Research Proposal: A data structure to support the simulation of random events

Hubert Chan

December 6, 2005

1 Introduction

Consider the problem of selecting an object at random from a set of objects where each object has a different probability of being selected. In this proposal, I describe a specific variation of this problem, inspired by a method of simulating the reaction-diffusion equation, and I propose to investigate creating a data structure for solving the problem efficiently.

The proposed data structure will store events along with their rates, and will support the following operations: *selecting* an event at random with probability proportional to its rate, *updating* an event's rate, *splitting* one event into multiple events, and *merging* multiple events into one event. Alternatively, instead of supporting splitting and merging, we may support *inserting* and *deleting* events.

Our reason for supporting these operations is motivated by a method of simulating the reaction-diffusion equation. This method will be described below. The reaction-diffusion equation describes a model for the behaviour of chemicals or other types of contaminants in a fluid. The equation models movement of the contaminant due to diffusion or fluid flow, as well as increases or decreases in concentration due to, for example, chemical reactions, or natural births and deaths in the case of a biological contaminant.

The problem of selecting an object at random is also related to a problem known as the selectable partial sums problem, as well as optimal search trees and coding systems, which we will discuss later in this proposal. These problems will serve as starting points for our research.

The organization of the rest of the proposal is as follows. First, we will describe in greater detail the random object selection problem and the method of simulating the reaction-diffusion equation. We will then give a more detailed description of the problem that we are examining. We will then explain the connection between the selectable partial sums problem and the random object selection problem and give a survey of previous work on the selectable partial sums problem. Next, we will explain the connection

between optimal search trees and coding systems and the random object selection problem, and give a survey of previous work on optimal search trees and coding systems. Finally, we will describe our research goals and approaches.

2 Selecting an object at random

Consider a set of n objects and a set of probabilities $p_1, \ldots, p_n$, with $\sum_{i=1}^{n} p_i = 1$, that represents the relative probability of each object being selected. We wish to select an object at random such that the probability of selecting the ith object is equal to p_i. Equivalently, we have a set of weights $w_1, \ldots, w_n$, and the total weight $w = \sum_{i=1}^{n} w_i$, and we wish to select an object at random such that the ith object is selected with probability $\frac{w_i}{w}$. We call this the *random object selection problem*.

This problem can be solved by generating a uniform-$[0, 1)$ random variable x, and determining k for which $\sum_{i=1}^{k-1} p_i \leq x < \sum_{i=1}^{k} p_i$. When the sets of objects and probabilities are fixed, finding k can be done by precomputing the partial sums $\sigma_k = \sum_{i=1}^{k} p_i$, and performing a binary search. However, if the set of objects or probabilities is subject to change, we must use a different strategy.

We also note that with simple binary search, all objects are treated equally, no matter what their probability. However, if we can arrange the data such that objects that have higher probability can be found faster, at the expense of objects that have lower probability, we may be able to improve the expected performance.

3 Simulating the reaction-diffusion equation

The random object selection problem is a common subtask in various randomized simulations. One such simulation is a method of simulating the reaction-diffusion equation, which describes how a chemical, or other contaminant, behaves in a fluid. The reaction-diffusion equation models the flow of the contaminant due to diffusion or fluid flow, as well as increases or decreases in concentration. These increases and decreases can be due to various causes. For example, the increases and decreases can be caused by a source or sink that produces or reduces the contaminant. In the case of a chemical, they can be caused by chemical reactions. In the case of a biological contaminant, they can be caused by natural births and deaths.

To simulate the behaviour of the contaminant, we first divide the fluid into cells using a grid, and by representing the concentration of the contaminant in each cell as a discrete number of particles. Thus movement of the contaminant through the fluid is modeled by movement of particles between neighbouring cells, and increases or decreases in concentration are modeled by particle "births" and "deaths". Mathematical treatments of this method of simulation and variants, showing convergence results as the cell size approaches zero, are given by Arnold and Theodosopulu [AT80], Blount

[Blo96], and Kouritzin and Long [KL02]. These results are beyond the scope of this proposal.

