Subject: Efficient comparison of arrays
Posted by Andy Loughe on Fri, 08 Aug 1997 07:00:00 GMT
View Forum Message <> Reply to Message

Hi!

I feel I should know the answer to this one, but I don't, so here goes.

Given vectors of the type...

a = [1,2,3,4,5]
b = [3,4,5,6,7]

What is the most efficient way to determine which values that occur in
a also occur in b (i.e., the values [3,4,5] occur in both a and b).

Presumably this needs to be done without loops (to be efficient), but an
obvious solution escapes me.

Thanks for your help.

--
Andrew F. Loughe |
afl@cdc.noaa.gov
University of Colorado, CIRES Box 449 |
http://cdc.noaa.gov/~afl
Boulder, CO 80309-0449 | phn:(303)492-0707
fax:(303)497-7013
--- ---------------
"I do not feel obliged to believe that the same God who has endowed us
with
sense, reason, and intellect has intended us to forego their use."
-Galileo

Subject: Re: Efficient comparison of arrays
Posted by J.D. Smith on Mon, 11 Aug 1997 07:00:00 GMT
View Forum Message <> Reply to Message

David Fanning wrote:
>
> David R. Klassen writes in response to an Andy Loughe question:
>
>>> Given vectors of the type...
>>>
>>> a = [1,2,3,4,5]
>>> b = [3,4,5,6,7]
>>>

```
>>>  What is the most efficient way to determine which values that occur in
>>>  a also occur in b (i.e., the values [3,4,5] occur in both a and b).
>>  How about:
>>       x=where(a eq b)
>>  This will give you the index numbers in a of those values that are also in b.
>>  So, to vector of the actual values would be a(x).
>
>  Ugh, I don't think so. Maybe it *should* work that way, but it
>  doesn't. At least not on my computer. :-)
>
>  I don't know if this is the most efficient way (it probably isn't),
>  but this is my off-the-cuff way of solving this problem.
>
>     FUNCTION A_in_B, a, b
>     num = N_Elements(a)
>     vector = FltArr(num)
>     FOR j=0,num-1 DO BEGIN
>        index = Where(a(j) EQ b, count)
>        IF count GT 0 THEN vector(j) = 1 ELSE vector(j)=0
>     ENDFOR
>     solution = a(Where(vector EQ 1))
>     RETURN, solution
>     END
>
>  When I run the example case above, I get a vector with the values
>  [3,4,5].
>
>  It might be more efficient to sort the arrays and then use some
>  kind of bubble-sort routine to find the first instance of a in b.
>  The WHERE function is going to find *all* instances, which is
>  probably the most inefficient part of this program.
>
>  Cheers,
>
>  David
```

Just to keep the Astronomy department here from being one-upped....

In addition to inefficiency, there is another problem with a_in_b.
Try, for instance:

```
IDL> a=[2,4,5,6,5,5,8] & b=[5,8,2,6,5,6,4]
IDL> print,a_in_b(a,b)
      2       4       5       6       5       5       8
```

As you can see, some of the values are replicated, when what I assume you would want is the unique values in this returned vector.   You could add a uniq() call, but that would make it even less efficient. The

repeated calls to where() make your routine quite slow for large vectors
(and unsymmetric with respect to argument interchange given one large
and one small vector). It also fails for no common elements. Here is
an implementation I just made up:

```
function contain,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 ne 0 then return, s[wh]
   return,-1
end
```

I ran some time tests on the two implementations. While a_in_b is adequate for small vectors, it is prohibitively slow for large ones. An example averages the result in seconds for two 10000 element random integer vectors on the range [0,20000].

```
Results for a_in_b:
 Average Time:       19.669667

Results for contain:
 Average Time:      0.19233332

Ratio:    102.269
```

And for 100 element vectors in the range [0,200]:

```
Results for a_in_b:
 Average Time:     0.010666664

Results for contain:
 Average Time:    0.0015666644
```

Hope it's useful.

JD

Subject: Re: Efficient comparison of arrays
Posted by davidf on Mon, 11 Aug 1997 07:00:00 GMT
View Forum Message <> Reply to Message

Andy Loughe wrote the other day:

> Given vectors of the type...
>
> a = [1,2,3,4,5]
> b = [3,4,5,6,7]
>
> What is the most efficient way to determine which values that occur in
> a also occur in b (i.e., the values [3,4,5] occur in both a and b).

A friend (wishing anonymity) wrote to me with this solution.
I am not sure how general it is, but it worked with this
test case and several others I made up.

