Partition Maps

Leonid Rozenberg*

Abstract

A partition map is a data structure to represent functions where we privilege merging, generating new functions, above other operations. I motivate the use of partition maps in lieu of other data structures and describe challenges in implementing the necessary logic.

1 Introduction

A mathematician can think of a partition map as a way to represent a function ($f : D \to R$), an association, a map. A programmer can think of it as a way to track, non-scalar state[1].

There are many data structures that one can use to represent functions, or state, such as arrays, association lists, trees, hash tables and variants of these. The deciding factor of which implementation to use depends upon the stored data and desired access pattern. Most data structures privilege value setting and getting; accessing and mutating the value associated with any key in the domain. A partition map, is a different technique, where we want to prioritize *merging* above other access patterns.

As a running motivating example, let D be the positive integers up to 100, and consider the functions

$$f_1(x) = \begin{cases} 1 & x \in [10, 80] \\ 0 & \text{otherwise,} \end{cases} \qquad f_2(x) = \begin{cases} 1 & x \in [20, 90] \\ 0 & \text{otherwise.} \end{cases}$$

Merging takes two functions and computes a new one such as

$$g(x) = f_1(x) + f_2(x) = \begin{cases} 0 & x \in [1, 9] \cup [91, 100] \\ 1 & x \in [10, 19] \cup [81, 90] \\ 2 & x \in [20, 80]. \end{cases}$$

The range, R, is specified by each function. It may vary, but for our purposes we want to emphasize that it is smaller than the domain. Lastly, we are not concerned with composition, cases such as $g(x) = f_j(f_i(x))$.

Briefly, as a point of comparison, consider the storage and merging costs for various data structures that could be used to model f_1, f_2 and then create g. An array, where we use one array position for each element in the domain, would require $O(|D|)$ storage and $O(|D|)$ evaluations for merging. This is the naive case. We can have similar performance using other data-structures (lists, trees and hash-tables) following the naive approach, one value per domain element, but with the extra overhead of pointer management.

*leonidr@gmail.com

[1] I will mix the two nomenclatures and ways of thinking as domain and range are particularly succinct and useful terms.

A couple of the conditions, present in the example, should highlight what is inefficient about the naive solution.

1. The size of the range (R) is much smaller than the domain (D). For f_1 or f_2 we have 2 elements vs 100, and for g the case is 3 vs 100. The specific relation of the two values is not as important as emphasizing that we want to create bounds proportional to $|R|$ as opposed to $|D|$.

2. Traditionally, when programmers are concerned with excessive or redundant evaluations, they will use memoization. In this case it is dubious that it will improve the situation as performing the calculation is so simple that it might be cheaper than looking up the result. Thus, the merging operation is *simple*[2].

3. The merging operation is *bounded*[3], it does not grow the size of the range excessively. An example of a function that grows excessively would be $h(x) = x + f_1$. If we know that all *future* merging operations are also bounded, we are even more motivated to merge in $O(|R|)$ time.

4. The domain is fixed. We know the full state for which we want to keep track of values and it does not change. Many algorithms that we might use instead assume, *a priori* that the user does not know the full domain, consequently concerning themselves with how it might grow or shrink.

Usually, we think about such problems by allocating space sufficient for our domain and then operate on each element therein. But if we know that the range is much smaller, it will not grow much in size, and the domain is known ahead of time, can we do better? Specifically, can we operate over the range elements, and add extra bookkeeping operations to track the domain elements, that will make the resulting code faster?

For each function that we specify, working backwards, the values in the range specify a partition of the domain. For example, for g, $S_0 = [1, 9] \cup [91, 100]$, $S_1 = [10, 19] \cup [81, 90]$, $S_2 = [20, 80]$ and $S_0 \cup S_1 \cup S_2 = D = [1, 100]$. I will refer to this as the partition *implied* by a function.

For the functional language enthusiast, the *merge* that I describe is commonly thought of as a *map2*; a higher order function that takes another function m that is applied to each element of two other structures, where the arguments to m, are chosen based on the internals of the data structures. I eschew that name to emphasize that something different, something dependent on the domain elements, will occur when we merge. Moreover, *map* will have a slightly different interpretation. When we *map*, we consider all of the unique elements of the range and apply a function to each of these values, with two caveats: the transform does not take a domain element as an argument and we store only the unique elements of the resulting range. Similarly, when we merge we will again ignore the domain elements when computing the new value, keep track of only the unique resulting values, but also take elements from the two ranges such that their respective domain sets have an intersection.

[2]These terms are unfortunately left imprecise at the moment as making them precise is part of the ongoing effort.

[3]Ibid.

2 Example applications

It is important to stress that these conditions are unique to the problem that I encountered and may not be present in other scenarios. To give more motivation for this method it might be worthwhile to consider potential applications.

