Stack-based breadth-first search tree traversal

Pravin Kumar Sinha

June 04, 2013

This article explains the traditional breadth-first search (BFS) that has drawbacks in terms of extra space required by the queue data structure. If a tree has multiple children and is a balanced tree, then the queue size would grow exponentially and could cause serious memory threats. The solution in this article provides a parallel approach for printing the tree in BFS fashion, where the queue data structure in not required at all. Instead, stack memory is used but memory consumption would be log(n) with base N, where N is the average children count per node and 'n' is the number of nodes in the tree.

Introduction

This article describes breadth-first search (BFS) on a tree (level-wise search) using a stack, either using a procedure allocated stack or using a separate stack data structure. BFS is a way of scanning a tree while breadth is performed first. The first node to be printed is root followed by its immediate children, then followed by the next level children. Here, root is at level 1, whereas, its children are at level 2. Now, grandchildren would be printed next at level 3. It would continue in this fashion until it meets the last level. An example is shown in following tree.

```
                                (a)
                                 |
                                 v
             ---------------------------------------------
             |      |      |      |      |     |         |
             v      v      v      v      v     v         v
            (0)    (1)    (2)    (3)    (4)   (5)       (6)
             |      |      |      |      |     |         |
             v      |      v      --+    v     ------+   v
   -------------    | ------------- |  ------------- | -------------
   | | | | | | |    | | | | | | | | |  | | | | | | | | | | | | | | |
   A B C D E F G    | A B C D E F G |  A B C D E F G | A B C D E F G
                    |               |                |
                    v               v                v
                 -------------  -------------  -------------
                 | | | | | | |  | | | | | | |  | | | | | | |
                 A B C D E F G  A B C D E F G  A B C D E F G
```

This would be printed as "a 0 1 2 3 4 5 6 A B C D E F G A B C D E F G A B C D E F G A B C D E F G A B C D E F G A B C D E F G A B C D E F G A B C D E F G".

Scenario

The known solution to the problem is using an extra queue. So, as you encounter a node, push its children to the queue. Keep popping the queue to print the node and then simultaneously push all its children into the queue.

```
procedure bfs(root:NODE*);
var myqueue:Queue;
var node:NODE*;
BEGIN
  push(myqueue,root);
  while(myqueue not empty) do
  begin
    node=pop(myqueue);
    print node;
    for (all children of node) do
      push(myqueue,children);
  end
END
```

With this solution, we have a memory concern, especially when the tree is not a binary tree or it is a balanced kind of tree where at each level, the number of node grows exponentially. For example, in the above tree, at the third level itself we have 49 nodes. In this circumstance, if the second-level end node ('6') is printed, then the queue would have all the 49 entries. Now, calculating the Nth level, it would be 7^(N-1) nodes. So, if a tree has 11 levels, then there would be 300 million entries and a 4-byte address pointer per entry would lead to around 1 MB. It would be worse when the function is a re-entrant and is accessed through multiple threads.

Proposed solution

With the new solution, we can have something to sacrifice in time coordinate in order to achieve in space coordinate. If we find something recursive, then space would be dependent only upon the number of levels, and in a balanced tree, it would log(n) where 'n' is the number of total nodes. Pseudocode of the solution would be as shown below.

```
procedure bfs(root:NODE*);
var target=0;
var node=root;
BEGIN
  for each level in tree do
  begin
    printtree(node,target,0);
    target=target+1;
  end
END


procedure printtree(node:NODE*, target:int, level:int);
BEGIN
  if(target > level) then
  begin
    for each children of node do
      printtree(child,target,level+1);
  end
  else
    print node;
END
```

In a 32-bit system, if we calculate the stack occupied during each function call, it would be 'arguments + return address + current stack pointer + current base pointer'. Here, it would be typically 4*3+4+4+4=20 bytes. With 11 levels, it would 220 bytes, which is far less than 1 MB.

How it works

There are two entities: a *helper* and a *tree scanner*. In design, the *helper* contains a *tree scanner*.

```
    +------+         +------------+
    |Helper|<>------>|Tree Scanner|
    |      |         |            |
    +------+         +------------+
  <>--------> Aggregation
```

