Modulated jump-counting patterns - a method for conditionally skipping different types of objects

Matthew Bentley

May 2, 2021

Abstract

Modulated jump-counting patterns transform low and high-complexity jump-counting patterns [1] [2] into a form which may be more appropriate for some types of computational work. They allow for skipping different types of elements based on preset parameters. This includes elements with binary characteristics and greater-than-binary characteristics eg. 3 or more types. Familiarity with both the low and high-complexity jump-counting patterns is assumed in the reader of this paper.

1 Introduction

The low-complexity and high-complexity jump-counting patterns are replacements for boolean skipfields which have greater iterative efficiency. These have been used in the colony [3] container. These patterns are well-established in their own papers and require no further explanation here, however if one is unfamiliar with them they will need to familarise themselves before continuing reading.

Modulated jump-counting patterns take the normally empty space within jump-counting skipfields created by un-skipped elements in those patterns, and creates skip-blocks for those areas as well as for the skipped areas. This enables the use of these

skipfields to not just skip one specific type of element, but to selectively skip either type.

A further extension of these patterns allows for skipfields which support more than two states ie. non-binary skipfields. This can, for example, enable selective processing of one or more types out of potentially millions of different element states. We will first cover the more familiar binary implementation of a skipfield, then move on to non-binary skipfields.

2 Definitions

For the purposes of this document, the following terms are defined:

LCJC: Acronym of 'low complexity jump-counting'.

HCJC: Acronym of 'high complexity jump-counting'.

MJC: Acronym of 'modulated jump-counting'.

Skipfield: An array of integers used to skip over certain objects in an accompanying data structure during iteration or processing. Denoted by S in equations.

Node: A single integer within a skipfield, corresponding to some external object which may or may not be skipped, depending on the state of the skipfield. For a boolean skipfield this would be 0 or 1, while for a jump-counting skipfield this can be any non-negative integer.

Skipblock: A contiguous sequence of skipped nodes within a skipfield. In the following LCJC skipfield:

0 4 0 0 4 0 0 0 1 0

all of the nodes from the first '4' to the second '4' are part of a skipblock. In addition the '1' is also a skipblock (of length = 1). In a boolean skipfield, any node which is '1' instead of '0' is part of a skipblock. For example:

1 1 1 0 0 0 1 1 0 0

Here the first 3 nodes are part of a skipblock of length 3. The 7th and 8th nodes are part of a skipblock of length 2.

Run: Another name for a contiguous sequence of objects which share a particular property eg. green objects vs red objects, or active objects vs erased objects.

Runblock: Another name for a skipblock, modified thus because all areas of a MJC skipfield are alternatively skippable or traversable, hence the name 'skipblock' becomes misleading.

Start node: The first node in any skipblock/runblock. In the LCJC skipfield above the first '4' is a start node, as is the '1'. In the Boolean skipfield above both the first and seventh nodes are start nodes. The index of this node is denoted by s in equations.

End node: The last node in any skipblock/runblock. In the LCJC skipfield above the second '4' is a start node, as is the '1'. In the Boolean skipfield above the third node is an end node, as is the 8th. The index of this node is denoted by e in equations.

Middle node: Any node in a skipblock/runblock which is not a start or end node. In the LCJC skipfield example above the third and fourth nodes are middle nodes. In the Boolean skipfield example above only the second node is a middle node.

Current node: The node currently being modified during processing. The index of this node is denoted by i in equations. **Zeroth node**: A node occurring before the skipfield which denotes the type of the first runblock in the skipfield, or 'type descrip-

tor node'. Detailed in the following sections. **Back node**: A node occurring after the skipfield which denotes the type of the last runblock in the skipfield. Another 'type descriptor node'. Detailed in the following sections.

3 Binary modulated skipfields

Take the LCJC skipfield below:

2 2 0 0 0 5 0 0 0 5

The equivalent binary modulated LCJC skipfield would be:

1 2 2 3 0 3 5 0 0 0 5 1

The first thing you will notice is that all previously unskipped areas eg. nodes 3-5, now have their own skipblocks. You will also note that there is now both a leading and trailing '1' on the skipfield. For the remainder of this paper we describe these as being the 'zeroth node' and 'back node' respectively. They are 'type descriptor nodes'.