2.1 Histograms

The functions that I used as my motivating examples might seem like toy examples without actual use. In practice, functions that are only a little bit more complicated, can arise when a user wants to track an empirical distribution with a histogram. Or even simpler uses such as frequency tables. One of the most common first steps in the construction of such tables is to decide the number of classes or bins; how to partition the domain. What if that was a question that was largely determined empirically?

This approach could have positive benefits in systems that want to integrate statistics from different observations, via such histograms. Distributed systems come to mind. For example, different routers might keep a map of packet size to latency (or other metric). But the domain space, packet size, could have a clustering effect where packets that have nearly the same size (eg. 100 or 105 bytes), have the same outcome. Or a large threshold effect, if a packet is larger than 1kb then it's metric is twice the value of those below. In both cases, a partition map would provide a sparser representation, and potentially simpler merging logic for aggregating algorithms.

2.2 Columnar Store

As another example consider merging columns of discrete values, in a table, such as a credit calculation. There are several columns all indexed by the same primary key UserId.

UserId	Education	Income Bracket	Age Group	Credit
1	High School	$< 10k$	< 20	Bad
2	High School	$< 10k$	< 20	Bad
...	...	...	...	...
103	High School	$< 10k$	20-30	Bad
104	High School	$< 10k$	20-30	Bad
...	...	...	...	...
206	High School	$10k - 50k$	20-30	Bad
...	...	...	...	...
1506	Bachelors	$50k - 100k$	20-30	Ok
...	...	...	...	...
11986	Bachelors	$> 200k$	30-40	Good
...	...	...	...	...
252321	Doctorate	$50k - 100k$	20-30	Ok
...	...	...	...	...

In this example, all of the inputs to our merge function (eg. Education can be one of "High School", "Bachelors" or "Doctorate"). and the output (Credit can be either Good, Medium or Bad) are discrete. If the individual datums are stored independently (eg. we have not already allocated the full table to store the data and result) a partition map approach might be warranted.

2.3 Original motivation

My objective, when developing partition maps, was to compute a likelihood function, a probability for a large set of genetic variants. The likelihood function was computed via a sequence of multiple recursions, in a dynamic programming approach.

$$
\begin{aligned}
M_{lkn} &= e_{lkn}(t_{MM}M_{l-1,k,n} + t_{IM}I_{l-1,k,n} + t_{DM}D_{l-1,k,n}) \\
I_{lkn} &= \frac{1}{4}(t_{MI}M_{l-1,k,n} + t_{II}I_{l-1,k,n}) \\
D_{lkn} &= t_{MD}M_{l,k,n} + t_{DD}D_{l,k,n}
\end{aligned}
$$

The genetic variants (the state/domain space indexed by n), were similar in many instances and the functions were bounded by their numerical accuracy. The calculations were also simple, a cross product of probabilities (M, I, D) and weights $(t_{MM}, t_{IM}, t_{DM} \ldots)$. The application had to perform this calculation several million times per sample. This was a big computational bottleneck and performing this computation via partition maps reduced the running time to less than 5% of the naive approach.

3 Related approaches

Before describing the implementation it would be helpful to quickly survey the literature of related data structures; to contrast how they do not meet the requirements and for inspiration. The common refrain is that they will make an undesired trade-off, where they favor look-ups versus merging.

3.1 Bidirectional maps

A bidirectional map[4], can be thought of as set whose elements are pairs that represent the association, coupled with lookup and alteration methods so that the set (maps) is (are) preserved regardless of which side is used as a key. For our purposes, naively, they would require storing an element of the domain. Non-naively, if one side was to represent a set of the implied partition, we are still willing to discard the convenience of look-ups for fast merges.

3.2 Fast Mergeable Integer Maps

Okasaki's classic description[4] of Patricia trees highlights how we can maximize the information contained within keys to build efficient data structures. Unfortunately, for our purposes, this technique has a slightly different interpretation of *merging*, combining disparate sets of keys and values that preserve fast lookup. We intent to merge two cases where values for all the keys are already known.

One could imagine mapping every set within a partition to a unique, large, integer by representing it as a bit vector (a bit per element), and then utilizing Patricia tree's as described in this work. The large key sizes ($O(|D|)$ bits), pose a substantial problem as the bit-twiddling necessary for look-ups is limited to a programs word size which might be a relatively small portion of $|D|$. Furthermore, the original fast lookup guarantees provided

[4] `https://en.wikipedia.org/wiki/Bidirectional_map`

by the integers are now swamped by this bigger size. Lastly, this method, like the others described, does not utilize our previous knowledge of a fixed domain.

One potential way to rescue this work would be an effective, possibly probabilistic, hash of sets in a partition to integers.

3.3 DIET

Discrete Interval Encoding Trees[2], describe an efficient representation for sets of types that are easily transformed into integers. The use of intervals to represent sets is an affirmation of my approach. Sadly, it is far from straightforward to figure out how to adapt these sets to represent maps, they will break the adjacency of nearby intervals. Finally, while traversing the leaf nodes of a tree is not difficult, constructing trees in that order can lead to pathological cases. At the end of the day, I am not certain that trees provide the right storage organization for our use case.