The helper asks the tree scanner to print all the nodes at a particular level. The tree scanner goes to the node at each level and asks if they belong to the level that the helper has sought for. If the node belongs to the particular level, it affirms and returns. All the nodes at the particular level get printed. Then the helper asks the tree scanner for the next level until the last level is met. In the following example, a tree helper asks for printing all the nodes at level 3.

```
           +-->(0)   +-->(A)   |-->(e)
           |-->(1)   |-->(B)-->|-->(f)
           |-->(2)-->|-->(C)   |-->(h)
  (a)----->| .       |.
           | .                           |-->(i)
   ^       | .       +-->(D)------------>|-->(j)
   |       +-->(6)-->|-->(E)   |-->(a)   |-->(k)
                     |-->(F)-->|-->(b)
   |                 |-->(G)   |-->(c)
                     +-->(H)
   |


 Tree scanner checks
 at first level, if
 it does not match
 to level number 3 it
 proceeds to next level



           +-->(0)   +-->(A)   |-->(e)
           |-->(1)   |-->(B)-->|-->(f)
           |-->(2)-->|-->(C)   |-->(h)
  (a)----->| .       |.
           | .                           |-->(i)
           | .       +-->(D)------------>|-->(j)
           +-->(6)-->|-->(E)   |-->(a)   |-->(k)
                     |-->(F)-->|-->(b)
                ^    |-->(G)   |-->(c)
                |    +-->(H)
               /
              /
 Tree scanner checks
 at second level next
 if it does not match
 to level number 3 it
 proceeds to next level
```

```
           +-->(0)   +-->(A)   |-->(e)
           |-->(1)   |-->(B)-->|-->(f)
           |-->(2)-->|-->(C)   |-->(h)
  (a)----->| .       |.
           | .                           |-->(i)
           | .       +-->(D)------------>|-->(j)
           +-->(6)-->|-->(E)   |-->(a)   |-->(k)
                     |-->(F)-->|-->(b)
                     |-->(G)   |-->(c)
                     +-->(H)

                          ^
                          |
                         /
                        /
       Tree scanner checks
       at third level next
       and it does match to
       level number 3 it prints
       this level nodes to next
```

Other benefits

Embellishment

Embellishment of output can be done in an easier way. Let us suppose, you need to insert a demarcation '-------------' between each child, as shown in the following example:

```
a
-------------
0 1 2 3 4 5 6
-------------
A B C D E F G A B C D E F G A B C D E F G A B C D E F G A B C D E F G A B C D E F G A B C
D E F G
```

This can be achieved by simply inserting a `println` command after the `printtree` command.

```
for each level in tree do
printtree(node, target, 0);
println('-------------');
.
.
.
```

Level specific

This can be useful in printing a particular level (that is, level number 4). Consider the following real-life example. In a company hierarchy, match this to a tree where the CEO sits at the top level and an entry-level engineer at the bottom level. Levels are typically distributed in bands. So, if senior engineers are at a specific band, (say band x) and you need to print all senior engineers who would match to level (N+1-x) where N is the maximum band, see the following example:

```
                        CEO            <---BAND N
                         ^
                         |
                     +------+
                     |      |
               PresidentA  PresidentB
                  ^
                  |
             +---------+
             |    |    |
            VP1   VP2  VP3
                       .
                       .
                       .
                       ^
                       |
                   +-------+
                   |   |   |
                  SE1  SE2  SE3      <----BAND x
                       .
                       .
                       ^
                       |
                  +--------+
                  |    |   |
                 ENG1 ENG2 ENG3      <----BAND 1
```

The program would be:

```
printtree(root,N+1-x,0);
```

Further, this function can also be directly used from scripts.

Generalization

If someone needs to have a different kind of lists and a different kind of list iterations, then the template method would be the best-suited pattern. The user would have the choice of going with several kinds of lists without actually meshing up with the algorithms (something similar to an abstract class) with the template method of list iteration. Generalization can be done on the possible type of list (such as array, single-linked list, double-linked list, and so on) iteration.

```
                   +----------------+
                   | BFS            |
                   |                |
                   |+iteratelist()=0|
                   +----------------+
                         ^
                        /_\
                         |
                       / | \
                      /  |  \
              /-------   |   -------\
             /           |           \
            /            |            \
           /             |             \
 +--------------+ +--------------+ +--------------+
 | ARRAY        | |SGL LINKED LST| |DBL LINKED LST|
 |              | |              | |              |
 |+iteratelist()| |+iteratelist()| |+iteratelist()|
 +--------------+ +--------------+ +--------------+
```

Program

A C program for BFS would be as shown below (where the child list is in the form of contiguous memory):

```
/*BFS sub routine*/
```

```
bool printtree(NODE* node, int target, int level) {
int i;
bool returnval=false;
if (target > level) {
for(i=0;i<CCOUNT;i++) if(printtree(node->child+i,target,level+1) ) returnval=true;
}
else {
printf("%c",(node->data));
if(node->child != NULL) returnval=true;
}
return returnval;
}


/*BFS routine*/
void printbfstree(NODE* root) {
if(root == NULL) return;
int target=0;
while(printtree(root,target++,0)) {
printf("\n");
}
}
```

Experimental data

```
Compile with -DSPACEDATA for space data, -DTIMEDATA for time data, -DLPRINT for
line printing and -DNOPRINT for not printing the data.
Note: Lines between '---------------' line are command and its output generated.
```

