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

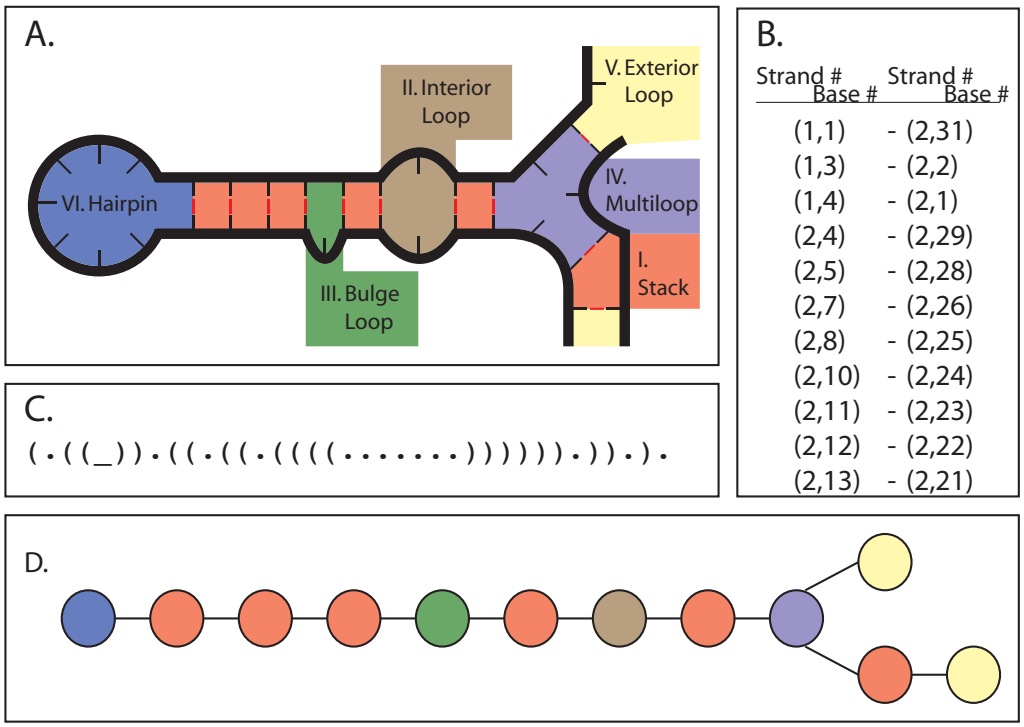

Figure 5.1: Example secondary structure, with different representations: (A) Original loop diagram representation. (B) Base pair list representation. Each base pairing is represented by the indices of the bases involved. (C) Dot-paren representation, also called the flat representation. Each base is 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.

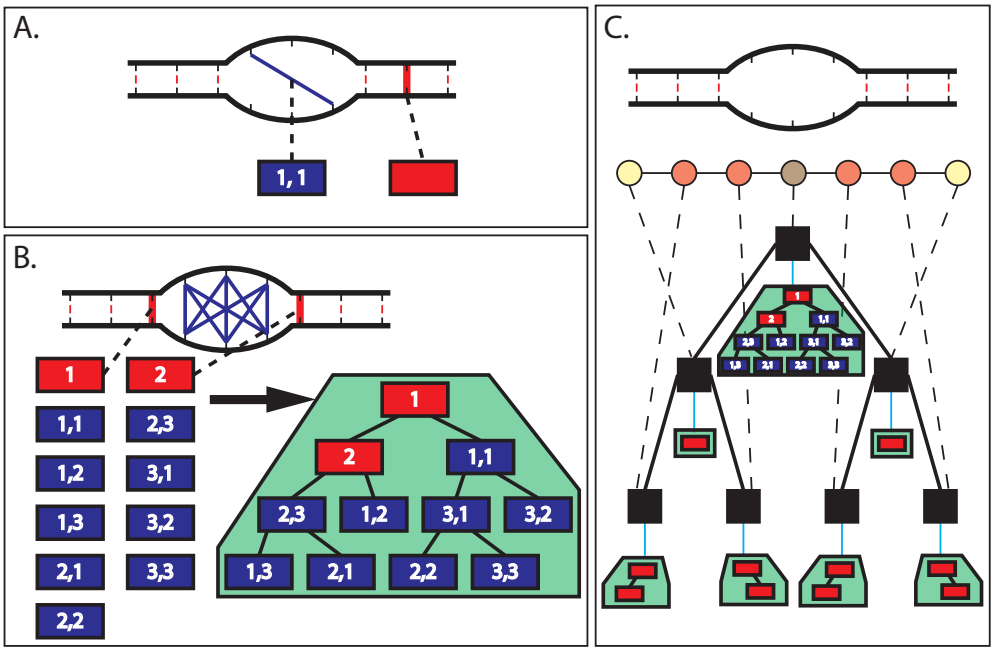

Figure 5.2: (A) Creation moves (blue line) and deletion moves (red highlight) are represented here by rectangles. Either type of move is associated with a particular loop, and has indices to designate which bases within the loop are affected. (B) All possible moves which affect the interior loop in the 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

