Subject: Re: Astronomys` Sixth Neighbour Needs Help
Posted by JD Smith on Mon, 28 Jul 2003 06:22:48 GMT
View Forum Message <> Reply to Message

On Fri, 25 Jul 2003 19:14:09 -0700, astronomer wrote:

> I can begin to imagine how people must have felt when electricity became
> widely available... or when the wheel was invented even!!!

Bruno,

This is an excellent test problem that demonstrates well the variety of tradeoffs one must make to get good performance out of IDL. Pavel's method demonstrates the vectorized version of a brute force technique, cast in terms of (potentially rather large) arrays. It is essentially the exact analog of your method, but designed to use array operations, which IDL is very good at (especially large arrays -- it's what it was designed for).

The reason why your method performed so poorly is that IDL "for" loops impose a fairly large (compared to other languages) overhead on each cycle. For the curious, this is likely because it must travel around the full interpreter loop on each iteration, possibly compiling things, processing background widget events, looking for keyboard input, processing signals, etc. -- very different from, e.g., a loop in C, which runs quite close to the hardware. As Pavel points out, if you can do a substantial amount of work in each loop iteration -- more than the large constant overhead imposed by that cycle -- "for" loops can give perfectly good performance (despite the quasi-fanatical facetious rantings to the contrary you may find on certain respected IDL resource sites).

So one way to get good speed to is to re-cast your problem in terms of the largest array operation you can, and then let IDL do what it does best: chew on those arrays. Let's see how this applies to your problem. Pavel had a few bugs in his code, and I like to build big arrays another way, so here's my version of the vectorized brute force search:

```
function nth_neighbor, x, y, k
  n = n_elements(x)
  dn = fltarr(n,/NOZERO)
  d=(rebin(transpose(x),n,n,/SAMPLE)-rebin(x,n,n,/SAMPLE))^2 + $
    (rebin(transpose(y),n,n,/SAMPLE)-rebin(y,n,n,/SAMPLE))^2
  for i=0L,n-1 do dn[i] = sqrt(d[(sort(d[*,i]))[k],i])
  return, dn
end
```

What am I doing here? I just make a huge array to compare every point with every other point, all at once. I do this by making pairs of arrays which are the tranpose of each other and subtracting them. E.g. for 4 points, computing dx looks like:

```
    x0 x0 x0 x0     x0 x1 x2 x3
    x1 x1 x1 x1     x0 x1 x2 x3
dx = x2 x2 x2 x2  -  x0 x1 x2 x3
    x3 x3 x3 x3     x0 x1 x2 x3
```

So, you see, this 16 element array contains all of the x deltas from each point to each other point (in fact, it wastefully contains them all twice, since for distance purposes x1-x2 and x2-x1 are equivalent, plus it contains distances from each point to itself). If I do something similar for the y values, subtract and add each squared, I get a large array which contains the distances from each point to each other point (actually the square of the distances... I save time by skipping the square-root until it's needed at the end, since sorting on d^2 and d gives the same result). I sort them row by row, and pick out the nth member of the sorted list.

This method works well, and is fast compared to your method, but it has a deep flaw, which you've already run up against. It requires the use of arrays of size n^2 (16 in this small example). This gets big fast! E.g., for 10,000 points, I might need 4 such arrays of size 10,000 x 10,000, which is around 1e8 x 4 x 4=1.6 GB of memory! You're entirely limited by how much memory you have, or, more accurately, how fast your disks and virtual memory sub-system are. While very fast for small arrays which can fit in memory, it just doesn't scale up to large data sets. Now you understand why the code choked on 15,000 points.

The reason brute force fails, cursed by slowness (in your method), or huge storage needs (in Pavel's) is all those comparisons. If you have to compare every single point to every single other point, that's n*(n-1)/2 comparisons, at best.... e.g. n^2. But intuitively, why should we compare all the points? We know the points way over on the other side aren't going to be the closest, but we compute their distances and sort on them as if they ever had a shot anyway. But how can we decide if things are close or not without comparing all their distances? A seeming catch-22.