3.4 Mergeable Interval Map

The Mergeable Interval Map[1] cleverly extends Okasaki's technique to ranges of integers, in the style of DIETs. This work is probably closest in spirit to the data structure that I intent to describe but the focus on retrieval, even if optimized for arbitrary intervals, is not the trade-off that I seek.

4 A non-naive implementation

A non-naive solution seeks smaller costs than one value per domain element. Arrays are not amenable to such approaches because the association between domain and range is implicit; each position in the array is associated with an element from the domain based on some enumeration. What is stored in each position is then the appropriate element in the range.

An association list is a linked list where each node contains a key and a value. They allow us $O(|R|)$ storage since we can store just one value per node. The problem then turns into how to represent and order the keys, the sets of a partition of D implied by f. For the moment, let us assume that we have such a representation, S_i, and order. For f_1, $S_1 = [10, 80]$ and $S_2 = [1, 9] \cup [81, 100]$. There is a simple algorithm (Alg.1) to merge two association lists. Traverse both lists looking for intersections between the sets. If an intersection between the keys exists, merge the two values and then insert that, keyed by the intersection, into an association list accumulator.

Algorithm 1: Merging Two Association Lists.

```
Function Merge1(L1, L2, f):
    Data: Two association lists, L1, L2, and the merge function, f.
    Result: L merged association list.
    L ← {}
    foreach S1, v1 ∈ L1 do
        R ← S1                                              // Remaining
        foreach S2, v2 ∈ L2 and R ≠ ∅ do
inter       I ← R ∩ S2
diff        R ← R\S2
            if I ≠ ∅ then
                InsertIntoList(I, v, L)
        end
    end
    Sort L by keys, the sets return L
Function InsertIntoList(I, v, L):
    Data: A set I, the key of value v and L, a list to insert into.
    Result: Modifies L, the lead pointer does not change.
    foreach Si, vi ∈ L do
        if vi = v then
            Si ← Si ∪ I                                     // Modify the set
            return
    end
    Append (I, v) to end of L.
```

The rub is in how we insert the new keyed value into the accumulator. One sensible strategy would be to prepend (cons) each set-value pair to the front of the accumulator after we find an intersection. While this is the fastest approach it does not bind the growth of the association list. In order to enforce that the association list contains only unique values we have to traverse the entire accumulator. If L_1 and L_2 have m, n elements respectively, and consider the worst case that the function creates no duplicate values, this leads to a pretty disappointing $O(n^2m^2)$ running time. This running time also excludes the set intersection and difference operations (labeled lines inter and diff in the algorithm), as if they were not expensive.

But they are expensive. One initially suitable approach to representing the S_i would be to use a bit vector. This is a common, well understood, data-structure, especially as the algorithms to compute set intersection and difference require simple bitwise logic operators. The problem is that this bit vector would still require $|D|$ bits, and $O(|D|)$ operations.

The solution that I propose is to use pairs to represent the sequential, inclusive intervals that cover each S_i stored in *ascending* order. The intervals partition the set $[1, |D|]$, representing the domain, using the same representation as the one if storing D in an array.

For example[5], for f_1, $S_0 = \{[1, 9], [81, 100]\}$ and $S_1 = \{[10, 80]\}$ and for g, $S_0 = \{[1, 9], [91, 100]\}$, $S_1 = \{[10, 19], [81, 90]\}$, and $S_2 = \{[20, 80]\}$ This approaches main advantage is that it allows us to escape from $O(|D|)$ as each S_i is bounded by $|R|$.

Using intervals has other advantages over bit vectors. Computing the intersection and

[5]I am using square brackets ([]) to denote intervals (and pairs) and curly brackets ({}) to denote lists. This is counter to many common functional programming languages, but it reinforces the mathematical notation. Using parentheses for intervals would be confusing as I explicitly want to use inclusive intervals.

difference requires a straight-forward but large case analysis of the ways that two intervals can intersect (Algorithm 2), that needs simple integer comparison. Because our key elements are discrete we know our borders exactly.

Algorithm 2: Interval Intersection and Difference.

