Subject: Re: matching lists
Posted by Craig Markwardt on Fri, 10 Mar 2000 08:00:00 GMT
View Forum Message <> Reply to Message

"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

Craig

http://cow.physics.wisc.edu/~craigm/idl/idl.html

--
 --- --------------
Craig B. Markwardt, Ph.D. EMAIL: craigmnet@cow.physics.wisc.edu
Astrophysics, IDL, Finance, Derivatives | Remove "net" for better response
 --- --------------

Subject: Re: matching lists
Posted by Andy Loughe on Fri, 10 Mar 2000 08:00:00 GMT
View Forum Message <> Reply to Message

; RESTRICTIONS:
; if the indices are not unique some matching elements may be
skipped...
; or is it worse than that?

Maybe worse than that...
  Isn't a significant restriction that the lists must be numeric?
  Or does it actually work for alpha-numeric lists?

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.
>
> Mark Fardal
> UMass
>
> ;+
> ; NAME:
> ;   LISTMATCH
> ;
> ; PURPOSE:
> ; find the indices where a matches b

```
>  ;   works only for SETS OF UNIQUE INTEGERS, e.g., array indices
>  ;
>  ;   for example: suppose you have a list of people with their ages and
>  ;   social security numbers (AGE, AGE_SS), and a partially overlapping
>  ;   list of people with their incomes and s.s. numbers (INCOME,
>  ;   INCOME_SS).  And you want to correlate ages with incomes in the
>  ;   overlapping subset.  Call
>  ;      LISTMATCH, AGE_SS, INCOME_SS, AGE_IND, INCOME_IND
>  ;   then AGE[AGE_IND] and INCOME[INCOME_IND] will be the desired
>  ;   pair of variables.
>  ;
>  ; AUTHOR:
>  ;   Mark Fardal
>  ;   UMass (fardal@weka.astro.umass.edu)
>  ;
>  ; CALLING SEQUENCE:
>  ;   LISTMATCH, a, b, a_ind, b_ind
>  ;
>  ; INPUTS:
>  ;   a and b are sets of unique integers (no duplicate elements)
>  ;
>  ; OUTPUTS:
>  ;   a_ind, b_ind are the indices indicating which elements of a and b
>  ;   are in common
>  ;
>  ; RESTRICTIONS:
>  ;   if the indices are not unique some matching elements may be skipped...
>  ;   or is it worse than that?
>  ; EXAMPLE:
>  ;   a = [2,4,6,8]
>  ;   b = [6,1,3,2]
>  ;   listmatch, a, b, a_ind, b_ind
>  ;    print, a[a_ind]
>  ;       2       6
>  ;    print, b[b_ind]
>  ;       2       6
>  ;
>  ;
>  ; MODIFICATION HISTORY:
>  ;   none
>  ; BUGS:
>  ;   tell me about them
>  ; ACKNOWLEDGEMENTS:
>  ;   trivial modification of SETINTERSECTION from RSI
>  ;
>  pro listmatch, a, b, a_ind, b_ind
>  minab = min(a, MAX=maxa) > min(b, MAX=maxb) ;Only need intersection of ranges
>  maxab = maxa < maxb
```

> ;If either set is empty, or their ranges don't intersect:
> ; result = NULL (which is denoted by integer = -1)
> if maxab lt minab or maxab lt 0 then begin
>   a_ind = -1
>   b_ind = -1
>   return
> endif
>
> ha = histogram(a, MIN=minab, MAX=maxab, reverse_indices=reva)
> hb = histogram(b, MIN=minab, MAX=maxab, reverse_indices=revb)
>
> r = where((ha ne 0) and (hb ne 0), count)
> if count gt 0 then begin
>   a_ind = reva[reva[r]]
>   b_ind = revb[revb[r]]
>   return
> endif else begin
>   a_ind = -1
>   b_ind = -1
>   return
> endelse
>
> end
>
>

--

Andrew F. Loughe email:loughe@fsl.noaa.gov phone:(303)497-6211

Subject: Re: matching lists
Posted by John-David T. Smith on Fri, 10 Mar 2000 08:00:00 GMT
View Forum Message <> Reply to Message

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.