Normal printing for both types at the child level count as 7 and tree depth as 3

```
---------------
>a.out -l3 -c7
queue based, allocated for queue size 50 ,each node size 4 bytes
printing queue based
a0123456ABCDEFGABCDEFGABCDEFGABCDEFGABCDEFGABCDEFGABCDEFG
printing stack based
a0123456ABCDEFGABCDEFGABCDEFGABCDEFGABCDEFGABCDEFGABCDEFG
printing done
---------------
```

Printing line-wise output for a stack-based BFS approach

```
----------------
>gcc -DLPRINT -lm stackbasedBFS.c
>./a.out -l3 -c7
queue based, allocated for queue size 50 ,each node size 4 bytes
printing queue based
a0123456ABCDEFGABCDEFGABCDEFGABCDEFGABCDEFGABCDEFGABCDEFG
printing stack based
a
0123456
ABCDEFGABCDEFGABCDEFGABCDEFGABCDEFGABCDEFGABCDEFG
printing done
----------------
```

Printing with space data and time data

```
---------------
>./a.out -l5 -c2
queue based, allocated for queue size 17 ,each node size 4 bytes
printing queue based
a01ABABababababab0101010101010101
queue based, queue usage size 16
diff time 0 sec 26 micro

printing stack based
a01ABABababababab0101010101010101
stack used 128
diff time 0 sec 14 micro

printing done
---------------
```

SPACE and TIME analysis

```
--------------
> cc -DNOPRINT -DSPACEDATA -DTIMEDATA -lm  stackbasedBFS.c
> ./a.out -l10 -c10
queue based, allocated for queue size 1000000001 ,each node size 4 bytes
> ./a.out -l10 -c9
queue based, allocated for queue size 387420490 ,each node size 4 bytes
printing queue based
Segmentation fault
> ./a.out -l9 -c10
queue based, allocated for queue size 100000001 ,each node size 4 bytes
printing queue based
queue based, queue usage size 100000000
diff time 28 sec 490083 micro

printing stack based
stack used 256
diff time 1 sec 469060 micro

printing done
> ./a.out -l5 -c10
queue based, allocated for queue size 10001 ,each node size 4 bytes
printing queue based
queue based, queue usage size 10000
diff time 0 sec 2891 micro

printing stack based
stack used 128
diff time 0 sec 164 micro

printing done
> ./a.out -l10 -c7
queue based, allocated for queue size 40353608 ,each node size 4 bytes
printing queue based
queue based, queue usage size 40353607
diff time 11 sec 874163 micro

printing stack based
stack used 288
diff time 0 sec 788580 micro

printing done
> ./a.out -l20 -c2
queue based, allocated for queue size 524289 ,each node size 4 bytes
printing queue based
queue based, queue usage size 524288
diff time 0 sec 333929 micro
```

```
printing stack based
stack used 608
diff time 0 sec 40476 micro

printing done
> ./a.out -l25 -c2
queue based, allocated for queue size 16777217 ,each node size 4 bytes
printing queue based
queue based, queue usage size 16777216
diff time 10 sec 635081 micro

printing stack based
stack used 768
diff time 1 sec 482634 micro

printing done
> ./a.out -l30 -c2
queue based, allocated for queue size 536870913 ,each node size 4 bytes
allocation failed, exiting

> ./a.out -l28 -c2
queue based, allocated for queue size 134217729 ,each node size 4 bytes
allocation failed, exiting
> ./a.out -l27 -c2
queue based, allocated for queue size 67108865 ,each node size 4 bytes
printing queue based
queue based, queue usage size 67108864
diff time 43 sec 22479 micro

printing stack based
stack used 832
diff time 5 sec 773319 micro

printing done
-----------------
```

In the space analysis, we can consider the worst-case scenario, that is, level 9 and child count 10. Here, the number of nodes used by the queue-based approach is 100,000,000, and 4 bytes for each node would be 100000000*4=400 MB(arr). Memory used by the stack-based approach is 256 bytes. In this program, it is calculated only for 8 levels instead of 9 levels. For 9 levels it would sum up to (9/8)*256=288 that is 288/9=32 bytes per level. In the following section, we will prove that it would be 20 bytes in an ideal case (where local variables would not be there).

In time analysis, we can take the worst-case scenario, that is, level 27 and children count 2. Time taken by the queue-based approach is 43 second and 22,479 microseconds. Time taken by the stack-based approach is 5 second and 773,319 microseconds.