Given a and b:

```
  a = [1,2,3,4,5]
  b = [3,4,5,6,7]
```

Let,

```
  array1 = BYTARR((MAX(a) > MAX(b)) - (MIN(a) < MIN(b)))
  array2 = array1
```

Then, let,

```
  ind1[a] = 1L
  ind2[b] = 1L
```

Finally, let,

```
  commonIndex = ind1 * ind2
```

The vector commonIndex now has 1s at the locations where there are
common values in the two sets. In other words,

```
  Print, commonIndex
       0   0   0   1   1   1
```

Something similar must be going on in the Venn diagram demo
I found recently in the IDL 5 demos, although a quick look
didn't find the relevant code snippet. Look at d_venn.pro
in the demo source directory.

Cheers,

David

David Fanning, Ph.D.

Fanning Software Consulting
Customizable IDL Programming Courses
Phone: 970-221-0438 E-Mail: davidf@dfanning.com
Coyote's Guide to IDL Programming: http://www.dfanning.com

Subject: Re: Efficient comparison of arrays
Posted by davidf on Mon, 11 Aug 1997 07:00:00 GMT

View Forum Message <> Reply to Message

Augh, it's too late for this:

I wrote this:

```
>  Given a and b:
>
>     a = [1,2,3,4,5]
>     b = [3,4,5,6,7]
>
>  Let,
>
>     array1 = BYTARR((MAX(a) > MAX(b)) - (MIN(a) < MIN(b)))
>     array2 = array1
>
>  Then, let,
>
>     ind1[a] = 1L
>     ind2[b] = 1L
>
>  Finally, let,
>
>     commonIndex = ind1 * ind2
>
>  The vector commonIndex now has 1s at the locations where there are
>  common values in the two sets. In other words,
>
>     Print, commonIndex
>          0   0   0   1   1   1
```

When I meant to write this:

Given a and b:

```
   a = [1,2,3,4,5]
   b = [3,4,5,6,7]
```

Let,

```
   array1 = BYTARR((MAX(a) > MAX(b)) - (MIN(a) < MIN(b)))
   array2 = array1
```

Then, let,

```
   array1[a] = 1L
   array2[b] = 1L
```

Finally, let,

```
   commonIndex = array1 * array2
```

The vector commonIndex now has 1s at the locations where there are common values in the two sets. In other words,

```
   Print, commonIndex
        0   0   0   1   1   1
```

--
David Fanning, Ph.D.
Fanning Software Consulting
Customizable IDL Programming Courses
Phone: 970-221-0438 E-Mail: davidf@dfanning.com
Coyote's Guide to IDL Programming: http://www.dfanning.com

Subject: Re: Efficient comparison of arrays
Posted by J.D. Smith on Tue, 12 Aug 1997 07:00:00 GMT

View Forum Message <> Reply to Message

David Fanning wrote:

```
>
>  Augh, it's too late for this:
>
>  I wrote this:
>
>>  Given a and b:
>>
>>     a = [1,2,3,4,5]
>>     b = [3,4,5,6,7]
>>
>>  Let,
>>
>>     array1 = BYTARR((MAX(a) > MAX(b)) - (MIN(a) < MIN(b)))
>>     array2 = array1
>>
>>  Then, let,
>>
```

```
>>     ind1[a] = 1L
>>     ind2[b] = 1L
>>
>>  Finally, let,
>>
>>     commonIndex = ind1 * ind2
>>
>>  The vector commonIndex now has 1s at the locations where there are
>>  common values in the two sets. In other words,
>>
>>     Print, commonIndex
>>           0   0   0   1   1   1
>
>  When I meant to write this:
>
>   Given a and b:
>
>      a = [1,2,3,4,5]
>      b = [3,4,5,6,7]
>
>   Let,
>
>      array1 = BYTARR((MAX(a) > MAX(b)) - (MIN(a) < MIN(b)))
>      array2 = array1
>
>   Then, let,
>
>      array1[a] = 1L
>      array2[b] = 1L
>
>   Finally, let,
>
>      commonIndex = array1 * array2
>
>   The vector commonIndex now has 1s at the locations where there are
>   common values in the two sets. In other words,
>
>      Print, commonIndex
>           0   0   0   1   1   1
>
>  --
```

There is an error in this code. Alex Schuster presents a similar solution, but without the error. The problem is you should be subtracting (min(a) < min(b)) from a and b as such:

```
array1[a- (min(a) < min(b))]=1L
```

and then add the minimum of the two vectors to the location in the commonIndex vector to get the final common values.

Otherwise, there will not, in general, be enough room in the index arrays to mark all the data values. It is just an accident that it works for [1,2,3,4,5],[3,4,5,6,7] ... try [1,2,3,4,5,6,7],[3,4,5,6,7,8] and you'll see the problem.