The *state* of the simulation refers to the collection of all particle counts in all cells. An *event* refers to the movement of a particle from one cell to an adjacent cell, the birth of a particle in a cell, or the death of a particle. The events are modeled as a Markov process: the occurrence of an event depends only on the current state, and not on any previous events. The time until the occurrence of a single event e, whether it be a particle movement, birth, or death, is an exponential random variable with rate λ_e. Intuitively, this means that the expected value of the time until the occurrence of the event e is $\frac{1}{\lambda_e}$. The rate λ_e is a function of the particle counts: the birth and death rates are functions of the particle count of the cell in which the birth or death is to occur, and the rate for a particle movement is a function of the difference in concentration between the source and destination cell. The rates could also be functions of the spatial location of the cells.

The straightforward method of simulating the set of events is to, for each event e, generate a λ_e-exponential random variable to determine the time of the next occurrence of e. We then determine which event will occur next by finding the event whose random variable has the smallest value. We then simulate the event, which causes the particle counts to change in some cells, and hence causes some of the rates to change. Thus we may need to regenerate some of our λ_e-exponential random variables before we determine the time for the next event to occur.

However, there is a faster method of determining the time of occurrence of the next event. Due to properties of the exponential distribution, the time until the occurrence of any event is also an exponential random variable, with rate $\lambda_E = \sum_{e \in E} \lambda_e$, where E is the set of all events. As well, given that some event has occurred, the probability that the event that occurred was event e is $p_e = \frac{\lambda_e}{\lambda_E}$. Thus to determine the time of the next event, we only need to generate a λ_E-exponential random variable, and to determine which event had occurred, we select an event randomly according to their relative probabilities p_e by solving the random object selection problem.

In order to obtain an efficient simulation, therefore, we need a data structure that allows us to store the rates in order to efficiently select an object at random, and to update the rates.

In our simulation, some areas may be of greater interest than others. For example, if we are modeling the behaviour of pollution in a lake or other body of water, we may be more interested in the areas close to shore, near populated areas. In order to conserve memory, then, our grid may not be uniform; areas of greater interest may have a finer grid, while areas of lesser interest may have a coarser grid. This allows us to conserve memory without sacrificing the accuracy of the simulation in the areas of interest. Our areas of interest may also change dynamically. For example, we may be more interested in areas with higher concentration of chemicals, and less interested in areas with little or no concentration. Since the concentration in the fluid changes over time, we may wish to be able to dynamically adjust our grid. Thus we wish to be able to split cells,

if their concentration is high, or merge neighbouring cells, if their concentration is low. Thus our data structure needs to be able to support splitting and merging of cells, in addition to selecting and updating. When splitting, we may either split along only one dimension at a time, or we may split along all dimensions at the same time. We will refer to the cells that we obtain from splitting as a *mergable set*. When merging, we only merge cells if they form a mergable set.

4 Problem description

We now describe in detail the abstract data type that we wish to construct. Given a set of events $e_1, \ldots, e_n$ with their rates $\lambda_1, \ldots, \lambda_n$, we wish to be able to randomly select an event proportionally according to the set of rates, such that the event e_k is selected with probability $\frac{\lambda_k}{\lambda_E}$ where $\lambda_E = \sum_{i=1}^{n} \lambda_i$. We also wish to modify the set of rates and events, and we wish to perform our selections and modifications efficiently by minimizing the expected time required for each operation.

In particular, the goal is to design a data structure for storing rates that efficiently supports the following operations:

1. *select:* select an event at random such each event is selected with probability $\frac{\lambda_k}{\lambda_E}$;
2. *update:* set the rate of an event to a new value;
3. *split:* split one event into a number of new events, distributing the rate of the old event evenly among the new events; and
4. *merge:* merge a number of events into one new event, where the rate of the new event is equal to the sum of the old events.

If we cannot obtain a data structure that supports splitting and merging natively, these operations can also be simulated using two other operations:

5. *insert:* add a new event and its associated rate; and
6. *delete:* delete an event and its associated rate.

We can simulate splitting and merging by deleting the old event(s), and inserting the new event(s), hence supporting insertions and deletions may be sufficient. However, it is most likely that such an arrangement would give worse performance than a data structure that supports splitting and merging natively.