For sparse integer data sets, such as the SSN example you give in the routine documentation, it will be vastly superior. Let's take your SSN for an example. If collected from random geographic locations and age brackets, the range of integers might span a billion. Suppose you had 1000 entries in your survey. The histograms will each return a vector of length 1-billion (almost all zeroes), whereas two 1-million entry arrays would be made using where_array... the difference between uncomfortable and easy.

In the other extreme, in which the data are well grouped (not sparse) and you have a lot of them (e.g. a set of particles labelled from 1-1e6), you will be better off with histogram... and now that I think about it, why can't we formulate it like this:

```
n_hist=maxab-minab
n_arr= n_elements(a) * n_elements(b)
if n_hist lt n_arr then
 ; use set_intersection method
else
 ; use array inflation method
endif
```

to get the best of both worlds. Hmm... maybe if I'm bored someday. By the way, you could fix your method to deal with repeated entries by dealing more carefully with the reverse indices, to collect every index in reva[reva[r[i]]:reva[r[i]+1]-1] for all i. I'll leave it as a challenge to do that without a loop over i ;)

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 |*|

```
;+
; NAME:
;       WHERE_ARRAY
;
```

```
; PURPOSE:
;  Return the indices of those elements in vector B which
;       exist in vector A.  Basically a WHERE(B IN A)
;       where B and A are 1 dimensional arrays.
;
; CATEGORY:
;       Array
;
; CALLING SEQUENCE:
;       result = WHERE_ARRAY(A,B)
;
; INPUTS:
;       A       vector that might contains elements of vector B
;       B       vector that we would like to know which of its
;               elements exist in A
;
; OPTIONAL INPUTS:
;
; KEYWORD PARAMETERS:
;       iA_in_B         return instead the indices of A that are in
;                       (exist) in B
;
; OUTPUTS:
;       Index into B of elements found in vector A.  If no
;       matches are found -1 is returned.  If the function is called
;       with incorrect arguments, a warning is displayed, and -2 is
;       returned (see SIDE EFFECTS for more info)
;
; OPTIONAL OUTPUTS:
;
; COMMON BLOCKS:
;               None
;
; SIDE EFFECTS:
;       If the function is called incorrectly, a message is displayed
;       to the screen, and the !ERR_STRING is set to the warning
;       message.  No error code is set, because the program returns
;       -2 already
;
; RESTRICTIONS:
;       This should be used with only Vectors.  Matrices other then
;       vectors will result in -2 being returned.  Also, A and B must
;       be defined, and must not be strings!
;
; PROCEDURE:
;
; EXAMPLE:
;       IDL> A=[2,1,3,5,3,8,2,5]
```

```
;       IDL> B=[3,4,2,8,7,8]
;       IDL> result = where_array(a,b)
;       IDL> print,result
;                  0          0          2          2          3          5
; SEE ALSO:
;       where
;
; MODIFICATION HISTORY:
;       Written by:     Dan Carr at RSI (command line version) 2/6/94
;                       Stephen Strebel         3/6/94
;                           made into a function, but really DAN did all
;                           the thinking on this one!
;                       Stephen Strebel         6/6/94
;                           Changed method, because died with Strings (etc)
;                           Used ideas from Dave Landers.  Fast TOO!
;                       Strebel 30/7/94
;                           fixed checking structure check
;                       Smith, JD 9/1/98
;                        Minor Tweak to case of no overlapping members
;-
FUNCTION where_array,A,B,IA_IN_B=iA_in_B

; Check for: correct number of parameters
;                       that A and B have each only 1 dimension
;                       that A and B are defined
   if (n_params() ne 2 or (size(A))(0) ne 1 or (size(B))(0) ne 1 $
       or n_elements(A) eq 0 or n_elements(B) eq 0) then begin
      message,'Inproper parameters',/Continue
      message,'Usage: result = where_array(A,B,[IA_IN_B=ia_in_b]',/Continue
      return,-2
   endif

;parameters exist, let's make sure they are not structures
   if ((size(A))((size(A))(0)+1) eq 8 or $
       (size(B))((size(B))(0)+1) eq 8) then begin
      message,'Inproper parametrs',/Continue
      message,'Parameters cannot be of type Structure',/Continue
      return,-2
   endif

; build two matrices to compare
   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

; normally (without keyword), return index of B that exist in A
   if keyword_set(iA_in_B) then return, I mod Na
   return,I/Na

END
```

