Efficient Generation of Discrete Random Variates

Bram van de Klundert

December 16, 2019

Abstract

In this paper we look at some of the fastest algorithms to generate discrete random variates. All of these algorithms support updating the rate of an event after the data structure has been build. As part of this study we propose extensions for the Table method and for the Alias method which allow these algorithms to perform updates on events. We show that using these extensions the Table method can be updates in $O(\Delta r)$ time. We also show that the Alias method can do updates bounded by $C \cdot r_{ave}$ in $O(1)$ amortized time, where r_{ave} is the average rate of events in the data structure.

To back up these theoretical results an experimental evaluation of the algorithms was executed. In this evaluation we scale the number of events, average rate of events and variance of the rate of events to see how each of the algorithms react to these changes.

1 Introduction

Generating discrete random variates is a common problem in simulations. It occurs in many fields including but not limited tot biochemistry [6] and chemistry [2]. This problem has been studied thoroughly. This resulted in a number of good algorithms for this problem like the Rejection method [7], Table method [3], Square histogram method [10] and the Alias method [9]. This last one allows us to generate variates in $O(1)$ time after $O(n)$ preprocessing time which is clearly optimal.

Maurer [5] studied a variant where we generate variates without replacement. For this problem they developed their own algorithm namely the Differential Search Tree method. They mention that the tree based method by Wong and Easton [11] can be used for the same problem. They also suggest that the Table method could possibly be adapted to allow for updates.

There is another variant of this problem which has been studied a lot less. In this variant the probabilities of each event can change. This variant thus requires algorithms to allow for updates, additions and removals of events. Many of the current algorithms used to generate discrete random variates do not allow for these operations. Two algorithms that do allow for updates are the ones proposed by Matias et al. [4] and the Rejection method.

In this paper we propose two extensions to existing algorithms. First, we extend the Table method to allow for updates to happen on the data structure. The resulting updates can be done regardless of the number of existing events in the data structure. The time required to complete the update scales with the size of the update i.e., the difference between the old and new rate of the updated item.

The second extension is for the Alias method. By forgoing the guarantee of getting an event by pulling two random numbers we can do updates of size $O(n)$ in constant time.

To back up the theoretical results of these algorithms we implemented our algorithms and a some of the fastest known algorithms. We used these to run a number of experiments to show how these algorithms scale. In these experiments we not only look at the number of events, but also the average rate and variance.

2 Problem description

Given is a set of N distinct events $e_0, \ldots, e_n$ with their associated probabilities $p_0, \ldots, p_n$. We want to sample from these events such that the probability of getting event e_i is equal to p_i. This version of the problem has been well studied. Using the Alias method proposed in Walker [9] this can be done in $O(1)$ time after building the data structure in $O(n)$ time.

The problem we will be focusing on in this paper differs from the above definition in two points. We will be using rates to describe the probabilities of the events on our data set. The rate r_i of some event e_i is defined as $p_i = r_i/r_{tot}$ where $r_{tot} = \sum_{i=1}^{N} r_i$. For example: consider three events A, B and C. $p_A = 50\%$, $p_B = 20\%$ and $p_C = 30\%$. A representation of these probabilities in rates could be $r_A = 5$, $r_B = 2$ and $r_C = 3$. However rates can be more flexible than that, because if we want changes in the rates to have a smaller impact on the probabilities we can chose to represent these probabilities with the rates $r_A = 25$, $r_B = 10$ and $r_C = 15$. In this second example each point of rate corresponds to 2% of the total, where in the first example each points represented 10%. Using rates instead of probabilities has two large advantages. First rates are integers, as such we do not have to worry about slight errors due to floating points. More importantly, the rate of any event can be changed without having to normalize all other rates. When we are using probabilities they always have to sum to 1. So whenever we change the probability of some event we have to normalize all other probabilities. Using rates circumvents this issue.

The second difference is that we have some extra requirements for our data structures. In the original problem the algorithms only have to generate a variate while respecting the probabilities of each variate. For the data structures in this paper we require that they support the following operations:

- Select an element, where each element gets selected with a probability proportional to its rate.

- Change the rate of an element.
- Add an element to the data structure.
- Remove an element from the data structure.

Of these requirements we will mainly be focusing on the first two, as adding and removing an element both require that we can change the rate of an event in the data structure. This gives us three benchmarks to compare the algorithms: build time, generation time and update time.

3 Algorithms

3.1 Rejection

The Rejection method [7] is one of the oldest algorithms for generating discrete random variates. It is also the simplest of the algorithms we look at in this paper.

The Rejection method works by selecting one of the events at random. We then check if we accept this chosen event. If we do not accept it we restart.

For this algorithm we just need a list of all events with their rates.

Algorithm 1 Generating an variate using the Rejection method

```
1: procedure GenerateVariate
2:     while TRUE do                          ▷ Repeat until we select a variate.
3:         Select an index i at random
4:         Get an uniform random number R between 0 and 1
5:         if R ≤ r_i/r_max then              ▷ Check if we accept the variate.
6:             return i
7:         end if
8:     end while
9: end procedure
```

3.1.1 Generating a variate

To generate an variate using the Rejection method we select an index from the list of events at random. We then check if we reject that variate by calculating its acceptance probability. If we reject the variate we to back to the first step and try again. To calculating the acceptance probability for some event i we use the formula r_i/r_{max} where r_{max} is the largest rate in the data structure.

Due to the possibility to endlessly reject variates the algorithm never terminates in the worse case. We can determine a worse case expected time by assuming we always select the variate with the lowest rate. In this case the probability of accepting will be r_{min}/r_{max} which gives us r_{max}/r_{min} expected

number of attempts before we accept the event. As such we can generate an variate in $O(r_{max}/r_{min})$ expected time.

Due to the acceptance probability the performance of the Rejection method depends a lot on the distribution of the rates of the events we are generating. D'Ambrosio et al. [1] have shown that the Rejection method is expected to run in constant time if the events follow an non-decreasing rate distribution.

3.1.2 Updating variates

Updating the rate of an event for the Rejection method is done by changing the rate in the internal array of rates. This can obviously be done in $O(1)$ time.

When doing updates there is one more thing we have to do: Find the new largest rate in the data structure. If the largest rate increases we can simply take the new rate and use that as maximum rate. When the largest rate decreases we have two ways to handle it: first we can ignore any decreases in r_{max}. The advantage of this is that we do not have to spend time finding the next largest rate. This has the downside that the maximum rate will never decrease, which will result in an larger expected number of tries before we accept an event. The second solution is to search for the new largest rate whenever we lower the largest rate in the data structure. While this keeps r_{max} at the lowest possible values, searching for the largest rate in the data structure takes $O(n)$ time. To mitigate this large cost we could run this update once every n updates such that we still end up with a $O(1)$ amortized expected update time.

For our experiments later in the paper we chose to ignore reductions to the maximum rate. Due to the setup of the experiments the downside of this choice does not come up during the experiments.

3.2 Alias

The Alias method was proposed by Walker [9] and is closely related to the Rejection method. In the Alias method we also pick one of the indexes at random and check if we accept them. The difference comes when we reject the event. In the Alias method we precompute an alias for each index. If we reject an event we select this alias instead of restarting.