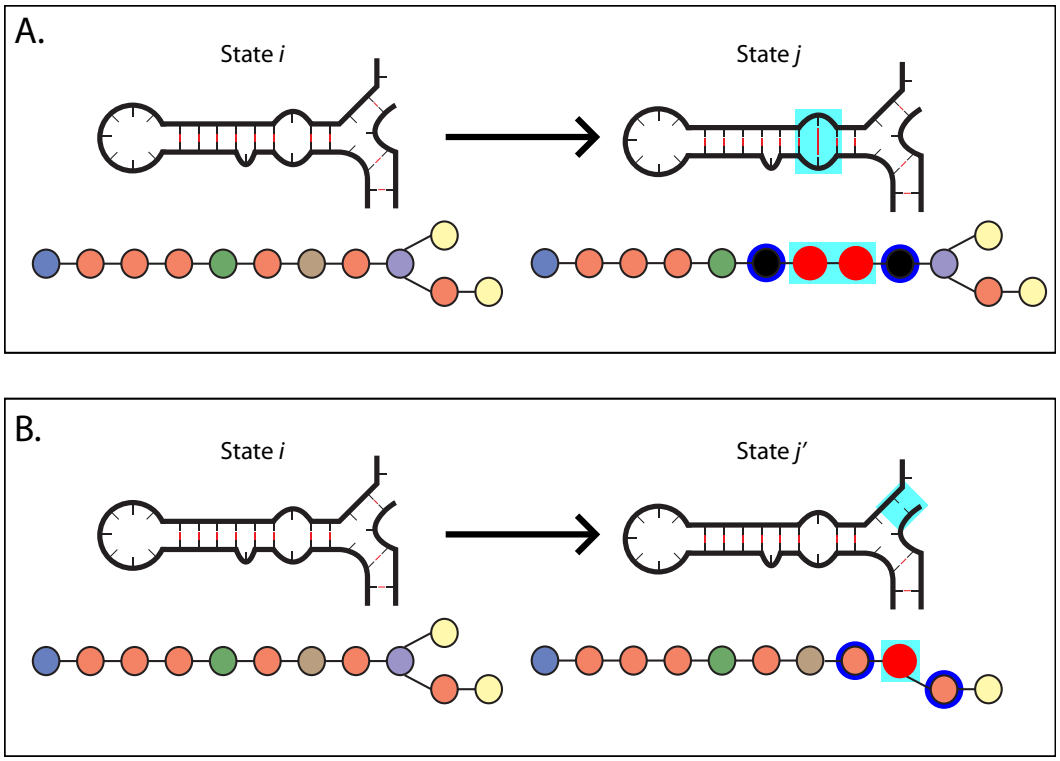

Figure 5.3: Moves of varying types which take current state i to state j. The changed region is highlighted in cyan. Loops that are in j but not i are highlighted in red (in the loop graph) and must be created and have their moves generated. Loops shown highlighted in blue have had an adjacent loop change,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

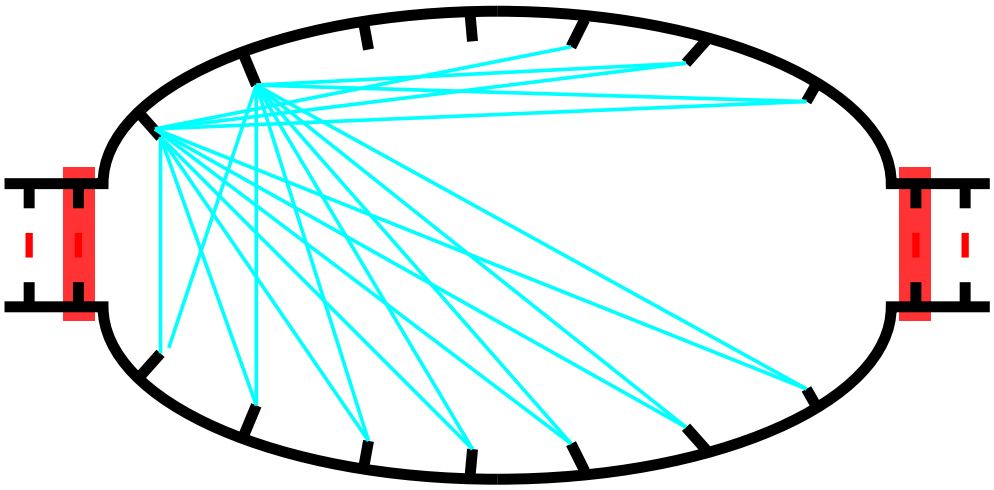

Figure 5.4: A interior loop, with all theoretically possible creation moves for the first two bases on the top strand shown as cyan lines, and all possible deletion moves shown as red boxes. Note that for each creation move shown here, we must check whether the bases could actually pair, and only then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

