforward shortcuts which keep us from having to check all the stop structures at every step. For example, if we check a particular state against a stop structure s_1 and find that they differ by 10 basepairs, we do not have to check against stop structure s_1 until we have made ten more elementary steps.

We summarize the additive components of the main loop's complexity in the following table:

Step	**Complexity**
Move Choice	$O(\log N)$or$O(N^2)$
Move Update	$O(N)$
Move Generation	$O(N^2)$
Output	$O(N)$
Stop Structures	$O(S*N)$

From this table we can conclude that the worst case complexity for this algorithm, *per step of the main loop* is $O(N^2)$. We now have to calculate how many of these basic steps are necessary in order to simulate T seconds in our system. To do this we must make an estimate of the average total rate per system state. We make a fairly poor estimate that this is on the same order as the number of loops in the system, and thus $O(N)$ at maximum.

Our average simulated time per step is then the inverse of this total flux, and so it must take $O(N)$ steps to simulate one second of simulation time. Thus for an entire trajectory of time T, it takes $O(T * N^3)$, and to run V of these trajectories it is then $O(V * T * N^3)$.

To see why we might wish to use a logarithmic time complexity algorithm for the Move Choice step (such as the one prevously presented in section B.6) rather than the current implementation (which is linear in the number of moves, thus $O(N^2)$ in the worst case), we look at a type of structure where the Move Choice algorithm becomes the limiting step. Imagine a secondary structure that is mostly composed of stack loops, e.g. two complementary strands that are fully hybridized. Here the number of moves is $O(N)$, but the move update and move generation steps are $O(1)$ due to the simplicity of the resulting structures. Thus if we are only considering the three move-based steps in the algorithm, the rate limiting step will be the move choice step. Each individual loop will have a constant number of moves (so using an array data structure isn't really any worse than a tree), but the top level move container will need to collect the $O(N)$ loops. So the move choice step is $O(N)$ if we use an array data structure, and $O(\log(N))$ if we use a tree. In actual performance testing for the simulator, we found that the move choice step does not take a significant portion of the per-step time even when using array-based move containers.

Appendix C

Equivalence between Multistrand's thermodynamics model and the NUPACK thermodynamics model.

We now return to examine the thermodynamics model presented here (Section 3) and compare it to the NUPACK thermodynamics model [5]. As we noted previously, our energy model differs in exactly two ways: the lack of any symmetry terms in the energy of a complex, and the addition of the ΔG_{volume} term. These, however, are merely natural consequences of a much more basic difference: the Multistrand energy model deals with systems of uniquely labelled strands in a stochastic regime, while the NUPACK energy model allows populations of strand species and operates in a mass action regime.

In this section we will examine these two models in greater detail. We introduce a particular class of macrostate which we call an "indistinguishability" macrostate, i.e. a way of grouping Multistrand's system microstates (which have all strands uniquely identified) into macrostates that are equivalent to the states on which the NUPACK thermodynamics model is defined. We then show that using these macrostates our models are mathematically equivalent, that is to say, they predict the same partition function over these macrostates as well as the same probability of observing each macrostate.

In order to build up to this proof of equivalence, we will need to introduce some extra framework detailing the differences between the two models and discussing the main concepts necessary (such as the "indistinguishability" macrostates). To make this discussion easier, we note the following conventions: whenever we refer to a system microstate, we always mean a Multistrand system microstate, defined in section 2.3. Once we have defined indistinguishability macrostates, we will always refer to them as simply macrostates

(or perhaps ind. macrostates), with the understanding that they are a specific class of macrostates, rather than a general definition such as in section 6.2.

C.1 Population Vectors

We previously introduced the set of (uniquely labelled) strands Ψ^*, but since we are now allowing duplicate strands we need a few more terms. Thus we have the set of strand species Ψ^0, which we can think of as the set of all the types of strands: one entry for each unique strand in the system, and we can then have multiple copies of that species in our system. We also have the set of strand complexes Ψ, where a strand complex is defined as some number of each strand species in a connected secondary structure. This is just the combinations of strand species we could have formed into a complex. Note that for a system with a finite number of strands, the set of possible strand complexes is also finite, though it can be quite large. Also, $\Psi^0 \subset \Psi$, and by convention two distinct species may not have the same sequence or label.

We can now think of an abstract representation of the current system's state: a population vector $m \in \mathbb{N}^{|\Psi|}$, where we note $\mathbb{N} = \mathbb{Z}_{\geq 0}$, which has an entry for each type of strand complex, indicating the number of complexes of that type in the current system's state. The initial population vector $m^0 \in \mathbb{N}^{|\Psi^0|}$ indicates how many of each type of strand are present in the system. We relate the two by the strand matrix $A \in \mathbb{N}^{|\Psi^0| \times |\Psi|}$, whose entries A_{ij} correspond to the number of strands of species i in complex type j. By our previous definition of strand complex, the rows of A are distinct. Thus, if we have a system which starts with m^0 of each strand species, $\Lambda = \{m \mid Am = m^0\}$ is the set of all population vectors consistent with conservation of strand counts.

C.2 Indistinguishability Macrostates

We now need to introduce a specific class of macrostate which we call **indistinguishability** macrostates, or **ind.** macrostates for short. The idea behind these is that in Multistrand, we assume every strand is uniquely labelled, but if we were given a set of strand species

and a mapping from our unique labels to a species, we then could partition our existing system microstates into groups representing those which would be indistinguishable when considering the strand species.

First, recall that every complex microstate c contained an ordering $\pi^*(c)$ on the strand ids. While this wasn't used very much in our original discussion, we now need to define this further: the ordering $\pi^*(c)$ is a *circular permutation* on the strand ids of c (which are $ST(c)$). We note that *circular permutations* are the permutations of the objects which are distinct when they are laid out on a fixed circle without a defined starting point. E.g. for strands labelled $1, 2, 3$ there are only two circular permutations on the strand ids: $(1, 2, 3)$ and $(1, 3, 2)$. As it will be used later, a *cyclic permutation* is a permutation which does NOT lead to a distinct ordering when laid out on a fixed circle – that is, they "rotate" the strands without rearranging their relative order. E.g. for a strand ordering $(1, 2, 3)$, there are three cyclic permutations of that strand ordering: $(1, 2, 3), (2, 3, 1)$ and $(3, 1, 2)$. For a given set of n elements, there are $(n-1)!$ circular permutations and n cyclic permutations of the elements.

