1 The problem

A configuration is made up of a collection of N associated simplices of ranks 0-4, each associated with a unique move that converts that simplex of rank d into another simplex of rank $4 - d$.

Each move is associated with some change in the action $\Delta\mathcal{S}_i$ and an associated Metropolis probability $P(i)$. If that move is forbidden by the local geometry, then $P(i) = 0$. Per Jack and Scott the current accept rate is $\mathcal{O}(10^{-4})$ for fine lattices, which have $N \simeq 10^5$. (Note that N includes simplices of all ranks.)

2 The algorithm

To design an algorithm that generates the same results as the current one, we need to compute the following as quickly as possible:

- Which move i will be the first one to be accepted by Metropolis
- How many tries it took to accept that move (since per the Metropolis algorithm simulation time is measured in attempted moves, not accepted ones)

Rather than trying moves until one is accepted, I propose an algorithm that knows the probability of all possible moves ahead of time, and then uses only *one* random number to choose which is accepted. Call the probability that any given move i will be the one eventually accepted $\tilde{P}(i)$. Then

$$\tilde{P}(i) = \frac{P(i)}{\sum_j P(j)},$$

i.e. the probability of eventually accepting any move is the fraction of the total Metropolis probability that it has.

For the second point, the number of Metropolis suggestions that are rejected before one is eventually accepted is independent of which one is eventually chosen. Thus it suffices to determine the probability that any given Metropolis suggestion will result in a rejection.

The probability of accepting any move i on any particular trial is equal to the probability of choosing it times the probability of accepting it, *i.e.*

$$P_a(i) = \frac{P(i)}{N}$$

Since these probabilities are mutually exclusive, we can just add them, to determine that the probability of accepting the move that we choose is

$$P_a = \frac{\sum_i P(i)}{N}$$

i.e. the probability that the move that is chosen at random will be accepted is just the average accept probability of all the moves.

The probability of accepting any move after rejecting n previous trials is thus

$$P(n) = P_a(1 - P_a)^n$$

and the number of previously rejected trials can be determined from a single random number r from (0,1) as

$$n_{\text{reject}} = \text{floor}(\log_{1-\text{P}_\text{a}}(\text{r})).$$

(I tested this with a quick Monte Carlo and it checks out.)

3 The implementation and speed gains

The key is that making any move i:

- Creates and/or destroys certain simplices, changing N and adding/removing certain moves from the pool of possibilities
- Alters the probability $P(j)$ only of 10-100 moves in the local neighborhood of i (if I'm reading Scott's thesis correctly, moves j which share triangles with those simplices altered by move i – is this correct?)

I propose to create an ordered binary tree (in the traditional computer science sense) of all possible moves. The index of each element on the binary tree is a simplex index i; these do not need to be continuous. Each node on the tree contains:

- Information about the move that it corresponds to (*i.e.* the simplex number and rank)
- The probability $P(i)$ of accepting that move if chosen, based on $\Delta\mathcal{S}(i)$ (zero if that move is impossible)
- Pointers to the left and right child nodes, and a pointer upward to the parent node
- The total probability of this node and all of its children

Note that the relation between nodes on the binary tree has nothing at all to do with the geometry of the lattice; it is only a tool for searching and indexing.

To determine which move j will be the one that the Metropolis algorithm eventually accepts:

1. Generate a random number r between 0 and the total probability $\sum P(i)$ in the whole tree (which you can read off of the root node).
2. Start at the root of the tree and traverse it, looking for the node j that will be accepted.

3. At each node, we have three options: it's either this node, or it lives somewhere along the left or the right branch.

4. Define P_{here} as the probability of the current node, P_{left} as the *total* probability along the left branch, and P_{right} as the total probability along the right branch.

5. Traverse the tree according to the following:

    - If r is less than P_{left}, then go left
    - If r is greater than P_{left} but less than $P_{left} + P_{here}$, then we are at the node j: accept it
    - If r is greater than $P_{left} + P_{here}$, then $r \leftarrow r - (P_{left} + P_{here})$ and go right

This allows us to determine the move j that will eventually be chosen in $\mathcal{O}(log(N))$ time. (I can't think of a way to do it in $\mathcal{O}(1)$ time; this binary tree business seems harder than it necessarily needs to be, but I can't think of a better alternative.)

Then, once we've chosen the move j:

1. Actually make that move (change the geometry)

2. If any simplices were destroyed, delete them from the tree:

    - Traverse the tree looking for the node to remove
    - We've then got to update the probabilities: traverse the tree back upward, reducing the "total probability here and below" field by the appropriate amount
    - This requires $\mathcal{O}(\log N)$ pointer and arithmetic operations

3. If any simplices were created, add them to the tree:

    - Traverse the tree looking for the spot to insert the new node
    - Calculate the probability associated with the new node by examining the change in action if the move corresponding to it is made
    - Traverse the tree back upward, increasing the "total probability here and below" field by the appropriate amount

4. The Metropolis probabilities $P(i)$ of the simplices which share a triangle with those simplices that were altered by the move that was just made:

    - Compute them (using the preexisting method)
    - Traverse the tree looking for them and update their probability
    - Traverse the tree back upward and update the "total probability here and below" field

I imagine that all of the $\mathcal{O}(\log N)$ operations will be extremely cheap, since they are just pointer traversal and arithmetic. The expense will come in the computation of $P(i)$ for the new simplices and those whose local neighborhood was altered by the chosen move. But this shouldn't be more than a few dozen per accepted move – far fewer than the $\mathcal{O}(10^4)$ that are required on fine lattices using standard Metropolis.

4 Global plus local action

In DT simulations there is also a global piece of the action S_{global} that affects all Pachner moves of the same order equally. We seek a way to incorporate this into the algorithm while avoiding any step whose cost is proportional to the number of possible moves, *i.e.* we need to avoid traversing the whole tree.