One method is triangulation. IDL has a lovely function called TRIANGULATE which can compute a so-called Delaunay triangulation (or DT -- do read about it) which has some nice properties. One of these properties is that the nearest neighbors of a point are either connected to it directly by the DT, or connected through some closer

point which is (mathematicians would say the nearest neighbors graph is a "sub-graph" of the Delaunay). Here's a function based on this property:

```
function nearest_neighbors,x,y,DISTANCES=nearest_d,NUMBER=k
  if n_elements(k) eq 0 then k=6

  ;; Compute Delaunay triangulation
  triangulate,x,y,CONNECTIVITY=c
  n=n_elements(x)
  nearest=lonarr(k,n,/NOZERO)
  nearest_d=fltarr(k,n,/NOZERO)

  for point=0L,n-1 do begin
    p=c[c[point]:c[point+1]-1] ;start with this point's DT neighbors
    d=(x[p]-x[point])^2+(y[p]-y[point])^2
    for i=1,k do begin
      s=sort(d)             ; A wasteful priority queue, but anyway
      p=p[s] & d=d[s]
      nearest[i-1,point]=p[0] ; Happy Day: the closest candidate is a NN
      nearest_d[i-1,point]=d[0]
      if i eq k then continue

      ;; Add all its neighbors not yet seen
      new=c[c[p[0]]:c[p[0]+1]-1]
      nnew=n_elements(new)
      already_got=[p,nearest[0:i-1,point],point]
      ngot=n_elements(already_got)
      wh=where(long(total(rebin(already_got,ngot,nnew,/SAMPLE) ne $
                          rebin(transpose(new),ngot,nnew,/SAMPLE),1)) $
               eq ngot, cnt)
      if cnt gt 0 then begin
        new=new[wh]
        p=[p[1:*],new]
        d=[d[1:*],(x[new]-x[point])^2+(y[new]-y[point])^2]
      endif else begin
        p=p[1:*]
        d=d[1:*]
      endelse
    endfor
  endfor
  if arg_present(nearest_d) then nearest_d=sqrt(nearest_d)
  return,nearest
end
```

I won't go through all the details. Roughly, it just starts searching out to nearby points on the DT, expanding outwards until it has

enough. For each of the n points, instead of computing distances to and sorting n points, it operates on a much smaller number. It gives you back *all* of the indices (and, optionally, distances) of the k nearest points, and has one quirk: for points on the boundary, the point itself is included first on the list. You could obviously test for this and reject them (since, with half their neighbors "missing", they're suspect anyway). One other nice feature: once you build the DT, you can cache it and use it again and again, if, e.g., you're finding the nearest neighbors interactively.

Here are some timings comparing the two, for finding the 6th/1st-6th neighbors of a random set of points:

```
     100 Points, 6 Nearest Neighbors
nxn:    0.0082000494
 DT:     0.030847073
```

Hmm... nxn does quite well there. But 100x100 is pretty small ;)

```
    1000 Points, 6 Nearest Neighbors
nxn:      0.79809201
 DT:      0.31871009
```

Starting to suffer, but we've still got a few MB of memory to spare.

```
    5000 Points, 6 Nearest Neighbors
nxn:       23.420803
 DT:       1.5785370
```

Here comes trouble. The relative timings will depend strongly on how much memory you have. He're we're just managing to fit about 400MB in memory, but danger looms ahead.

```
    7000 Points, 6 Nearest Neighbors
nxn:       152.13264
 DT:       2.3276160
```

With 800MB or so needed for that big nxn calculation on a 500MB laptop, we really hit the disk this time. Notice how the DT method doesn't miss a beat, just scaling up roughly as n.

Keep in mind that as the number of nearest neighbors you want gets bigger, the brute force method starts looking better and better (i.e. the problem gets harder and harder), but it still won't scale:

```
    1000 Points, 50 Nearest Neighbors
nxn:      0.81207108
 DT:       3.1622850
```

And now, just to show it can be done:

```
      100000 Points, 6 Nearest Neighbors
 DT:       32.612031
```

I left out nxn since I didn't have a spare 200GB of memory to throw around ;).

So, should we congratulate ourselves for having found a truly fast record-setting algorithm? No, probably not. My code is actually fairly sloppy in the inner loop, re-sorting things that are already mostly sorted, finding and comparing more DT-connected points than strictly necessary to get to the nearest set, concatenating and re-allocating small arrays like mad. The point is, I could afford to be sloppy, since the competition just couldn't scale. By working on the inner loop, I could probably squeeze another factor of 3 out of it in IDL. Carefully re-coded in C, you could easily see a factor of 50-100 speedup. But try something like http://www.cs.umd.edu/~mount/ANN/ before you resort to that. The take home lesson? IDL probably isn't the best tool if you need raw speed on problems it isn't optimized for (adding arrays and the like), but with a nice collection of tricks in your toolkit, you can take it places it wasn't really designed to go.

JD

---