This is accomplished by computing n blocks. Each of these blocks contains two events, an event and their alias. As we have n blocks each of these represents an rate equal to average rate (r_{ave}) over all events.

As an example assume we have 3 variates A, B and C. A has a rate of 70, B a rate of 20 and C a rate of 30. In the Alias data structure we would have 3 pairs, where each pair has a rate of 40 (r_{tot}/n). To form these pairs we only have one option: AB, AC and AA. To balance out the probabilities the first pair AB has a $20/40 = 50\%$ chance to select A and a 50% chance to select B. Similarly AC has an 25% chance to select A and 75% chance to select C. Finally there is also a block which only contains rate from A thus has a 100% chance to select A.

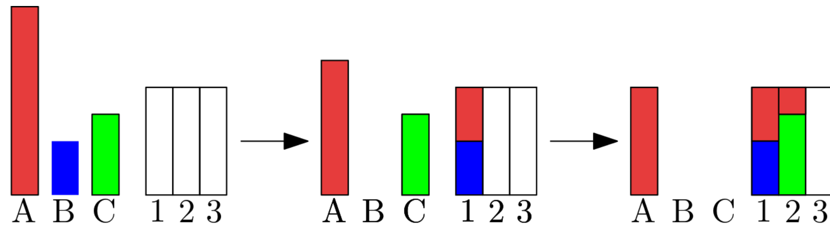


Figure 1: We build the Alias structure from three events. A with $r_A = 70$, B with $r_B = 20$ and C with $r_C = 30$. We first take B and complete block 1 using weight from A. After that we take C and finish the block with some weight from A. After this step we are left enough rate from event A to fill the last block.

3.2.1 Data structure

In the Alias method we keep track of three things for each block: Which event is in the block, the alias of the block and the probability of accepting the event if we select the block. As such the data structure consist of three arrays of size n.

To fill these arrays we use Algorithm 2 proposed by Vose [8] which takes $O(n)$ time. For this algorithm we split our events into 3 groups. An under full group containing all events with rate less than the block size. An over full group with all events which have more rate than the block size. The last group consists of events with rate equal to the block size. We start with the last group, as these are equal to the size of a block we can simply add these events as their own blocks. For the remaining blocks we take an event from the small list and add it to an block. Then we take an event from the big list and fill the remaining space in the block with that event. We then check if the remaining rate of the event is larger or equal to the block size. If it is we return it to the over full group. Otherwise we put it in the under full group.

This algorithm works perfectly in theory where we have infinite precision on floating point numbers. Unfortunately this is not the case when we implement this using most programming languages. The main thing that can go wrong is events which are exactly the block size being assigned to the small list on line 3 and 12. To prevent this we need to slightly modify the algorithm. On line 6 we change the condition of the while loop to check if both the small and big list have items left. If the big list ever empties before the small list we have at least one small item with size blocksize. In this case we need to add items from the small list to the data structure.

3.2.2 Generating variates

Generating an variate using the Alias method similar to the method for the Rejection method. We start by selecting an event e_i at random. We then check if we accept this event by generating an uniform random number between 0 and

Algorithm 2 Building the Alias data structure

```
 1: procedure buildAlias(events)
 2:     blockSize ← ave(events)
 3:     Fill small with all events e with r_e < blockSize
 4:     Fill big with all events e with r_e > blockSize
 5:     Add a block for each event e with r_e = blockSize
 6:     while small is not empty do
 7:         b ← big.pop()                          ▷ Pop the last element of big.
 8:         s ← small.pop()                        ▷ Pop the last element of small.
 9:         Add the rate of s to an new block b
10:         Add b as alias for block b
11:         Reduce the rate of b by blocksize − r_s
12:         if r_b < blockSize then       ▷ Check if b is larger than the block size.
13:             small ← b                          ▷ If not add b to small.
14:         else
15:             big ← b                            ▷ Otherwise add b to big
16:         end if
17:     end while
18:                       ▷ All remaining items in big should be of size blockSize.
19:     while Big is not empty do
20:         g ← big.pop()
21:         Add a block for event g
22:     end while
23: end procedure
```

1, if that number is smaller than r_i/r_{ave} we accept it. If we reject the variate we simply return the corresponding alias.

3.2.3 Updating

The main downside of the Alias method is that there is no known way to efficiently update one of the variates in the data structure. This is due to the fact that all blocks in the Alias data structure have to be the same size. Whenever we change the rate of one of the variates we would increase or decrease the size of one block. To correct for this we would have to re balance all other blocks. If we have to modify all blocks we might as well rebuild the entire data structure which takes $O(n)$ time.

Reducing the time bound on accepting an variate in $O(1)$ to $O(1)$ expected time it is possible to update the Alias method. To do this we make it possible for the alias to be rejected. This allows us to lower the amount of rate in a block of the Alias method without having to rebuild the entire structure. Because blocks do not have to be completely filled now we can also add weight to a variate by creating a new block and partially filling it. The downside of both these updates is that they increase the expected time use to generate a variate.

The probability of restarting added by each update depends on the number of events in the data structure. Increasing weight adds uncertainty of at most 1 block, by adding a block of rate 1. Removing an item can add more uncertainty, but on average it will remove r_{ave} weight, which is equal to the size of one block. This means that in the expected worse case scenario after $\frac{1}{2}n$ updates, we have to on do on average two attempts before we generate an variate. Thus if we rebuild every $\frac{1}{2}n$ updates we can generate variates in $O(1)$ expected time.

This extended Alias algorithm has some nice properties.

Theorem 1. *Given a data structure for the Alias extended algorithm, we can increase or decrease the rate of a single event in this data structure in $O(C)$ amortized worse case time as long as the change of rate is bounded by $C \cdot r_{ave}$ where where C is some constant and r_{ave} is the average rate of events in the data structure.*

Proof. For increases in rate this is easy to show. The data structure has blocks of size r_{ave}. So we can add a rate of $C \cdot r_{ave}$ by adding C blocks to the data structure.

Decreases require the following observation: After building the data structure the average number of block an containing a given event has an upper bound of 2. This follows from the facts that there are n blocks and each block contains at most 2 events. There for right after we build the data structure we have to alter an average of 2 blocks to do the maximum possible decrease in rate. After this each update can add at most C block to the data structure which we can remove in an equal amount of time. □

Algorithm 3 Updating the Alias Enhanced data structure

```
1: procedure updateAlias(e, newWeight)
2:     Δr ← events[e] − newWeight                ▷ Calculate the change in rate
3:     if Δr > 0 then                                ▷ The weight is increasing
4:         while Δr > blockSize do
5:                                   ▷ Add blocks till Δr can be contained in one block.
6:             Add a full block b of event e
7:             Add block b to the list of block containing e
8:             Δr ← Δr − blockSize
9:         end while
10:                                  ▷ Add the remaining rate in Δr as another block.
11:        Add a block b of event e with r_e = Δr
12:        Add block b to the list of block containing e
13:    else if Δr < 0 then                           ▷ The weight is decreasing
14:                                         ▷ Repeat until we removed all rate in Δr.
15:        while Δr < 0 do
16:            Get a block b containing e
17:            if Δr > the rate of event e in b then
18:                Reduce Δr by the rate of e in b
19:                Remove event e from b
20:                Remove event b from the list of event containing e
21:            else                          ▷ Δr is smaller than the rate of e in b
22:                Reduce the rate of e in b by Δr
23:                Δr ← 0
24:            end if
25:        end while
26:    end if
27: end procedure
```

3.3 MVN

A very promising data structure was proposed by Matias et al [4]. This data structure splits all events into ranges. Each range i consist of events with rate in $[2^{i-1}, 2^i)$. The size of this range was chosen such that the largest rate in the range was at most two times the size of the smallest range. This way we can use the Rejection method to select one item in the range in constant expected time.

