Chapter 1

Distributed System Debugging and Depth First Search

Contributed by Nathaniel Kremer-Herman

1.1 Introduction

One common algorithm for traversing the entirety of a graph is **depth first search** (**DFS**). 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. Depth first search is less memory-intensive than other common graph traversal algorithms 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 specification is contingent upon other tasks which ran before it.

For example, take task n and task $n + 100$ within a single parallel application. 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 (perhaps this is specified in a configuration file that has been passed down from 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. Some parallel applications consist of $O(10)$ tasks up to $O(1,000,000)$ tasks, though at the University of Notre Dame typically they range from $O(100)$ to $O(100,000)$ as they execute on the campus cluster. Each task is a vertex in the DAG. The edges in the DAG represent the dependencies between tasks. There exists only one edge between two vertices. This typically denotes a dependency of a file existing. 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 1.2, distributed applications at the University of Notre Dame 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. This was done using a Perl script. These synthetic DAGs are composed of vertices which are tagged as failed or succeeded. A configurable rate of failure will determine how likely it is a vertex will be tagged as failed during graph generation. 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 a scientific workflow application. The workflow is 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, following a scatter-gather execution pattern. This means there is one task at the beginning of the workflow which partitions a large input dataset (the reference in BWA-GATK). Then, a number of tasks can run concurrently as the query dataset is applied to each partition of the reference dataset. Finally, a task gathers all the output data from the concurrent queries and aggregates them into a single set of condensed output files. BWA-GATK runs Java codes which are dependent upon both a specific version of Java and the correct setting of the Java `$CLASSPATH` environment variable. Its successful execution is also dependent upon each task generating the expected output files. This application consists of over $1,000$ tasks and executes for multiple hours. The amount of tasks and total execution time are both configurable.

1.4 DFS - A Key Graph Kernel

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. Given graph G and vertex v, we begin a depth first search starting by pushing v on the 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 worst case time complexity for the algorithm is $O(|V| + |E|)$, 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 apparent since a stack maintains each vertex the

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 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 Prior and Related Work

There are many commons problems in distributed debugging which tools attempt to solve. These include developing a consistent definition of time [11], reaching a consensus of distributed state [12], and the development of distributed snapshots [3]. However, there has yet to be a *generalized* solution to parsing distributed debug logs and output logs to find, with absolute certainty, the root cause of errors and failures. This remains an open problem.

One aspect of distributed application failure is the misconfiguration of the runtime environment. Part of this is due to a lack of transparency to the user what all must be specified in an environment configuration which is the driving impetus behind [19]. Other tools attempt to do the work of specifying a complete environment configuration *for* the user. Lightweight virtualization technologies such as containers [13, 10] are commonly used for this purpose as they provide a full software and library stack to support user applications. Other tools attempt to make the configuration of an environment easy at the user level without virtualization [14, 4, 8, 7, 1, 18, 6]. As much as these tools provide a user with environment configuration assistance, they do not prevent misconfigurations from occurring. Distributed applications can still alter the state of the environment within virtualized spaces or user-level sandboxes, potentially causing errors as they would outside the confines of these tools. Further, they may obfuscate the root cause of distributed application errors since they may intercept these errors.

There are two typical methods of debugging distributed application errors. The first is to include tracing mechanisms at runtime to catch error messages, intercept system calls, and report failures [5, 9]. Tools which do this create a performance overhead on a distributed application, but they provide an exact trace of causation for errors (so long as the tracer captures all that information). Performance is traded in exchange for a complete listing of all relevant messages, calls, and failures to make debugging easy. These kinds of tools are also typically designed for only one distributed architecture [5] or a specific use case rather than general distributed debugging [9]. The second method of debugging distributed applications is to use after-the-fact analysis tools to trawl debug logs [2, 16, 15, 17]. These tools sacrifice a complete trace in exchange for producing no overhead at runtime. Because a trace does not exist, these tools must *infer* whether one event in the log is causally linked to another. There are many statistical models which guide these tools, with varying degrees of accuracy when compared to a ground truth, but overall they are meant to provide a quick method of combing through logs. When done by a human (even a system administrator or domain expert), the manual investigation of errors in debug logs takes a long time and is tedious. Both the tracing tools and debug log analysis tools attempt to decrease the burden for manual debugging as much as possible.

Using depth first search on debug logs is an attempt to merge both types of tools together. The DFS debugger relies upon a trace of certain system calls as well as *a priori* knowledge of task dependencies to investigate the causes of errors and failures. However, the debugger is an after-the-fact traverser of debug logs. What it gains from the tracer is an absolute lineage of relationships between processes based upon which environment variables they consume and which files they access. What the debugger gains from log analyzers is a level of interactivity and the production of condensed, actionable output to aid a user in their debugging.