File Attachments

1) where_array.pro, downloaded 114 times

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 |*|

Subject: Re: matching lists
Posted by John-David T. Smith on Mon, 13 Mar 2000 08:00:00 GMT
View Forum Message <> Reply to Message

Mark Fardal wrote:
>
> Hi,
>
> Somehow I knew that if I mentioned HISTOGRAM, that would stir up some
> real IDL programmers. :->
>
> J.D.'s sort method seems like the winner. The modest speed advantage
> of the histogram method in certain cases is not important. If you are
> in a situation where just matching up the elements is the limitation,
> you are probably going to be in trouble doing any analysis with them
> (let alone reading them in).
>
> The problem of repeated elements, which is the only advantage of
> WHERE_ARRAY, is not of any concern, at least to me. The point of
> the key variables a and b is that they are supposed to be unique
> identifiers. I would just like the routine not to break completely
> in case the same element was copied into the arrays twice. The
> sort method does fine in that respect (finds the last of the
> duplicate elements in a and the first in b).

By the way, I found an implementation I had mentioned a while back on the news group but had forgotten about from the nasa lib called "match" which does pretty much the same thing. It's probably less efficient, since it uses an auxiliary list of indices in addition to the flag vector, instead of just using the sort() results directly, and performs a few more "where" tests as a result. But a similar idea, written first in 1986! Match() is also more immune to changes in sort() than my routine, as a result of carrying around these additional index arrays.

Take a look.

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 |*|

Subject: Re: matching lists
Posted by Mark Fardal on Mon, 13 Mar 2000 08:00:00 GMT
View Forum Message <> Reply to Message

Hi,

Somehow I knew that if I mentioned HISTOGRAM, that would stir up some
real IDL programmers. :->

J.D.'s sort method seems like the winner. The modest speed advantage
of the histogram method in certain cases is not important. If you are
in a situation where just matching up the elements is the limitation,
you are probably going to be in trouble doing any analysis with them
(let alone reading them in).

The problem of repeated elements, which is the only advantage of
WHERE_ARRAY, is not of any concern, at least to me. The point of
the key variables a and b is that they are supposed to be unique
identifiers. I would just like the routine not to break completely
in case the same element was copied into the arrays twice. The
sort method does fine in that respect (finds the last of the
duplicate elements in a and the first in b).

The only flaw with the sort method is that sooner or later RSI is
going to break its own SORT function, just like it does with all of
its other code...

> 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.

Hope WHERE_ARRAY does not become "standard", since it's clearly inferior
to the sort method.

For completeness, using the sort method inside the calling sequence
I originally posted would look like:

```
pro listmatch, a, b, a_ind, b_ind
   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 ne 0 then begin
    a_ind = srt[wh]
    b_ind = srt[wh+1] - n_elements(a)
  endif else begin
    a_ind = -1
    b_ind = -1
    return
```

```
  endelse
end
```

Mark Fardal
UMass

Subject: Re: matching lists
Posted by Craig Markwardt on Mon, 13 Mar 2000 08:00:00 GMT
View Forum Message <> Reply to Message

"J.D. Smith" <jdsmith@astro.cornell.edu> writes:
> ...
>
> 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 truly excellent analysis. Thanks.

Craig

--
---------- ----------
Craig B. Markwardt, Ph.D. EMAIL: craigmnet@cow.physics.wisc.edu
Astrophysics, IDL, Finance, Derivatives | Remove "net" for better response
---------- ----------