An Algorithm on String Numeric Mixed Data - Parsing and Sorting Using C++

Shiming Tao[1], and Zhuping Lu[2]

***Abstract*—**This paper discusses a type of string numeric mixed data, which is stored in the form of string, but consists of two parts: the numeric part and the nonnumeric part. A data structure is defined with an algorithm developed for the work of parsing the original data (string) into the right form and sorting this type of data efficiently.

***Keywords*—** data structure, string numeric mixed data, parsing, sorting.

I. Introduction

A string is traditionally a sequence of characters, either as a literal constant or as some kind of variable ([3]). It may store a certain amount of internal structural information. There is plenty of work to do to exploit the potential information from a string. One way to do it is creating a database to store the structured information, which may consume much resource. We try to do some work to exploit the internal structural information in a light weighted way. Our study in this paper is to extract and split numeric content and nonnumeric content from such strings, study the data structure and explore sorting algorithms, explain the work with some examples.

II. Structure of String Numeric Mixed Data

A string numeric mixed data node should contain two arrays to store information, one stores the numeric content (numeric array, na), the other stores the nonnumeric content (string array, sa). Any element in the numeric array is called "one numeric unit", any element in the nonnumeric array is called "one nonnumeric unit" or "one string unit". A method called "parse" is defined to extract numeric data (by calling method "parse_numeric") and nonnumeric data (by calling method "parse_string") from a raw string. Furthermore, to make this data structure a "comparable class" so that it can be implemented to many mature comparison-based sorting algorithm ([1]), the less than operator "<" and the copy assignment operator operator "=" should be provided.

[1]Faculty of Network Science of Haikou College of Economics.
[2]School of Tourism and Civil Aviation Management of Haikou College of Economics.

III. Parsing and Sorting

A. Parsing the original data into the right form

As described above, a parsing procedure consists of two sub procedures: the numeric parsing procedure and the string parsing procedure. The main idea is scanning the raw string, start a numeric parsing procedure to get a numeric unit when reaching a numeric, start a string parsing procedure to get a nonnumeric unit when reaching a nonnumeric. To make things simpler, some empty strings were added to the nonnumeric part to make string units and numeric units appear alternatively. Since the parsing procedure of numeric part is irreversible, an additional array is needed to store the raw characters of the numeric part (for a backup, the string form of numeric array, sna). The steps for the parsing procedure are as follows (suppose the raw data is string s).

1. Set i=0, sa={}, na={}, sna={}.
2. Test i'th character of s (s[i]). If it is the start of a numeric unit (a digit or a dot followed by a digit), there are 3 cases:
   1) if i=0 then add an empty string to the nonnumeric part, go to setp 3;
   2) if i>0 and s[i] is right after the last nonnumeric unit, go to setp 3;
   3) if i>0 and s[i] is right after a numeric, then add an empty string to the nonnumeric part, go to setp 3.

   If it is the start of a nonnumeric character serial, go to step 4. If it is the terminator '\0', end parsing.
3. Begin constructing a numeric unit using s[i], s[i+1], ..., s[k] until s[k] is the end of the numeric unit or the terminator '\0'. The numeric unit is constructed as follows: set base=0, exponent=0, for each j=i,i+1,…,k,
   1) if s[j] is a digit and there are no dots nor power characters in {a[i], a[i +1], … , a[j-1]}, set base = base * 10 + nv(s[j]) (nv stands for "numeric value of ");
   2) if s[j] is a digit and there is one dot (s[l], $i \le l \le j-1$) but no power characters in {a[i], a[i +1], … , a[j-1]}, set base = base + nv(s[j]) * $10^{-\lambda}$,where $\lambda = j - l$;
   3) if s[j] is a digit and there are one dot and one power character in {a[i], a[i +1], … , a[j-1]}, in addition, the dot occurs before the power character, set exponent = exponent * 10 + nv(s[j]);
   4) if s[j] is a digit and there are one or two dots and one power character in {a[i], a[i +1], … , a[j-1]}, in addition, one dot (a[l]) occurs after the power character, set exponent = exponent + nv(s[j]) * $10^{-\lambda}$,where $\lambda = j - l$;

Finally, set numeric = base * $10^{exponent}$. Add it to the numeric part. Set i=k+1 if s[k] is not '\0' or i=k otherwise. Go back to step 2.

