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.