Dartmouth College

# Dartmouth Digital Commons

6-1-1998

# An Implementation of External-Memory Depth-First Search

Christopher S. Leon
*Dartmouth College*

Follow this and additional works at: https://digitalcommons.dartmouth.edu/senior_theses

Part of the Computer Sciences Commons

## Recommended Citation

# An Implementation of External-Memory Depth-First Search

*Senior Honors Thesis*

by
Chris Leon
Dartmouth College
Hanover, New Hampshire
June, 1998

**Abstract**

In many different areas of computing, problems can arise which are too large to fit in main memory. For these problems, the I/O cost of moving data between main memory and secondary storage (for example, disks) becomes a significant bottleneck affecting the performance of the program.

Since most algorithms do not take into account the size of main memory, new algorithms have been developed to optimize the number of I/O operations performed. This paper details the implementation of one such algorithm, for external-memory depth-first search.

Depth-first search is a basic tool for solving many problems in graph theory, and since graph theory is applicable for many large computational problems, it is important to make sure that such a basic tool is designed to minimize the affect of the bottleneck of moving data between main memory and secondary storage.

The algorithm whose implementation is described in this paper is sketched out in an extended abstract by Chiang et al. We attempt to improve the given algorithm by minimizing the number of I/O operations performed. We also extend the algorithm by finding disjoint trees and by classifying all the edges in the problem.

# 1 Introduction

This paper details the process of implementing an algorithm for depth-first search in external memory. Beginning with a faithful adaptation of the algorithm, attempts were made to add additional functionality to the given algorithm while maintaining or improving the number of input/output operations between main memory and secondary (disk) storage (I/O's) performed during the running of the program.

Although the I/O complexity was not improved by anything more than a constant factor, it was possible to add the ability to process disjoint trees and to classify all the edges in the graph while maintaining the I/O complexity bounds of the algorithm.

Some terms will be helpful in analyzing the performance of the program:

- $V$ = The number of vertices in the graph.

- $E$ = The number of edges in the graph.

- $M$ = The number of items that can fit in main memory.

- $B$ = The number of items that can fit in a single disk block.

1

- $D$ = The number of disks in the system.

This paper traces the path taken from the first attempt at implementing the external-memory depth-first search algorithm, to adding disjoint tree searching and then edge classification. Each section details the changes that were made in each step to the structures and routines used by the program, as well as how main memory is used.

# 2 Background

When solving problems that are too large to fit in main memory, the I/O system between main memory and external memory becomes the bottleneck that can determine the performance of the application. Problems this large need to be dealt with by minimizing the number of I/O's performed, and new algorithms need to be developed that take this I/O bottleneck in to account. This paper deals particularly with depth-first search.

A paper by Chiang, Goodrich, Grove, Tamassia, Vengroff, and Vitter [CGG$^+$95] includes a sketch of an algorithm to perform depth-first search efficiently with respect to the number of I/O's performed. Their algorithm follows:

## 2.1 Algorithm

From Chiang et al., Section 7. To clarify part of this algorithm, $scan(E)$ is shorthand for $E/BD$.

> ...We analyze the performance of sequential DFS, modifying the algorithm to reprocess the graph when the number of visited vertices exceeds $\Theta(M)$. We present a graph with $V$ vertices and $E$ edges by three arrays. There is a size-$E$ array $A$ containing the edges, sorted by source. Size $V$ arrays $Start[i]$ and $Stop[i]$ denote the range of the adjacency list of $i$. Vertex $i$ points to vertices $\{A[j]|Start[i] \le j \le Stop[i]\}$.
>
> DFS maintains a stack of vertices corresponding to the path from the root to the current vertex in the DFS tree. The pop and push operations needed for a stack are easily implemented optimally in I/O's. For each current vertex, examine the incident edges in the order given on the adjacency list. When a vertex is first encountered, it is added to a search structure, put on the stack, and made the current vertex. Each edge read

is discarded. When an adjacency list is exhausted, pop the stack and retreat the path one vertex.