There are severial cases when s[k] is an end of a numeric unit:

1) s[k] is not a numeric character ('0'-'9' (digit), '.' (dot), 'D' or 'd' or 'E' or 'e' (power character));
2) s[k] is a dot but it is the 3rd one in a single numeric parsing;
3) s[k] is a dot and it is the 2nd one in a single numeric parsing, but there is no power character between the 2 dots (eg. 3.56.7 is not a numeric);
4) s[k] is a dot but s[k+1] is not a digit;
5) s[k] is a power character and it is the 2nd one in a single numeric parsing;
6) s[k] is a power character but s[k+1] is not a digit nor a dot.

4. Begin constructing a string (nonnumeric unit) with s[i], s[i+1], ..., s[k] until s[k] is '\0' or the end of the nonnumeric unit but s[k+1] is a start of a numeric unit. Add it to the nonnumeric part. Set i=k+1 if s[k] is not '\0' or i=k otherwise. Go back to step 2.

To construct the compare operator "<", the only thing to do is compare sa[0], na[0], sa[1], na[1], ... The routine is like this.

```
bool DataNode::operator<(DataNode& rhs.)
{
  int i;
  //sa[i] and na[i] appear alternatively, most import, na[i] arises after sa[i]
  for(i=0; i<na.size() && i<rhs.na.size(); i++)
  {
    //compare sa[i] and na[i] alternatively
    if(sa[i]<rhs.sa[i])
      return true;
    else if(sa[i]>rhs.sa[i])
      return false;

    if(na[i]<rhs.na[i])
      return true;
    else if(na[i]>rhs.na[i])
      return false;
  }
  if(i==sa.size() && i<rhs.sa.size())
    return true;
  if(i<sa.size() && i<rhs.sa.size() && sa[i]<rhs.sa[i])
    return true;
  if(i==sa.size()-1 && i<rhs.na.size() && sa[i]==rhs.sa[i])
    return true;
  return false;
}
```

Fig. 1: Code for operator "<".

To construct the compare operator "=", the only thing we need to do is copy vector sa, na and sna.

```
DataNode& DataNode::operator=(const DataNode& rhs)  //avoid deep copy
{
  if (this == &rhs) return *this; // handle self assignment
  sa=rhs.sa;
  na=rhs.na;
  sna=rhs.sna;
  return *this;
}
```

Fig. 2: Code for operator "=".

B. Sorting the data

Now that we have finished the parsing procedure and constructed a kind of comparable data structure, we can go to the next procedure -- the sorting procedure. There are many comparison-based sorting algorithms (A comparison-based sorting algorithm makes ordering decisions only on the basis of comparisons [1]) for use to do this job. Since the data is occasionally very large and is too expensive to copy, we try to choose some indirect sorting routine based on QuickSort (which is the fastest sorting algorithm known in practice [1]). The sorting routine(published by [1], p352-355) run as this

Step 1 Create an array of pointers to the array to be sorted. Rearrange the pointers according to their pointees.

```
vector<Pointer<Comparable>> p( a.size( ) );
int i, j, nextj;
for( i = 0; i < a.size( ); i++ )
p[ i ] = static_cast<Pointer<Comparable>>( &a[ i ] );
quicksort( p );
```

Step 2 Shuffle items in place.

```
for( i = 0; i < a.size( ); i++ )
    if( p[ i ] != &a[ i ] )
    {
      Comparable tmp = a[ i ];
      for( j = i; p[ j ] != &a[ i ]; j = nextj )
      {
        nextj = p[ j ] - &a[ 0 ];
        a[ j ] = *p[ j ];
        p[ j ] = static_cast<Pointer<Comparable>>( &a[ j ] );
      }
      a[ j ] = tmp;
      p[ j ] = static_cast<Pointer<Comparable>>( &a[ j ] );
    }
```

Fig. 4: Code for shuffling items in place

IV. Application with Examples

A. Analysis of the algorithm

The algorithm discussed above can be divided into two stages, the parsing stage and the sorting stage. Generally, the number of numeric units and the number of numeric characters in each data node are not predictable, the accurate storage and amount of computation can not be indicated. The approximate storage and amount of calculation can be computed with the assumptions listed below (these assumptions are reasonable in many situations, such as the following examples)

Asumption 1 For any character, the probability that it is a numeric character is p.