Another question with the process... what happens when you don't have a well grouped set of integers... e.g [1,2,3,4,5] and [3,4,10000,900,2] ... lots of wasted zeroes in those index arrays to determine this one.

JD

Subject: Re: Efficient comparison of arrays
Posted by William Clodius on Wed, 13 Aug 1997 07:00:00 GMT

View Forum Message <> Reply to Message

Andy Loughe wrote:

>
> Hi!
>
> I feel I should know the answer to this one, but I don't, so here goes.
>
> Given vectors of the type...
>
> a = [1,2,3,4,5]
> b = [3,4,5,6,7]
>
> What is the most efficient way to determine which values that occur in
> a also occur in b (i.e., the values [3,4,5] occur in both a and b).
>
> Presumably this needs to be done without loops (to be efficient), but an
> obvious solution escapes me.
>
> Thanks for your help.
> <snip>

It is not clear whether you want the values or the positions of the values (the second is harder). For the first case it is possible to this with an algorithm that approximately scales as of order N ln N.

Assume you have two vectors of length N and M respectively.

If unsorted, sort them, => operations of order N ln N and M ln M.

If duplicates within a vector can exists, delete duplicates. Operations of order N and M.

Concatenate arrays. An operation of order N + M

Sort concatenated array. An operation of order (N+M) ln (N+M)

Inspect adjacent elements of the sorted array to find duplicates. An operation of order N+M.

Done.

--

William B. Clodius Phone: (505)-665-9370
Los Alamos Nat. Lab., NIS-2   FAX: (505)-667-3815
PO Box 1663, MS-C323   Group office: (505)-667-5776
Los Alamos, NM 87545   Email: wclodius@lanl.gov

Subject: Re: Efficient comparison of arrays
Posted by J.D. Smith on Wed, 13 Aug 1997 07:00:00 GMT

View Forum Message <> Reply to Message

David Foster wrote:

```
>
>  J.D. Smith wrote:
>>
>>  Just to keep the Astronomy department here from being one-upped....
>>
>>  <SNIP>
>>
>>  Here is an implementation I just made up:
>>
>>  function contain,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 ne 0 then return, s[wh]
>>     return,-1
>>  end
>>
>>  I ran some time tests on the two implementations.  While a_in_b is
>>  adequate for small vectors, it is prohibitively slow for large ones.  An
>>  example averages the result in seconds for two 10000 element random
>>  integer vectors on the range [0,20000].
>>
```

```
>>  Results for a_in_b:
>>         Average Time:       19.669667
>>
>>  Results for contain:
>>         Average Time:      0.19233332
>>
>>  Ratio:                        102.269
>>
>>  And for 100 element vectors in the range [0,200]:
>>
>>  Results for a_in_b:
>>         Average Time:     0.010666664
>>
>>  Results for contain:
>>         Average Time:    0.0015666644
>>
>
>  When you choose a method make sure you test the solutions on
>  data that is typical to your operations; don't rely on time
>  postings based on artificial situations. Below are results
>  comparing FIND_ELEMENTS.PRO (my routine that I've posted already)
>  and JD Smith's CONTAIN.PRO function listed above.
>
>  The test data are:
>
>        A = BYTARR(65536)
>           A 256x256 image which is a section of the brain that
>           has been coded into discrete values to represent the
>           different structures in the brain. Roughly in the
>           range 0-128, many repeated values (compresses well).
>           Very typical for my situation.
>
>        B = BINDGEN(50)
>
>  Here are the results:
>
>    IDL> t1=systime(1) & c = FIND_ELEMENTS(a,b) & t2=systime(1) & $
>                print, t2-t1
>       2.3154050
>    IDL> t1=systime(1) & d = CONTAIN(a,b) & t2=systime(1) & $
>                print, t2-t1
>       132.54824
>
>  In some situations the more primitive approach may be better
>  (JD Smith's solution is certainly much more elegant and clever).
>  Also be aware that some solutions like FIND_ELEMENTS() and
>  WHERE_ARRAY() return *all* subscripts for items found, including
>  repeats, whereas CONTAIN() does not.
```

>
> Dave

This is an important point. However, I don't quite understand you're timings. I ran your code with a=round(randomu(sd,65536)*128) and b=bindgen(50). Now granted that I didn't have your brain data, but the values I got were:

```
contain() time (sec):
 Average Time:      0.73649999

find_elements() time (sec):
 Average Time:       1.2958000
```

Not an extreme advantage (and one which would fail for smaller b's), but clearly different from the values you indicate. I am running on a Pentium 166 Linux machine. Perhaps just another indication of the hardware subtleties we've all grown accustomed to.