So, every complex microstate c contains an ordering $\pi^*(c)$ that is a circular permutation on the strand ids. We define the set of *indistinguishable* orderings of c as the circular permutations of c's strand labels (labels represent the strand species). For example, with a complex of 4 strands, where strand id's 1 and 2 have label A, and strand ids 3 and 4 have label B, the circular permutations on strand ids are $\{(1, 2, 3, 4), (1, 2, 4, 3), (1, 3, 2, 4), (1, 3, 4, 2), (1, 4, 2, 3), (1, 4, 3, 2)\}$, and the indistinguishable orderings are $\{(A, A, B, B), (A, B, A, B)\}$. Thus, we can think of a complex microstate as having the ordering $\pi^*(c)$ from the circular permutations on the strand ids, but also having an ordering $\pi(c)$ from the circular permutations on the strand labels which represents the indistinguishable ordering for the complex if strands with the same sequence are considered indistinguishable. This $\pi(c)$ tells us which complex ind. macrostate this complex microstate is a member of.

We can now define the concept of two complex states being indistinguishable, and thus approach a definition for a macrostate of the system.

Definition:

The complex microstates c, c' are indistinguishable if the following properties hold:

1. The indistinguishable orderings $\pi(c), \pi(c')$ correspond to the same indistinguishable ordering π_{eq}. Note that this is the case iff $\pi(c)$ is a cyclic permutation of $\pi(c')$.

2. There exists a one-to-one mapping ξ between the strand ids of c and the strand ids of c' such that for every base pair $(i_j \cdot k_l)$ in c, there is a base pair $(i_{\xi(j)} \cdot k_{\xi(l)})$ in c', and for every base pair $(i_{\xi(j)} \cdot k_{\xi(l)})$ in c' there is a base pair $(i_j \cdot k_l)$ in c. In other words, the mapping on strand ids induces a one-to-one mapping between the base pairs of each complex. Note that if ξ exists, it is not necessarily a unique mapping, with the cases where it is not corresponding to symmetric complexes, examined below. □

From this definition it is easy to see when two system microstates are indistinguishable: System microstates s, s' are indistinguishable if there exists a one-to-one mapping between the complex microstates of each system microstate, such that complex microstates map onto indistinguishable complex microstates.

We now put these all together to define an indistinguishability macrostate of the system (or, for want of a clunky name, a system ind. macrostate). A system ind. macrostate v is a set of system microstates from S (the set of all system microstates), such that for every pair of microstates $s, s' \in v$ with $s \neq s'$, s and s' are indistinguishable, and for any system microstate $s'' \notin v$, s'' is distinguishable from any system microstate $s \in v$. Note that in many cases when talking about ind. macrostates, we will use an alternate form, in which the ind. macrostate is given by a particular system microstate representatitive from the set. This is convenient for energy functions in particular, as every system microstate in an ind. macrostate has the same energy, and it is sometimes useful to know what a (microstate) energy function would give for a macrostate's energy. Note that this is **different** than the ind. macrostate's energy, as that is given by equation 6.2 from section 6.2.

We also have a notion of an ind. macrostate of a complex (a complex ind. macrostate, as used previously). This is defined similarily - a complex ind. macrostate v' is a set of

complex microstates from C (the set of all complex microstates for a particular complex), such that for every pair of complex states $c, c' \in v'$ with $c \neq c'$, c and c' are indistiguishable, and for any microstates $c'' \notin v', c \in v'$, c'' is distinguishable from c. Finally, a complex ind. macrostate has an additional property which will be used in energy functions, which is the symmetry factor. The symmetry factor of a complex ind. macrostate c (where we now use the representative complex microstate), $R(c)$ is the number of cyclic permutations on the strand ids which map the base pairs of a representative complex state onto themselves. That is to say, it is the number of cyclic permutations ξ' such that for every base pair $(i_j \cdot k_l)$ in s, $(i_{\xi'(j)} \cdot k_{\xi'(l)})$ is also a base pair in s.

In the upcoming sections we will examine the size of complex ind. macrostates, as well as energy functions which apply to complex microstates, system microstates, and ind. macrostates of both types. Finally, when talking about a single complex type $j \in \Psi$, we will occasionally refer to the set of all indistinguishable orderings of the strands, Π_j, as well as the complex ind. macrostates in a given indistinguishable permutation $\pi \in \Pi_j$, which is $\Omega_j(\pi)$.

C.3 Macrostate Energy

Recall that we can calculate the probability of a specific system microstate s using the energy $\Delta G^*_{box}(s)$ and the partition function $Q_{kin} = \sum_i e^{-\Delta G^*_{box}(i)/RT}$ as $Pr(s) = \frac{1}{Q_{kin}} e^{-\Delta G^*_{box}(s)/RT}$. Macrostate probabilities can be calculated in the same manner, using the energy function for an ind. macrostate and the appropriate ind. macrostate partition function (or by summing the probabilities of the constituent microstates).

Recall that we previously gave the standard free energy of a complex microstate c, containing L strands as:

$$\overline{\Delta G}(c) = \left(\sum_{loop \in c} \Delta G(loop) \right) + (L-1)\Delta G_{assoc}$$

And for our kinetic system, we use the following, which includes the volume-dependent

term in the complex's energy.

Energy of a complex microstate c, containing L strands:

$$\begin{aligned}\Delta G^*(c) &= \left(\sum_{loop\in c}\Delta G(loop)\right) + (L-1)*(\Delta G_{assoc}+\Delta G_{volume}) \\ &= \overline{\Delta G}(c) + (L-1)\Delta G_{volume}\end{aligned}$$

And combined, the energy of a system microstate s is:

$$\Delta G^*_{box}(s) = \sum_{complex\ microstate\ c\in s}\Delta G^*(c)$$

The volume term ΔG_{volume} for the remainder of this discussion will be the quantity RT log M_s, where M_s is the number of solvent molecules in the box. This will be a useful comparison later, as the NUPACK partition function uses the same terms.

Lastly, the energy of a complex ind. macrostate c, containing L strands with a symmetry factor $R(c)$ as computed directly using the NUPACK energy model [5], is:

$$\Delta G(c) = \left(\sum_{loop\in c}\Delta G(loop)\right) + (L-1)\Delta G_{assoc} + RT\log R(c) \tag{C.1}$$

C.4 Partition Function

We now wish to compare the partition function over our system microstates with the partition function Q_{box} examined in [5], which is over (our) system ind. macrostates.