Asumption 2 The average number of calculation for each numeric character during parsing is q.

Asumption 3 The average length of each string that forms a numeric is r.

Asumption 4 The average number of numeric units and string units in each data node is S_1 and S_2.

Further, suppose there are N (not too small) data nodes containing L characters in average to be sorted. There are approximately NLp numeric characters to be compared and calculated into numeric. The total number of calculation during numeric parsing is NLpq. In the sorting stage, as the average-case running time of QuickSort is proved to be

O(NlogN) ([1]). This should be modied to $O(s_1 N \log N)$ in our case. Thus, the total running time for numeric data is approximately $NLpq \times O(s_1 NlogN)$. Similarly, the total running time for string data is approximately $NL(1-p) \times O(s_2 NlogN)$. The total running time for the data is (with the backup array sna to be considered, this formula should be modified, but the final result is the same)

$$\begin{aligned} & NLpq \times O(s_1 N\,ogN) + NL(1-p) \times O(s_2 N\,ogN) \\ &= O\{N^2 L\,ogN[pqs_1 + (1-p)s_2]\} \\ &= O(N^2 l\,ogN) \end{aligned} \tag{1}$$

The approximate storage for the raw data is approximately NL characters. But during the parsing stage, there are NLp characters parsed into NLp/r numeric units, the rest NL(1-p) characters are split into some strings during the parsing procedure, with the size unchanged. Since the total number of data copies during the sorting procedure is given by $N + lnN - 1.423$ [1], the approximate storage used should be

$$(N+\ln N-1.423)[Lp/r \text{ numerics} + L(1-p) \text{ characters}] \tag{2}$$

In the C++ environment, with the numeric treated as float type (4 bytes) and the characters as utf-8 (2 bytes), the approximate storage should be

$$\begin{aligned} & (N + \ln N - 1.423)[4Lp / r + 2L(1-p)] \\ &= 2(N + \ln N - 1.423)L(1 - p + \frac{2p}{r}) \end{aligned} \tag{3}$$

If sna is considered, the formula should be modified to $2(N + lnN - 1.423)L(1 + 2p / r)$.

B. Example 1

The following data is from Linux System, items from directory /dev/. By parsing and sorting, we get

TABLE I
SORTING DATA FROM COMPUTER DIRECTORY

Description	Data
Original	… tty1 … tty63 tty7 tty8 tty9 … tty60 tty61 tty62 … tty53 tty54 …
Normal sort (0.000131507s)	… tty0 tty1 tty10 tty11 tty12 tty13 … tty2 tty21 tty22 … tty3 tty30 …
String numeric parse and sort (0.00503554s)	… tty0 tty1 tty2 tty3 tty4 tty5 … tt9 tty10 tty11 … tty19 tty20 …

C. Example 2

Suppose we have in our work some files to sort and manage. Such files may be from our students (who submit the files individually) in the format as <IP address>-<Class>-<Student name>-<Score>.doc". The result by traditional sort and the Parsing-Sorting procedure are like this