The zeroth node describes the type of the first skipblock - 0 for one type of object (which could be erased objects, or something else), 1 for the opposite type. Likewise the back node describes the type of the last skipblock and is used for iterating in reverse (if reverse iteration is undesired, this node may be omitted).

How is this useful? Well, imagine if you will a scenario where we are not using jump-counting skipfields for, as is common, skipping erased or otherwise de-activated objects. Think instead of the colours red and green. We want to skip red objects, but not green objects. This is easily accomplished with a regular LCJC/HCJC. But consider the opposite situation - if, while using the same skipfield, we decide that we only want to process red objects now, we cannot do so while achieving O(1) iteration time.

To do that, we need a modulated jump-counting skipfield. Consider the modulated LCJC below:

objects :		r	r	r	r	g	g	g	r	r	g
skipfield :	1	4	0	0	4	3	0	3	2	2	1	0

Here we have assigned 1 to red, and 0 to green. The zeroth node of the skipfield describes a red initial skipblock, and the back node describes a green final skipblock. In between those two skipblocks we have another green and another red skipblock. Here the binary modulated LCJC/HCJC gives the ability to choose which type of object we are interested in skipping over.

Using the example above, if we decide to iterate over red objects, we know from the zeroth node that the first skipblock is red, and so we iterate over it rather than skipping it. Since this is a binary sequence, we know that a green skipblock must follow a red skipblock, so the next skipblock is skipped over using it's start node's value as the length of the run to skip, in standard LCJC/HCJC jump-counting fashion. At this point we will use the term 'runblock' instead of skipblock since these areas are not necessarily skipped, depending on which type of object one is interested in iterating over.

3.1 Forward iteration

1. We establish 0 as denoting one type of object while 1 denotes the other. This is the type number. We decide which type of object to we wish to iterate over, and set t to that type's number.

2. To initialise our iterator, we set the current skipfield index (i) to 1, and read the zeroth node's value.

3. If the zeroth node is not equal to t, the first run in the skipfield is of objects we wish to skip over, so we add the value of the skipfield node at index 1 to i.

4. We then read the skipfield node's value at index i, add 1 to it and store the result as j. Now the iterator is in the beginning position and we are ready for iteration, as follows:

5. We add 1 to i.

6. If $i = j$, we read the skipfield at index i, add this to i, then re-read the skipfield at the new index i, add i to it and store the result as j.

7. To iterate again we return to step 5.

In pseudocode this can be expressed as follows (where S denotes the skipfield and i the current index into the skipfield):

// Iterator initialisation:

$$i := 1 \tag{1}$$

$$IF\ (S_0\ !=\ t)\ THEN\ i := 1 + S_1 \tag{2}$$

$$j\ := i + S_i \tag{3}$$

// Single iteration:

$$i := i + 1 \tag{4}$$

$$IF\ (i = j)\ THEN\ \{ \tag{5}$$

$$i := i + S_i \tag{6}$$

$$j := i + S_i\ \} \tag{7}$$

You will notice that although per-iteration this equation involves fewer reads of the skipfield than the unmodulated LCJC/HCJC patterns, it also contains more instructions and a branch, which the unmodulated patterns do not have (LCJC/HCJC iteration has no branches and is only 2 instructions). This means it could be slower than unmodulated LCJC/HCJC iteration depending on the platform.

It is important to note that increased iterative performance over the unmodulated LCJC/HCLC skipfields is not the purpose of a modulated skipfield; the purpose is to enable the ability to choose which types of elements we wish to skip over, as opposed

to having one predetermined type.

One might question the advantage of a MJC over a boolean skipfield, given that the latter can also be used for this sort of selective processing and its iteration also involves branching. The advantages are (a) a MJC skipfield involves no looping and hence (b) iteration skips the skipped elements in O(1) time, as opposed to a boolean skipfield's undefined time complexity for the same operation.

For example, a single iteration over a boolean skipfield could involve 1 or 1000 branching instructions depending on the state of the skipfield and how many skipped objects there are between any two unskipped objects. A MJC skipfield will only ever invoke 1 branching instruction between any two unskipped objects.

