Solving Puzzles With PostScript

Don Lancaster
Synergetics, Box 809, Thatcher, AZ 85552
http://www.tinaja.com  don@tinaja.com
copyright c2006 as GuruGram #67.
(928) 428-4073

Way on back in the late 1950’s when I first learnt engineering, many math calculations were difficult, crude, and very time consuming. You often tended to avoid doing anything involving extensive math. Or else went far out of your way to find some simplifying alternatives.

These days, of course, the average kitchen blender has far more computational capability than a fifties student engineer could have even dreamt of. And the routine math power of a plain old desktop PC was unimaginably beyond anything anywhere at any cost. The paradigm has obviously shifted.

To the point where we now have new and better math tools available. Some of which can be based on nothing but plain old brute force…

Throwing a few million extra math calculations at any problem is not that big a deal any more.

I’ve already used brute force methods for several previous projects. Ferinstance, in our Fun With Fields of GuruGram #43, all of the horrendously ugly coordinate transformations and complex trig functions involved in traditional electromagnetic fields problems got completely blown away. Simply handled by using a plain old rectangular area and brute force recalculating the boundary conditions a few hundred thousand times. Then using the classic relaxation method.

And all of the exciting developments in our Magic Sinewaves started off by brute force calculating every possible harmonic result for all possible digital words of a given binary length.

When the combinations got out of hand, longer words were dealt with by pre-applying suitable filters. Which eventually revealed that something rare but exotic was possible in the way of greatly improved digital sinewave generation. Zeroing the maximum harmonics using the minimum pulse edges.

You’ll find much more of the story in STALAC.PDF. Along with many other related "fun with math" adventures.

— 67.1 —
Puzzling over PostScript

I’ve long been overly enamored by the superb general purpose PostScript computing language. Which I routinely use for an amazing variety of tasks. And many thousands of examples of which you’ll find on these Web Pages.

Let’s see how PostScript can be used as a sledgehammer solution to some common classes of puzzles.

Ferinstance, some of the more insidious time wasters are known as alphametic puzzles. And a typical example might be...

```
ONE
ONE
TWO
TWO
THREE
ELEVEN
======
TWENTY
```

The object of the game being to find one or more unique numeric solutions. That obey the obvious rules of base 10, each letter always a uniquely specific digit, and no leading zeros.

There are a surprisingly large number of alphametic puzzles. Several web sites devoted to them appear, here, here, and here. Or you can search Google under "alphametic puzzles" for nearly 5000 more.

One of the simplest known examples is the classic \textit{SEND + MORE = MONEY}.

We will look at two methods of solving this class of problems. The first attack method will use direct brute force without regard to the numeric relations in the problem. With direct brute force, we will pay no attention whatsoever to any internal "clues" the puzzle gives us.

Our second will use modified brute force to dramatically speed up the results. For this, we will use any obvious internal clues. Such as partial totals, relations between numerals, and ranges of allowed carries. With either method, you’ll still end up exploring many thousands to many millions of possible permutations.

Let’s see. Since there are ten numerals maximum, the total possible number of solution combinations is $10^9 \times 8 \times 7 \times 6 \ldots$. Otherwise known as factorial ten, $10!$, or a grand total of 3,628,800 permutations.

PostScript can easily whump up these combinations in a minute or less and often get us at least one solution in about the same time. Here is your basic $10!$ factorial permutation generator...
Not much rocket science here. Simply nine loops inside each other.

But instead of comparing each new variable against the previously used ones, a `valid` array gets used instead. This turns out to be quite a bit faster. A "0" in the array means the numeral remains available; a "1" means it is in use. Values change to "1" on any loop continuance and change back to "0" when any loop is temporarily finished.

Similarly, while the needed tenth loop could be used, a direct calculation is faster. This totally general code executes `dotask` precisely 3,628,800 times. Once for each possible combination of numerals.

For any particular problem, you will substitute the `x0` ... `x9` values with your desired letters for the particular puzzle at hand. Your `dotask` routine will then make the needed specific calculations for you. And then spit out the desired results for you.

Here is an example of a `dotask` routine...
Interestingly, there are **four** possible solutions. Column specific $R$ and $V$ can have their values interchanged. And column specific $W$ depends upon its own column specific $L$.

This routine exhaustively searches all $10! = 3,628,800$ permutations in a brute force search. It gives you the first answer in thirty seconds on an 800 MHz XP.

**Digging Deeper**

By going to problem specific restrictions, you can dramatically reduce the cases needing tested. And speed up your results. Leading to our **modified brute force** method.

Such problem specific restrictions are set by column carries, restrictive interactions between variables, summations, and such. In this case, some careful thought should show us that...

- $O$, $T$, and $E$ cannot be zero because they lead.
- Carry5 has to be 1, so $1 + E = T$.
  Which further restricts $E$ to not being 9.
- Possible values for carry3 are 1 through 4.
  Since carry3 + $H + E = E$, carry3 + $H = 10$.
  $H$ can thus only be 6, 7, 8, or 9.
- From column 1, $10 \mod (3*E + 2*O + N)$ equals $Y$.

Each restriction can be added as a condition once all needed variables for that condition are defined. Modifying **scanall** leaves us with...
The modified code evaluates 11,208 cases with the first answer in 0.3 seconds.

**And Deeper?**

There’s little point in further optimization if you are already running in a fraction of a second and you are only going to run the code once. But if you really want to, a second column restriction drops you down to \(1206\) trips with the first answer in \(50\) milliseconds. Beyond that, the law of diminishing returns sets in. With the extra calculations for qualifications actually increasing your processing time. And the reporting time becoming a major fraction of anything left.

**For More Help**

Some ready-to-use companion code appears as **PUZZ01.PSL**. Similar topics in our **PostScript** and **Math Stuff** libraries. Our **Gonzo Utilities** greatly simplify and improve your use of **PostScript**. As always, **Custom Consulting** is available. You can email me for details.