TABLE II
SORTING DATA FROM COMPUTER DIRECTORY

Normal sort (0.000673411s)	Parsing-Sorting (0.00280358s)
192.168.1.10C5-David Elk-SC261.25.doc	192.168.1.1C1-Mic Mala-SC165.31.doc
192.168.1.10C5-Wilia Ly-SC344.12.doc	192.168.1.1C2-Chris Rob-SC141.1.doc
192.168.1.11C2-Kev Mcou-SC455.02.doc	192.168.1.10C5-David El-SC261.25.doc
192.168.1.12C2-Mara Wet-SC482.7.doc	192.168.1.1C5-Jo Thon-SC482.02.doc
192.168.1.13C2-Mike Seelig-SC30.73.doc	192.168.1.1C5-Paul Dira-SC444.45.doc
192.168.1.14C2-Ania Co-SC267.14.doc	192.168.1.10C5-Wliam Ly-SC344.12.doc
192.168.1.15C2-Robert En-SC237.13.doc	192.168.1.11C2-Kevi Mo-SC455.02.doc
192.168.1.16C1-Alber Seg-SC198.12.doc	192.168.1.12C2-Marta Wat-SC482.7.doc
192.168.1.17C2-Sergey Elf-SC467.05.doc	192.168.1.13C2-Mike Seelig-SC30.73.doc
192.168.1.18C1-Rui Zhao-SC443.49.doc	192.168.1.14C2-Ant Coen-SC267.14.doc
192.168.1.19C1-Irma Fu-SC104.65.doc	192.168.1.15C2-Rob Ent-SC237.13.doc
192.168.1.1C1-Mic Manet-SC165.31.doc	192.168.1.16C1-Albet Seg-SC198.12.doc
192.168.1.1C2-Chris Robet-SC141.1.doc	192.168.1.17C2-Serg Efon-SC467.05.doc
192.168.1.1C5-Jose Thom-SC482.02.doc	192.168.1.18C1-Rui Zhao-SC443.49.doc
192.168.1.1C5-Paul Dira-SC444.45.doc	192.168.1.19C1-Irma Fusi-SC104.65.doc
192.168.1.20C1-Ni Kar-SC469.64.doc	192.168.1.20C1-Nios Stew-SC469.64.doc
192.168.1.21C2-Robin Wil-SC292.39.doc	192.168.1.2C2-Nash Wene-SC463.67.doc
192.168.1.22C2-Will Fer-SC294.25.doc	192.168.1.2C3-Enric Eular-SC434.49.doc
192.168.1.23C2-Bill Mur-SC341.15.doc	192.168.1.2C3-Warren Rol-SC292.09.doc
192.168.1.24C4-Jim Green-SC271.47.doc	192.168.1.2C3-Willard Lib-SC136.16.doc
192.168.1.25C4-Jim Car-SC136.74.doc	192.168.1.21C2-Rin Will-SC292.39.doc
192.168.1.26C4-Eddi Mark-SC377.6.doc	192.168.1.22C2-Will Fer-SC294.25.doc
…	…

D. Unsolved problems and suggestions

Although most data can be parsed and sorted by the algorithm we study. There still exist several problems when we take a deep look at the algorithm. We are not going into more details for these but list some of them with corresponding suggestions.

Problem 1 During the parsing stage, all numeric units are not bounded (for convenience of the parsing procedure). There exist some cases when the objective numeric unit is too large to be identified. In these cases, our suggestion is set a bound on each numeric unit (the base and the exponent) during parsing. If some numeric unit would reach the bound after adding a next numeric character, end the numeric parsing and begin a new numeric parsing, or go back to the beginning of such numeric unit and try parsing such characters into a string unit.

Problem 2 There are some cases when dot should be treated as nonnumeric character (In Example 2, the IP Address is such a case). For these cases, our suggestion is to

rewrite the numeric parsing procedure, treat dot as nonnumeric character.

Problem 3 There are some cases when the first K characters should be ignored during numeric parsing. In these cases, our suggestion is rewrite the numeric parsing procedure, treat the first K characters as nonnumeric characters (The data in Example 2 can be modified and sorted using the ignore K characters case when one need to sort the data by "IP address", "Class" or "Score").

REFERENCES

[1] Mark Allen Welss, *Data Structures and Problem solving Using C++ , 2nd ed*. Upper Saddle River. N.J.: Pearson Education International, 2003, ch. 9, pp. 321-356.
[2] cppreference.com, http://en.cppreference.com/w/
[3] wikipedia.org, https://en.wikipedia.org/wiki/Main Page