1.6 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 vertex.
- A key-value array to store properties (i.e. files and environment variables).
- A flag indicating whether the vertex has been visited or not.
- A flag indicating whether the vertex has failed 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 $O(|V| + |E|)$.

When the iterative DFS algorithm is used for debugging, there are a few additions which need to be made. Rather than simply traversing all vertices, the algorithm should only push vertices onto the stack which have some common property with the current one. In the average case, this should reduce the number of vertices visited. At worst case, all vertices are still traversed. Most importantly, the algorithm should be called using a vertex that is labeled as failed rather than an arbitrary root node as defined in the original algorithm. In order for the traversal to have any use in debugging, we must return an ordered set of vertices visited (this represents causal relationships). Algorithm 2 demonstrates the added checking which must be done to traverse only vertices which have some commonality with the current vertex.

Algorithm 2 Iterative debugging algorithm

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

1.7 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.

1.8 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 1.8. 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.

1.9 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.

1.10 Response to Reviews

From the feedback received, I noted there was a fair amount clarifications needed. To avoid confusion, I removed all mentions of breadth first search since the comparison between BFS and DFS seemed to muddy the introduction. Also included are various clarifications about minor points such as that the example in the introduction is for a single parallel application. The introductory paragraph of Section 1.4 was removed. A forward-reference to a section was fixed. This was a reference to the wrong section. Most importantly, a related work section has been included. I was initially unsure what was to be included in that section, but it has now been fleshed out. A more complete explanation of how the debugging is done in-traversal has been added to Section 1.6.

Bibliography

[1] Minos: portable binaries for linux. http://s.minos.io.

[2] Diogo Behrens, Marco Serafini, Sergei Arnautov, Flavio P. Junqueira, and Christof Fetzer. Scalable error isolation for distributed systems. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, pages 605–620, Berkeley, CA, USA, 2015. USENIX Association.

[3] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, February 1985.

[4] Eelco Dolstra, Merijn de Jonge, and Eelco Visser. Nix: A safe and policy-free system for software deployment. In *18th Large Installation System Administration Conference (LISA '04)*, pages 79–92, 2004.

[5] Nikoli Dryden. Pgdb: A debugger for mpi applications. In *Proceedings of the 2014 Annual Conference on Extreme Science and Engineering Discovery Environment*, XSEDE '14, pages 44:1–44:7, New York, NY, USA, 2014. ACM.

[6] Bastian Eicher and Thomas Leonard. Zeroinstall. https://0install.net.

[7] John L. Furlani and Peter W. Osel. Abstract yourself with modules. In *Proceedings of the Tenth Large Installation Systems Administration Conference (LISA '96)*, pages 193–204, 1996.

[8] Todd Gamblin, Matthew LeGendre, Michael R. Collette, Gregory L. Lee, Adam Moody, Bronis R. de Supinski, and Scott Futral. The spack package manager: Bringing order to hpc software chaos. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 40:1–40:12, New York, NY, USA, 2015. ACM.

[9] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 71–85, Berkeley, CA, USA, 2014. USENIX Association.

[10] Gregory M. Kurtzer, Vanessa Sochat, and Michael W. Bauer. Singularity: Scientific containers for mobility of compute. *PLOS ONE*, 12(5):1–20, 05 2017.

[11] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[12] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982.

[13] Dirk Merkel. Docker: Lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239), March 2014.

[14] Hisham Muhammad and André Detsch. An alternative for the unix directory structure£. In *Proceedings of the III WSL-Workshop em Software Livre, Porto Alegre*, 2002.

[15] Tongqing Qiu, Zihui Ge, Dan Pei, Jia Wang, and Jun Xu. What happened in my network: Mining network events from router syslogs. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, IMC '10, pages 472–484, New York, NY, USA, 2010. ACM.

[16] Colin Scott, Vjekoslav Brajkovic, George Necula, Arvind Krishnamurthy, and Scott Shenker. Minimizing faulty executions of distributed systems. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 291–309, Santa Clara, CA, 2016. USENIX Association.

[17] Li Tan, Min Feng, and Rajiv Gupta. Lightweight fault detection in parallelized programs. In *Proceedings of the 11th ACM/IEEE International Symposium on Code Generation and Optimization (CGO)*, pages 1–11. IEEE, 2013.

[18] B. Tovar, N. Hazekamp, N. Kremer-Herman, and D. Thain. Automatic dependency management for scientific applications on clusters. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*, pages 41–49, April 2018.

[19] Tianyin Xu, Jiaqi Zhang, Peng Huang, Jing Zheng, Tianwei Sheng, Ding Yuan, Yuanyuan Zhou, and Shankar Pasupathy. Do not blame users for misconfigurations. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 244–259, New York, NY, USA, 2013. ACM.