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.

JD

Subject: Re: Efficient comparison of arrays
Posted by David Foster on Wed, 13 Aug 1997 07:00:00 GMT

View Forum Message <> Reply to Message

J.D. Smith wrote:
>
> Just to keep the Astronomy department here from being one-upped....
>
> <SNIP>

```
>
> Here is an implementation I just made up:
>
> function contain,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 ne 0 then return, s[wh]
>    return,-1
> end
>
> I ran some time tests on the two implementations.  While a_in_b is
> adequate for small vectors, it is prohibitively slow for large ones.  An
> example averages the result in seconds for two 10000 element random
> integer vectors on the range [0,20000].
>
> Results for a_in_b:
>        Average Time:       19.669667
>
> Results for contain:
>        Average Time:      0.19233333
>
> Ratio:                        102.269
>
> And for 100 element vectors in the range [0,200]:
>
> Results for a_in_b:
>        Average Time:     0.010666664
>
> Results for contain:
>        Average Time:    0.0015666644
>
```

When you choose a method make sure you test the solutions on
data that is typical to your operations; don't rely on time
postings based on artificial situations. Below are results
comparing FIND_ELEMENTS.PRO (my routine that I've posted already)
and JD Smith's CONTAIN.PRO function listed above.

The test data are:

```
 A = BYTARR(65536)
    A 256x256 image which is a section of the brain that
    has been coded into discrete values to represent the
    different structures in the brain. Roughly in the
    range 0-128, many repeated values (compresses well).
```

Very typical for my situation.

B = BINDGEN(50)

Here are the results:

```
  IDL> t1=systime(1) & c = FIND_ELEMENTS(a,b) & t2=systime(1) & $
 print, t2-t1
      2.3154050
  IDL> t1=systime(1) & d = CONTAIN(a,b) & t2=systime(1) & $
 print, t2-t1
      132.54824
```

In some situations the more primitive approach may be better
(JD Smith's solution is certainly much more elegant and clever).
Also be aware that some solutions like FIND_ELEMENTS() and
WHERE_ARRAY() return *all* subscripts for items found, including
repeats, whereas CONTAIN() does not.

Dave
--

```
   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ ~~~~~~
    David S. Foster         Univ. of California, San Diego
     Programmer/Analyst     Brain Image Analysis Laboratory
     foster@bial1.ucsd.edu  Department of Psychiatry
     (619) 622-5892         8950 Via La Jolla Drive, Suite 2200
                            La Jolla, CA  92037
   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ ~~~~~~
```

Subject: Re: Efficient comparison of arrays
Posted by J.D. Smith on Thu, 14 Aug 1997 07:00:00 GMT

View Forum Message <> Reply to Message

> <snip>
>
> All very true. With any method there are going to be some tradeoffs.
> But I am skeptical of a method that relies on the sorting of the
> arrays in question. In my timing above for CONTAIN(), of the 10.10
> seconds, 9.40 are spent sorting the array! On some hardware and with
> some data this may not be a problem; on my hardware and with my
> data it most definitely is.
>

It all boils down to this... the bulk of the time taken by contain() is
in sorting. This is obvious. Let a and b be the two vectors in

question. Let a have n elements and b have m elements. The approximate
number of operations to do the sorting is then (n+m)log(n+m) for an
efficient sorting algorithm, on average. On the other hand,
find_elements() necessarily takes on order (n x m) operations (for each
of the m elements in b, compare it with all n elements in a). If n>>m
then the sorting term is approximately nlog(n). Which method takes more
operations? The ratio of the two operation counts is r=log(n)/m. When
this is unity, the two methods will be roughly on equal footing. If r
is much greater than 1, find_elements() will be faster. For r much less
than one, contain() with it's sorting will be faster. In the case of
n=m, r=2log(2n)/n << 1 for any sizeable n (> 10, say).

So, truly, it does depend critically on your data. I found, on my
machine, an equality at approximately m=25 for n=65536. Log(n)=16 in
this case, so it's not too far off. For larger, n, the test gets more
accurate (until memory becomes an issue).