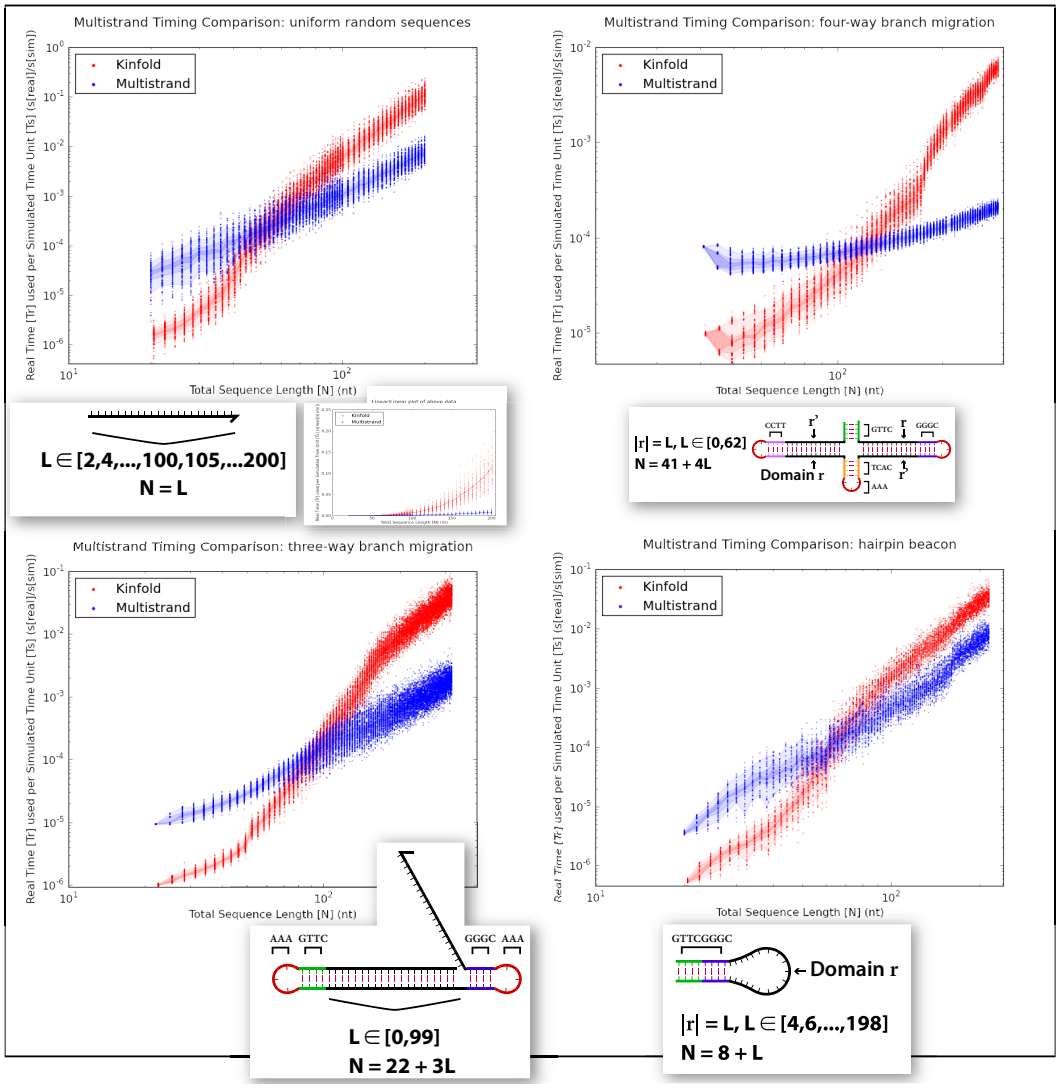

Figure 5.5: Comparison of real time used per simulated time unit between Multistrand and Kinfold 1.0, for four different single stranded systems with varying total length. All plots are log/log except for the inset, which is a linear/linear plot of the same data is in the uniform random sequence plot. 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.