By stacking these ranges in multiple layers we create a forest of trees in which we can search very quickly. We can build this forest in $O(n)$ time. With the modification this data structure also allows for $O(\log^* n)$ updates and generation of variates. In the remainder of the paper we will be using the abbreviation MVN to refer to this algorithm.

3.3.1 Constructing the forest

The MVN data structure consists of an forest of trees, a list of root nodes for each level and a list of the weight of level. While we construct the forest we add all root nodes to the correct lists.

To build the trees we start form the bottom on level 0. Level 0 contains all events in the data structure. To build level 1 we group the level 0 events in ranges. Range i contains all events with rate within $[2^{i-1}, 2^i)$. Each of these ranges gets a rate equal to the sum of the rates of all events in the range. To create level 2 ranges we do the same but on the ranges of level 1. There is one exception: If a range only has one child we consider it a root node and do not use it to build the next level. Instead we add the root node to the list of root nodes for this level and add the rate of this range to the rate of this level.

By repeating the above steps we end up with a number of trees, each represented by a root node.

Algorithm 4 Generating an variate using the MVN method

```
 1: procedure GenerateVariateMVN
 2:     Select a level L based on the total rate of that level
 3:     Select a root node Node from level L based on the rates of the root node.
 4:     while Node.level > 0 do                    ▷ Repeat until we reach level 0.
 5:         Uniformly select a range Cr from the children of Node
 6:         Get an uniform random number R between 0 and 1
 7:         if R ≤ r_Cr / r_Node then
 8:             Node ← Cr
 9:         end if
10:     end while
11:     return The event index of Node
12: end procedure
```

3.3.2 Query

To generating a random event from the data structure we have to go through three steps. We will look at these steps in further details later in this section

- Select a level.
- Select a tree from the level.
- Walk down the tree to select an event.

For the first step we select one of the levels where the probability of selecting a level depends on the rate of all the trees in that level. This means that the probability to select level i is equal to the sum of the rate of all root nodes on level i divided by the total rate of the data structure. The easiest way to implement this is by selecting a uniform random number between 0 and r_{tot}. We can than loop over all levels, check if the rate of the level is lower than the random number. If it is we select that level. When the random number is larger than the rate of the level we subtract the rate of the level from the random number and move on to the next level.

Once we have a level we do the same to select a root node. But this time we want to select some tree t from level l. The probability of selecting t is r_t / r_l. We can use the same strategy to select the root node as we did for selecting a level.

The final step is to traverse the tree until we reach one of the variates. To do this we will use the Rejection method on each level of the tree to select one of the children until we get to on of the events at level 0. Because each range r_i has a limit on the rate of the ranges that can be inside it of $[2^{i-1}, 2^i)$ we know that the largest value in the range can at most be twice as large as the smallest one. This property allows us to select a child at random in $O(1)$ expected time using the Rejection method. As the depth of the trees is bounded by $\log^* n$ we can generate one of the variates in $O(\log^* n)$ time.

3.3.3 Update

Whenever we change the rate of an event or range there are six possible situations:

- The rate of the event stays within the same range as it was before the update.
- The rate of the event changes such that it falls its outside current range and there exist some range that can now contain it.
- The rate of the event changes such that it falls its outside current range and there is no range that can contain it.
- The range was a root node and after the update it rate does not fall within another range.

- The range was a root node and after the update the rate does fall within another range.
- The range was a root node but its last member was removed.

Whenever we update an range we also have to update both the range that used to contain it and the range that now contains it. This way the rates of all ranges stays up to date. In the worse case we split at every over the $\log^* n$ levels of the tree, resulting in an $(2^{\log^* n})$ expected worse case update time.

Whenever we change the rate of a root node or add/remove a root node we have to update the list of root nodes on a level and the associated rate of the level.

3.3.4 Getting down to $O(\log^* n)$

With the implementation of the update as described in the update section an update requires $O(2^{\log^* n})$ expected worse case time. The main cause of this time bound is the possibility of ranges changing parents. Whenever an range switches from one parent to another we create two update paths we have to follow upwards.

To improve this Matias et al. proposed that we introduce two tolerances in the algorithm. First, when we determine if a root has the correct parent we use the bounds $[(1-b)2^{i-1}, (2+b)2^{i-1})$ for range r_i. In this formula b has a value $0 \leq b < 1$. This will allow the algorithm some slack before we have to move a range to a different parent. The second addition is a change to how we determine if a range becomes root node. In the original algorithm any range with only 1 child is a root node. We change the requirement to become a root node to having less than $d = 1/2((2+b)/(1-b))^2 2^c$ where c is some non negative integer.

By introducing these extra tolerances we can drastically change the number of expected steps required in each update. The base algorithm is equivalent to setting $b = 0$ and $c = 0$. In their paper Matias et al. show that by increasing these tolerances to $b = 0.4$ and $c = 2$ we can improve the time bound for generating and updating an event to $O(\log^* n)$ amortized expected time.

These modifications also have a downside, in the paper they mention that these modifications add a $1/b$ multiplicative factor to the generation time due to the increased bounds for the Rejection method. Similarly there is an $\log d$ additive factor added to the generation time due to a larger number of root nodes. They suggest that for practical purposes, where the worse case time is not essential, an implementation can use the original algorithm or only implement the change to the maximum size of a root node.

For out experiments we compare both the algorithm without tolerances ($b = 0$ and $c = 0$) and with the suggested tolerances ($b = 0.4$ and $c = 2$)

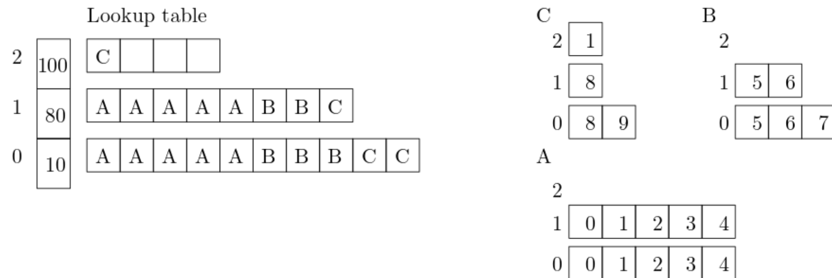


Figure 2: This figure shows the data structure from the Table method containing three events A, B and C. $r_A = 55$, $r_B = 23$ and $r_C = 112$. Before each line of the lookup table there is the total rate of that level. Level 1 has a total rate of 80. For each of the events on each level we have a linked list containing the indexes of all item of the given event on that level.