5 The selectable partial sums problem

Let us consider again the basic random object selection problem, a simplified version of the problem that we are trying to solve in which we only need to support the select

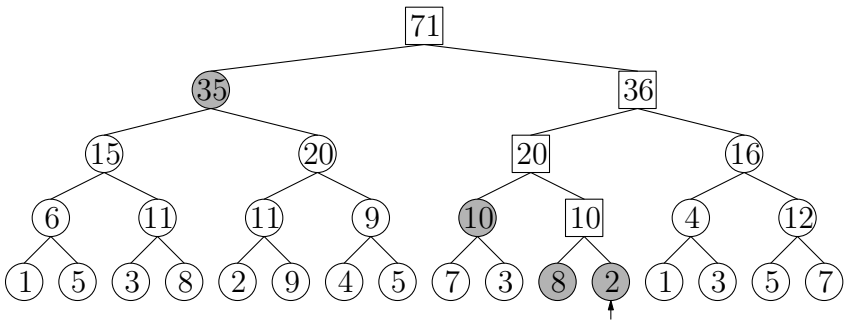

Figure 1: Answering a sum query. The arrow indicates the sum that we want to determine. The square nodes are the ancestors of the leaf k. The shaded nodes are the nodes whose weights we add together to determine the sum.

and update operations. This problem is related to the selectable partial sums problem, which is defined as follows. We are given a sequence of n non-negative integers $a_1, \ldots, a_n$, which we will call the *keys*, we must support the following operations:

1. *update:* change the value of a key;
2. *sum:* given k with $1 \leq k \leq n$, calculate the partial sum $\sigma_k = \sum_{i=1}^{k} a_i$; and
3. *select:* given a target value t, find the value k for which $\sigma_{k-1} \leq t < \sigma_k$.

Thus, for the random object selection problem, to pick a random object, we let $a_i = p_i$, and let our uniform-$[0, 1)$ random variable x be our target value t.

The simplest sub-linear data structure for supporting selectable partial sums is a balanced binary tree, in which each operation takes $O(\log n)$ time, and most of the previous work in this area is based on this idea. We store the numbers $a_1, \ldots, a_n$ in the leaves of the tree, and each interior node stores the sum of the leaves in its subtree. We will refer to this sum as the node's *weight*. Thus an update can be performed by updating the corresponding leaf and the weight of all its ancestors. To find a sum σ_k, we find all ancestors of the leaf k, and sum together the weights of all the left children of the ancestors, provided that the left child is not also an ancestor, plus the weight of the leaf itself. (See figure 1.) To perform a selection, we proceed recursively starting from the root: if our target t is less than the weight of the left child, we recurse on the left subtree; if t is greater than the weight of the left child w, we set $t \leftarrow t - w$, and recurse on the right subtree.

Pătraşcu and Demaine [PD04] give an optimal data structure for the partial sums with select problem: on a b-bit machine, when updates are limited to δ-bit additive changes, both updates and queries can be done in $\Theta(1 + \lg n / \lg(b/\delta))$ time. They also prove matching lower bounds. The bounds that they give are amortized.

Raman, Raman, and Rao [RRR01] give a succinct data structure, which uses only $kn + o(kn)$ space, and takes $O(\lg n / \lg \lg n)$ worst-case time for each operation. A *suc-*

cinct data structure is one that has size close to the information-theoretic lower bound. They also consider a succinct data structure that has different time bounds for updates and queries in the case where all the integers are of size 1 bit. For any parameter u, $\lg n / \lg \lg n \leq u \leq n^{\epsilon}$, updates can be performed in $O(u)$ amortized time, and a sum can be calculated in $O(\log_u n)$ worst-case time. Hon, Sadakane, and Sung [HSS03] improve on this result by allowing the integers to have size up to $O(\lg \lg n)$ bits, and obtaining the same time bounds with the improvement that the update time is now worst-case instead of amortized. Hon et al. also extend the selectable partial sums problem to support the insert and delete operations, and offer succinct data structures when the integers are of size $O(1)$ bits or, if we are given a large precomputed table, up to $O(\lg \lg n)$ bits.

Moffat [Mof99] gives a simple data structure that performs updates and queries in $O(\log(1 + k))$ time when updating or querying the kth key a_k in the sequence. Moffat presents this result in the context of arithmetic encoding by relating the running time to the size of the code words; he shows that if the elements are sorted in descending order of probability, the time to perform an update or query is proportional to the number of bits needed to encode a symbol. However, since the sequence must be kept in sorted order, the data structure can only efficiently support increments or decrements by one.

Hampapuram and Fredman [HF98] investigate the case where updates and sum queries have different probabilities: given p_k being the probability of updating node k, and q_k being the probability of querying the partial sum σ_k, they examine how to minimize the expected time for an operation. However, their data structure does not support the select operation.

