Depth-First Search and Its Use Case in Distributed Systems Debugging

Nate Kremer-Herman
Depth-first search kernel

Given a large graph.
Start at a root node.
Find all reachable vertices.
Measured in TEPS, just like BFS.

Worst case performance:
$O(|V| + |E|)$
Iterative pseudocode

1 procedure DFS-iterative(G,v):
2     let S be a stack
3     S.push(v)
4     while S is not empty
5         v = S.pop()
6         if v is not labeled as discovered:
7             label v as discovered
8         for all edges from v to w in G.adjacentEdges(v) do
9             S.push(w)
Implementation techniques

- Implemented in Perl
  - Regex matching
  - Data structures

- Used the iterative algorithm
  - Can only use recursion to a certain depth
  - Faster albeit less elegant

- Is essentially a bottom-up DFS (kinda)
  - I cheat and make the leaves the roots
For Perl wizards:

```perl
sub iterative {
  my ($v) = @_;
  my $return = "";
  my @stack;
  push(@stack, $v);
  while(scalar(@stack)) {
    my $n = pop(@stack);
    if(($nodes{$n}{'v'} != $i)) {
      $nodes{$n}{'v'} = $i;
      $return = $return . "$n:";
      $traversed++;
    }
    my @children = split(";", $nodes{$n}{'c'}});
    my @attrs = split(";", $nodes{$n}{'a'}});
    foreach my $c (@children) {
      my @cattrs = split(";", $nodes{$c}{'a'}});
      my $cf = 0;
      foreach my $ca (@cattrs) {
        foreach my $na (@attrs) {
          if(substr($na, 1) eq substr($ca, 1)) {
            if($c >= 0) { push(@stack, $c); }
            $cf = 1;
          }
        }
      }
      if($cf) { last; }
    }
  }
  return $return;
}
```
Notional summary (for everyone else)

1 \textbf{procedure} DFS-iterative(G,v):
2 \hspace{1em}push v on a stack, S
4 \hspace{1em}while S is not empty
5 \hspace{2em}v = S.pop()
6 \hspace{1em}if v has not been visited in this round:
7 \hspace{2em}label v as visited
8 \hspace{1em}for all child edges of v do
9 \hspace{2em}if child has a matching attribute with v:
10 \hspace{3em}S.push(child)
Updated complexity analysis

- Algorithm is still $O(|V| + |E|)$
  - Worst case, we look at all vertices
  - Best case, we look at no vertices

- Added overhead for attribute analysis
  - We only keep vertices which share attributes (for debugging)
  - Attribute checking slows down traversal by an order of magnitude or two (fun)
- Measured performance in TEPS
- Captured error nodes in separate graphs
  - Examples to come
Datasets

- All datasets are currently synthetic
  - Each is a binary graph
  - Generated via Perl script

- Number of nodes ranges from 10 - 1,000,000
  - Realistic dataset size $O(100) - O(10,000)$
  - Tiny: 10 nodes
  - Small: 100 nodes
  - ... 
  - Colossal: 1,000,000 nodes
  - Any bigger runs into memory limitations
Small DAG example:

Only 100 nodes

Still a headache to parse through by hand.

No highlighting of failed task lineage! I have to switch between a debug log and this graph.
Much more manageable output per failed task.
Sometimes a task fails on its own, not because of a previous task.
### Implementation performance results

<table>
<thead>
<tr>
<th>Nodes</th>
<th>TEPS</th>
<th>Nodes</th>
<th>TEPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>361,347.81</td>
<td>6 (10)</td>
<td>60,676.46</td>
</tr>
<tr>
<td>100</td>
<td>375,014.11</td>
<td>71 (100)</td>
<td>74,231.57</td>
</tr>
<tr>
<td>1,000</td>
<td>367,753.35</td>
<td>872 (1,000)</td>
<td>69,404.00</td>
</tr>
<tr>
<td>10,000</td>
<td>500,512.47</td>
<td>9,724 (10,000)</td>
<td>118,423.95</td>
</tr>
<tr>
<td>100,000</td>
<td>476,622.42</td>
<td>99,032 (100,000)</td>
<td>99,258.17</td>
</tr>
<tr>
<td>1,000,000</td>
<td>458,047.21</td>
<td>998,270 (1,000,000)</td>
<td>120,060.52</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Only Traversal</th>
<th>Traversal + Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes</td>
<td>TEPS</td>
</tr>
<tr>
<td>10</td>
<td>361,347.81</td>
</tr>
<tr>
<td>100</td>
<td>375,014.11</td>
</tr>
<tr>
<td>1,000</td>
<td>367,753.35</td>
</tr>
<tr>
<td>10,000</td>
<td>500,512.47</td>
</tr>
<tr>
<td>100,000</td>
<td>476,622.42</td>
</tr>
<tr>
<td>1,000,000</td>
<td>458,047.21</td>
</tr>
</tbody>
</table>
Plans for parallel implementation

- Use Work Queue master-worker framework to parallelize traversal
  - Cascading traversal pattern
  - May not be faster than serial implementation for a realistic dataset (resource acquisition)

- Use real data if there is time
  - Only roadblock is transforming debug logs into graphs, traverser is done
  - Each type of log has its own syntax, so each requires a handwritten parser
Questions?