4.1 Adding the global action

Since $\mathcal{S}_{\text{global}}$ will change frequently, we can only store $\mathcal{S}_{\text{local}}$ in the tree.

I propose that we create five "subtrees", one for each Pachner move type, and store only the probability based on $\mathcal{S}_{\text{local}}$, called P_{local} in those trees. The head of each tree knows the total probability P_{local} of the moves beneath it, as always.

Then – based on the values of $\mathcal{S}_{\text{global}}$ and the total probabilities in each Pachner-type subtree, we first make a five-way choice of which move type we will make. If changes in $\mathcal{S}_{\text{global}}$ do not affect the relative values of P_{local} within a subtree, then nothing needs to change: we can apply the effect of $\mathcal{S}_{\text{global}}$ as an overall weight factor for each subtree, generate one random number to choose which subtree to look at, and descend it searching for the right move as before.

4.2 Factorizing the probability

However, this is not the case without some modifications. Recall the formula for $P_{A\to B}$, the probability that a move from A to B will be accepted:

$$P_{A\to B} = \begin{cases} 1, & \text{if } \mathcal{S}_B <= \mathcal{S}_A \\ e^{S_A - S_B}, & \text{if } \mathcal{S}_B > \mathcal{S}_A \end{cases}$$

The action here, in the Metropolis prescription, is the entire action $\mathcal{S}_{\text{local}} + \mathcal{S}_{\text{global}}$. Since it is piecewise defined, we cannot easily separate global and local effects on the probability and write $P_{A\to B} = (P_{A\to B,\,\text{global}})(P_{A\to B,\,\text{local}})$. For instance, if a certain move greatly reduces the local action, the global action may not affect its probability at all, since it will be equal to 1 by the first case above.

We need an alternate definition for $P_{A\to B}$ that generates the same canonical ensemble as Metropolis but which meets the criterion $P_{A\to B} = (P_{A\to B,\,\text{global}})(P_{A\to B,\,\text{local}})$. The only thing required for detailed balance is that

$$\frac{P_{A\to B}}{P_{B\to A}} = e^{\mathcal{S}_A - \mathcal{S}_B}.$$

4.3 The square root approach

I propose a different approach here that allows us to factorize the local and global effects. The Metropolis algorithm's accept-reject step requires that these probabilities be no greater than unity, but since this

algorithm is "rejection-free", we have no such constraint. These are thus no longer *probabilities* (since they can be greater than one). To allow us to use the same symbol P, and to distinguish them from Monte Carlo configuration weights (which I introduce later), I'll refer to them as the *ponderance* P – which is a generalized version of the Metropolis accept probability.

Then we can also maintain detailed balance with the definition

$$P_{A\to B} = \sqrt{e^{\mathcal{S}_A - \mathcal{S}_B}},$$

which is not piecewise defined and thus factorizes neatly as we require.

$$P_{A\to B,\text{local}} = \sqrt{e^{\mathcal{S}_{A,\text{local}} - \mathcal{S}_{B,\text{local}}}} \tag{1}$$

$$P_{A\to B,\text{global}} = \sqrt{e^{\mathcal{S}_{A,\text{global}} - \mathcal{S}_{B,\text{global}}}} \tag{2}$$

(Of course, in implementing this it makes sense to code the square root as a factor of 1/2 in the exponent.)

Then we store P_{local} in the five subtrees, and first use P_{global} along with the total ponderance contained in each subtree to choose which order Pachner move we want to make before descending the chosen subtree and choosing one particular move.

4.4 Handling the reject count

However, this creates a problem with calculating n_{reject}. Recall that this is done using the relation

$$n_{\text{reject}} = \text{floor}(\log_{1-\text{P}_\text{a}}(\text{r})).$$

where P_a is the average Metropolis accept probability (or ponderance) of all the moves. If we use the ponderance as calculated above, this leads to a negative logarithm when P_a is greater than one. This should not be a surprise; if we shift from thinking about accept probabilities to thinking about more abstract ponderances, it might be expected that things connected to transition probabilities might break!

Nonetheless, we need a way to account for this effect: that some configurations are likely to have a longer dwell time in the Markov chain than others.

4.5 The weighted ensemble approach

One might also imagine using this algorithm to generate a weighted ensemble rather than to reproduce exactly the Metropolis Markov chain. Thus, instead of producing something that looks like `AABCCCDDDDDAB`, it would produce something that looks like `A (2); B (1); C (3); D (5); A(1); B(1)`. Then an analysis code would simply take a weighted average over configurations, where each configuration in the sequence is weighted with its average dwell time in the MCMC sequence.

To do this, the weights w of each configuration should be proportional to the average number of Metropolis repeats that it would have in a traditional MCMC approach. I believe that this approach will have

statistical and performance gains. It will eliminate a further source of stochasticity (the random nature of the dwell time of each move), although this will not be meaningful in DT simulations where analysis code typically only examines one configuration out of many because of long autocorrelation times. It will also require the generation of only one random number per step.

This calculation is much simpler. The average dwell time d_{avg} of a configuration in the sequence, which is proportional to its weight in the ensemble w, is just equal to the inverse of the average ponderance of all moves leading away from it:

$$w \propto d_{\text{avg}} = \frac{1}{P_a}.$$

This approach has no such issue: if the average ponderance is greater than one, then the weight in the final ensemble will be less than one. In the traditional Metropolis sense, this means that the probability of accepting the next suggested move is greater than one, and the dwell time of that configuration in the sequence is less than one – both things that are absurd, in an "unweighted" Metropolis sequence. But this poses no problem in the weighted ensemble.