The selectable partial sums problem is a well studied problem. However, most previous work assumes that all updates and queries are equally likely. Hampapuram and Freedman give the only analysis that considers the case where the keys have different access probabilities.

6 Optimal search trees and coding systems

As we showed in the previous section, we can use binary trees to solve the selectable partial sums problem. Without any knowledge of the access frequencies of the leaf nodes, one generally attempts to construct a balanced binary tree such that each node is accessed in $O(\log n)$ time, where n is the number of elements stored in the tree. However, if some nodes are known to be accessed with higher probability, it may make sense to place these nodes closer to the root. In this way, there is a higher probability that a search will take less than $O(\log n)$ time, and hence we will obtain a better running time on average, or over a large number of searches. An *optimal search tree* is a search tree that minimizes the expected time for a search, given the access probabilities for each element, and hence if we use an optimal search tree (or near-optimal search tree) to solve the selectable partial sums problem, we will obtain a better expected running time.

Given a set of probabilities $(p_1, \ldots, p_n)$ that each of the elements will be accessed, the expected search time in an optimal binary search tree is related to the *entropy* of the

set of probabilities: $H = \sum_{i=1}^{n} p_i \log \frac{1}{p_i}$. The expected search time in an optimal binary search tree is at least $H - \log H - \log e + 1$ and at most $H + 2$ [Meh77].

Optimal binary trees are also related to Huffman codes [Huf52], a compression scheme that assigns to each character a binary code word of varying length, depending on its frequency in the message, and in which no code word is a prefix of another. For such types of compression schemes, Huffman codes are optimal; a character that occurs with probability p has a codeword of length at most $\lceil \log \frac{1}{p} \rceil$. A (binary) code tree can be constructed from a code in the following way: given a code word, a 0 in the word corresponds to going left in the tree, and a 1 to going right. When we reach the end of the code word, the node that we are at, which will be a leaf node, contains the encoded character. Since each codeword has length at most $\lceil \log \frac{1}{p} \rceil$, the average codeword length is at most $\sum_{i=1}^{n} p_i \lceil \log \frac{1}{p_i} \rceil$, which is very similar to the definition of the entropy of a set of probabilities. Thus a Huffman tree can also be used as a near-optimal search tree, and hence can also be used in solving the selectable partial sums problem and the random object selection problem, giving us a better expected running time than $O(\log n)$. There are other types of coding systems as well; we are interested in those where the code can be represented by a tree, since this allows us to solve the selectable partial sums problem.

When compressing a message using Huffman's algorithm, two passes over the message are required: one pass to count the number of occurrences of each character, and another pass to encode each character. Huffman coding also does not allow for updating the character frequencies. Faller [Fal73] and Gallager [Gal78] independently propose a one pass algorithm that dynamically constructs a Huffman tree; when the algorithm processes the ith character of the message, the code tree is a Huffman tree for the first i characters of the message. A system than creates a Huffman code dynamically is known as a *dynamic Huffman coding*. Knuth [Knu85] improves the algorithm by increasing the speed and by supporting weight decrements in addition to increments. Knuth's algorithm updates the Huffman tree by incrementing the weight of the node corresponding to the next input character, along with its ancestors, and performs a number of subtree swaps to ensure that the tree is still optimal, potentially a number of subtree swaps equal to the length of the path from the node to the root. The algorithm thus performs work proportional to the depth of the node, or the length of the codeword. Knuth also shows how to support weight decrements. This algorithm is known as the Faller-Gallager-Knuth (or FGK) algorithm, and uses on average at most 2 extra bits per character over Huffman's algorithm [ML97].

Vitter [Vit87] presents a similar algorithm that uses only at most 1 extra bit per character over Huffman's algorithm. Vitter's algorithm differs from the FGK algorithm in the data structure used; Vitter introduces a data structure called a *floating tree* that does not store parent-child relationships explicitly. This allows greater flexibility when performing subtree swaps in time similar to the FGK algorithm. This extra flexibility allows Vitter to maintain an invariant that improves the code size. In fact, Vitter shows that the code generated is optimal over all dynamic Huffman codes.