Function `IntervalIntersectionDifference`(I_1, I_2)**:**
  Data: Two intervals $I_1 = [s_1, e_1]$ and $I_2 = [s_2, e_2]$ of non-negative integers that are non-empty, $s_1 \leq e_1$ and $s_2 \leq e_2$.
  Result: A quintuple of potentially empty ($[\,]$) intervals: the intersection, the part of I_1 that come before the intersection[a], the part of I_2 that come before the intersection, the part of I_1 that come after the intersection, and the part of I_2 that come after the intersection.
  if $s_2 < s_1$ **then**
    if $e_2 < s_1$ **then**
      return $[\,], [\,], I_2, I_1, [\,]$
    else if $e_2 < e_1$ **then**
      return $[s_1, e_2], [\,], [s_2, s_1 - 1], [e_2 + 1, e_1], [\,]$
    else if $e_2 = e_1$ **then**
      return $I_1, [\,], [s_2, s_1 - 1], [\,], [\,]$
    else // $e_2 > e_1$
      return $I_1, [\,], [s_2, s_1 - 1], [\,], [e_1 + 1, e_2]$
  else if $s_2 = s_1$ **then //** $e_2 \geq s_1$
    if $e_2 < e_1$ **then**
      return $[s_1, e_2], [\,], [\,], [e_2 + 1, e_1], [\,]$
    else if $e_2 = e_1$ **then**
      return $I_1, [\,], [\,], [\,], [\,]$
    else // $e_2 > e_1$
      return $I_1, [\,], [\,], [\,], [e_1 + 1, e_2]$
  else if $s_1 < s_2$ and $s_2 < e_1$ **then //** $e_2 > s_1$
    if $e_2 < e_1$ **then**
      return $I_2, [s_1, s_2 - 1], [\,], [e_2 + 1, e_1], [\,]$
    else if $e_2 = e_1$ **then**
      return $I_2, [s_1, s_2 - 1], [\,], [\,], [\,]$
    else $e_2 > e_1$
      return $[s_2, e_1], [s_1, s_2 - 1], [\,], [\,], [e_1 + 1, e_2]$
  else if $e_1 = s_2$ **then //** $e_2 \geq e_1$
    if $e_2 = e_1$ **then**
      return $I_2, [s_1, s_2 - 1], [\,], [\,], [\,]$
    else // $e_2 > e_1$
      return $[s_2, e_1], [s_1, s_2 - 1], [\,], [\,], [e_1 + 1, e_2]$
  else // $e_1 < s_2$
    return $[\,], I_1, [\,], [\,], I_2$

[a]Even though the intersection may be empty (such as the first case) I_1 or I_2 may still be before it in the sense that they are before the midpoint. The same interpretation applies to the parts that come after an intersection (such as the last case).

The interval intersection-difference operation is then coupled with a fold over the two lists that contain the entire set.

Algorithm 3: Set Intersection and Difference.

Function `IntersectionAndDifference`(S_1, S_2)**:**
  Data: Two lists of *ascending* intervals S_1, S_2.
  Result: A list of intersecting interval S_i, and two lists of the set differences for each input lists S_{d1}, S_{d2}.
  $S_i \leftarrow \{\}$
  $S_{d1} \leftarrow \{\}$
  $S_{d2} \leftarrow \{\}$
  while $S_1 \neq \{\}$ and $S_2 \neq \{\}$ **do**
    $I_1 \leftarrow$ pop first element of S_1
    $I_2 \leftarrow$ pop first element of S_2
    $I, B_1, B_2, A_1, A_2 \leftarrow$ `IntervalIntersectionDifference`(I_1, I_2)
    if $I \neq [\,]$ **then** Append I to end of S_i
    if $B_1 \neq [\,]$ **then** Append B_1 to end of S_{d1}
    if $B_2 \neq [\,]$ **then** Append B_2 to end of S_{d2}
    if $A_1 \neq [\,]$ **then** Prepend A_1 to front of S_1
    if $A_2 \neq [\,]$ **then** Prepend A_2 to front of S_2
  end
  if $S_1 \neq \{\}$ **then** Append S_1 to end of S_{d1}
  if $S_2 \neq \{\}$ **then** Append S_2 to end of S_{d2}
  return S_i, S_{d1}, S_{d2}

Unfortunately, I do not have way to think about the running time of Algorithm 3 that is satisfactory. On the one hand, we can think about about the running time in terms of the lengths of the interval lists. In this case, it easy to construct pathological cases that have running time $O(|S_1||S_2|)$ such as $S_1 = \{[1,1],[3,3],\ldots\}$ and $S_2 = \{[2,2],[4,4],\ldots\}$. On the other hand, one can think about the size of D that is represented by each S_i, in this case, the performance of the algorithm would depend upon the implied partitions. In other words, it would be highly domain dependent.

In practice this algorithm may be sufficient and has an advantage over using bit vectors. Since the intervals are ascending, every call to *IntervalIntersectionDifference* the size of S_1 and S_2 (which could represent $|D|$), shrinks by either I, B_1, or B_2. Which, if one considers the first three return values in Algorithm 2, always contains one non empty interval.

With this representation of sets it is easier to build a more efficient algorithm to merge two association lists. In this case we will, once again, add the constraints that the sets are ascending. Since our set representation consists of a list of intervals, by ascending, we mean that the lowest element, of the lowest interval, of each set are ascending. Therefore we would represent f_1 as $\{\{[1,9],[81,100]\} \to 0, \{[10,80]\} \to 1\}$ and $g = \{\{[1,9],[91,100]\} \to 0, \{[10,19],[81,90]\} \to 1, \{[20,80]\} \to 2\}$.