3.4 Table Method

The Table method was proposed by Marsaglia [3]. It is based on the idea that we can pick an element from an array in $O(1)$ time. We can use this to build a lookup table to generate random variates. To do this we create an array of size $r_{tot} = \sum_{i=0}^{N} r_i$. To this array we add elements such that event e_i has r_i elements.

To generate a random variate we start by generating an uniform random number r between 0 and r_{max}. We then lookup the event we generate by looking at index r in our lookup table.

The obvious downside to this setup is that is requires an array of size r_{tot}. In his paper Marsaglia aims to resolve this issue by compressing the data structure.

3.4.1 Compressing the lookup table

To improve the space requirement on the data structure we can compress the lookup table. To this end we will group items into multiple levels. Where items on higher levels represent more rate. We do this using Algorithm 5

Algorithm 5 Compressing the the lookup table

1: $l \leftarrow 0$
2: **while** There is an event with more than 10 items at level l **do**
3: Remove 10 items from an event e
4: Add 1 item of e to level $l + 1$
5: **if** There are no events with more than 10 items at level l **then**
6: $l \leftarrow l + 1$
7: **end if**
8: **end while**

This algorithm leaves us with a lookup table with $\log_{10} r_{max}$ levels. Each

item at level l represents 10^l times in the original lookup table. By doing this we reduce the size of the lookup table from r_{tot} to $\log_{10} r_{max} \cdot 9$ items.

When building the actual data structure we do not actually want to build the original lookup table. Instead we will directly determine how many items we want on each level. To determine the number of items for a given event e on level l we need to find the number of items of size 10^l. Its important to note that we should exclude those that fit on level $l+1$, with an value of 10^{l+1}. To accomplish this we use the formula $\lfloor r_e \bmod 10^{l+1} \rfloor$

To generate an variate from this new data structure we can use Algorithm 6. In this algorithm we select an position in the lookup table at random. We then check each level if that level contains that index. Finally when we reach the right level we select the item from the level.

Algorithm 6 Generate a variate using the Table method

```
1: ur ← random()                          ▷ Get a uniform random value between 0 and 1.
2: r_scaled ← ur · r_tot                  ▷ Scale it using the total rate.
3: l ← The highest level of the data structure
4: while r_scaled > levelWeigth[l] do     ▷ Find the correct level.
5:     r_scaled ← r_scaled − levelWeigth[l]
6:     l ← l − 1
7: end while
8: return levelItems[l][r_scaled/10^l]    ▷ Return the event from the lookup table
```

Theorem 2. *Given an compressed lookup table with maximum rate r_{max} we can generate an variate from this table in $O(\log r_{max})$ time.*

Proof. When generating a variate we have to, in the worse case, look at all levels of the data structure to see if they contain our random index. Checking if the level contains a certain index takes $O(1)$ time. We do this check for each of the $\log_{10} r_{max}$ levels. Thus we can find the correct level in $O(\log r_{max})$ time.

After we find the correct level we have to select the correct index form the array on this level. This also takes $O(1)$ time.

Thus in total we spend $O(\log r_{max})$ time to generate an variate. □

Up till this point we have always taken for granted that each level increase the rate of each item by a factor 10. There is no reason to restrict ourselves to using 10 as base factor for the Table method. By changing the base factor of the data structure we can influence the size of each level and total number of levels of the data structure. If we take B as base for our data structure each level can have up to $B-1$ items. The number of layers of the data structure is equal to $\lfloor \log_B r_{max} \rfloor$.

3.4.2 Update

In Marsaglia [3] only the static problem of generating variates is considered. We have developed an extension to this data structure that allows us to modify the

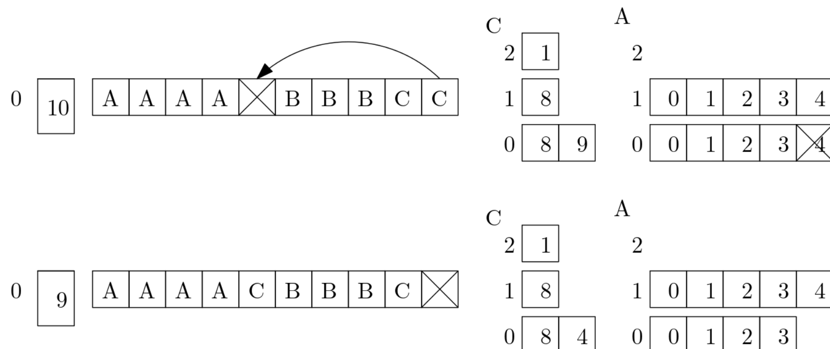


Figure 3: This image shows part of the data structure from Fig 2. We only show the lookup table on level 0 and the index arrays from A and C. It shows what happens if we lower the rate of A by one. The last item from A gets replaced by and item from C as it happens to be the last in the lookup table.

rate of any event in $O(\log \Delta r)$ amortized expected time where Δr is the change in rate of the update.

When doing updates to the Table method we have to add and remove a number of items for a given event. To do this efficiently we need to be able to quickly find items of that event. Secondly we need to remove items in the middle of an list. To solve the first point we maintain an linked list of indices of items for each event on each level. This way we can, given an event and level, find an item to remove in $O(1)$ time. The second point is solved by using a simple trick. If we want to remove an item in the middle of an list we move the last item in the list to that location. Then we remove the last item. As removing the last index only takes $O(1)$ time this allows us to remove any element in the array in $O(1)$ time. The second solution does add some complications for our linked list of indices as we now have need some way to quickly update the linked list. To find the index of an item on the linked list we store an reference to the list item with each item. This way we can quickly find the item in the linked list we have to update.

These modifications result in Algorithm 7. All the linked lists for together store one item for each item in the data structure. Thus the linked list uses $O(n \log r_{max})$ space. For each level the new or old rate is in we can add or remove up to $B-1$ items. As the number of levels is equal to $1 + \log_B r_{max}$. An update can take at most $O(\log r_{max})$ time.

Theorem 3. *Given an compressed update table with the modifications in Algorithm 7 we can do an update of size* 1 *on a single event in* $O(1)$ *amortized time.*

For the proof of this result we will first need to show that we can do increments and decrements of weight 1 in constant time. Following that we will introduce an modification for the Table method that will allow us to combine

Algorithm 7 Updating the Table method

```
1: procedure UpdateTable(e, newWeight)
2:     updateDepth ← the highest level we have to update
3:     for i ← 0 . . . updateDepth do                          ▷ Loop over all levels
4:         newLevelWeight ← the new rate of e on level i
5:         newLevelItems ← the difference in items of e on level i
6:         if newLevelWeight > r_{l,e} then                     ▷ We have to add items.
7:             for j ← 0 . . . newLevelItems do           ▷ Do newLevelItems times.
8:                 Add an item to the lookup table on level i for event e
9:                 Add the item to the list of items for event e
10:            end for
11:        else                                              ▷ We have to remove items
12:            for j ← 0 . . . abs(newLevelItems) do
13:                Get an item from e at depth i
14:                Remove the item from the list of items of event e.
15:                Remove the item from the lookup table.
16:            end for
17:        end if
18:        Update the weight of the level
19:    end for
20: end procedure
```

these two proofs such that it will work for all updates of size 1.