First it is important to know what system we are working with. The general system is that of a box containing initial strands according to the labelled population vector m^0, thus we are considering a system with possibly indistinguishable strands. First, we define a system microstate as being consistent with a population vector m (this is a population vector over labelled complex types) if it contains the correct number of complexes of each type in the population vector, if we consider strands indistinguishably (according to strand species). Then, note that every individual system microstate must be consistent with some population vector m which satisfies $Am = m^0$. Let S be the set of all system microstates

which are possible with initial strands m^0, then we can use $\Lambda = \{m \mid Am = m^0\}$ and let $S(m)$ be the set of all microstates which have population vector m.

We can then write the system microstate partition function as:

$$Q_{kin} = \sum_{s \in S} e^{\Delta G^*_{box}(s)/RT}$$

And rewriting in terms of Λ as:

$$\begin{aligned} Q_{kin} &= \sum_{m \in \Lambda} \sum_{s \in S(m)} e^{\Delta G^*_{box}(s)/RT} \\ &= \sum_{m \in \Lambda} q_{kin}(m) \end{aligned}$$

Where we define $q_{kin}(m)$ as $\sum_{s \in S(m)} e^{\Delta G^*_{box}(s)/RT}$.

Next we wish to examine the system ind. macrostate partition function Q_{box}, which considers the system ind. macrostates with initial population vector m^0. We will explain the components of Q_{box} in detail, as presented in [5]:

$$Q_{box} \approx Q_{ref} \sum_{m \in \Lambda} q(m)$$

This is the basic equation[1]. Q_{ref} is the reference state of the box, which includes terms related to the volume and the reference state s for which $\Delta G_{box}(s) = 0$, as will be discussed later.

$$q(m) \equiv \prod_{j \in \Psi} \frac{M_s^{m_j} Q_j^{m_j}}{m_j!} \tag{C.2}$$

This is the partition function over the macrostates of a particular population vector m, and it is the product of the partition functions of the individual complex types (Q_j's), corrected for solvent (the $M_s^{m_j}$ term) and symmetry (the $m_j!$ term, as well as the particular Q_j used).

[1] We note that this approximation is due to an assumption that the number of solvent molecules in the box is much greater than the number of strands.

$$Q_j = \sum_{\pi \in \Pi_j} Q_j(\pi)$$

This is the partition function over all the macrostates (of a complex) for a particular type of complex j. Since each type of complex could have many different orderings of the strands, this is a sum over all the indistinguishable orderings for that complex type (Π_j), of the partition function for that type of complex for a particular indistinguishable ordering $Q_j(\pi)$.

$$Q_j(\pi) = \sum_{c \in \Omega_j(\pi)} \exp(-\Delta G(c)/RT)$$

Finally, this is the partition function over all the macrostates of a complex of type j, in indistinguishable ordering π, and it is just the sum over the symmetry corrected energy for each macrostate.

C.5 Proof of equivalence between Multistrand's partition function and the NUPACK partition function

C.5.1 Proof Outline

Our goal now is to show the equivalence between the partition function over system microstates, Q_{kin} and that for system ind. macrostates Q_{box}. While we would like to also show that the probability of observing a given system ind. macrostate is equivalent between the two models, it is beyond the scope of this already quite long proof. However, the same general structure used for the partition function proof can be used to show the equivalence of the probability between the two models. The two key insights for the probability proof are to count the number of system microstates composing a particular ind. macrostate, and to derive from [5] the probability of a specific ind. macrostate as in that work the smallest component considered in the "box" partition function is the probability distribution for a particular population vector m.

It is helpful to understand what direction we take to prove the equivalence between these two formulations of the partition function. We first examine entire systems where there is only a single complex of a particular type and the complex is always connected throughout time. This allows us to work out the microstate partition function for a very small case where it is easy for us to count the number of microstates in each macrostate. This also leads to examining the difference in energy functions used, and explains why we match the symmetry corrections even though the microstate energy function doesn't include any symmetry terms.

Once we have established the very small case, we can examine a larger system, where we have any number of complexes of a single type, and that number doesn't change. This is a building block step towards being able to write out the microstate partition function for any population vector m, at which point we can use the Q_{kin} equation above to handle any system (recall that individual states in a macrostate must always be within the same population vector).

The key here is to remember that in each of these three steps, the **system** we are dealing with is a different one, whose microstates are those possible given a particular choice of population vector m.

C.5.2 System with a single complex

Consider a kinetic system whose states are entirely in the population vector $m : m_j = 1, m_i = 0 \ \forall i \neq j$. This corresponds to a system where there is a single complex, which is never allowed to disassociate. Let S be the set of microstates of the system (which satisfy population vector m). This is equivalently the $S(m)$ discussed before, but we simplify the notation here as we are always working with a single m. We then have:

$$q_{kin}(m) = \sum_{s \in S} e^{-\Delta G^*_{box}(s)/RT}$$

Note that for this simple system, the microstates s are each a set containing a single

corresponding complex state c. It is interested to note that for this system, since we have restricted the complex to always be connected, the system is not necessarily connected by kinetic moves in the state space S. Thus $\Delta G^*_{box}(s) = \Delta G^*(c)$ for these microstates. Note that they can still be from different indistinguishable permutations of the strands, so we group them by the indistinguishable permutations Π, letting $S(\pi)$ be the set of microstates which are consistent with the indistinguishable ordering π:

$$q_{kin}(m) = \sum_{\pi \in \Pi} \sum_{\substack{s \in S(\pi) \\ s = \{c\}}} e^{-\Delta G^*(c)/RT}$$

We now consider the macrostates of the complex j, $s' \in \Omega_j(\pi)$. We wish to know how many microstates $s \in S(\pi)$ are contained in the macrostate s' (or put differently, indistinguishable from the macrostate s', if we treat the macrostate as a representative of the set). Let $R(s')$ be the symmetry factor of the macrostate. First, we note that for $R(s') = 1$, any permutation on the indistinguishable strands will lead to a unique microstate that is indistinguishable from s'. Thus there are $\prod_{i \in \Psi^0} (Am)_i!$ microstates corresponding to each non-symmetric macrostate s'. (Note that they must all have the same energy.) In symmetric states, some of these permutations (the circular ones with the appropriate symmetry) lead to equivalent microstates, and thus the total number of microstates corresponding to a symmetric macrostate is lower, by the symmetry factor R. Thus for either case we have the general formula of $\frac{1}{R(s')} \prod_{i \in \Psi^0} (Am)_i!$ microstates for each macrostate s'.