Algorithm 4: Merging Two Association Lists.

Function `Merge2`(L_1, L_2, f)**:**
  Data: Two association lists, L_1, L_2, and the merge function, f. The association lists consist of lists of intervals, sets (S_i), as the keys and arbitrary values.
  Result: L merged association list.
  $L \leftarrow \{\}$
  while $L_1 \neq \{\}$ and $L_2 \neq \{\}$ **do**
    $S_1, v_1 \leftarrow$ pop first element of L_1
    $S_2, v_2 \leftarrow$ pop first element of L_2
    $S_i, S_{d1}, S_{d2} \leftarrow$ `IntersectionAndDifference`(S_1, S_2)
    $v_n \leftarrow f(v_1, v_2)$
    `MergeOrAddToEnd`(S_i, v_n, L)
    if $S_{d1} \neq \{\}$ **then** `InsertIntoAscending`(S_{d1}, v_1, L_1)
    if $S_{d2} \neq \{\}$ **then** `InsertIntoAscending`(S_{d2}, v_2, L_2)
  end
  return L
Function `MergeOrAddToEnd`(S, v, L)**:**
  Data: A set S, value v and an association list L.
  Result: Modifies L, the lead pointer is unchanged.
  foreach $S_i, v_i \in L$ **do**
    if $v_i = v$ **then**
      Traverse the interval lists S_i and S, ordering and merging intervals.
      return
  end
  Append (I, v) to end of L.
Function `InsertIntoAscending`(S, v, L)**:**
  Data: A set S, value v and an association list L. Assume that v does not equal any value in L.
  Result: Returns L.
  foreach $S_i, v_i \in L$ **do**
    if $S < S_i$ and $i = 0$ **then**
      Insert (S, v) before (S_i, v_i) and **return** *pointer to* (S, v)
    else if $S < S_i$ **then**
      Insert (S, v) before (S_i, v_i) and **return** L
  end
  Append (I, v) to end of L and **return** L.

In this version, *MergeOrAddToEnd* must maintain the property that the sets store their intervals in ascending order. This requires walking two ascending lists of intervals, talking the lower element and merging if the intervals are adjacent (eg. $s_i = e_j$). Similarly, *InsertIntoAscending* is important to make sure that each call to *IntersectionAndDifference* is aligned; the start of the lowest intervals in S_1 and S_2 are equal. This is the invariant that allows us to avoid nested loops over the two lists and process them in tandem instead.

The ascending property of the sets guarantees that the call to *IntersectionAndDifference* always returns a non-empty intersection. In the pathological cases we can construct partitions so that neither S_{d1} nor S_{d2} are empty, consequently there are at most mn calls to f. Furthermore, in the pathological case where each call to f generates a unique value,

we will continue to walk an increasingly longer list with every call to *MergeOrAddToEnd*, recreating the $O(m^2n^2)$ running time of Algorithm 1.

Are the faster set intersection and difference calculations, which should not be understated, the only benefit? Consider the running time of the ideal case, where we merge two lists with identical partitioning. It is easy to see that in this case, Algorithm 1, still checks for intersections m^2 times, while Algorithm 4 only m times. Another important point to consider is the case of merging more than 2 functions, (eg.$g = f_1 + f_2 + f_3$). In this case, the pathological case would still be $O(m^2n^2o^2)$ (where $o = L_3$), which can be much better than $O(|D|^3)$. A final appeal of this approach is that it feels pretty natural, it is how I think most human "computers" would solve this problem.

There are many aspects that affect Algorithm 4's running time besides the number of calls to f. For example with this approach, as opposed to the first approach with bit vectors, how we order the domain matters in terms of computational speed. It is important to avoid potentially pathological cases that might partition the domain, such as the even and odds example. We want our domain to have clusters of value, so that they represent big intervals. If we revisit the merging columns example, we want to know that successive rows will have the same values for multiple attributes.

Another practical aspect are the allocations necessary to support this algorithm. In the naive case we can explicitly define the space necessary by the size of D and the number of merges that we may perform. In this case we are repeatedly allocating lists of pairs of integers to keep track of the partitions.

At the moment, the only convincing way to think about Algorithm 4 as opposed to using arrays to store data is to benchmark. This is constructive as it demonstrates how many different trade-offs a developer has to take into account.

5 Implementation, Benchmarks and Details

A standalone library implementing partition maps in OCaml[3] is available at `https://github.com/rleonid/partition_map`. The library also contains benchmarking applications to allow an interested programmer to see if partition maps are a good fit for a computation.