Lemma 1. *Given an compressed update table with the modifications in Algorithm 7 we can process an increase in rate of* 1 *on a single event in* $O(1)$ *amortized time.*

Proof. To show this we will be using the accounting method. We take the price of adding or removing one item from the data structure as 1 coin. We will pay 2 coin whenever we add an item to the data structure, one to add the item, the other as a down payment for later. Assume we start with an empty table. After adding 9 items the 10th item will overflow the level. Here we will use our saved up 9 coins to remove all 9 items we have added on level 0 so far. Following this we will add 1 item to level 1 and do a down payment of 1 coin. Because we always put down a coin whenever we add an item on any level removing the item is always free. This way any increase of 1 takes 2 coins. which shows that it takes $O(1)$ amortized time. □

The proof that we can do decreases in $O(1)$ amortized time is similar to the one of Lemma 3.4.2.

Unfortunately we can construct a situation where every update is expensive. For this we need a lookup table where level 0 contains $B-1$ items of event e_i. If we add 1 item to event e_i we will trigger an expensive step. After this step

the number of items of e_i at level 0 is 0. We can now cause another expensive step by removing 1 item from e_i. We can repeat this process indefinitely.

To prevent two successive updates of size one from triggering an expensive step we can increase the number of items of e can exist on level l. Let us look at what happens after we do an expensive step if we double threshold on the number of items allowed on a level. This means that a level can have up to $2B - 1$ items of any given event. After we do an expensive step due to a decrement we are left with $B - 1$ items on the level. Doing an increment at this point only increase the number of items to B. We need B more increments before we trigger another expensive step.

Doubling the capacity of our levels does bring some more complications with it. First of all we cannot calculate the exact number of items that should be on a level after an update as that would limit the number of items that can exist on the level to $B - 1$. Instead of the absolute number of items we look at the difference in rate. From this difference we calculate the increase/decrease of items on each level. The second complication is that level $l-1$ can now contain more rate than 1 item of level l. This can cause complications when removing weight from level l. Imagine the follow scenario: We have 2 items of event e on level l. We also have $B + 3$ items on level $l - 1$. If there is an update which removes 3 items from level l we not only have to check the levels above l but also those below l. This strange update where we have to check if the levels below l contain enough weight to create another item at level l can only happen at the highest level of an event. If there was some higher level that contains weight we could simply do an expensive update and reduce the weight of that level. The second condition for this update to happen is if we remove more rate than the total rate on the highest level of the event. It is thus impossible to trigger this update when doing updates of size 1, but any implementation of this algorithm will have to be able to do this update.

All of these changes combined result in Algorithm 8.

Now that we can do updates of size 1 in $O(1)$ time we can improve the general time bound for updates. To do this we have to make the observation that an update of size Δr can cause an increase or decrease of at most 1 on level $(\log_B \Delta r) + 1$. As we can do any updates of size 1 in $O(1)$ amortized time our time bound only has to look at the updates to levels $\log_B \Delta r$ and below. So we can do updates in $O(\log \Delta r)$ time.

4 Experiments

Beside giving an overview of available algorithms and contributing some of our own we want to compare these algorithms. For our experiments we have implemented a number of algorithms: MVN, MVN with tolerances, Table method, Rejection method, Alias method and Alias method with enhancement. We will be comparing these algorithm on time to build the data structures, time to generate a variate and time to update the rate of an event.

To ensure consistency between experiments all algorithms will be using the

Algorithm 8 Updating the Extended Table method

1: Determine the change in rate Δr
2: **if** Δr is positive **then**
3:     **while** Δr >blockSize **do**
4:         Add a block of event e and probability $= 1$.
5:     **end while**
6:     Add a block of event e and probability $= \Delta r / blockSize$
7: **else**
8:     **while** $\Delta r < 0$ **do**
9:         Find a block b which contains rate of e
10:         **if** The rate of e in b >Δr **then**
11:             Reduce the rate of e in b by Δr
12:             $\Delta r \leftarrow 0$
13:         **else**
14:             Remove e from block b
15:             Increase Δr by the rate of e in b.
16:         **end if**
17:     **end while**
18: **end if**

same data sets of rates for the experiments. These data sets are randomly generated using an uniform distribution. To reduce the influence of a lucky data set we run each of the experiments with 10 different seeds. This way each algorithm will be run on the same 10 data sets.

As some of the algorithms use random numbers we run each of the experiments 10000 times for each data set. This serves two purposes: First of all it reduces the impact of spikes in cpu load on the running time. Secondly it reduces the influence of some random seed.

For each of the experiments we will be looking at the scaling of algorithm on three parameters: The number of events in the data structure, the average rate of events and the variance in rate of the events. To manipulate the average rate of events we scale the minimum and maximum rate by the same amount. To manipulate the variance we start with all variates with a fixed rate. Each following experiment will have a bigger difference between the minimum and maximum rate. By scaling the minimum and maximum values by the same amount we maintain the same average rate.

The first set of experiments will look at the time required to build the data structure. As all of the algorithms are bounded by $O(n)$ running time while setting up the data structure we will be scaling N this experiment linearly. We do the same for the average rate and variance.

The second set of experiments aims at comparing the time required to generate variates. To see some difference between algorithms we will be scaling the number of events exponentially. This allows us to see some difference between algorithms which scale logarithmic and those that are constant. The average

rate and variates are still scaled linearly.

The last set looks at the time required to change the rate of events. The scaling for these will be the same as those for the generation time.

The algorithms were implemented in python 2.7. To time the running time of each algorithm we used the timeit module. As this module disables garbage collecting we can our experiments in batches of 100 runs of the algorithm. After 100 runs we restarted the program to clean up old memory. The implementation of the algorithms is available on https://github.com/randoomed/DiscreteRandomVariates. The computer used for the experiments was an Lenovo g505 laptop running Ubuntu 18.04. This laptop has an A4-5000 CPU with 4gb memory.

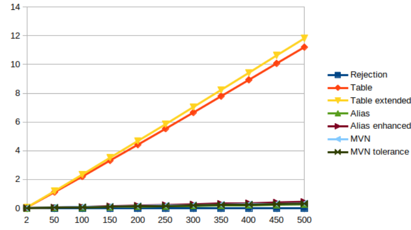
5 Results

5.1 Building

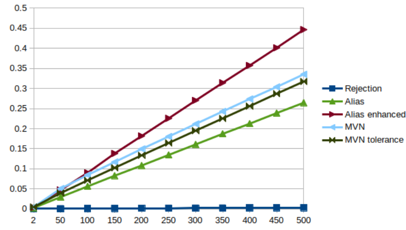
For our first set of experiments we look at the time required to build our algorithms and how this scales based on the number of events. The build times for the algorithms are shown in fig 4a. Clearly the both the Table method and Extended Table method take more time to build than the others. This was expected based on the theoretical running time of $O(N \cdot r_{ave})$. As the range for rates for this experiment was $1 - 10000$ this adds on average an 5000x multiplier to the running time. We included Fig 4b to show the details on the scaling on the other algorithms. The first thing that really stands out in this figure is that the Rejection method takes nearly no time to build. When building the Rejection method we only copy the array of events instead of building our own internal data structure. As this is an very common operation it is most likely implemented in a very optimized way in the compiler resulting in a very quick operation. The last interesting observation we can make from this graph is that the MVN algorithm with tolerances out performs the MVN algorithm without tolerances. This increase in performance is due to ranges have a lower threshold to become a root node.

