Subject: Re: matching lists
Posted by John-David T. Smith on Sun, 12 Mar 2000 08:00:00 GMT
View Forum Message <> Reply to Message

Craig Markwardt wrote:
>
> "J.D. Smith" <jdsmith@astro.cornell.edu> writes:
>>
>> Mark Fardal wrote:
>>>
>>> Hi,
>>>
>>> I have been looking at properties of particles in a simulation, and
>>> sometimes need to match up the particles in two different subsets. I
>>> typically have (quantity A, index #) for one set of particles, and
>>> (quantity B, index #) for another set, and want to compare quantities
>>> A and B for the particles that are in both sets.
>>>
>>> As of late last night I could not think of a good way to do this;
>>> WHERE inside a for-loop would be very slow. Maybe I'm missing
>>> something easy, but in any case here's a solution inspired by the
>>> recently submitted SETINTERSECTION function. Hope somebody finds
>>> it useful.
>>>
>>
>> The standard where_array, as posted a few years back, and modified
>> slightly for the case of the null intersection, is attached. It
>> will work with floating point and other data types also. It works
>> by inflating the vectors input to 2-d and testing for equality in
>> one go. It will also handle the case of repeated entries.
>> ...
>
> I also submit CMSET_OP, a function I recently posted on my web page.
> (Actually, I'm not sure if Mark is referring to that by
> SETINTERSECTION).
>
> Advantages are:
> * works on any numeric or string data type
> * works in order (n1+n2)*log(n1+n2) time or better, rather than n1*n2
> * uses the histogram technique for short integer lists as JD suggests
> * also does "union" and "exclusive or"
> * also does A and NOT B or vice versa
>
> Disadvantages:
> * it removes duplicates, treating the two lists strictly as sets.
> * returns values, not indices
>

The flag+shift method which I came up with a few years back (search for "Efficient comparison of arrays", circa 1997) was surrounded by a great deal of discussion about the best type of algorithm for this method. I'll quote from one of my postings at that time:

 ==
On another point, it is important to note that the problem of finding where b's values exist in a (find_elements()) is really quite different from the problem that contain() attempts to address: finding those values which are in the intersection of the vectors a and b (which may be of similar sizes, or quite different). The former is a more difficult problem, in general, which nonetheless can be solved quite rapidly as long as one vector is quite short. But the time taken scales as the number of elements in b, as opposed to the comparative size of b (to the total elements in a and b) -- i.e. nearly constant with increasing length of b. Anyway, it is important to understand the various scales, sizes and efficiencies in the problem you are trying to solve if you hope to come up with an effective solution.
 ==

The point of which is that cmset_op, while a very complete and feature-packed collection of the various *value-based* set algorithms proposed over the years, does not solve the *index-based* set intersection operation originally requested. This is a harder problem, also solveable for non-sparse sets with histogram and reverse indices, or with the full n-squared array comparison.

A sort-based method which solves this problem, but which also lack grace when dealing with repeated values (just selecting the last one which exists in the intersection), is as follows:

```
;; Return the indices of values in a which exists anywhere in b (one only for
repeated values)
function ind_int_SORT, a, b
    flag=[replicate(0b,n_elements(a)),replicate(1b,n_elements(b) )]
   s=[a,b]
   srt=sort(s)
   s=s[srt] & flag=flag[srt]
   wh=where(s eq shift(s,-1) and flag ne shift(flag, -1),cnt)
   if cnt eq 0 then return, -1
   return,srt[wh]
end
```

Which can be compared to the ARRAY, and HISTOGRAM methods:

```
function ind_int_ARRAY, a, b
   Na = n_elements(a)
   Nb = n_elements(b)
```

```
  l = lindgen(Na,Nb)
  AA = A(l mod Na)
  BB = B(l / Na)

  ;;compare the two matrices we just created
  I = where(AA eq BB,cnt)
  if cnt eq 0 then return,-1
  return,I mod Na
end

function ind_int_HISTOGRAM, a, b
  minab = min(a, MAX=maxa) > min(b, MAX=maxb)
  maxab = maxa < maxb
  ha = histogram(a, MIN=minab, MAX=maxab, reverse_indices=reva)
  hb = histogram(b, MIN=minab, MAX=maxab)
  r = where((ha ne 0) and (hb ne 0), cnt)
  if cnt eq 0 then return, -1
  return,reva[reva[r]]
end
```

A time comparison of the three methods, revealed the following for unique integer vectors:

1. ARRAY is almost always slowest due to the large memory requirement. Only for very small input vectors (10 elements or so), will ARRAY beat SORT. It is, however, the only method which correctly identifies repeated entries (not used in this test).

2. SORT is faster than HISTOGRAM for sets sparser than about 1 in 20, otherwise histogram is faster, nearly independent of input vector size. This will of course depend most critically on the amount of memory you have.

3. HISTOGRAM slows down rapidly with increasing sparseness (just more memory required).

For fun, I also simulated the Social Security number test, with 1000x1000 number distributed randomly between 0 and 999 99 9999:

```
SORT Method:
Average Time ===>    0.0045424044
ARRAY Method:
Average Time ===>      0.61496135
HISTOGRAM Method:
% Array requires more memory than IDL can address.
% Execution halted at:  IND_INT3          27
```

As I noted, the histogram will of course fail on trying to allocate the huge memory it needs for all those zeroes! ARRAY only squeaked by, and increasing the test to 10,000 SSN's yields:

```
SORT Method:
Average Time ===>     0.059468949
ARRAY Method:
% Unable to allocate memory: to make array.
```

and even the array method fails. How high can sort go? It was happy to do an index intersection of 1 million x 1 million SSN's (finding 41734 overlapping indices) in 10 seconds.

Just to be fair to ARRAY, if a and b are of very different sizes, and one of them is quite small (say less than 5 elements or so), it can dominate over SORT and HISTOGRAM.

So, if you want a generic solution which works in the same n log(n) time using a fixed amount of memory for any type of integer input data, sparse or not, use SORT. If you know your data is non-sparse (better than 1 in 10 say), you can see a speedup of a few with HISTOGRAM. If you are looking for the points of intersection in a huge array of a small set of values, you might consider ARRAY. If you want to do strings or floats or other types of data, you cannot use HISTOGRAM. And if you want information for repeated values in the same input vector, you're stuck with ARRAY.

As always, your results will vary with individual machine and operating systems. Be sure to test using your data and computer if you need to optimize for speed.

JD

--
J.D. Smith |*| WORK: (607) 255-5842
Cornell University Dept. of Astronomy |*| (607) 255-6263
304 Space Sciences Bldg. |*| FAX: (607) 255-5875
Ithaca, NY 14853 |*|