Gagie [Gag04] investigates dynamic Shannon coding. Shannon coding is similar in flavour to Huffman coding, but differs in the way the code tree is generated; in the code tree for a Shannon code, the weight of the internal node that has children with weight w_i and w_j will be $\max(w_i, w_j) + 1$ rather than $w_i + w_j$. In the static case, Shannon coding is less efficient than Huffman coding in terms of code lengths. However, Gagie shows an algorithm that produces dynamic Shannon codes, but is more efficient than the dynamic Huffman codes.

Optimal binary trees and some coding systems allow us to obtain trees in which leaves that have a higher probability of being accessed are closer to the root, and hence take less time to access. We can use these trees to solve the selectable partial sums problem. By considering the different access probabilities of the keys, this allows us to decrease the expected cost of the operations.

7 Research goals

Similar to the cases of optimal search trees and coding systems, we are considering different probabilities of access, and thus our goal is not to minimize the worst-case performance of the data structure, but to minimize the expected cost of the operations. Thus we will try to obtain running times that are expressed as functions of the probabilities p_e, or the rates λ_e, instead of just the number of elements n. Since we will have a large sequence of operations, and we are only concerned with the total running time, it may suffice to obtain amortized results instead of worst-case results.

Our aim is to design a data structure that supports the operations with the following running times. From results on optimal search trees and Huffman coding, we can obtain trees in which each element is accessed in at most $\log \frac{1}{p_e}$ time, where p_e is the access probability of that element. Thus it seems reasonable to aim for updates and selections to be performed in $O(\log \frac{1}{p_e})$ time. The time to perform an update may also be a function of the rate change; larger updates may require more time since updates may require the tree to be reorganized in order to ensure that events can still be selected in $O(\log \frac{1}{p_e})$ time. We hope to also be able to perform insertion and deletion in $O(\log \frac{1}{p_e})$ time. Of course, being able to perform insertion and deletion in $O(\log \frac{1}{p_e})$ time implies that updates can be performed in the same time, since an update can be simulated by deleting the old rate and inserting the new rate.

It is much less clear what running time should be expected for splitting and merging. Splitting is performed when an event's relative probability is high, and merging is performed when the relative probability is low; hence, we would want to obtain results such that the operations can be performed quickly in those situations. In a tree structure that does not use any special representation, splitting should be trivial: since we are replacing a node with a set of k nodes, equally weighted, and whose total weight is equal to the weight of the original node, it suffices to make the original node be the root of a balanced tree that contains the new nodes as leaf nodes. Thus if the original node had

probability p_e, the new nodes will each have probability $\frac{p_e}{k}$. Since the root node of the subtree is at depth $O(\log \frac{1}{p_e})$ in the tree, due to the fact that the original node was at that depth in the original tree, all the leaf nodes will be at depth $O(\log \frac{1}{p_e} + \log k) = O(\log \frac{k}{p_e})$, and hence we will maintain the correct running time. However, if the tree is stored in a special structure (such as the array used by Vitter [Vit87]), or if the data structure requires the tree to satisfy other constraints, it is not as simple to modify the tree, and hence splitting may require more time.

Merging seems more difficult than splitting. In the ideal case, if we are merging, for example, two events that have the same parent, then we can do the reverse of what we do for splitting; we can replace the subtree consisting of the two events and their parent with a single node whose rate is equal to the sum of the rates of the two events. However, if the events do not have the same parent, we will have to remove the two events, and determine the best place to insert the new node in order to minimize the time taken by the operation. One way to do this is, if we are only merging two events, e_1 and e_2 that have the same depth in the tree, is to swap e_2 with the sibling of e_1 in the tree, updating the weights of the ancestors[1]. (See figure 2.) We can then easily merge e_1 and e_2 since they now have a common parent. When we swap e_2 with the sibling of e_1 and update the weights of the ancestors, we need to update only the weights up to the lowest common ancestor of e_1 and e_2; the nodes above the lowest common ancestor of e_1 and e_2 do not change weights since the set of leaf nodes in their subtrees do not change. Thus in order to minimize the time for a merge, we wish to keep nodes that may be merged "close" in the sense that their lowest common ancestor is as deep as possible. This means that when we perform an update operation, and we must reorganize the tree in order to ensure that all nodes are still accessed in $O(\log \frac{1}{p})$ time, we may need to be more careful in how we choose to reorganize the tree. It is unclear what type of running time we should expect for merging.

Our goal is to not only obtain theoretical results for the running times, but to also implement our data structure and compare its performance experimentally against using just a balanced binary tree to determine the difference in performance in real-life situations.