Figure 5 shows the build time but this time when we scale the average rate of the events. The lowest average was 6 where the events could have a rate between 1 and 11. The highest average was 50, with an minimum value of 45 and maximum of 55. Figure 5a clearly shows that the Table method scale linearly with the average rate until the average reaches 15 after which it slows down. After 15 every 2 steps add 10 to the average which is exactly equal to the base. This way it only adds on average one item to each event. The next slowdown would probably happen around an average rate of 100. The only other algorithm that changes with an increase in rate is the MVN algorithm. With a higher average all events consolidate into one or two ranges. This slightly speeds up the algorithm.

Our final experiments on build times scales the variance of the rates. The average rate during all the runs was 5000. The minimum and maximum values scaled from 0 away from the average up to 4500 away from the average.

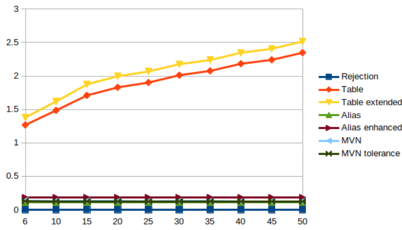


(a) The Table method and Extended Table method are significantly slower than the other algorithms.

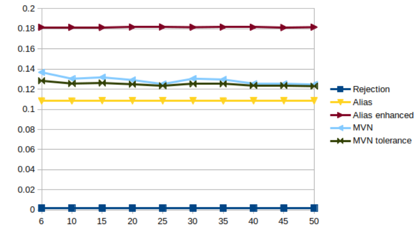


(b) Most other algorithms scale at a similar rate with the Rejection method being the only exception. while the Rejection method still scales with the number of items, it does so very slowly.

Figure 4: This figure shows the time required to build the data structures for each of the algorithms. The X-axis show the number of events in the data structure. The Y-axis show the average time in seconds used to build the data structures. The Table method is significantly shower to build due to it scaling with the rate of the events. The Rejection method on the other hand

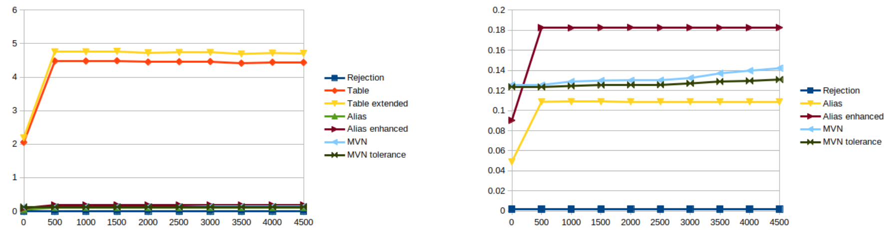


(a) The Table method scales with the average rate of events. At an average rate of 15 the scaling slows down due to the fact that a rate of 10 only using one item. If this graph was extended we would see another slowdown around an average of 100.



(b) The other algorithms do not scale significantly with the average rate. The MVN algorithm gets slightly faster due to more events being compacted in a single range at higher rates.

Figure 5: These figures show the time required to build the data structured. The x-axis shows the average rate of events in the data structure. The maximum value of an event was 5 higher than the average, the minimum rate was 5 below than the average.



(a) The Table method is significantly slower than other algorithms at building, but it does not scale with the variance in rate. The large increase in speed at variance= 0 is due to this point being exactly at 5000. this only adds 5 items to the data structure for each event.

(b) The Alias method builds very fast at 0 variance. This is because each event fills exactly 1 block. The MVN algorithm becomes slightly slower when the variance increases. With a high variance the algorithm has to create more ranges to contain all events.

Figure 6: These figures show the time required to build the data structured. The x-axis shows the average rate of events in the data structure. The maximum value of an event was 5 higher than the average, the minimum rate was 5 below than the average.

Mostly, the Alias algorithms and Table methods got an large increase in speed with an variance of 0. For the Alias algorithms this was because all items were equal to the average so each event turned into a block. The Table method preforms well at 0 variance because it only has to add 5 items to the level representing a rate of 1000 for each event.

The MVN algorithms seem to suffer a bit when the largest and lowest values are far apart. This causes the algorithm to make some extra ranges slowing the build time down.

5.2 Generating

The second set of experiments looks at the time required to generate an variate from the various data structures. The first of these experiments examines how the generation time scale based on the number of events in the data structure. In this experiment we can see that the MVN algorithm is the slowest to generate an variate. It also scales faster with the number of events than any of the other algorithms. That the MVN algorithm scales worse than the others corresponds to the theoretical result. In the second experiment on generating variates we look at scaling based on the average rate. While none of the algorithms scale directly based on the average rate the Rejection method actually becomes more efficient. While the variance of the rates does not change, the percentage difference between the maximum and minimum value decreases. The Table method also shows a small improvement in efficiency as the average rate grows. This can be explained by more of the rate being concentrated in the highest level of

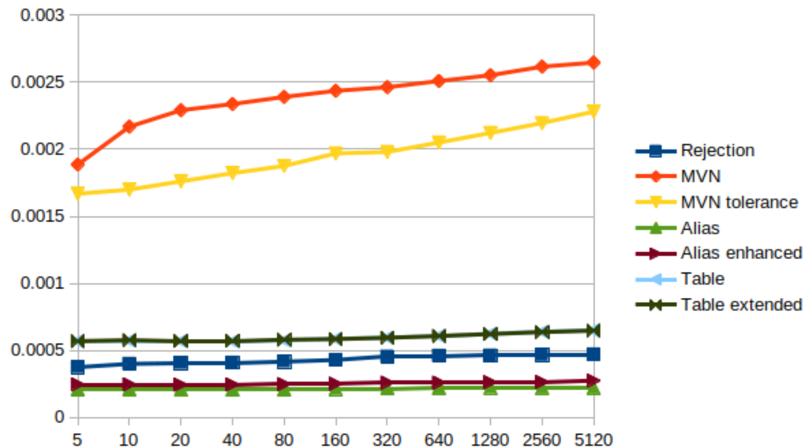


Figure 7: This graph hows how the time required to generate a variate scales with the number of items in the data structure. The Y-axis show the time required to generate 100 variates in seconds. The X-axis shows the number of items in the data structure, note that this axis follows an exponential scale. The MVN algorithms scale the worse of the algorithms.

the data structure. When an variate is generated we check the data structure from high to low, so when more of the rate is in the higher levels we select a level faster. The MVN algorithm also shows some interesting spikes. These spikes generally correspond to powers of two. At these average rates the events get split over multiple ranges. This effect is reduced when we use tolerances with the MVN algorithm. The final experiment that looks at generating variates look at the variance of rates. The two interesting results again come from the MVN and Rejection algorithms. The Rejection algorithm suffers from an increase in variance. The MVN algorithm has a similar problem where, when the rates have a high variance, the algorithm has to make a lot of ranges to encapsulate all variates.