Here we will discuss the 32 bytes taken by each level of the tree.

```
----------------
> gdb
GNU gdb 6.3
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i686-pc-linux-gnu".
(gdb) file a.out
Reading symbols from /home/pravinsi/a.out...done.
Using host libthread_db library "/lib/tls/libthread_db.so.1".
```

```
(gdb) b printtree
Breakpoint 1 at 0x8048802: file stackbasedBFS.c, line 79.
(gdb) run
Starting program: /home/pravinsi/a.out
queue based, allocated for queue size 50 ,each node size 4 bytes
printing queue based
a0123456ABCDEFGABCDEFGABCDEFGABCDEFGABCDEFGABCDEFGABCDEFG
queue based, queue usage size 49
diff time 0 sec 82 micro

printing stack based

Breakpoint 1, printtree (node=0x804b008, target=0, level=0) at stackbasedBFS.c:79
79          __asm__("movl %%ebp, %0;" : "=r" ( i));
(gdb) c
Continuing.

Breakpoint 1, printtree (node=0x804b008, target=1, level=0) at stackbasedBFS.c:79
79          __asm__("movl %%ebp, %0;" : "=r" ( i));
(gdb) bt
#0  printtree (node=0x804b008, target=1, level=0) at stackbasedBFS.c:79
#1  0x080488d6 in printbfstree (root=0x804b008) at stackbasedBFS.c:99
#2  0x08048f51 in main (argc=1, argv=0xbffff894) at stackbasedBFS.c:216
(gdb) c
Continuing.

Breakpoint 1, printtree (node=0x804b018, target=1, level=1) at stackbasedBFS.c:79
79          __asm__("movl %%ebp, %0;" : "=r" ( i));
(gdb) bt
#0  printtree (node=0x804b018, target=1, level=1) at stackbasedBFS.c:79
#1  0x0804886c in printtree (node=0x804b008, target=1, level=0) at stackbasedBFS.c:84
#2  0x080488d6 in printbfstree (root=0x804b008) at stackbasedBFS.c:99
#3  0x08048f51 in main (argc=1, argv=0xbffff894) at stackbasedBFS.c:216
(gdb) f 1
#1  0x0804886c in printtree (node=0x804b008, target=1, level=0) at stackbasedBFS.c:84
84    for(i=0;i<CCOUNT;i++) if(printtree(node->child+i,target,level+1) ) returnval=true;
(gdb) info reg ebp
ebp            0xbffff7a8       0xbffff7a8
(gdb) f 0
#0  printtree (node=0x804b018, target=1, level=1) at stackbasedBFS.c:79
79          __asm__("movl %%ebp, %0;" : "=r" ( i));
(gdb) info reg ebp
ebp            0xbffff788       0xbffff788
(gdb) x/10x 0xbffff788
0xbffff788:     0xbffff7a8      0x0804886c      0x0804b018      0x00000001
0xbffff798:     0x00000001      0x08048d5b      0x000490cf      0x00000000
0xbffff7a8:     0xbffff7c8      0x080488d6
(gdb) x 0x0804886c
0x804886c <printtree+112>:      0x8410c483
(gdb) x 0x08048d5b
0x8048d5b <BFS+413>:    0xc910c483
-----------------
```

When we see the memory region between two consecutive base pointers, (that is, 0xbffff7a8 and 0xbffff788) it is:

```
0xbffff788:     0xbffff7a8      0x0804886c      0x0804b018      0x00000001
0xbffff798:     0x00000001      0x08048d5b      0x000490cf      0x00000000
0xbffff7a8:     0xbffff7c8
```

There is a difference of 32 bytes from one base pointer (0xbffff7a8) to the next base pointer (0xbffff788). Where two are local variables 0x00000000('i') and 0x000490cf('returnval'), and one cached data 0x08048d5b. If we remove the implementation details and talk about generic memory usage, then it would be 8-3=5. 5*4=20 bytes per level.

Left to the reader

As the function recursions use stacks internally, it would be another approach to use stack explicitly and not using the function recursion. It is left to the reader to use another nonrecursive approach. It can save more memory.

Conclusion

Experimental data matches to theoretically placed hypothesis. It has been noted that in the example program provided in this article, stack-based BFS takes 32 bytes per level (of the tree), which includes 8 bytes as the local variable of the algorithm tree recursive function. In case of the tree implemented in another fashion, a programmer has a choice to exclude 8 bytes of local variable. Meanwhile, in this example, we have 32 bytes per level and are calculating for 27 levels in the binary tree, it comes to 864 bytes usage. Whereas, queue-based BFS usage is 67108864*4 bytes (around 256 MB). This is considerable space saving. Time domain (which is a new approach) still holds better in experimental data. In the worst-case scenario, stack-based BFS takes 5.7 seconds whereas queue-based BFS takes 43 seconds.

References

Fundamentals of data structure by Ellis Horowitz and Sartaj Sahni, 1983, ISBN 0716780429

Downloadable resources

Description	Name	Size
algorithm c program file for this tutorial	stackbasedBFS.c	3KB