5.1 Benchmarks

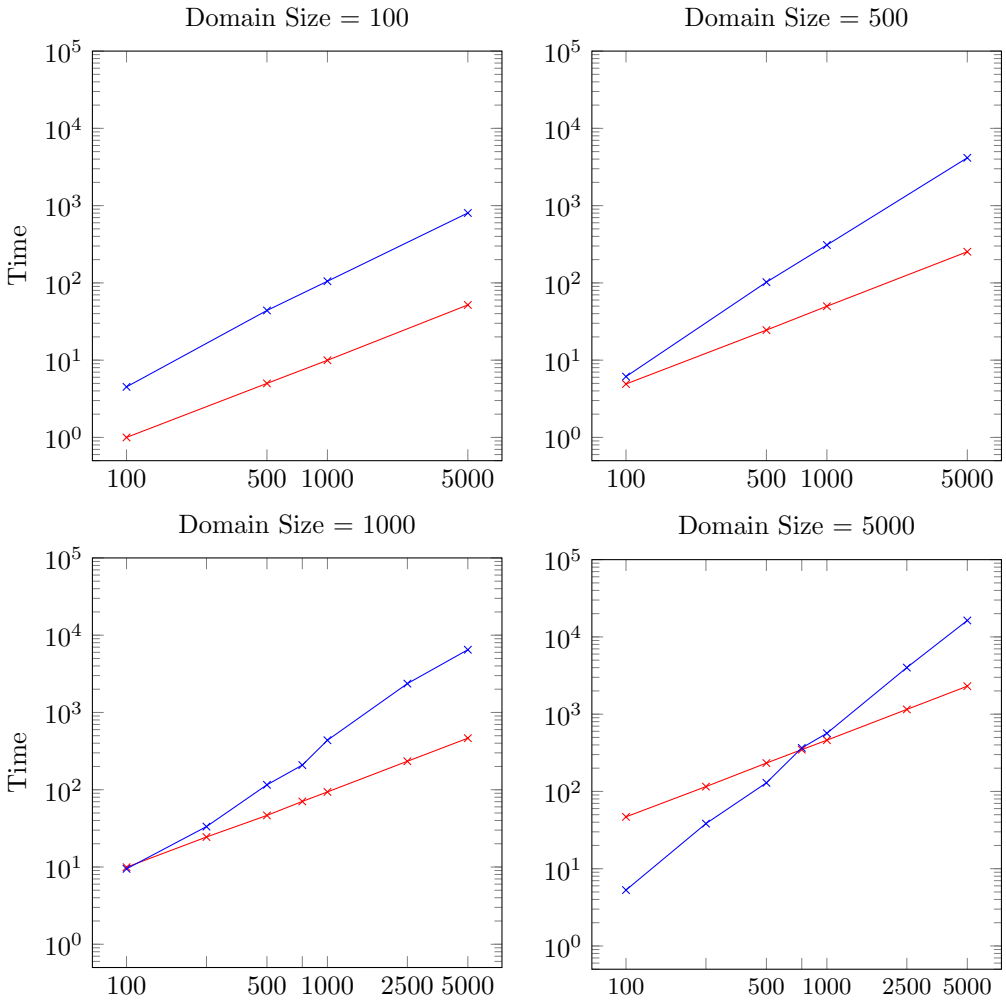
To analyze the performance of Algorithm 4 versus a naive solution using arrays, I created a hypothetical scenario where we merge functions similar to f_1 and f_2 that we defined previously. For the benchmarks we test domain sizes of 100, 500, 1000 and 5000. For the first two domain sizes we tested merging 100, 500, 1000 and 5000 states/functions and for the last two domain sizes we also tested merging 250, 750 and 2500 states.

To constrain the size of the range for each state I choose the number of unique values v, from a Poisson distribution ($\lambda = 1.5$)[6]. Afterwards, I again sample from a Poisson distribution ($\lambda = 2.5$) to determine the number of intervals, i. Finally, I assign the values in $[1, v]$

[6]Technically, I drew the number of values greater than or equal to 1 from a Poisson, since one could sample 0 from a Poisson distribution, which would not make sense. I chose the Poisson distribution because it provides a simple way to draw a small integer and this procedure in aggregate generates simple functions like f_1, f_2 and g. One can interpret this to mean that on average there will be 2.5 values per domain. I could have used a more complicated form for function generation but I thought that would further obscure the benchmark.

to the i intervals by randomly choosing a starting value in $[1, v]$ sequentially (wrapping back to 1 after v) assigning values. For example, for a domain size of 1000 some sample functions (represented as partition maps) are: $\{\{[1, 496]\} \to 0, \{[497, 745]\} \to 1, \{[746, 1000]\} \to 2\}$, $\{\{[1, 886], [937, 978]\} \to 1\{[887, 936], [979, 1000]\} \to 0\}$, and $\{\{[1, 481], [538, 987]\} \to 0, \{[482, 537], [988, 1000]\} \to 1\}$. The merging operation consists of adding the two states, for the above three we get: $\{\{[1, 481]\} \to 1, \{[482, 496], [538, 745], [887, 936], [979, 987]\} \to 2, \{[497, 537], [746, 886], [937, 978], [988, 1000]\} \to 3\}$. We do not time the allocation, calculation or deallocation of the intermediate states. Lastly, the times are normalized so that 100 merges of an array of size 100, takes time 1.