3.2 Reverse iteration

1. t is set to the number of the type we wish to iterate over.

2. To initialise our iterator, we set the current skipfield index (i) to the index of the back node (b) minus 1, and read the back node's value.

3. If the back node is not equal to t, the last run in the skipfield is of objects we wish to skip over, so we subtract the value of the skipfield node at index i from i.

4. We then read the skipfield node's value at index i, subtract that number from i and store the result as j. Now the iterator is in the beginning position and we are ready for iteration, as follows:

5. We subtract 1 from i.

6. If $i = j$, we read the skipfield at index i, subtract this from i, then re-read the skipfield at the new index i, subtract this from i and store the result as j.

7. To iterate again we return to step 5.

In pseudocode this can be expressed as follows:

// Iterator initialisation:

$$i := b - 1 \tag{8}$$

$$IF\ (S_0\ !=\ t)\ THEN\ i := i - S_i \tag{9}$$

$$j\ := i - S_i \tag{10}$$

// Single iteration:

$$i := i - 1 \tag{11}$$

$$IF\ (i = j)\ THEN\ \{ \tag{12}$$

$$i := i - S_i \tag{13}$$

$$j\ := i - S_i\ \} \tag{14}$$

4 Non-binary modulated skipfields

Consider the problem of adding blue elements to our set of objects. Even if we modified the binary above code to allow for 3 types of object instead of 2, we would still have to assume a set sequence of runblocks in the skipfield - type 0 (green) followed by type 1 (red) followed by type 2 (blue), followed by type 0 etcetera. Otherwise alternating between them does not work. For example the following sequence would be perfectly valid:

$objects:$		r	r	r	r	b	b	b	g	g	r	
$skipfield:$	1	4	0	0	4	3	0	3	2	2	1	0

But the following sequence would not work:

$objects:$		r	r	r	r	g	g	g	b	b	r	
$skipfield:$	1	4	0	0	4	3	0	3	2	2	1	0

This is because blue does not follow the established sequence in the second example, so therefore an iterator would assume that the green objects were blue and the blue objects were green.

There is a way around this, which is to utilise the upper or lower n number of bits in each skipfield node, where n is the number of bits necessary to represent the total number of different object types (eg. 2 bits in the example above). These bits would describe the type of object in each runblock, and would be stored in the start and end nodes.

This reduces the potential length of a run, based on the bit-depth of the skipfield. For example, a skipfield made up of unsigned 8-bit integers would be able to describe runs of up to 255 objects, but if 2 of those bits are used to describe the type of object in the run, the maximum length of the described run is 63 (note: though in fact, since a runblock cannot be of length 0, we could simply assume that the value represented by all start and end nodes has 1 subtracted from it, thereby increasing the maximum run-length to 64 in this example).

In this case there is no need for zeroth or back nodes in the skipfield to describe the type of object in the first and last runblocks, as is necessary in the binary MJC patterns. These are already be encoded in the start and end skipfield nodes for both of those runblocks.

If we interpret the lower 2 bits in our skipfield nodes as the element type for the above example, this shifts the actual run-length numbers up by 2 bits, resulting in the latter:

$objects:$	r	r	r	r	g	g	g	b	b	r
$skipfield:$	16	0	0	16	12	0	12	8	8	4

Once we add the element type data into the lower 2 bits (0 = green, 1 = red, blue = 2), this becomes:

$objects:$	r	r	r	r	g	g	g	b	b	r
$skipfield:$	17	0	0	17	12	0	12	10	10	5

This means that our red, green and blue runblocks can follow any possible sequence. Of course, this adds overhead to both the writing and reading of the skipfield due to the shifting and masking operations necessary to separate the object type numbers from the run-length numbers. But, dependent on the situation this trade-off could be

worth the additional instructions.

The potential number of different object types can be as large as one has room to store in the skipfield integer type. A 32-bit unsigned integer skipfield could, for example, sacrifice 10 bits per node in order to be able to describe 1024 different types of element, while retaining a maximum runblock length of 4194304. While a 16-bit unsigned integer skipfield could sacrifice 3 bits per node in order to describe 8 different types of element while retaining maximum runblock lengths of 8182.

