Chapter 1

Distributed System Debugging and Depth First Search

Contributed by Nathaniel Kremer-Herman

1.1 Introduction

There are two common algorithms for traversing the entirety of a graph. These are **depth first search** (**DFS**) and **breadth first search** (**BFS**). The DFS algorithm begins at an arbitrary root node and traverses as far as it can down one branch of the graph until it cannot traverse any further. It then backtracks up to the closest unexplored branch and begins its traversal, repeating this exploratory step. BFS works in an inverse manner. It traverses each unexplored node adjacent to the root node in the first step. Each adjacent node from these newly-traversed nodes become a frontier which BFS explores one node at a time rather than one branch at a time like in DFS. While this allows BFS to explore each branch of a graph in a somewhat concurrent manner, it can leave a large memory footprint. Depth first search, on the other hand, is less memory-intensive since it traverses an entire branch in one iteration then relinquishes the memory used for those nodes once the branch has been exhausted. Depth first search is explored in this chapter.

An application for which DFS is useful is debugging a distributed system. In this chapter, a distributed system is defined as a set of machines which operate within some shared resource ecosystem such that the processes which make up an application can run on these multiple machines. Example applications include scientific workflows, machine learning applications, large-scale simulations, etc. Each of these applications has some inherent sense of scalability which a distributed system can provide. An application is composed of a set of atomic units of work called *tasks*. A task is typically a process which consumes some pre-defined input data to produce expected output data which is used by subsequent tasks. An application is thus a pipeline (or multiple concurrent pipelines) of tasks which consume some original input data to produce some final result.

At the University of Notre Dame, researchers have access to a distributed system consisting of multiple machines totaling at approximately $25,000$ cores. These machines are connected by multiple resource management services. These includes an Apache Hadoop cluster, the HTCondor batch system, and the UGE batch system. Researchers write parallel applications which make use of these resources via Hadoop, HTCondor, and UGE (as well as other boutique resource managers they stand up on their own). These researchers are typically used to running their analyses, simulations, and other applications sequentially on their laptop or workstation. When asked to parallelize their work, a researcher is usually sent outside of their coding comfort zone.

It is often the case that the researcher will not fully understand the level of scaling their application can handle. This could mean they write their application to be embarrassingly parallel when in fact it will not operate as such. On top of understanding the behavior of an application is the necessity of understanding how each task of an application will interact with its runtime environment. A runtime environment consists of environment variables, files, libraries, and resources like cores, memory, and disk. If a user does not properly specify what the environment for each task should be, there will be issues at runtime. This is especially true of tasks whose environment is specified by other tasks which ran before it.

For example, take task n and task $n + 100$. Task n must successfully complete before task $n + 100$ can execute. Task n performs some work and sets an environment variable on machine A. Machine A is located in the UGE cluster. For the sake of realism, let us assume task n sets the Java `$CLASSPATH` environment variable because the task is in charge of noting where some Java libraries are located. A few hours later, task $n + 100$ executes on machine B. Machine B is in the HTCondor cluster, it has a different operating system than machine A, and it does not have access to the shared filesystem which machine A uses. We can infer what will happen to task $n+100$. The process will read in the `$CLASSPATH` originally set by task n and run a Java program. It is at this point that task $n + 100$ will catastrophically fail! It is not able to link to the libraries specified in `$CLASSPATH` because those libraries are located in a shared filesystem only accessible by machines in the UGE cluster.

It is our experience that tasks often obfuscate these kinds of failures, making them difficult for end-users to debug. This may be due to error handling by the process which returns a different error message than the root cause, obfuscation by resource management software like virtual machines or containers intercepting the environment failure, error obfuscation by the distributed system resource managers like a batch system, or perhaps by custom error handling the researcher has written into their own application. It would be preferable to insert some logging mechanism within each task which tracks how it interacts with its environment. This removes all obscuring of the underlying cause of failure at the task level. However, this does not tell us *why* this task's environment induced a failure. For that, we would need to find out from where this task received its environment specification. This second problem can be implemented as a graph and solved via depth first search.

1.2 The Problem as a Graph

We can represent the history of tasks of an application as a directed, acyclic graph (DAG). The tasks which comprise an application have dependencies between each other. In the previous example, we stated task $n + 100$'s execution was contingent on task n's successful execution. An application can consist of $O(10)$ tasks up to $O(1,000,000)$ tasks, though typically they range from $O(100)$ to $O(100,000)$. Each tasks is a vertex in the DAG. The edges in the DAG represent the dependencies between tasks. Each vertex may be dependent upon multiple prior vertices, and it may serve as a dependency for multiple subsequent vertices.

