Not The Maximum Segment Sum Problem
Introduction to Functional Programming

Yu Liu

December 14, 2015

Not the Maximum Segment Sum (NMSS for short) problem is an interesting problem from the context of [1]. In this short report I introduce how to compute NMSS problem in **liner time** and also introduce some concepts and skills of functional programming (FP). As a related topic the *Maximum Segment Sum* problem has been studied by many researchers [2–4].

1 Notations

The syntax is mainly based on Haskell but a Java implementation is given. List is denoted as $[x, y, z]$ and $++$ means list concatenation. Function application is denoted with a space with its argument without parentheses, i.e., $f\ a$ equals to $f\ (a)$. Functions are curried and bound to left and thus $f\ a\ b$ equals to $(f\ a)\ b$. Functions can be composed by "$\cdot$" and it has higher priority than other operators.

2 The Definition of Non-Segment

By definition of segment, contiguous subsequence of a list, the non-segment is defined: for a list at shortest with 3 elements, the non-segments are the sequences that are not contiguous subsequences (segments), formally, denoted by a regular expression as follows.

$$F^*T^+F^+T(T+F)^*$$

For example, given a list:

$$[-4, -3, -7, +2, +1, -2, -1, -4]$$

The subsequence $[-4, -3, -7]$ is a segment, while $[-4, -7]$ is a not-segment and $[-4, -3, -7, +1]$ is also a not-segment of the original list. We can use a automaton to recognize this regular expression:

$$dataState = E|S|M|N$$

- State E for F^* (empty)
- State S for F^*T^+ (suffix)

- State M for $F^*T^+F^+$ (middle)
- State N for $F^*T^+F^+T(T+F)^*$

The sum of a non-segment is called *Non-Segment Sum* (NSS).

3 Specification

The MNSS is to find a NSS which has the maximum sum.

$$\begin{array}{lcl} mnss & :: & [Int] \rightarrow Int \\ mnss & = & maximum \cdot map\ sum \cdot nonsegs \end{array}$$

To define the *nonsegs* function, firstly, we introduce a *markings* function:

$$\begin{array}{lcl} markings & :: & [a] \rightarrow [[(a, Bool)]] \\ markings\ xs & = & [zip\ xs\ bs | bs \leftarrow booleans(length\ xs)] \\ booleans\ 0 & = & [[\ ]] \\ booleans\ (n+1) & = & [b : bs | b \leftarrow [True, False], bs \leftarrow booleans\ n] \end{array}$$

So that, the *nonsegs* can be defined as

$$\begin{array}{lcl} nonsegs & :: & [a] \rightarrow [[a]] \\ nonsegs & = & extarct \cdot filter\ nonsegs \cdot markings \\ extarct & :: & [[(a, Bool)]] \rightarrow [[a]] \\ extarct & = & map(map\ fst \cdot filter\ snd) \end{array}$$

By making use of the automaton, *nonsegs* can also be defined as

$$nonseg = (==N) \cdot foldl\ step\ E \cdot map\ snd$$

Here, the *Step E* means start the automaton form E state, and here are the states transformation functions:

$$\begin{array}{ll} step\ E\ False\ = E & step\ M\ False\ = M \\ step\ E\ False\ = S & step\ M\ False\ = N \\ step\ S\ False\ = M & step\ N\ False\ = N \\ step\ S\ False\ = S & step\ N\ False\ = N \end{array}$$

4 Derivation

The *mnss* can be redefined as

$$\begin{array}{lcl} mnss & = & maximum \cdot map\ sum \cdot\ extract \cdot filter\ nonseg \cdot markings \\ extarct & = & map(map\ fst \cdot filter\ snd) \\ nonseg & = & (==N) \cdot foldl\ step\ E \cdot map\ snd \end{array}$$

the problem is turned to find a way to apply the fusion law of $foldl$ to get a better algorithm, that $extract \cdot filter\ nonseg \cdot markings$ can be treated as an instance. So, we can define a *pick* function as follows.