We then rewrite q_{kin} in terms of the macrostates and L, the total number of single strands used in the system ($L(m) = \sum_{i \in \Psi^0} (Am)_i$):

$$
\begin{aligned}
q_{kin}(m) &= \sum_{\pi\in\Pi}\sum_{\substack{s'\in\Omega(\pi)\\ s'=\{c'\}}}\left(\frac{1}{R(s')}\prod_{i\in\Psi^0}(Am)_i!\right)e^{-\Delta G^*(c')/RT} \\
&= \sum_{\pi\in\Pi}\sum_{\substack{s'\in\Omega(\pi)\\ s'=\{c'\}}}\left(\prod_{i\in\Psi^0}(Am)_i!\right)e^{-\overline{\Delta G}(c')/RT-(L(m)-1)\log(M_s)-\log(R(c'))} \\
&= \sum_{\pi\in\Pi}\sum_{\substack{s'\in\Omega(\pi)\\ s'=\{c'\}}}\left(\prod_{i\in\Psi^0}(Am)_i!\right)\frac{M_s}{M_s^{L(m)}}e^{-\Delta G(c')/RT} \\
&= \left(\prod_{i\in\Psi^0}(Am)_i!\right)\frac{M_s}{M_s^{L(m)}}Q_j
\end{aligned}
$$

And so we have shown that the case of a single complex comes out to exactly the ind. macrostate partition function $q(m)$ (Equation C.2) of the complex, with an extra factor of $\left(\prod_{i\in\Psi^0}(Am)_i!\right)\frac{1}{M_s^{L(m)}}$ due to reference state and volume.

C.5.3 System with multiple copies of a single complex type

We now build on this result by using it as the base case for induction to cover a system composed entirely of complexes of a single type. In particular, we are examining a system where $m : m_j = n, m_i = 0\,\forall i \neq j$, and our inductive hypothesis is:

$$
q_{kin}(m) = \left(\prod_{i\in\Psi^0}(Am)_i!\right)\frac{M_s^{m_j}}{M_s^L}\frac{Q_j^{m_j}}{m_j!}
$$

We have shown the base case ($n = 1$), so now we need to show the inductive step. Again, we define S as the set of microstates consistent with m, and our normal definition of q_{kin} is:

$$
q_{kin}(m) = \sum_{s\in S} e^{-\Delta G^*_{box}(s)/RT}
$$

To cover this case we need to look at an expanded set of states, where the difference is that we add a "marking" to some part of the microstate (complexes).

We expand the system as follows:

$$\begin{aligned} S' &= \bigcup_{s \in S} Mark(s) \\ Mark(s) &= \{(s,c) | c \in s\} \end{aligned}$$

Consider now $q_{exp}(m) = \sum_{(s,c) \in S'} e^{-\Delta G^*_{box}(s)/RT}$. $Mark(s)$ expands every original state into m_j new states, all of which have the same energy for the system microstate component. Thus we know that $q_{kin}(m) = \frac{1}{m_j} q_{exp}(m)$.

We now use the markings to group microstates from S'. Consider the following grouping of microstates, based on a complex microstate c (note that c by definition includes a specific set of strand ids used in the complex):

$$F(c) = \{s | (s,c) \in S'\}$$

Thus $F(c)$ is all system microstates in S' that contain a complex microstate c (with specific set of strand ids). Now consider the sum over all the microstates within one grouping:

$$\sum_{s \in F(c)} e^{-\Delta G^*_{box}(s)/RT} = e^{-\Delta G^*(c)/RT} * \sum_{s \in F(c)} e^{-\Delta G^*(s \backslash c)/RT}$$

where we let $s \backslash c$ represent the microstate that would result if the marked complex microstate c were removed from the system microstate s. Looking at the resulting summation, note that it is over all the microstates for a system with one less complex of type j! To see why it is exhaustive, note that if it were not, the original system would be missing valid microstates consistent with m, similarily, it cannot overcount any states, as that would imply duplicated states in the original S. Thus we can use our inductive hypothesis to express the result:

$$\sum_{s \in F(c)} e^{-\Delta G^*_{box}(s)/RT} = e^{-\Delta G^*(c)/RT} * q_{kin}(m')$$

Where $m' : m'_j = m_j - 1, m'_i = m_i = 0 \, \forall i \neq j$ is our smaller system. Note that the choice of strand ids only matters for the complex in state c, not for the rest, as in general we can use any set of strand ids corresponding to the correct numbers of strand types. Let us expand on this idea to break down our base q_{exp} in terms of $F(c)$ and combinations of strand ids. We introduce a new set E', which will be the set of all valid combinations of strand id's for a complex of type j (j is a labeled complex). It is helpful to note the size of this set, which will be the number of ways for each strand species, to select the appropriate number of strand ids of that species for a single complex of type j. This is, in two different forms: $|E'| = \prod_{i \in \Psi^0} \binom{m_i^0}{A_{ij}} = \prod_{i \in \Psi^0} \binom{(Am)_i}{(Am')_i}$

For example, if our complex of type j is two A strands, and we have 4 A strands with ids $\{1, 2, 3, 4\}$, then $E' = \{\{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 3\}, \{2, 4\}, \{3, 4\}\}$ are the valid sets of strand ids. Finally, let $U(E)$ be the set of all microstates for a single complex of type j which use only the set of strand id's E. We can then rewrite q_{exp} in terms of these, noting again that we are not overcounting any states in S' nor leaving any out (for the same reasons as before, if either were the case it would indicate a defect in the original S):

$$\begin{aligned} q_{exp}(m) &= \sum_{E \in E'} \sum_{c \in U(E)} \sum_{s \in F(c)} e^{-\Delta G^*_{box}(s)/RT} \\ &= \sum_{E \in E'} \sum_{c \in U(E)} e^{-\Delta G^*(c)/RT} q_{kin}(m') \end{aligned}$$

Note that m' is defined as before: $m' : m'_j = m_j - 1, m'_i = m_i \, \forall i \neq j$. Finally, note that the sum over $U(E)$ is just the sum over all microstates of a single complex, and is thus equivalent to our base case!