7.1 Approaches

In order to obtain a data structure that satisfies the requirements, we will first look at existing data structures and try to adapt them to suit our needs. Among the previous work on the partial sums problem, only Hampapuram and Fredman [HF98] consider query and update probabilities and try to minimize the expected cost. However, their data structure does not support the select operation, nor splitting, merging, insertions, or deletions. And while it allows the keys to be updated, it does not allow modifications

[1] If e_1 and e_2 do not have the same depth in the tree, we may not be able to swap e_2 with the sibling of e_1. Otherwise this may result in a node being too deep in the tree, giving an access time of more than $O(\log \frac{1}{p})$.

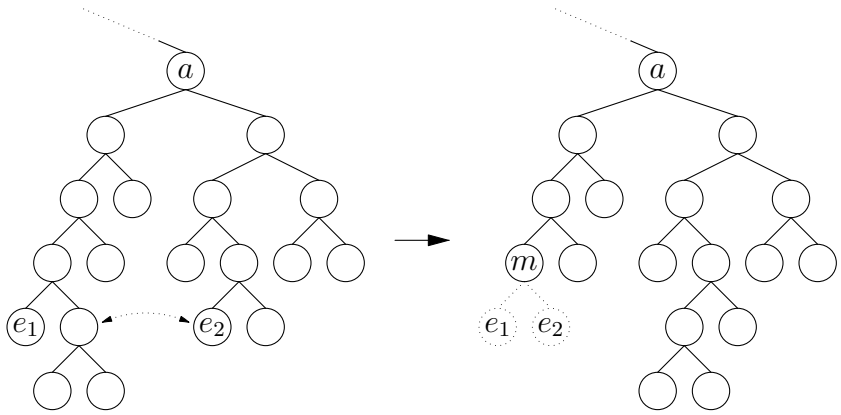

Figure 2: Merging two events e_1 and e_2 that do not have the same parent. The dotted arrow indicates the nodes that must be swapped. The node a is the lowest common ancestor of e_1 and e_2. Note that after swapping e_2 with the sibling of e_1, the children of a are the highest nodes for which the sets of leaf nodes in their subtrees change. The dotted nodes indicate e_1 and e_2 before merging, and the node m is the node resulting from merging e_1 and e_2.

of the access probabilities. Thus it seems like much work would be required to adapt their data structure. However, it may still be worth considering some of the ideas that they present. It may also be worth examining whether their structure can be easily modified to support the select operation.

The data structures used in optimal search trees and coding systems already support the selecting in $O(\log \frac{1}{p_e})$ time. However, the optimal search tree structures do not support changing access probabilities, so updates cannot be done, nor can insertions or deletions.

The data structures used for dynamic Huffman coding support updating of weights, but only support incrementing or decrementing weights by one. Thus an update operation would require $O(\Delta\lambda_e \log \frac{1}{p_e})$ total time, where $\Delta\lambda_e$ is the change in a node's rate. We can also perform insertions as follows: these data structures maintain a special node called the "zero node," which is a node with zero weight and represents all characters that have zero weight. Whenever a character c that previously had zero weight changes to have non-zero weight, the zero node is replaced by a subtree consisting of three nodes, one parent and two children, all with zero weight. One of the children becomes the new zero node and the other child becomes the new node for the character c, and its weight is incremented. This allows us to perform insertions since it allows us to add new nodes to the tree. As well, deletions could be performed by decrementing the desired node until its weight becomes zero. When its weight is zero, we can then simply remove the node from the tree, replacing its parent with its sibling. However this method will most likely be too slow for our purposes; insertions and deletions would

take $O(\lambda_e \log \frac{1}{p_e})$ time.

A different approach would involve grouping together nodes whose rates fall within a given range. Within these groups, we can store the nodes using a standard balanced binary tree — since all the nodes have similar rates, this should give us our desired performance, as long as we can select the correct group in the desired time bounds. Using this type of organization should provide support for selects, updates, insertions, and deletions in $O(\log \frac{1}{p_e})$ time. However it is unclear whether this structure will support the split and merge operations, and what performance it will give if it does support these.