4.1 Memory-saving measures

Since the non-binary modulated pattern describes the object type at the beginning and end of each run, it is possible to have several sequential runblocks of the same object type. For example, in the non-binary modulated pattern the following LCJC skipfield is completely valid:

$objects:$	r	r	r	r	r	r	r	b	b	r
$skipfield:$	17	0	0	17	12	0	12	10	10	5

In the binary modulated pattern this would not be possible since it would be assumed that we are switching object type at the end of each run. What this means is, for the non-binary modulated pattern we can use a lower bit-width for the skipfield type and simply have multiple sequential runblocks when the runblock length encountered is larger than the skipfield node's bit-width can describe.

If we have a skipfield bit-width of 8 bits for example, and we use 2 of those bits to describe the object type for each runblock, that limits each run to 63 objects, however we could place several runblocks in a row to effectively create runs which are unlimited in length. This would increase the number of instructions incurred per full iteration over the skipfield, but might reduce the size of the skipfield substantially, and depending on the typical length of runs in a given scenario, may not affect performance at all.

5 Applications

It is expected that MJC skipfields would be most useful in the scenario where we wish to iterate over all objects in a sequence at times, and over only certain types of objects in that sequence at other times. An example could be a very specific form of multiset, where only certain types of objects are to be selected a given point in time, but it is also possible to iterate over all objects in the sequence.

6 Modification of singular skipfield nodes

Largely, skipfield modification follows the same rules that are already established in the LCJC and HCJC pattern papers. Up until this point we have used LCJC patterns to demonstrate the MJC pattern, but either can be used. It is merely easier to update and describe the low-complexity variant, so that is what we will continue to use in this section for the most part. Those familiar with the HCJC pattern can easily extrapolate this information to create equivalent operations for that.

In addition, since non-binary MJC skipfields involve so many additional calculations, and since the layout is changed significantly by the requirement of needing to storing object type information in the upper or lower bits of nodes, we will explain node modification only in the context of a binary MJC skipfield. The reader is assumed to be able to extrapolate this information to create equivalent operations for the non-binary variant.

For a LCJC, one must know the location of the start or end node of the runblock which contains the node we wish to change - either by having iterated over the skipfield or some other method. All modification operations involve either splitting runblocks (for middle nodes) or truncating runblocks. We'll start with middle nodes.

6.1 Middle node modification

Take the following modulated binary LCJC skipfield:

$objects:$		r	r	r	g	g	g	g	g	r
$skipfield:$	1	3	0	3	5	0	0	0	5	1	1

Say we wish to change the 2nd node from red to green. At this point we subtract the index of this runblocks start node from the current nodes index, to obtain the distance from the start node to the current node. We store this as j, then set the value of the start node, and the node to the left of the current node, to j. In this case j is 1, so we end up writing to the start node twice. We then set the current node's value to 1:

$objects:$		r	g	r	g	g	g	g	g	r
$skipfield:$	1	1	1	3	5	0	0	0	5	1	1

Lastly we subtract the index of the current node from the end node to find the distance between the current node and the end node. We store this as k, then write k to both the end node, and the node to the right of the current node. Again k is 1 here, so we write to the end node twice:

$objects:$		r	g	r	g	g	g	g	g	r
$skipfield:$	1	1	1	1	5	0	0	0	5	1	1

The process is the same for a HCJC skipfield except that the middle nodes of the two new runblocks on the left and right must be set to the sub-index of the middle node in the context of the runblock (ie. the 2 + n pattern):

$objects:$		r	g	r	g	g	g	g	g	r
$skipfield:$	1	1	1	1	5	2	3	4	5	1	1

For the above modification example there are no middle nodes in the left or right runblocks resulting from the runblock split, so nothing changes in that area. However if we were to change the 7th node from green to red, the result would be as follows:

$objects:$		r	g	r	g	g	g	r	g	r
$skipfield:$	1	1	1	3	3	2	3	1	1	1	1

6.2 Start node modification

In this case, for a binary modulated skipfield, we are truncating the size of the runblock which the start node belongs to, and increasing the size of any adjacent runblock to the left. There are four possible situations here:

1. Node is at start of skipfield and is equal to 1.
2. Node is at start of skipfield and is not equal to 1.
3. Node is not at start of skipfield and is equal to 1.
4. Node is not at start of skipfield and is not equal to 1.

