Search Optimization: Uses of Breadth first and Depth first searches

Steffen L.

Artificial Intellegence CS 4242
Kennesaw Sate University
Marietta, 1100 South Marietta Parkway 30060

27 Feb 2017

Abstract

The uses of tree structures with organizing data and breadth and depth-first searches have been the best candidates to extract data once placed in the tree structure. A breadth-first search is a level order algorithm that traverses all the parent nodes before the children nodes. A depth-first search would traverse a path until it reaches a leaf node and backtracks to a new branch until it finds the right node. Its fundamental underlying difference in structure can be interpreted as the use of stacks and queues and a recursive algorithm. This project aims to demonstrate the uses of depth-first and breadth-first searches in an implementation for a behavior tree.

1 Introduction

Tree searches can come in many forms. The different forms can exist because of the different tree structure or different optimization benefits. Tree structures such as binary search trees would use a binary search to search the tree since it is the most optimized search algorithm for an ordered binary tree. The issue with that particular scenario is getting to a situation where one would be able to implement a binary tree structure into their projects. In the case of the behavior tree, there is not really a way to organize the behaviors from least to greatest that would make sense. The purpose of a behavior tree would be to create branching choices that an AI could make in specific situations. Reordering it to an alphabetical order, for example, would break that decision tree pathing that the behavior tree has. In this case, a behavior tree would be classified as an unordered tree and would require search algorithms that do not need an ordered data structure to run correctly. Two such examples would be the breadth-first and depth-first searches.

Depth-first search was first investigated in the 19th century by a French mathematician named Charles Pierre Trémaux. He created an undirected graph where he then used a depth-first search to create a strategy for solving mazes [1]. The depth-first search algorithm has two implementations, one that is recursive and one that is non-recursive. In a recursive implementation, the algorithm would search through all the children of a node, and it would call itself on each children node. It would return the searched node by passing it up the recursive calls. An example implementation of a recursive depth-first search would look like this:

Listing 1: Recursive Depth-First Search

```
template <typename type>
type* depth_search_recursive(std::
    string search, type* node, int&
    count){
  if(node != NULL){
    // count number of checks here
    ++count;
    if(node->get_data().get_behavior()
        == search || node->get_data().
       get_response() == search){
      return node;
    }
    type* temp = NULL;
    for(size_t i=0; i < node->
       get_num_children(); i++){
      temp = depth_search_recursive(
          search, node->get_child(i),
           count);
```

```
            if(temp != NULL){
            return temp;
          }
        }
    }
    // node == NULL or not found
    return NULL;
}
```

The non-recursive implementation of depth-first search uses a stack of children nodes to traverse. The algorithm would append the list of children as it traverses the tree into a stack. Then in a while loop, the top element of the stack is popped off and checked to be the resulting search. If it is not the result the algorithm was looking for, then the child of that top node is appended to the top of the stack. This implementation is slightly different from the recursive implementation in that it searches the children from the last child first. So instead of searching for the first child then second child, it would iterate through the second child before iterating through the first child. An example of that code would look something similar to this:

Listing 2: Non-Recursive Depth-First Search

```
template <typename type>
type* depth_search(std::string search,
     type* node, int& count){
  if(node != NULL){
    std::stack<type*> list;
    list.push(node);
    while(!list.empty()){
      type* top = list.top();
      list.pop();
      // count number of checks here
      count++;
      if(top->get_data().get_behavior
          () == search || top->get_data
          ().get_response() == search){
        return top;
      }
      for(size_t i=0; i < top->
          get_num_children(); i++){
        list.push(top->get_child(i));
      }
    }
  }
  // node == NULL or not found
  return NULL;
}
```

For the breadth-first search implementation, the algorithm is very similar to the non-recursive implementation of the depth-first search algorithm. Instead of using a stack where the first input is the last output, the algorithm uses a queue where the first input is the first output. In the loop, all the children are appended at the end of the queue and therefore the parents are checked before their children. This makes the breadth-first search a level order search. It iterates through the entire level in the tree before iterating through the next level. Here is an example implementation:

Listing 3: Breadth-First Search

```
template <typename type>
type* breadth_search(std::string
    search, type* node, int& count){
  if(node != NULL){
    std::queue<type*> list;
    list.push(node);
    while(!list.empty()){
      type* top = list.front();
      list.pop();
      // count number of checks here
      count++;
      if(top->get_data().get_behavior
          () == search || top->get_data
          ().get_response() == search){
        return top;
      }
      for(size_t i=0; i < top->
          get_num_children(); i++){
        list.push(top->get_child(i));
      }
    }
  }
  // node == NULL or not found
  return NULL;
}
```

Note that the only difference between the non-recursive implementation of depth-first search and the breadth-first search is the use of queue data structure rather than stack data structure. Switching the order of when the children should get iterated changes the search order from depth to breadth.

Comparing both depth-first implementations and breadth-first searches, they all have the same worst case scenario of $O(n)$ where n is the number of nodes in the tree. The worst possible case of a search is a linear correlation between the number of nodes in the tree. The algorithms, however, have a different distribution of searches and there are different benefits to all searches in the right circumstances. Rather than looking at just the Big-O notation, the algorithms are different in the order in which the tree is searched. To

look at the benefits of the search order, the algorithms should be tested on different shaped trees on different search probabilities.

2 Related Works

Alexandria University focused on space efficiency due to the ubiquitous use of hand-held technology that have limited processing and memory capabilities. They created an efficient breadth-first search algorithm that was able to compute in $\Theta(loglogn)$ in number of bits [2]. This was compared to an unbounded memory algorithm. Some instances however were limited to at best a $\Theta(nlogn)$ bit efficiency without dramatically increasing processing time. The Institute of Mathematical Sciences in India improved on their design by reducing the memory requirement to $1.585n + o(n)$ bits [3]. This implementation required more time to compute $O(mlognf(n))$ where m and n are the two dimension sizes of the graph/tree and f(n) is a slow growing function of n.

On a different note, the implementation of behavior trees are also of interest. The Delft University of Technology was able to simulate and test in real world environment a flying robot capable of autonomously traversing through a window using a behavior tree and on-board processing and stereoscopic camera [4]. This was able to run comparably well to a user.

3 Methodology

The tree set up for the project is a behavior tree for a possible game AI decision tree. It consists of nodes containing strings of behavior and responses. The leaf nodes are designed to be the responses for the branching behaviors. The structure of the tree is an unordered m-way tree. It has an unbounded number of children for each node. The behavior tree is an unordered tree where the structure is dependent on the possible choices an AI could make from the subsequent branch.

The tree is set up for three different algorithms, a breadth-first search, a recursive depth-first search, and a non-recursive depth-first search. With the three search algorithms, there are five different test trees to search through. The first tree is a top-heavy tree where there are more branches than there are leaves. The second is a bottom-heavy tree where there are more leaves than there are branches. The third is a left-heavy tree where there are more leaves on the left side of the tree with a balanced amount of branches, and vice versa with a right-heavy tree. The final tree is a balanced tree with the same number of branches and leaves.

In the project, there were two ways the search algorithms could be tested. The first way would be to iterate through a search on every node once and only once. Then a counter would measure the number of iterations each search would take and it would be averaged for the whole tree. The issue with this test is that all the search algorithms perform with the same number of iterations total and averaged over the entire tree. More details are explained in the Results.