$$q_{exp}(m) = \sum_{E \in E'} q_{kin}(m'') * q_{kin}(m')$$

where m' is as before, and $m'' : m''_j = 1, m''_i = 0 \, \forall i \neq j$. Now we see that the q_{kin} in the sum don't depend on E', and thus we have:

$$q_{exp}(m) = |E'| * q_{kin}(m'') * q_{kin}(m')$$

where $|E'|$ is the number of ways to choose strand ids for one complex out of the strand ids for the whole system: $|E'| = \prod_{i\in\Psi^0} \binom{(Am)_i}{(Am'')_i} = \prod_{i\in\Psi^0} \frac{(Am)_i!}{(Am'')_i!*(Am')_i!}$. Putting it all together:

$$\begin{aligned}
q_{exp}(m) &= \left(\prod_{i\in\Psi^0} \frac{(Am)_i!}{(Am'')_i! * (Am')_i!}\right) * \left(\prod_{i\in\Psi^0} (Am'')_i!\right) \frac{M_s}{M_s^{L(m'')}} Q_j * q_{kin}(m') \\
&= \left(\prod_{i\in\Psi^0} \frac{(Am)_i!}{(Am')_i!}\right) * \frac{M_s}{M_s^{L(m'')}} Q_j * q_{kin}(m') \\
&= \left(\prod_{i\in\Psi^0} \frac{(Am)_i!}{(Am')_i!}\right) * \frac{M_s}{M_s^{L(m'')}} Q_j * (\prod_{i\in\Psi^0} (Am')_i!) \frac{M_s^{m_j-1}}{M_s^{L(m')}} \frac{Q_j^{m_j-1}}{(m_j-1)!} \\
&= \left(\prod_{i\in\Psi^0} (Am)_i!\right) * \frac{M_s}{M_s^{L(m'')}} Q_j * \frac{M_s^{m_j-1}}{M_s^{L(m')}} \frac{Q_j^{m_j-1}}{(m_j-1)!} \\
&= \left(\prod_{i\in\Psi^0} (Am)_i!\right) * \frac{M_s^{m_j}}{M_s^{L(m)}} \frac{Q_j^{m_j}}{(m_j-1)!} \\
q_{kin}(m) &= \frac{1}{m_j} * q_{exp}(m) \\
q_{kin}(m) &= \left(\prod_{i\in\Psi^0} (Am)_i!\right) * \frac{M_s^{m_j}}{M_s^{L(m)}} \frac{Q_j^{m_j}}{m_j!}
\end{aligned}$$

And the inductive step holds. Note again that $L(m) = \sum_{i\in\Psi^0}(Am)_i$, which is the total number of strands in the system with population vector m.

C.5.4 Full System

We now have the components necessary to give a formula for $q_{kin}(m)$ for a general m, which we can prove via induction. We wish to show for a general m:

$$q_{kin}(m) = \left(\prod_{i\in\Psi^0} (Am)_i!\right) * \frac{1}{M_s^{L(m)}} * \prod_{j\in\Psi} \frac{M_s^{m_j} Q_j^{m_j}}{m_j!}$$

We proceed via induction on the number of non-zero entries in m, and have already shown the base case for a single non-zero entry in m, above. In order to break down the

general case into smaller components so that we can apply the inductive hypothesis, we wish to follow a process similar to the previous marking, in which we separate out a particular component of the system microstate. This component t is the system microstate corresponding to a system with population vector m', which has $m'_j = m_j > 0, m'_i = 0 \forall i \neq j$, using a particular set of strand ids. Thus we can define a 'marked' set of states S' as before, using $T(m')$ as the set of all system microstates corresponding to a system with population vector m' and an arbitrary choice of strand ids:

$$S' = \{(s,t) | s \in S, t \in T(m'), s.t. \forall c \in t, \text{if } t \subseteq s \text{ then } c \in s\}$$

Note that we must pick a j such that $m_j > 0$, and that this is an arbitrary choice which remains for the rest of this step (j never changes, we just use it to pick a complex type which appears in m.) We can now group the system microstates as follows:

$$F(t) = \{s \in S | (s,t) \in S'\}$$

Again, if we sum over the states in each grouping, we get:

$$\begin{aligned}
\sum_{s \in F(t)} e^{-\Delta G^*_{box}(s)/RT} &= \sum_{s \in F(t)} e^{-\Delta G^*_{box}(t)/RT} * e^{-\Delta G^*_{box}(s \backslash t)/RT} \\
&= e^{-\Delta G^*_{box}(t)/RT} \sum_{s \in F(t)} e^{-\Delta G^*_{box}(s \backslash t)/RT} \\
&= e^{-\Delta G^*_{box}(t)/RT} q_{kin}(m'') \\
&= e^{-\Delta G^*_{box}(t)/RT} q_{kin}(m - m')
\end{aligned}$$

where $m'' : m''_j = 0, m''_i = m_i \, \forall i \neq j$. We proceed exactly as before, letting E' be the set of all combinations of strand ids which could be used in the complexes of type j, and $U(E)$ the set of all microstates $T(m')$ which use the set of strand ids E. Note that the following decomposition doesn't involve a q_{exp}, because in this marking we didn't expand our system at all (each $s \in S$ corresponds to exactly one $(s,t) \in S'$). It must also be exhaustive and non-redudant for the same reasons as before, if it were otherwise it would imply either a

missing system microstate in S, or the duplication of some microstate in S. We then get:

$$\begin{aligned} q_{kin}(m) &= \sum_{E \in E'} \sum_{t \in U(E)} \sum_{s \in F(t)} e^{-\Delta G^*_{box}(s)/RT} \\ &= q_{kin}(m'') * \sum_{E \in E'} \sum_{t \in U(E)} e^{-\Delta G^*_{box}(t)/RT} \\ &= q_{kin}(m'') * q_{kin}(m') * \sum_{E \in E'} 1 \\ &= q_{kin}(m'') * q_{kin}(m') * |E'| \end{aligned}$$

where m' is as stated above: $m' : m'_j = m_j, m'_j = 0 \; \forall i \neq j$. Note that the second step is possible because $q_{kin}(m'')$ is dependent on the choice of j, but not E or t, and similarily for the third step. These terms are again the base case for the induction ($q_{kin}(m')$), the inductive hypothesis ($q_{kin}(m'')$) and the number of ways to pick the strand ids for each component. We continue, substituting for $|E'| = \prod_{i \in \Psi^0} \frac{(Am)_i!}{(Am'')_i! * (Am')_i!}$. Putting it all together:

$$\begin{aligned} q_{kin}(m) &= q_{kin}(m'') * q_{kin}(m') * \prod_{i \in \Psi^0} \frac{(Am)_i!}{(Am')_i! * (Am'')_i!} \\ &= \left(\prod_{i \in \Psi^0} \frac{(Am)_i!}{(Am')_i! * (Am'')_i!} \right) * \left(\prod_{i \in \Psi^0} (Am')_i! \right) * \frac{M_s^{m_j}}{M_s^{L(m')}} \frac{Q_j^{m_j}}{m_j!} * q_{kin}(m'') \\ &= \left(\prod_{i \in \Psi^0} \frac{(Am)_i!}{(Am'')_i!} \right) * \frac{M_s^{m_j}}{M_s^{L(m')}} \frac{Q_j^{m_j}}{m_j!} * q_{kin}(m'') \\ &= \left(\prod_{i \in \Psi^0} \frac{(Am)_i!}{(Am'')_i!} \right) * \frac{M_s^{m_j}}{M_s^{L(m')}} \frac{Q_j^{m_j}}{m_j!} * \left(\prod_{i \in \Psi^0} (Am'')_i! \right) * \frac{1}{M_s^{L(m'')}} * \prod_{k \in \Psi} \frac{M_s^{m''_k} Q_k^{m''_k}}{m''_k!} \\ &= \left(\prod_{i \in \Psi^0} (Am)_i! \right) * \frac{M_s^{m_j}}{M_s^{L(m')}} \frac{Q_j^{m_j}}{m_j!} * \frac{1}{M_s^{L(m'')}} * \prod_{k \in \Psi} \frac{M_s^{m''_k} Q_k^{m''_k}}{m''_k!} \\ &= \left(\prod_{i \in \Psi^0} (Am)_i! \right) * \frac{1}{M_s^{L(m)}} * \prod_{k \in \Psi} \frac{M_s^{m_k} Q_k^{m_k}}{m_k!} \end{aligned}$$

And the inductive step holds. We conclude that with $Q_{ref} = \prod_{i \in \Psi^0} (Am)_i! * \frac{1}{M_s^L}$ (which is independent of m for $m \in \Lambda$), we have:

$$Q_{kin} = \sum_{m \in \Lambda} q_{kin}(m) = Q_{ref} * \sum_{m \in \Lambda} \prod_{j \in \Psi} \frac{M_s^{m_j} Q_j^{m_j}}{m_j!} = Q_{ref} \sum_{m \in \Lambda} q(m) = Q_{box}$$

Unsurprisingly, [5] uses the same reference state as the our model, where every strand is a separate complex with no internal base pairings. ($\Delta G^*_{box}(no\,bp) = 0$), so it is unsurprising that the Q_{ref} is exactly the same for both, and thus we have shown agreement between Multistrand's microstate partition function formulation and the NUPACK partition function formulation.

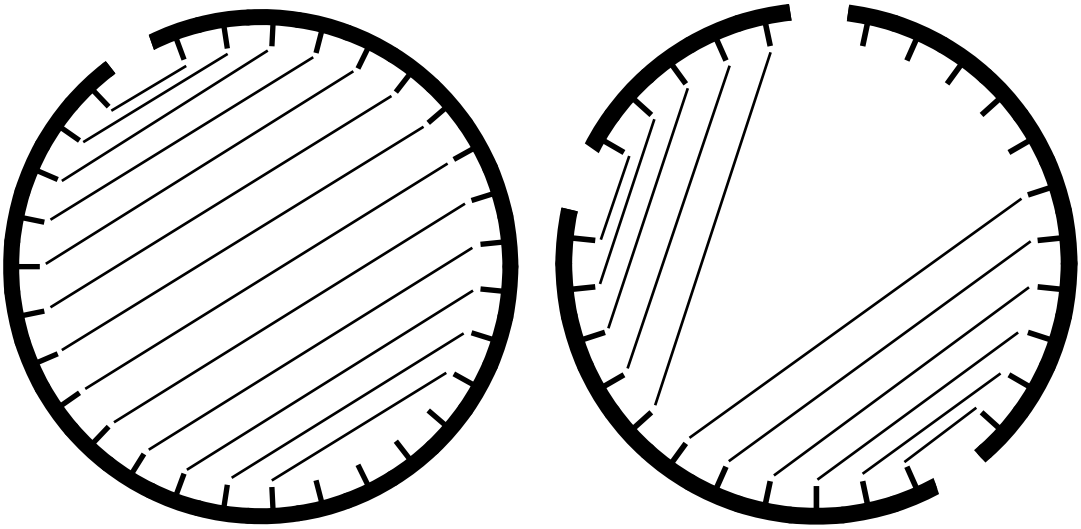
Appendix D

Strand Orderings for Pseudoknot-Free Representations

Given a complex microstate c, we can draw a *polymer graph* representation (also called the *circle-chord* representation) by laying out the strands on a circle, in the ordering $\pi^*(c)$, and representing the base pairs by chords connecting the appropriate locations (Figure D.1). In the case of a complex microstate with a single strand, we call the secondary structure *pseudoknotted* if there are crossing chords. However, the case where a complex microstate contains multiple strands requires a slightly more complex definition. We note that in this case, a strand ordering $\pi^*(c)$ corresponds to one particular way of arranging the strands on the circle; the circular permutations of $\pi^*(c)$ are the $(L-1)!$ permutations of the strands which are distinct when arranged on a circle, e.g. for three strands labeled A, B, C, there are only two distinct circular permutations: (A, B, C) and (A, C, B). With that in mind, we call an arbitrary secondary structure pseudoknotted if every circular permutation of the strand ordering has a polymer graph representation that contains a crossed chord.

While our simulator is not constrained to using a strand ordering $\pi^*(c)$ whose polymer graph representation does not contain a crossed chord, it is convenient for us to do so, as the output representation is easier to generate in these cases. The following heuristics allow us to maintain the property that our complex microstates always use a strand ordering $\pi^*(c)$ whose polymer graph does not contain a crossed chord:

The initial strand orderings we generate are based on a dot-paren structure, which naturally translate to a polymer graph with no crossed chords. The only time strand orderings can change is when performing a bimolecular move (either a break move or join move). For a break move, the resulting pair of complexes maintain the same orderings