1.3 Some Realistic Data Sets

As stated in Section 1.4, distributed applications typically range from $O(100)$ to $O(100,000)$ tasks, but an application may have any arbitrary number of tasks. The only logical constraint on task number is whether the distributed system can handle the resources consumed by the outputs of those tasks (e.g. disk space). Debugging information is tracked on a per-task basis, leading to the creation of a preponderance of log data throughout the lifetime of a distributed application. The domains from which these applications arise vary greatly. Domain scientists from the natural sciences like high-energy physics, bioinformatics, and computational chemists as well as social scientists all have a burgeoning need for large-scale computational resources. Digital humanities have experienced a similar uptick in distributed computing applications.

To demonstrate the effectiveness of depth first search in distributed systems debugging, we can derive the necessary divergence from a traditional implementation of DFS from first principles. To do this, we must first generate arbitrary DAGs. These DAGs will be composed of vertices which are tagged as failed or succeeded. Each vertex will also have associated data about properties it has inherited from it parent(s) and properties it passes down to its descendants. We can do this by writing a DAG generator program. This abstracted view allows us to both construct a modified depth first search algorithm for the purpose of distributed system debugging *and* provides a method by which to test the scalability of the algorithm.

After testing with the synthetically generated DAGs, we can then test the modified DFS on real log data. The logs will be created after execution of two different applications: a scientific workflow and a machine learning application. Both are executed in a master-worker framework, meaning there is a centralized master process in charge of transmitting input and output data as well as dispatching tasks to worker processes on different compute nodes. The scientific workflow is a bioinformatics application called BWA-GATK. It makes genomic comparisons from a small query dataset to a larger reference dataset. The machine learning application is called SHADHO, and it is used on the MNIST dataset to train on character recognition. Both applications consist of over $1,000$ tasks and execute for multiple hours. The amount of tasks and total execution time are both configurable.

1.4 DFS - A Key Graph Kernel

Depth first search as a graph kernel makes the most sense for the application of distributed debugging. Although there are more intricate methods of traversing graph structures, we will need to visit every vertex without much preference for which branch to visit first. DFS allows for this total exploration. Breadth first search is another valid candidate, however we have to consider memory constraints which may hinder the performance of BFS. This is especially true if we implement a parallel method of traversing the graph since we cannot assume the size of available memory to be large across multiple compute vertices.

Depth first search begins at an arbitrary root vertex. This vertex is labeled as visited. From that root vertex, the algorithm gathers all the adjacent vertices to the selected root. For each adjacent vertex, the algorithm checks if it has been visited. If it has not been visited, the algorithm traverses to the vertex and repeats that step of gathering adjacent vertices. This process is repeated until all vertices have been visited.

The iterative implementation of depth first search, shown in Algorithm 1, requires a stack. vertices are pushed onto this stack if they are adjacent to the vertex most recently popped off the stack. For each unvisited vertex, DFS labels it as visited and pushes its neighbors onto the stack. This is repeated until all vertices have been visited and the stack is empty.

The recursive implementation of DFS, shown in Algorithm 2, is sleeker. There is no space overhead of a stack. For each vertex, the algorithm recursively calls itself on all vertices adjacent to the current one. On each recursive call, the vertex passed into the algorithm is labeled as visited. The algorithm continues until no new unlabeledvertices exist.

The worst case time complexity for running both implementations of the algorithm is $O(|V| +$

Algorithm 1 Iterative algorithm

```
procedure DFS($G, v$)
    let $S$ be a stack
    $S$.push($v$)
    while $S$ is not empty do
        $v \leftarrow S$.pop()
        if $v$ is not labeled as visited then
            label $v$ as visited
            for all edges from $v$ to $w$ in $G$.adjacentEdges($v$) do
                $S$.push($w$)
```

Algorithm 2 Recursive algorithm

```
procedure DFS($G, v$)
    label $v$ as visited
    for all edges from $v$ to $w$ in $G$.adjacentEdges($v$) do
        if $w$ is not labeled as visited then
            call DFS($G, w$)
```

$|E|$), where $|V|$ represents the number of vertices and $|E|$ represents the number of edges in the graph. This time complexity is due to the fact that the algorithm must traverse the whole graph, visiting each vertex only once. Space complexity for DFS is $O(|V|)$. This is obvious in the case of the iterative implementation since a stack is kept of each vertex the algorithm must visit.