We'll deal with each of these in turn:

6.2.1 Node is at start of skipfield and is equal to 1

In this case we have a single-node runblock. We invert the zeroth node, take the value of the right-hand node, add 1 to it, and set the current node to that value. We then use the current node's value as the index of the end node in the right-hand runblock, and set that node to the same value as the current node. For example, with the skipfield below:

$objects:$		r	g	r	g	g	g	r	g	r
$skipfield:$	1	1	1	3	3	2	3	1	1	1	1

Changing the 1st node of the skipfield to green results in the following:

$objects:$		g	g	r	g	g	g	r	g	r
$skipfield:$	0	2	2	3	3	2	3	1	1	1	1

In equation form this is:

$$S_0 = !(S_0) \tag{15}$$

$$e = S_2 \tag{16}$$

$$S_e = S_1 = S_2 + 1 \tag{17}$$

6.2.2 Node is at start of skipfield and is not equal to 1

We invert the zeroth node, take the value of the current node, subtract 1 and store the result as j. We use the value of the current node as the index of the end node, and set both the end node and the right-hand node to j. For example, with the skipfield below:

$objects:$		r	r	r	g	g	g	r	g	r	
$skipfield:$	1	3	0	3	3	0	3	1	1	1	1

Changing the 1st node of the skipfield to green results in the following:

$objects:$		g	r	r	g	g	g	r	g	r	
$skipfield:$	0	1	2	2	3	0	3	1	1	1	1

In equation form this is:

$$S_0 = !(S_0) \tag{18}$$

$$j = S_1 - 1 \tag{19}$$

$$e = S_1 \tag{20}$$

$$S_e = S_1 = j \tag{21}$$

6.2.3 Node is not at start of skipfield and is equal to 1

In this case we have a single-node runblock in between two runblocks. We take the value of the left-hand node and subtract it from the current node's index to find the index of the lefthand runblock's start node (s). We then take the value of the right-

hand node and add it to the current node's index to find the index of the right-hand runblocks end node (e). Finally, we set the values of s and e to the value of s plus the value of e plus 1. For example:

$objects:$		r	g	g	g	g	g	r	g	r
$skipfield:$	1	1	5	0	0	0	5	1	1	1	1

Changing the 7th node above to green results in the following:

$objects:$		r	g	g	g	g	g	g	g	r
$skipfield:$	1	1	7	0	0	0	5	1	7	1	1

In equation form this is:

$$s = i - S_{i-1} \tag{22}$$

$$e = i + S_{i+1} \tag{23}$$

$$S_e = S_s = S_e + S_s + 1 \tag{24}$$

6.2.4 Node is not at start of skipfield and is not equal to 1

Here we take the current node's value, subtract 1 from it and store this as j. We add j to the current node's index to find the index of this runblocks end node (e), then set both the right-hand node and the end node to j. Finally we take the value of the left-hand node, subtract that from the current node's index to find the index of the left-hand runblocks start node (s). We then set both the current node and the left-hand runblocks start node to the value of the left-hand runblocks start node plus 1. For example:

$objects:$		r	r	r	g	g	g	r	g	r
$skipfield:$	1	3	0	3	3	0	3	1	1	1	1

Changing the 4th node of the skipfield below to red results in the following:

$objects:$		r	r	r	r	g	g	r	g	r
$skipfield:$	1	4	0	3	4	2	2	1	1	1	1

In equation form this is:

$$j = S_i - 1 \tag{25}$$

$$e = i + j \tag{26}$$

$$S_{i+1} = S_e = j \tag{27}$$

$$s = i - S_{i-1} \tag{28}$$

$$S_s = S_i = S_s + 1 \tag{29}$$

6.3 End node modification

In this case, for a binary modulated skipfield, we are truncating the size of the runblock which the end node belongs to, and increasing the size of any adjacent runblock to the right. There are four possible situations here:

1. Node is at end of skipfield and is equal to 1.
2. Node is at end of skipfield and is not equal to 1.
3. Node is not at end of skipfield and is equal to 1.
4. Node is not at end of skipfield and is not equal to 1.