Another approach would be to consider a structure similar to B-trees, with the additional requirement that the elements or subtrees within each node all fall within a specific range of rates. For example, we can require that every element in a node has a rate between 2^i and 2^{i+1}. In this way, as we go from the root towards the leaves, the weight decreases by a factor of at least two each time we go to a deeper level in the tree, which will give us an $O(\log \frac{1}{p_e})$ time for selections. Using a B-tree-like structure may help in performing insertions and deletions while ensuring that each event is at the correct depth in the tree to ensure that we have an $O(\log \frac{1}{p_e})$ time for selections.

Since our data structure represents cells in a geometric space, it may also be useful to examine previous work related to quadtrees to look for relevant methods.

We hope that by creating a data structure described in this proposal, we can obtain speed improvements in certain randomized algorithms. In particular, we hope to obtain significant improvements in the simulation of the reaction-diffusion equation. We will be using results from selectable partial sums, optimal search trees, and coding systems as starting points for our research. We also have some new approaches based on grouping nodes according to ranges of rates, and we also plan on examining previous work in the area of quadtrees.

References

[AT80] L. Arnold and M. Theodosoopulu. Deterministic limit of the stochastic model of chemical reactions with diffusion. *Advanced Applied Probability*, 12:367–379, 1980.

[Blo96] Douglas Blount. Diffusion limits for a nonlinear density dependent space-time population model. *Annals of Probability*, 24(2):639–659, 1996.

[Fal73] N. Faller. An adaptive system for data compression. In *Record of the 7th Asilomar Conferences on Circuits, Systems and Computers*, pages 593–597, 1973.

[Gag04] Travis Gagie. Dynamic shannon coding. In Susanne Albers and Tomasz Radzik, editors, *Proceedings of 12th annual European Symposium on Algorithms 2004*, volume 3221 of *Lecture Notes in Computer Science*, pages 359–370, 2004.

[Gal78] R. G. Gallager. Variations on a theme by Huffman. *IEEE Transations on Information Theory*, IT-24:668–674, November 1978.

[HF98] Haripriyan Hampapuram and Michael Fredman. Optimal biweighted binary trees and the complexity of maintaining partial sums. *SIAM Journal on Computing*, 28(1):1–9, 1998.

[HSS03] Wing-Kai Hon, Kunihiko Sadakane, and Wing-Kin Sung. Succinct data structures for searchable partial sums. In Toshihide Ibaraki, Naoki Katoh, and Hirotaka Ono, editors, *Algorithms and Computation, 14th International Symposium, ISAAC 2003, Kyoto, Japan, December 15-17, 2003, Proceedings*, volume 2906 of *Lecture Notes in Computer Science*, pages 505–516. Springer, 2003.

[Huf52] David A. Huffman. A method for the construction of minimum redundancy codes. In *Proceedings of the Institute of Radio Engineers*, volume 40, pages 1098–1101, 1952.

[KL02] Michael A. Kouritzin and Hongwei Long. Convergence of markov chain approximations to stochastic reaction-diffusion equations. *The Annals of Applied Probability*, 12(3):1039–1070, 2002.

[Knu85] Donald E. Knuth. Dynamic Huffman coding. *Journal of Algorithms*, 6:163–180, 1985.

[Meh77] Kurt Mehlhorn. A best possible bound for the weighted path length of binary search trees. *SIAM Journal on Computing*, 6:235–239, 1977.

[ML97] Ruy Luiz Milidiú and Eduardo Sany Laber. Improved bounds on the inefficiency of length-restricted prefix codes. Technical report, Pontificia Universidade Católica do Rio de Janeiro, September 1997.

[Mof99] Alistair Moffat. An improved data structure for cumulative probability tables. *Software — Practice and Experience*, 29(7):647–659, 1999.

[PD04] Mihai Pătraşcu and Erik D. Demaine. Tight bounds for the partial-sums problem. In J. Ian Munro, editor, *Proceedings of the 15th annual ACM-SIAM Symposium on Discrete Algorithms, New Orleans, January 11–13, 2004*, pages 20–29. SIAM, 2004.

[RRR01] Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct dynamic data structures. In Frank K. H. A. Dehne, Jörg-Rüdiger Sack, and Roberto Tamassia, editors, *Proceedings of 7th International Workshop on Algorithms and Data Structures, Providence, RI, August 8–10, 2001*, volume 2125 of *Lecture Notes in Computer Science*, pages 426–437, 2001.

[Vit87] Jeffrey Scott Vitter. Design and analysis of dynamic Huffman codes. *Journal of the Association for Computing Machinery*, 34(4):825–845, October 1987.