$$\begin{array}{l} pick :: State \rightarrow [a] \rightarrow [[a]] \\ pick\ q = extract \cdot filter((==q) \cdot foldl\ step\ E \cdot map\ snd) \cdot markings \end{array}$$

just let q = N , $nonsegs = pick\ N$. Here clime these equations hold:

$$
\begin{array}{lcl}
pick\ xs & = & [[\,]] \\
pick\ S\ [\,] & = & [\,] \\
pick\ S\ (xs ++ [x]) & = & map(++[x])(pick\ S\ xs ++ pick\ E\ xs) \\
pick\ M\ [\,] & = & [\,] \\
pick\ M\ (xs ++ [x]) & = & pick\ M\ xs ++ pick\ S\ xs \\
pick\ N\ [\,] & = & [[\,]] \\
pick\ N\ (xs ++ [x]) & = & pickNxs ++ map(++[x])(pick\ N\ xs ++ pick\ M\ xs)
\end{array}
$$

recast the definition of $pick$ as an instance of $foldl$: firstly we make a tuple:

$$pickallxs = (pick\ E\ xs, pick\ S\ xs, pick\ M\ xs, pick\ N\ xs)$$

so, we have

$$
\begin{array}{lcl}
pickall & = & foldl\ step([[\,]],[\,],[\,],[\,]) \\
step(ess, nss, mss, sss)x & = & (ess, map(++[x](sss ++ ess), \\
 & & mss ++ sss, nss ++ map(++[x])(nss ++ mss))
\end{array}
$$

so that $mnss = maximum \cdot map\ sum \cdot fourth \cdot pickall$. by define

$$tuple\ f(w,x,y,z) = (f\ w, f\ x, f\ y, f\ z).$$

Then we have :

$$maximum \cdot map\ sum \cdot fourth = fourth \cdot tuple(maxmium \cdot map\ sum)$$

Then $mnss$ is changed to: $mnss = fourth \cdot tuple(maximum \cdot map\ sum) \cdot pickall$

By fusion law of $foldl$:

$$f(foldl\ g\ a\ xs) = foldl\ h\ b\ xs$$

if $f\ a = b$, and $f(g\ x\ y) = h(f\ x)y$ hold for all x and y.

f,g,and a are instantiations:

$$
\begin{array}{lcl}
f & = & tuple(maximum \cdot map\ sum) \\
g & = & step \\
a & = & ([[\,]],[\,],[\,],[\,])
\end{array}
$$

We need to find the h and b, satisfy the fusion conditions.

$$tuple(maximum \cdot map\ sum)([[\,]],[\,],[\,],[\,]) = (0, -\infty, -\infty, -\infty)$$

Here h and b is given: b is $(0, -\infty, -\infty, -\infty)$ and

$$h(e,s,m,n)x = (e, (s \uparrow\ e) + x, m \uparrow s, n \uparrow ((n \uparrow m) + x))$$

so that

$$mnss = fourth \cdot foldl\ h(0, -\infty, -\infty, -\infty)$$

to simplify the definition of $mnss$ we can do this :

$$
\begin{array}{lcl}
mnss\ xs & = & fourth(foldl\ h\ (start(take\ 3\ xs))\ (drop\ 3\ xs)) \\
start\ [x,y,z] & = & (0, \uparrow [x+y+z, y+z, z], \uparrow [x, x+y, y], x+z)
\end{array}
$$

5 Remarks

Also, for at-least-length-k problem we can derive O(nk) algorithm. And, for non-regular conditions such as $F^*T^nF^*T^nF*$ ($n \geq 0$) is susceptible to the same method. The Java implementation is here: `https://github.com/moyun/Mnss`. Readers can evaluate the performance of the linear algorithm and compare with your own implementation.

References

[1] Richard Bird. *Pearls of Functional Algorithm Design.* Cambridge University Press, New York, NY, USA, 1st edition, 2010.

[2] M.~Cole. Parallel Programming, List Homomorphisms and the Maximum Segment Sum Problems. Report {csr}-25-93, Department of Computing Science, The University of Edinburgh, 1993.

[3] Kazutaka Morita, Akimasa Morihata, Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi. Automatic inversion generates divide-and-conquer parallel programs. In *ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007)*, pages 146–155. ACM Press, jun 2007.

[4] Z.~Hu, H.~Iwasaki, and M.~Takeichi. Formal Derivation of Parallel Program for 2-Dimensional Maximum Segment Sum Problem. In *Annual European Conference on Parallel Processing, LNCS 1123*, pages 553–562. Springer-Verlag, 1996.