6.3.1 Node is at end of skipfield and is equal to 1

In this case we have a single-node runblock. We invert the back node, take the value of the left-hand node, add 1 to it, and set the current node to that value. We then subtract the value of the left-hand node from the current node to find the index of the left-hand runblocks start node. We then set that node to the same value as the current node. For example, with the skipfield below:

$objects:$	r	g	r	g	g	g	r	g	r
$skipfield:$	1	1	1	3	3	2	3	1	1	1	1

Changing the last node of the skipfield to green results in the following:

$objects:$	g	g	r	g	g	g	r	g	g
$skipfield:$	0	2	2	3	3	2	3	1	2	2	0

In equation form this is:

$$S_b = !(S_b) \tag{30}$$

$$s = i - S_{i-1} \tag{31}$$

$$S_s = S_i = S_{i-1} + 1 \tag{32}$$

6.3.2 Node is at end of skipfield and is not equal to 1

We invert the back node, take the value of the current node, subtract 1 and store the result as j. We then subtract j from the index of the current node to find the start node, and set both the start node and the left-hand node to j. Finally we set the current node to 1. For example, with the skipfield below:

$objects:$	r	r	r	g	g	r	g	g	g
$skipfield:$	1	3	0	3	2	2	1	3	0	3	0

Changing the last node of the skipfield to green results in the following:

$objects:$	r	r	r	g	g	r	g	g	r
$skipfield:$	1	3	0	3	2	2	1	2	2	1	1

In equation form this is:

$$S_b = !(S_b) \tag{33}$$

$$j = S_i - 1 \tag{34}$$

$$s = i - j \tag{35}$$

$$S_s = S_{i-1} = j \tag{36}$$

$$S_i = 1 \tag{37}$$

6.3.3 Node is not at end of skipfield and is equal to 1

The procedure here is the same as with the start node. We have a single-node runblock in between two runblocks. We take the value of the left-hand node and subtract it from the current node's index to find the index of the lefthand runblock's start node (s). We then take the value of the right-hand node and add it to the current node's index to find the index of the right-hand runblocks end node (e). Finally, we set the values of s and e to the value of s plus the value of e plus 1. For example:

$objects:$		r	g	g	g	g	g	r	g	r
$skipfield:$	1	1	5	0	0	0	5	1	1	1	1

Changing the 7th node above to green results in the following:

$objects:$		r	g	g	g	g	g	g	g	r
$skipfield:$	1	1	7	0	0	0	5	1	7	1	1

In equation form this is:

$$s = i - S_{i-1} \tag{38}$$

$$e = i + S_{i+1} \tag{39}$$

$$S_e = S_s = S_e + S_s + 1 \tag{40}$$

6.3.4 Node is not at end of skipfield and is not equal to 1

Here we take the current node's value, subtract 1 from it and store this as j. We subtract j from the current node's index to find the index of this runblocks start node (s), then set both the left-hand node and the start node to j. Finally we take the value of the right-hand node, add that to the current node's index to find the index of the right-hand runblocks end node (e). We then set both the current node and the right-hand runblocks end node to the value of the right-hand runblocks end node plus 1. For example:

$objects:$		r	r	r	g	g	g	r	g	r	
$skipfield:$	1	3	0	3	3	0	3	1	1	1	1

Changing the 6th node of the skipfield below to red results in the following:

$objects:$		r	r	r	g	g	r	r	g	r	
$skipfield:$	1	3	0	3	2	2	2	2	1	1	1

In equation form this is:

$$j = S_i - 1 \tag{41}$$

$$s = i - j \tag{42}$$

$$S_{i-1} = S_s = j \tag{43}$$

$$e = i + S_{i+1} \tag{44}$$

$$S_e = S_i = S_e + 1 \tag{45}$$

7 Modification of multiple consecutive skipfield nodes at once

This becomes more complicated. If we have a range, say nodes 1-4 of the following LCJC sequence:

$objects:$		r	r	r	g	g	g	r	g	r
$skipfield:$	1	3	0	3	5	0	0	0	5	1	1

And we wish to convert this range to green, this is relatively simple - the first 3 nodes are a runblock of red, while the 4th node is the start node of a runblock of green. We negate add the number at node 1 to the number at node 4, set the 1st and 8th nodes to that new number, and the conversion is complete. Since the start of the range is the 1st node, we would also need to change the value of the zeroth node from 1 to 0.