Figure 1: Comparing Arrays vs Partition Map, $E[|R|] = 2.5$.



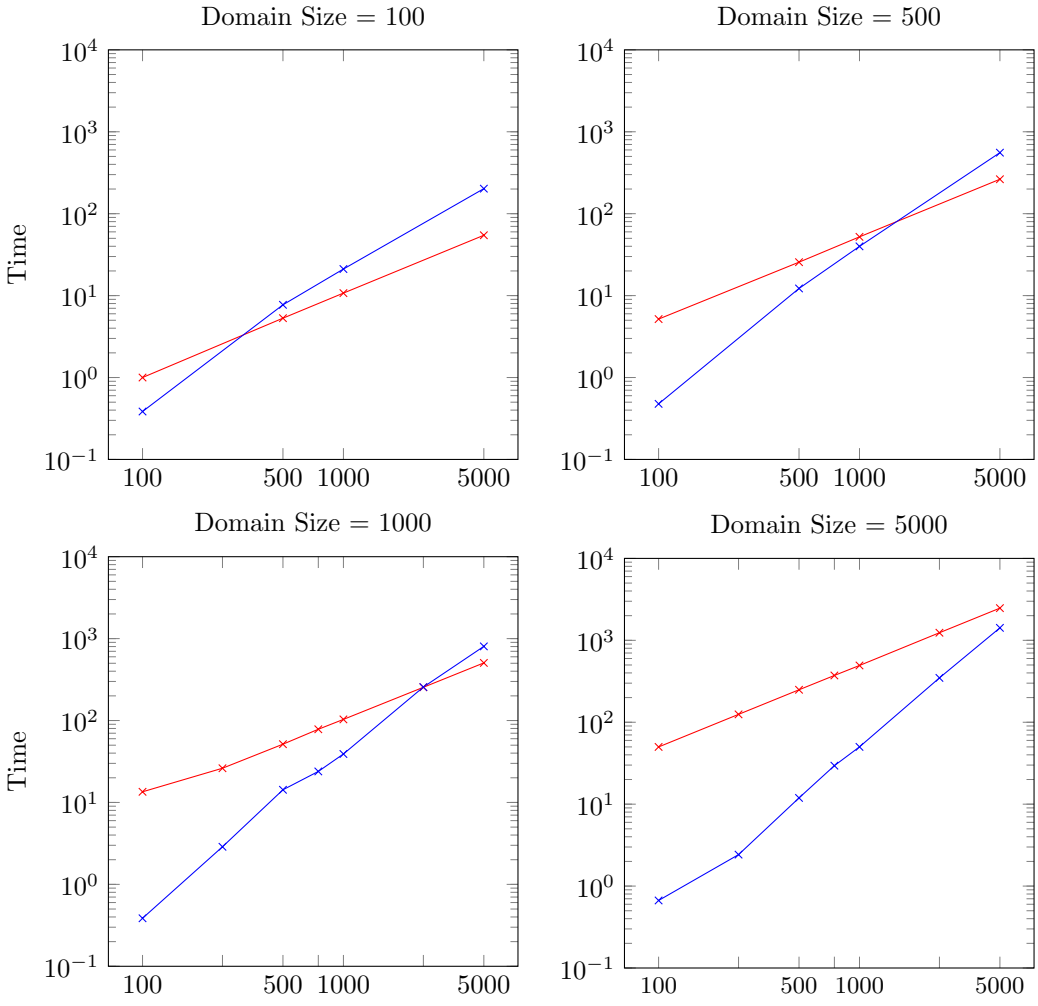
The immediate take away is that in only a limited set of scenarios do partition maps outperform the naive approach. But there are a couple points to consider. First, the array sizes that we are comparing against are relatively small and in all cases the entire data set,

including the domains to merge, can fit onto a CPU-cache. Therefore determining the two values to add is pretty fast and difficult for partition maps to compete against.

Second, notice that for a given domain size, the time increases proportional to the number of merges, as expected. But the time also increases as the domain increase: the curve rises. Partition maps do not exhibit the same behavior, rather their curve tilts! Consequently, for a small number of merges partition maps will be faster than arrays.

Regardless, the reader is probably disappointed; she was promised an interesting data-structure but in only a limited case does it actually prove to be useful. Fortunately, not everything is lost. Here is another benchmark where we lower the expected size of the range to 1.5. The take-away message is that the complexity of the computation matters; if it is simple, then partition maps can help.

Figure 2: Comparing Arrays vs Partition Map, $E[|R|] = 1.5$.