We want to end this section with a general note on the performance of the Alias Enhanced algorithm. All experiments were ran right after generating the data structure. This gives it a performance similar to the original Alias algorithm. In a later section we will look how the algorithm scales as more updates are done before generating the variates. The speed at which the data structure degrades depends on the number of events in the data structure. Figure 10 shows the average generation time for data structures with 5, 10, 15 and 20 events. The X-axis shows the number of updates done before generating the variate. From this graph we can see that the generation time scales linearly with the number of updates done.

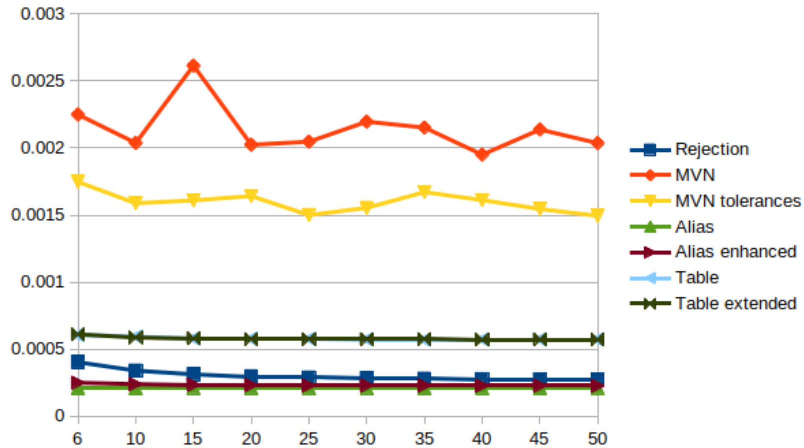


Figure 8: The graph shows how the time required to generate a variate scales with the average rate of the events. The MVN, Rejection and Table method all get a slight improvement in efficiency when the average rate increases while the variance stays the same.

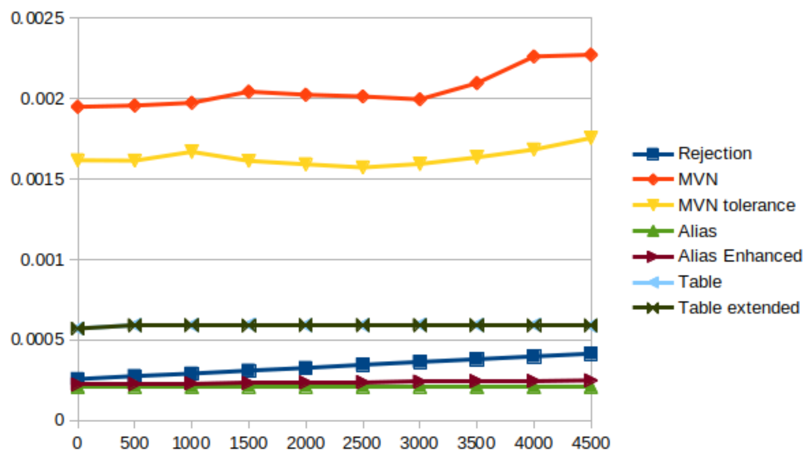


Figure 9: The generation time of the Rejection method and MVN scale with the variance in rate. For the MVN algorithm we see large spikes when a new power of two gets included in the range of possible rates. The Rejection method suffers from the difference between r_min and r_max increasing.

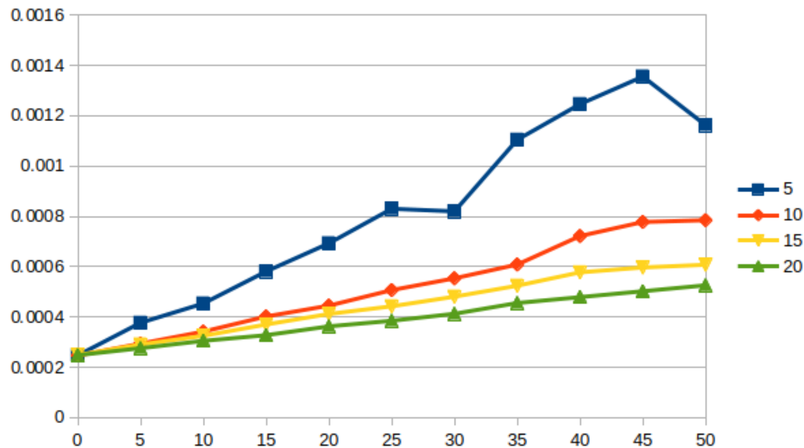


Figure 10: The Alias Enhanced data structure degrades as more updates are done on the data structure. This graph shows the generation time of the Alias Enhanced data structure after a certain number of updates. Each line represents a different number of items in the data structure. The graph clearly shows that if the data structure contains more items the impact of updates is lower.

5.3 Updating

For our last set of experiments we look at updating the events in the data structure. Just like the previous experiments we compare the algorithms based on their running time and how they scale with the number of events, average rate and variance of the rates. The graph in Figure 11 shows the results when we scale the number of events. From this graph its clear that the Table method is significantly slower than the other algorithms. The Table method shows a slight increase in running time up to $N = 2560$. We suspect this is due to the test method. In the experiment we ran batches of 100 updates on the same data structure before restarting. As the number of events increased the chance for an update to create new arrays to fit the new data also increased. We suspect this bump would vanish if we increased the number of updates per batch. The MVN method shows an more interesting trend. It became more efficient as the number of event increased. We think this happens because less ranges have to be created when an update happens as these ranges already exist. When an range already exists we only have to add the lower range/event.

For the experiment where we scale the average rate of events we ran two experiments. The first experiment scaled the average rate between 6 and 50, the second look at the average rates between 1000 and 10000. Both experiments show similar results. Like before the Table method is quite a bit slower than the others. This experiment also clearly shows that the Table method scales logarithmic with the average rate of events. The MVN algorithm shows some peeks. If we look closer at the data we notice that these peeks generally match

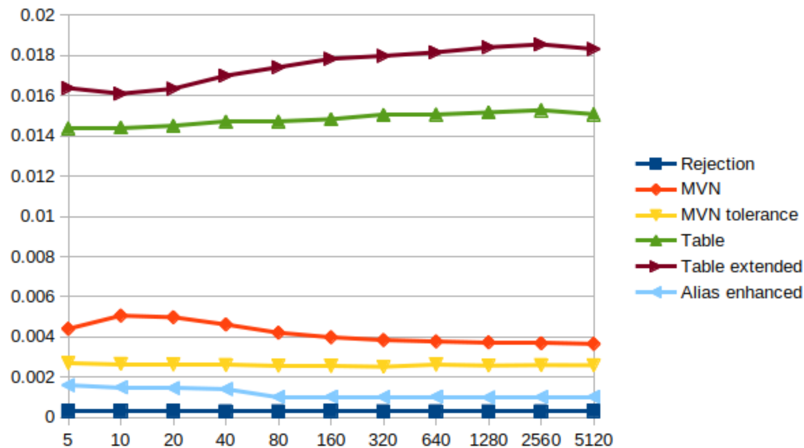


Figure 11: This graph shows how the time to do an update on the data structure scales with the number of events. The MVN algorithm becomes more efficient due to more ranges already existing. Thus less ranges have to be created and deleted. The increase on the Table method is most likely due to a similar effect where it becomes more likely that an update happens on a level that does not exist yet for an event, thus requiring us to build the data structure on that level for that event.

an average of a power of two. This can be a problem because it makes it more likely for an event to switch from one range to another. Thus requires the algorithm to look at more ranges.