However if we wished to convert nodes 2-5 of the above sequence instead, there is no way to establish from the values of the nodes themselves, where the beginning and end of those runblocks are. In this case, we would have to iterate from the very beginning of the container in order to locate the runblocks contained by the range, and modify them that way.

Alternatively one might modify the LCJC slightly so that middle nodes are always 0. This is not hard to do, but involves one or two additional instructions (see the Parallel Processing section of the low complexity jumnp-counting skipfield paper). In this case we could assume any two adjacent non-zero nodes are start and end nodes respectively of two separate runblocks, while zero nodes are always middle nodes. We could then scan to the left and right of the beginning and end nodes of the range to find runblocks.

One might think that a HCJC skipfield version would be better in this scenario:

$objects:$		r	r	r	g	g	g	r	g	r
$skipfield:$	1	3	2	3	5	2	3	4	5	1	1

But again it is impossible to establish whether the beginning and end nodes of a given range are start/middle/end nodes of the runblock, without scanning to the left and right (to find values which do not match the $S_x = S_{x-1} + 1$ pattern of the HCJC middle nodes. In either case, the modification of multiple consecutive nodes in a MJC skipfield is non-trivial.

However, if we specifically use iterators in the form described in the iteration section and retain i (the current index), t (the object type which we are interested in iterating over), s (the index of the current runblocks start node) and j (the index of the next runblocks start node), and use two of these iterators as a range for modifying nodes, it is possible to work out the beginning and end of each runblock involved and modify multiple consecutive nodes in this way.

8 Conclusion

It is expected that this pattern will find use in some form of data processing, though it is difficult to forecast what exactly. The non-binary variant of the pattern shows more promise as it is more flexible and also can save significant amounts of skipfield memory though bit-depth reduction. However in applications where types have a singular dominant binary characteristic, it would be of use in terms of processing both types of elements at times, then selectively processing only one type of element at other times. In those cases it would provide O(1) skipping of the other type of element, compared to a boolean skipfield, whose problematic aspects in terms of cache, branching and branch prediction [4] [5] [7] [6] have been fully examined in the low and high-complexity jump-counting papers.

9 References

References

[1] Bentley, Matthew R., *The low complexity jump-counting pattern*, `https://plflib.org/matt_bentley_-_the_low_complexity_jump-counting_pattern.pdf`, 2020.

[2] Bentley, Matthew R., *The high complexity jump-counting pattern*, `https://plflib.org/matt_bentley_-_the_high_complexity_jump-counting_`

pattern.pdf, 2019.

[3] Bentley, Matthew R., *Introduction of colony to the C++ standard library*, www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p0447r13.html, 2021.

[4] Arun Kejariwal, Center for Embedded Computer Systems University of California (Irvine, USA), Alexander V. Veidenbaum, Alexandru Nicolau, Xinmin Tian, Milind Girkar, Hideki Saito, Utpal Banerjee (2008), Comparative architectural characterization of SPEC CPU2000 and CPU2006 benchmarks on the intel® Core™ 2 Duo processor, Embedded Computer Systems: Architectures, Modeling, and Simulation, SAMOS.

[5] Babka, Vlastimil and Marek, Lukáš and Tuma, Petr (2009), *When misses differ: Investigating impact of cache misses on observed performance.*, Parallel and Distributed Systems (ICPADS), 2009 15th International Conference on, pp. 112-119. IEEE.

[6] Agner Fog (2017), Branch prediction in Intel Haswell, Broadwell and Skylake, The microarchitecture of Intel and AMD CPUs, http://www.agner.org/optimise/microarchitecture.pdf, pp.28.

[7] Claypool, Mark, and Kajal Claypool (2010). "Latency can kill: precision and deadline in online games." Proceedings of the first annual ACM SIGMM conference on Multimedia systems. ACM.