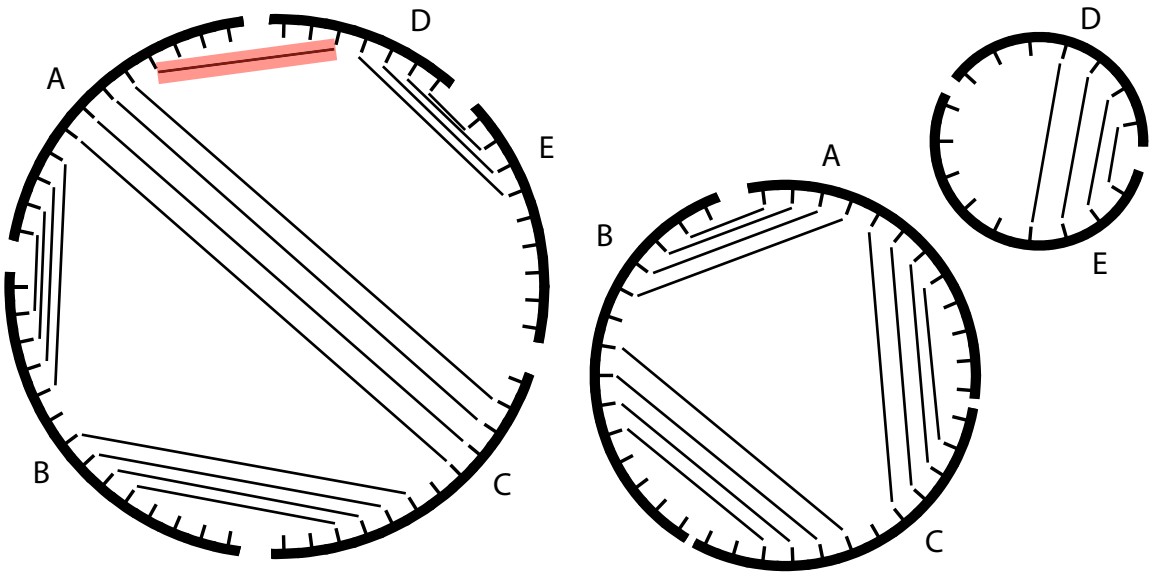


(a) A hairpin with long stem (b) Short three-way branch migration system

Figure D.1: Two different secondary structures using polymer graph representation. Strands are always arranged with $5' \to 3'$ orientation being clockwise around the circle.

the strands originally had in the original complex – e.g. if we had a complex of 5 strands (A through E) with ordering (A, D, E, C, B) which broke apart into two complexes with strands A, B, C and D, E, the resulting pair of orderings would be (A, C, B) and (D, E) (Figure D.2).

For a join move, we first note that a complex's open loops correspond to sequential pairs of strands in the circular strand ordering (this corresponds to the strands on either side of the "nick" in the dot-paren representation), so that a complex with ordering (A, B, D, C) has open loops corresponding to the pairs (A, B), (B, D), (D, C) and (C, A). When we perform a join move, we take each complex's strand ordering and find the cyclic permutation (these are the permutations that are identical when arranged on a circle, e.g. (A, B, C), (B, C, A) and (C, A, B) are cyclic permutations on the strand ordering (A, B, C)) which places the affected open loops at the edge of the permutation. For example, if we are joining (A, B, D, C) with (E, G, F) by the open loops around (B, D) and (E, G), we use the cyclic permutations (D, C, A, B) and (G, F, E), ending up with the strand ordering (D, C, A, B, G, F, E) for the resulting complex (Figure D.3). Why does this ordering have no crossing chords (assuming the starting ones did not)? We note that this join move was joining a base on either the $5'$ end of D or the $3'$ end of B, with a base on either the $5'$ end of G or the $3'$ end of E - in all four of these cases, the resulting chord cannot cross



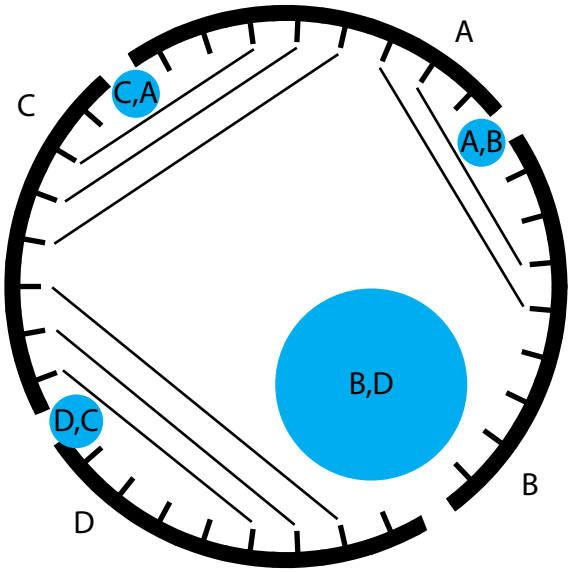
(a) (A,D,E,C,B) before break move

(b) (A,C,B) and (D,E) after break move

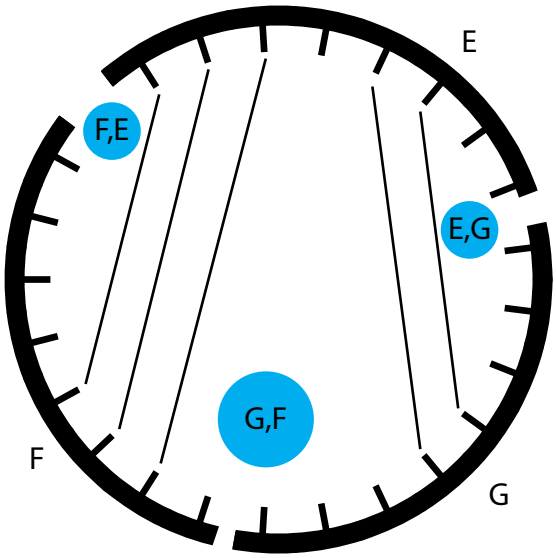
Figure D.2: Polymer graph representation before and after a break move (base pair highlighted in red). Note that the ordering is consistent, but we now have two separate complexes and thus two separate polymer graphs.

any existing chords if we use the given strand ordering. Note that in general, the bases available for a join move are not necessarily from the $5'$ and $3'$ edges near the nick but the same argument applies. For example, if we used a join from the (G, F) open loop (from the previous example), there is a single base on strand E that could be used to make the join, but since it's in the same open region as the $5'$ end of G and $3'$ end of F, it also could not create a crossed chord.

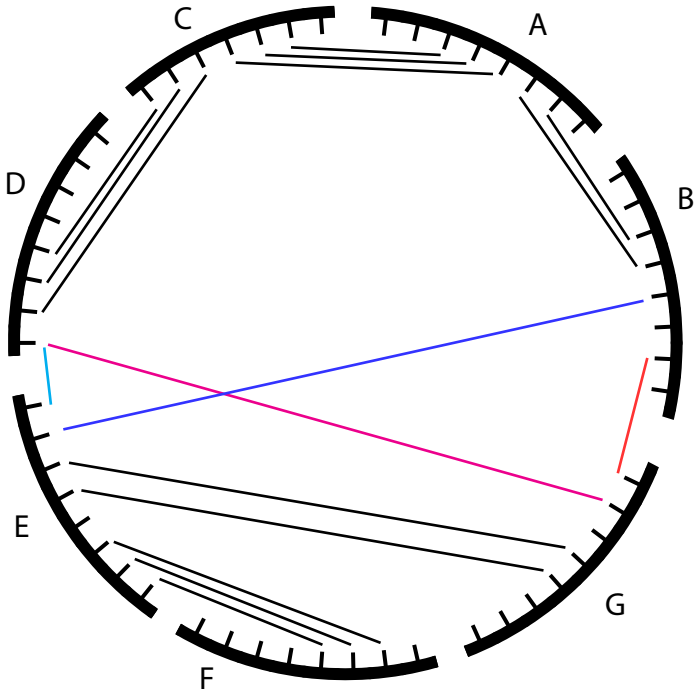
So we have shown that the strand ordering $\pi^*(c)$ maintained by our simulator for a complex microstate c has a polymer graph with no crossed chords. This leads naturally to the question of whether there is a different circular permutation of $\pi^*(c)$ which also has no crossed chords. The (surprising) answer is no - every other circular permutation has at least one crossed chord! This is stated in the following theorem:



(a) (A,B,D,C) before join move



(b) (E,G,F) before join move



(c) (D,C,A,B,G,F,E) after cyclic permutations and join

Figure D.3: Polymer graph representation before and after a join move. Open loop regions are noted with a cyan circle marker. Four of the (many) possible join moves between open loops (B, D) and (E, G) are shown in (c), using red, blue, magenta, and cyan.

D.1 Representation Theorem

For every non-pseudoknotted complex microstate c, there is exactly one circular permutation $\pi^*(c)$ *whose polymer graph has no crossed chords.*

While the above heuristic can be expanded on to prove this theorem via induction on the number of strands in a complex, a more thorough proof can be found in [5] and so we will not reproduce it here.

Bibliography

[1] Jonathan Bath, Simon J. Green, and Andrew J. Turberfield. A free-running DNA motor powered by a nicking enzyme. *Angewandte Chemie International Edition*, 44(28):4358–4361, 2005.

[2] Chris Berlind. California Institute of Technology, Computer Science Senior Thesis, 2011.

[3] Victor A. Bloomfield, Donald M. Crothers, and Ignacio Tinoco Jr. *Nucleic Acids: Structures, Properties, and Functions*. University Science Books, Sausalito, CA, 2000.

[4] Yi Chen, Mingsheng Wang, and Chengde Mao. An autonomous DNA nanomotor powered by a DNA enzyme. *Angewandte Chemie International Edition*, 43(27):3554–3557, 2004.

[5] Robert M. Dirks, Justin S. Bois, Joseph M. Schaeffer, Erik Winfree, and Niles A. Pierce. Thermodynamic analysis of interacting nucleic acid strands. *SIAM Review*, 49(1):65–88, 2007.

[6] Robert M. Dirks, Milo Lin, Erik Winfree, and Niles A. Pierce. Paradigms for computational nucleic acid design. *Nucleic Acids Research*, 32(4):1392–1403, 2004.

[7] Christoph Flamm, Walter Fontana, Ivo L. Hofacker, and Peter Schuster. RNA folding at elementary step resolution. *RNA*, 6:325–338, 2000.

[8] Daniel T. Gillespie. Exact stochastic simulation of coupled chemical reactions. *J Phys Chem*, 81(25):2340–2361, 1977.

[9] Ivo L. Hofacker. Vienna RNA secondary structure server. *Nucleic Acids Research*, 31(13):3429–3431, 2003.

[10] Kyozi Kawasaki. Diffusion constants near the critical point for time-dependent ising models. *Phys Rev*, 145:224–230, 1966.

[11] Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. Equation of state calculations by fast computing machines. *J Chem Phys*, 21:1087–1092, 1953.

[12] Steven R. Morgan and Paul G. Higgs. Barrier heights between ground states in a model of RNA secondary structure. *Journal of Physics A: Mathematical and General*, 31(14):3153, 1998.

[13] Jonathan A. Othmer. *Algorithms for mapping nucleic acid free energy landscapes*. PhD thesis, California Institute of Technology, 2009.

[14] Paul W. K Rothemund, Nick Papadakis, and Erik Winfree. Algorithmic self-assembly of DNA sierpinski triangles. *PLoS Biol*, 2(12):e424, 12 2004.

[15] John SantaLucia. A unified view of polymer, dumbbell, and oligonucleotide DNA nearest-neighbor thermodynamics. *Proceedings of the National Academy of Sciences*, 95(4):1460–1465, 1998.

[16] John SantaLucia, Hatim T. Allawi, and P. Ananda Seneviratne. Improved nearest-neighbor parameters for predicting DNA duplex stability. *Biochemistry*, 35(11):3555–3562, 1996.

[17] John SantaLucia and Donald Hicks. The thermodynamics of DNA structural motifs. *Annual Review of Biophysics and Biomolecular Structure*, 33(1):415–440, 2004.

[18] Georg Seelig, David Soloveichik, David Y. Zhang, and Erik Winfree. Enzyme-free nucleic acid logic circuits. *Science*, 314(5805):1585–1588, 2006.

[19] Friedrich C. Simmel. Processive motion of bipedal DNA walkers. *ChemPhysChem*, 10(15):2593–2597, 2009.

[20] Friedrich C. Simmel and Wendy U. Dittmer. DNA nanodevices. *Small*, 1(3):284–299, 2005.

[21] James G. Wetmur. Hybridization and renaturation kinetics of nucleic acids. *Annual Review of Biophysics and Bioengineering*, 5(1):337–361, 1976.

[22] Darren J. Wilkinson. *Stochastic Dynamical Systems*, pages 359–375. John Wiley & Sons, Ltd, 2011.

[23] Erik Winfree. Algorithmic self-assembly of DNA: Theoretical motivations and 2D assembly experiments. *Journal of Biomolecular Structure and Dynamics*, 11(2):263–270, 2000.

[24] Joseph N. Zadeh, Conrad D. Steenberg, Justin S. Bois, Brian R. Wolfe, Marshall B. Pierce, Asif R. Khan, Robert M. Dirks, and Niles A. Pierce. NUPACK: Analysis and design of nucleic acid systems. *Journal of Computational Chemistry*, 32(1):170–173, 2011.

[25] David Yu Zhang and Erik Winfree. Control of DNA strand displacement kinetics using toehold exchange. *Journal of the American Chemical Society*, 131(47):17303–17314, 2009.

[26] Wenbing Zhang and Shi-Jie Chen. RNA hairpin-folding kinetics. *Proceedings of the National Academy of Sciences*, 99(4):1931–1936, 2002.

[27] Michael Zuker. Mfold web server for nucleic acid folding and hybridization prediction. *Nucleic Acids Research*, 31(13):3406–3415, 2003.