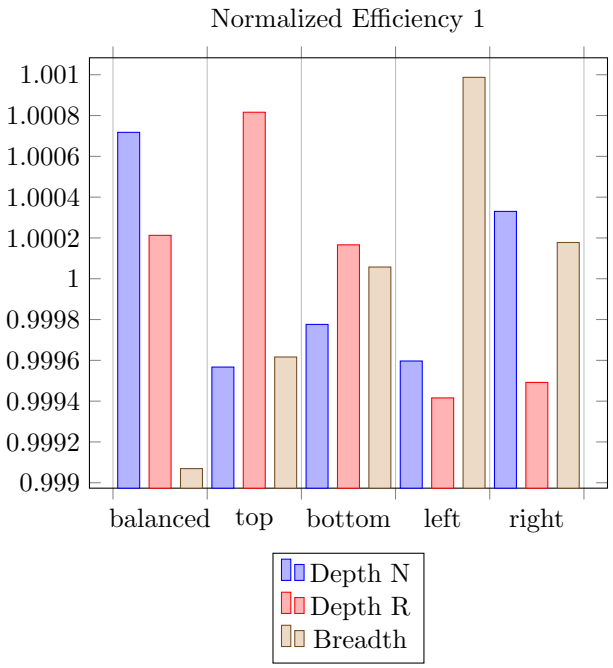
The second setup for tests would be to iterate a set number of times and search for a random node each time. In each iteration, the algorithm would generate a random search node r and each algorithm would search for r and return the number of iterations it took to find r. The random sample from the tree would simulate a random distribution of searches on the tree. This test represents a more real-world representation of usage in a behavior tree.

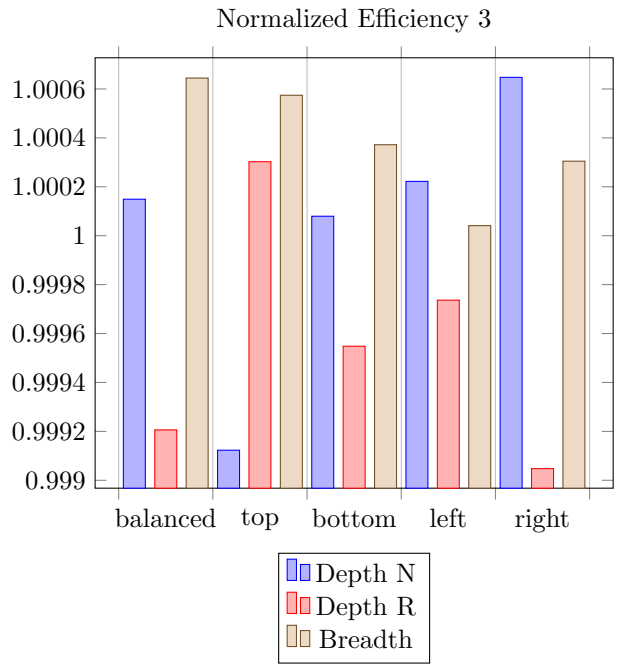
4 Experimental Results

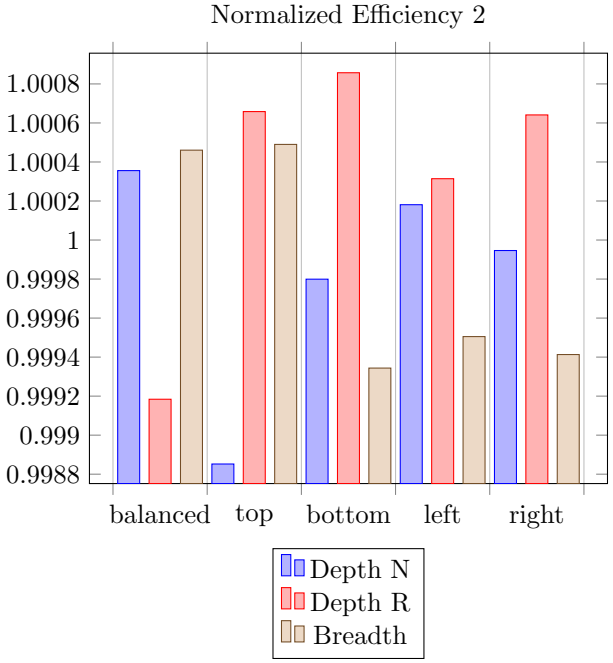
In the test where each search algorithm runs on a tree once and only once, the results are equal across the board. Each search algorithm searched through all the nodes in all trees with the same number of iterations. Since both algorithms have the same O(n), this test does not show any difference. This does not represent a good test on the different optimizations each search algorithm have. For each test, all three implementations of search algorithms have the same number of iterations it takes to search through all trees.