To gauge the performance of an exploratory graph algorithm, it makes sense to track how quickly it can traverse the entire graph. We can measure this in traversed edges per second (TEPS). This measurement is commonly used for depth first search and breadth first search.

1.5 A Sequential Algorithm

To implement the sequential depth first search algorithm, we must first take a look at how the graph should be represented in-memory. The graph should be structured as a collection of vertices whose edges are pointers to other vertices. An object-oriented approach, while viable, seems a bit heavy-handed since the complexity of the data structures needed to represent a DAG are quite low. Any standard language which allows for a data structure to store a few values and point to another data structure is sufficient. We want to emphasize that standard languages should convey a connotation of portability, which will be important for when the algorithm is scaled up to a parallel execution model.

In particular, a linked list or other array-like data structure provides the proper primitives storage and interface for DAG traversal. The properties of this data structure are:

- A unique vertex ID.
- A pointer to each child node.
- A key-value array to store properties (i.e. environment variables).
- A flag indicating whether the node has been visited or not.

Neither the time nor space complexity should increase in the implementation of the algorithm. Since no additional looping or significant functionality is added beyond simple bookkeeping during traversal, both space and time complexity remain linear. The whole graph must be read into memory, so the space complexity remains $O(|V|)$. The time complexity remains $O(|V| + |E|)$.

1.6 A Reference Sequential Implementation

The sequential implementation of depth first search was done in the Perl scripting language. Because it is an interpreted language, we can expect the performance to be slower (in TEPS) than a more performant, compiled program written in C, for example. The iterative version of the algorithm was implemented due to how it interacts cleanly with the Perl environment and data structures, namely the Perl `hash` structure.

The graph is represented as a Perl `hash`. Each element in the `hash` has a `visited` element, representing whether it has been visited by the algorithm. The stack in the iterative algorithm is implemented as a Perl `array` from which the vertices are pushed and popped. Rather than have a separate function to determine which vertices are adjacent to the current one, the `hash` keeps a record of a vertex's neighbors. In this case, since every graph traversed is a DAG, neighbors are child nodes.

1.7 Sequential Scaling Results

To evaluate the scalability of the sequential implementation of depth first search, synthetic DAGs of different sizes were generated. The generator created a binary DAG. This means that each vertex has *at most* two children. Some nodes only had one child. A pseudo-root and pseudo-leaf node were added at the top and bottom of the DAG to ensure a clean beginning and end to the graph.

Each successive DAG was an order of magnitude larger than the previous. This is shown in the left column of Table 1.7. Since only the scalability of DFS is to be measured, no debugging measurements or bookkeeping was kept. The DFS algorithm was the only operation performed on the generated graph. DFS was executed 100 times on each graph.

Vertices	**Avg. Time (s)**	**Avg. TEPS**
10	0.00003	361,347.81150
100	0.00027	375,014.11840
1,000	0.00272	367,753.35340
10,000	0.02108	500,512.47540
100,000	0.21075	476,622.42660
1,000,000	2.18351	458,047.21580
10,000,000	22.39310	446,715.40780

As expected from the complexity analysis of Section 1.4, the total time taken to traverse the graph scales linearly with the order of magnitude increase of the graph size. This validates that the sequential implementation of the algorithm remained lightweight in any additional bookkeeping that was added. In addition, it is worth noting the average traversed edges per second (TEPS) seemed to increase with scale. However, it is also worth noting that the experiments ran on a production machine which is in an active computer cluster. The machine used had 8 cores, 32GB of memory, and 2TB of disk space on the disk the experiment was executed. The experiment consumed only a single core and eventually consumed all available memory when attempting to scale to $100,000,000$ vertices. The seeming increase in performance could simply be due to less load on the machine at runtime of the later experiments. The granularity of this scalability benchmark makes it impossible to know for sure why TEPS increased with the increased scale of the graph.

1.8 Conclusion

The depth first search graph kernel was introduced as a potential candidate for distributed debugging, alongside breadth first search as another potential candidate. The smaller memory footprint of DFS, along with its behavior of searching an entire branch at a time, makes it the preferable algorithm for analyzing debugging logs of distributed applications. Its linear time and space complexities are demonstrated in two different implementations of a sequential algorithm, iterative and recursive. Also discussed were the needs of an implementation for this algorithm.

A Perl implementation of the iterative sequential algorithm was presented along with a brief evaluation of its scalability on a single production machine used in a research computing cluster at the University of Notre Dame. The next step is to implement a parallel version of depth first search and compare its performance to the sequential algorithm. It is expected that there is some scale at which the parallel algorithm will outperform the sequential one.