Lastly, observe that there is a slight kink in the partition maps performance trend[7]. I am not certain of the exact cause as I have not figured out an adequate model of this running time. I think that it represents something like a phase transition; at some point the relationship of length of the list storing the interval and values, versus the length of the intervals becomes more and then less favorable for the operation. I only highlight it because I have seen this odd behavior before and to emphasize that a better understanding of Algorithm 4 is needed.

5.2 Details

5.2.1 Interval representation

Given that the domain is limited. It is probable that for reasonable problems, it is much smaller than the set of representable integers (2^{32} or 2^{64}). Furthermore, one of the big practical burdens of this algorithm is the storage of the intervals as pairs. Creating a pair requires extra allocations, or a pointer to the data.

One implementation shortcut that I use is to pack the interval into one integer, where the start value occupies the upper half of the bits, and the end occupies the lower. In the 64-bit case, store $I = [s, e]$ as $s \cdot 2^{32} + e$. Aside from less allocations, this also makes comparing intervals much faster and a better version of Algorithm 2 can be implemented that takes only one comparison to see if two intervals are equal.

6 Conclusion and open questions

This is still a work in progress, and I am do not think that the current method and implementation is the best one possible. A big motivation for publishing and exposing this work is to gather feedback. I have tried my best to find connections to other approaches, and perhaps this technique is well understood and studied under a different name. I have also not been successful in providing proofs of various claims, nor even improve their somewhat unsatisfactory worst case performance.

None-the-less, I have found this technique useful and wanted to reach out to a broader community. I will close by describing what I see to be the interesting problems remaining. I have divided the problems into a practical and theoretical categories.

6.1 Practical

6.1.1 Is there a fully implicit form of partition maps?

The methods describe here use nested lists to represent the final association, and the implementation is in OCaml, a functional garbage-collected language. The cost of managing the pointers for our data-structure, is handled by the garbage-collector[8], and perhaps a hand-tuned solution might do better. But such a solution would rely upon carefully managing space for the sets of the partition and values (or pointers to them). This leads to an important question of whether the entire arrangement may be encoded in an implicit form. This approach could alleviate the current implementations cache-unfriendliness.

[7] The reason why I included 250 and 750 merges.

[8] In my use case the program spends around 25% of the running time in the GC.

In our custom implementation of intervals, we encoded the start and end within a single integer because the size of the domain was much smaller than what is ultimately representable by even half a 32bit word. But this leaves open the question of whether we can encode even more state into a given word.

6.1.2 SIMD

If a good implicit representation is possible where essentially arrays of integers are used to represent the sets of a partition and pointers to values. The operations that merge more than one partition map could be bottlenecked at comparing integers, in this case SIMD operations could be helpful.

This would be particularly useful to functions that merge more than 2 partition maps.

6.2 Theoretical

6.2.1 Define simple and bounded

The current work leaves these terms, used to describe when one might use partition maps, loosely defined, and gives examples as opposed to practical guidance. Consequently, the cost of evaluating the merge operation f does not permeate the run time analysis, of the total partition map merge.

6.2.2 Do we have to traverse the whole accumulator?

What techniques can we leverage to make adding to the accumulator faster than traversing the entire list. At the moment we are only asking for an equality test. But if we were to ask for a comparator could we use that to arrange the values, and the sets that they point at, to preserve fast merging yet provide faster ways to detect duplicate elements of the range and then preserve boundedness.

6.2.3 Hashing sets within a partition?

Does there exist a way to efficiently hash a set within a partition? The current work uses an association list to store data, but being able to hash a set would open up the possibility of using hash table like or Patricia tree like data structures (as previously described[4]). For most purposes, we can expect that $|R| < 2^{64}$, so a 64-bit word size would be sufficient. But we want two properties for this function. First, efficient computation, we do not have the resources for a cryptographic secure hash, as we would be competing against integer comparisons. And second, an ability to preserve intersections and set-difference operations. This last request, seems particularly daunting, so perhaps this approach is dubious.

7 Acknowledgments

This research was initiated and performed while the author was employed by Mount Sinai and previously supported by the Parker Institute for Cancer Immunotherapy. The author is indebted to helpful discussions with Sebastian Mondet.

References

[1] Richard Bonichon and Pascal Cuoq. A Mergeable Interval Map. `https://rbonichon.github.io/papers/rangemaps-jfla11.pdf`, 2010.

[2] Martin Erwig. Diets for fat sets. *Functional Pearls*, 1:3–8, 1993.

[3] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The Objective Caml system, documentation and user's manual – release 4.01*. INRIA, 2013.

[4] Chris Okasaki and Andrew Gill. Fast Mergeable Integer Maps. `https://www.usma.edu/eecs/SiteAssets/SitePages/Faculty%20Publication%20Documents/Okasaki/ml98maps.pdf`, 9 1998.