JD

Subject: Re: Efficient comparison of arrays
Posted by David Foster on Thu, 14 Aug 1997 07:00:00 GMT

View Forum Message <> Reply to Message

```
J.D. Smith wrote:
>  David Foster wrote:
>>
>>  Below are results
>>  comparing FIND_ELEMENTS.PRO (my routine that I've posted already)
>>  and JD Smith's CONTAIN.PRO function listed above.
>>
>>  The test data are:
>>
>>         A = BYTARR(65536)
>>            A 256x256 image which is a section of the brain that
>>            has been coded into discrete values to represent the
>>            different structures in the brain. Roughly in the
>>            range 0-128, many repeated values (compresses well).
>>
>>         B = BINDGEN(50)
>>
>>  Here are the results:
>>
>>       FIND_ELEMENTS() :  2.3154050
>>
>>       CONTAIN() :      132.54824
>
>  This is an important point.  However, I don't quite understand you're
```

> timings. I ran your code with a=round(randomu(sd,65536)*128) and
> b=bindgen(50). Now granted that I didn't have your brain data, but
> the values I got were:
>
> contain() time (sec):
>   Average Time: 0.73649999
>
> find_elements() time (sec):
>   Average Time: 1.2958000
>
> Not an extreme advantage (and one which would fail for smaller b's),
> but clearly different from the values you indicate. I am running on a
> Pentium 166 Linux machine. Perhaps just another indication of the
> hardware subtleties we've all grown accustomed to.

Your point is well taken about hardware subtleties. On a Sparc 2 running
Solaris 2.5 here are the timings on the same data you uses above:

contain() (sec) :   10.12
find_elements() (sec) : 4.40

God I hate working on a Sparc 2!

> 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.

All very true. With any method there are going to be some tradeoffs.
But I am skeptical of a method that relies on the sorting of the
arrays in question. In my timing above for CONTAIN(), of the 10.10
seconds, 9.40 are spent sorting the array! On some hardware and with
some data this may not be a problem; on my hardware and with my
data it most definitely is.

I'm just saying that people should check first. When WHERE_ARRAY()
was first posted, it was touted as a superior algorithm simply because
it was vectorized. But in fact it is slow and requires a hell of a
lot of memory. Your method is quite clever and probably works well
in many situations, but people shouldn't rely on posted timings to
compare methods; they should time the methods themselves, on their
machines and with their data.

I have always thought that IDL should provide this functionality, as
well as the removal of the intersection of two arrays from one of
the arrays.

Dave
--

```
  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ ~~~~~~~
   David S. Foster         Univ. of California, San Diego
    Programmer/Analyst     Brain Image Analysis Laboratory
    foster@bial1.ucsd.edu  Department of Psychiatry
    (619) 622-5892         8950 Via La Jolla Drive, Suite 2240
                           La Jolla, CA  92037
  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ ~~~~~~~
```

k

Subject: Re: Efficient comparison of arrays
Posted by William Clodius on Thu, 14 Aug 1997 07:00:00 GMT
View Forum Message <> Reply to Message

David Foster wrote:

> <snip discussion of problems with the use of sorting for an algorithm>

The problem is not the use of a sorting algorithm, the problem is the use of an inefficient sorting algorithm.

--

William B. Clodius Phone: (505)-665-9370
Los Alamos Nat. Lab., NIS-2   FAX: (505)-667-3815
PO Box 1663, MS-C323   Group office: (505)-667-5776
Los Alamos, NM 87545   Email: wclodius@lanl.gov

Subject: Re: Efficient comparison of arrays
Posted by John Votaw on Wed, 20 Aug 1997 07:00:00 GMT

View Forum Message <> Reply to Message

I very frequently have a task similar to finding common elements in arrays as discussed in this thread. I have a large array A and an array B which is known to be a subset of A. No elements in A or B are repeated. This situation occurs when A is an array of indicies into an image volume forming a region of interest and B are the indicies of some feature you would like to remove from the region of interest. The problem is to return an array that contains the elements of A that are not in B.

Following the lead of J. D. Smith, I wrote the following routine:

```
function eliminate,a,b
  c =[a,b]
  cs=c(sort(c))
  keepers=where(cs ne shift(cs,1) and cs ne shift(cs,-1), count)
  if count ne 0 then return,cs(keepers)
  return,-1
end
```

The brute force method:

```
function eliminate_bf,a,b
  mn=min(a)
  c=[mn-2,a,mn-1]  ;remove possibility of end effects
  for i=0,n_elements(b)-1 do begin
    j=(where(b(i) eq c))(0)
    c=[c(0:j-1),c(j+1:*)]
  endfor
  return,c(1:n_elements(c)-2)
end
```

In my applications, a has about 20000 elements and b has between 1 and 1000. If the number of elements in b is less than 35, then the brute force method is faster, otherwise eliminate is faster -- very much so. When the number of elements in b is 100, it is 3 times faster.

Does anyone have another algorithm or comments?

John R. Votaw
votaw@commander.eushc.org