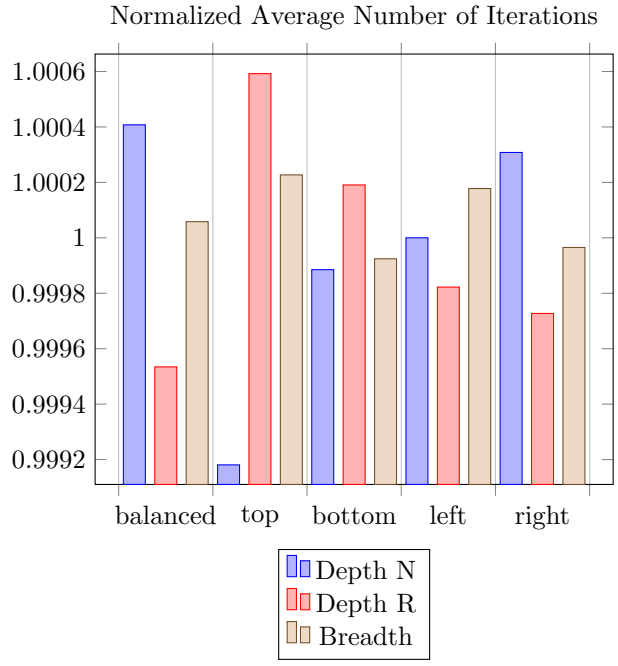
The next test performed uses a probabilistic distribution to search through the trees. By randomly picking which node to search, it creates a more realistic distribution of which nodes get picked. This way, in theory, the top-heavy trees where there are more branches than leaves, the search would be more likely to search a branch than a leaf. This is where the different unbalanced trees should show how some of the search algorithms behave in practical applications.

The results of the different trees averaged over 1,000,000 runs on all permutations of trees and search algorithms are as follows:









The graph is produced by normalizing all the number of iterations on all the trees by the average result for the implementations for each tree. For example, the search algorithms for the balanced tree are all normalized to the average result of all implementation for the balanced tree.

$$I_n = \frac{3s}{\sum s_i}$$

Where I_n is the normalized number of iterations, s is the sum of iterations over the 1,000,000 runs, and i is

iterating through all three search implementations.

The data results are very close to each other and may be within the error of the random function. After taking three different tests, the search algorithms are fluctuating in efficiencies on all the trees. They are not consistent.

5 Conclusion

Based on these data alone, there is no clear cut advantages for any implementation over another. The results are very sparse and they are mostly due to the random function than any efficiencies in the algorithms. Rather if the search space was limited to specific nodes where there are deep trees, maybe the efficiencies will be more pronounced. The general consensus for the benefits for a breadth-first or depth-first searches are the limitations of memory and any known target location for the search.

A breadth-first search or a non-recursive depth-first search tend to use more memory by storing the locations of all the children. In a deep tree with limited memory, those search implementations may not be practical. If the data to be searched tend to stay near the root of the tree, then a breadth-first search should be optimal since it should reach the target node in fewer iterations.

References

[1] Even, Shimon (2011), *Graph Algorithms* (2nd ed.), Cambridge University Press, pp. 4648, ISBN 978-0-521-73653-4.

[2] A. Elmasry, T. Hagerup, and F. Kammer. Space-efficient basic graph algorithms. In *32nd STACS*, pages 288301, 2015.

[3] Banerjee, N., Chakraborty, S., & Raman, V. (2016). Improved Space efficient algorithms for BFS, DFS and applications.

[4] Scheper, K. W., Tijmons, S., de Visser, C. C., & de Croon, G. E. (2016). Behavior Trees for Evolutionary Robotics. Artificial Life, 22(1), 23-48.