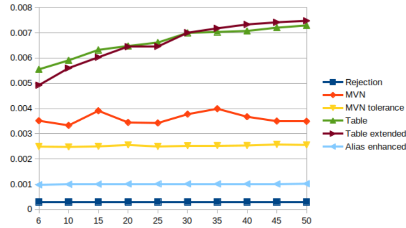
The last parameter we look at is the variance of the updates. In practice this means that the difference in rate for the update increases. The only algorithm that responds to this change is the Table method. The Table method seems to scale linearly with the size of the update.

6 Conclusions

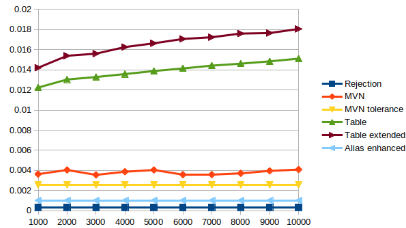
We proposed two extensions to algorithms for generate discrete random variates where the distribution does not change over time. Both of these algorithms can compete with the fastest known algorithms in terms of theoretical time scaling.

To really compare our extended algorithms to some of the fastest known solutions for this problem we implemented them. We also build implementations for the Rejection method and MVN algorithm. Using these implementations we ran a number of experiments comparing the build, update and generation time for these algorithms on the same data set.

The Rejection method was by fast the fastest of the algorithms to be build and to update due to the simplicity of these operations. It does however suffer a bit when the relative variance of the rate of the event increases.



(a) This graph shows the effect when we scale the average update from size 5 to size 50. With nice peeks around 15 and $30 - 35$



(b) This graph shows the scaling from 1000 to 10000

Figure 12: These figures show how the algorithms scale with the average rate of an update. The Table method scales according to a logarithmic function as more levels are changed. The MVN algorithm shows peeks around the powers of two where more events switch ranges.

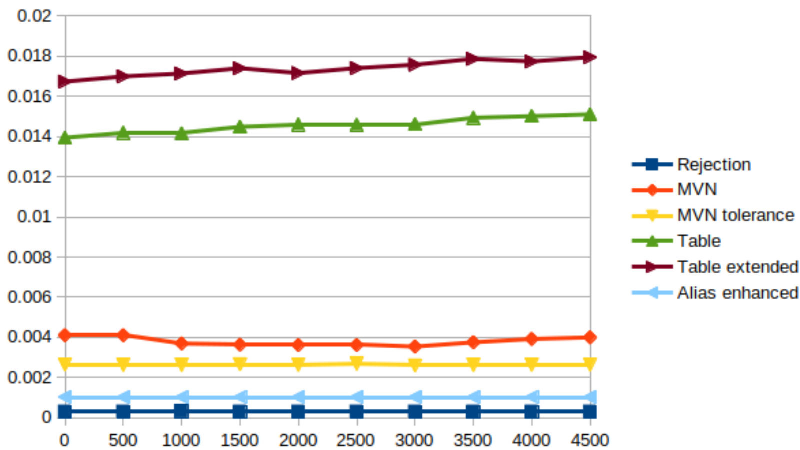


Figure 13: The graph shows how the running time of updates to the algorithms scale with the variance of possible rate values in updates. The Table method is the only algorithm that scales with an increase in variance. A higher variance results in on average larger updates. This hurts the running time of the Table method.

In our experiments the MVN algorithm shows a very consistent all round performance requiring around the same time to generate variates as for updates. While its generation time was slightly slower than the other algorithms it was faster to build and update than most.

The extension to the Table method managed to out perform the MVN algorithm in generation time. Unfortunately due to the the algorithm using items to represent rate it was rather slow to build and update as it had to do a relatively large amount of operations. In our experiments the Table method without modifications to get $O(\Delta r)$ update time out performs the one with this modification. While its a nice theoretical result, the modification makes the algorithm more complicated. This extra logic reduces the practical running time of the algorithm in favor of more consistency. We do not recommend implementing this modification when using the Table method.

The experiments show that our extension to the Alias method can, in an optimal situation, out perform any of the other algorithm in terms of generation time. At the same time it can also be updated and build in time similar to the other algorithms. It does however suffer an increase in generation time if multiple updates are done on the algorithm before generating an variate. We think this weakness can be resolved with further research by improving the way updates done. By filling up empty space in blocks updates might actually improve the generation time.

7 Future research

We believe that the Alias Enhanced algorithm can be further improved. In this paper we used a very basic update scheme that, while being fast, also increased the number of empty space in the algorithm. As the algorithm restart anytime a blank space is selected while generating an variates this reduces its performance. By improving the update scheme we expect it to be possible to reduce the impact of updates such that the algorithm will never have to rebuild its data structure.

It would also be interesting to see how the algorithms in this paper respond to different probability distributions. For the Table method we shortly looked at the case where updates were always of size one. Both of these subjects might show interesting results. This research might take the form of theoretical analysis similar to the work of D'Ambrosio et al. or might be the result of experimental research like in this paper.

References

[1] F. D'Ambrosio, H. L. Bodlaender, and G. T. Barkema. Dynamic sampling from a discrete probability distribution with a known distribution of rates. 2019. Unpublished paper.

[2] D. T. Gillespie. Exact stochastic simulation of coupled chemical reactions. *Journal of Physical Chemistry*, 81(25):2340–2361, 1977.

[3] G. Marsaglia. Generating discrete random variables in a computer. *Communications of the ACM*, 6(1):37–38, 1963.

[4] Y. Matias, J. S. Vitter, and W. C. Ni. Dynamic generation of discrete random variates. *Theory of Computing Systems*, 36(4):329–357, 2003.

[5] P. M. Maurer. Finite random variates using differential search trees. In *Proceedings of the Summer Simulation Multi-Conference*, SummerSim '17, pages 28:1–28:12, San Diego, CA, USA, 2017. Society for Computer Simulation International.

[6] A. Slepoy, A. P. Thompson, and S. J. Plimpton. A constant-time kinetic Monte Carlo algorithm for simulation of large biochemical reaction networks. *Journal of Chemical Physics*, 128(20):1–8, 2008.

[7] J. von Neumann and G. E. Forsythe. Various techniques used in connection with random digits. *National Bureau of Standards, Applied Math Series*, 38(12):36–38, 1951.

[8] M.D. Vose. A linear algorithm for generating random numbers with a given distribution. *IEEE Transactions on Software Engineering*, 17(9):972–975, 1991.

[9] A.J. Walker. An efficient method for generating discrete random variables with general distributions. *ACM Transactions on Mathematical Software*, 3(3):253–256, 2002.

[10] J. Wang, W. W. Tsang, and G. Marsaglia. Fast generation of discrete random variables. *Journal of Statistical Software*, 11(i03), 2004.

[11] C. K. Wong and M. C. Easton. An efficient method for weighted sampling without replacement. *SIAM Journal on Computing*, 9(1):111–113, 1980.