The only problem arises when the search structure holding visited vertices exceeds the memory available. When that happens, we make a pass through all of the edges, discarding all edges that point to vertices already visited, and compacting so that all the edges in each adjacency list are consecutive. Then we empty out the search structure and continue.

The algorithm must perform $O(1)$ I/Os every time a vertex is made the current vertex. This can only happen $2V$ times, since each such I/O is due to a pop or to a push. The total additional number of I/Os due to reading edge lists is $O(scan(E)+V)$. The search structure fills up memory at most $O(V/M)$ times. Each time the search structure is emptied, $O(scan(E))$ I/Os are performed.

**Theorem 7.1.** *Let $G$ be a directed graph containing $V$ vertices and $E$ edges in which the edges are given in a list that is sorted by source. DFS can be performed on $G$ with $O((1+V/M)scan(E)+V)$ I/Os.* [CGG$^+$95]

## 2.2   Tools

To implement this depth-first search algorithm, we used Dartmouth's ViC* programming interface, which simulates the Parallel Disk Model (PDM). ViC* lets the user implement and verify a PDM algorithm, such as the one above. More information about ViC* can be found at http://www.cs.dartmouth.edu/ thc/ViC/.

# 3   Implementing Depth-First Search

My implementation of the depth-first search algorithm, appropriately called dfs, went through several incarnations. I will present each of these separately and look at the decisions made at various points in the development of each.

## 3.1   dfs

The first version of dfs stuck fairly faithfully to the algorithm detailed above. There was however one departure, which is described below. First, we will examine how main memory was used. Then we will look at the structures and routines involved in the implementation and the changes that were made to the algorithm for this version of dfs.

### 3.1.1 Memory

ViC* works primarily with files. Files are read in to main memory stripe by stripe. To implement dfs, we divided memory up into two sections. Two stripes worth of memory were reserved to provide working space, while the rest of memory was dedicated to the search structure, which is described in the following section.

### 3.1.2 Structures

To generate the graph, a program named make-graph was used that generated a graph for a given number of vertices and edges. This program created three files, one for each array detailed in the algorithm: the adjacency list (A), the start index array (start), and the stop index array (stop). The dfs program could then be run, first making copies of those three files to work with, and then creating another file to represent the stack.

For the stack, dfs kept an offset variable, from which it could calculate what stripe and where on the stripe the current vertex is. A pop consisted of reading the stripe and returning the new current vertex, and a push read the stripe, wrote in the current vertex, and wrote the stripe back to disk.

The search structure took the form of a very basic linear, unsorted list, where vertices where simply appended on to the list when t hey were made the current vertex. When it filled up, dfs removed all edges going to any of the vertices in the search structure from A.

### 3.1.3 Routines

At this stage, dfs consisted mostly of three functions. The main function was the heart of the program, handling most of the depth- first search. It started at 0, and using calls to another function called GetNextVertex, performed the search. This main function also handled the search structure, checking to see if the vertex returned was already encountered, and if it wasn't, inserting the new vertex into the structure. Pushing and popping to and from the stack was handled here, as well as the output describing what stage the search was at. When the search structure filled up, EmptyStructure would be called.

GetNextVertex worked from the current vertex with the three arrays, A, start, and stop to find the next vertex that could be reached from the current vertex. The implementation of GetNextVertex contains the first main departure from the original algorithm. The algorithm describes a compaction step for removing edges that go to an already visited vertex from the adjacency list. Since the adjacency list of a vertex

could be arbitrarily large, conceivably larger than main memory, and probably larger than the two stripes that were reserved for workspace, a method would need to be devised to compact the list efficiently in external memory. Thus, we decided not to compact the adjacency lists during the emptying of the search structure. Instead, EmptyStructure will remove edges from the adjacency list by replacing the edge's entry in the adjacency list with a $-1$, which GetNextVertex understands to be a removed edge. The edges that have been replaced by $-1$ by EmptyStructure are in effect removed by GetNextVertex. GetNextVertex discards edges it encounters by incrementing the start index for the current vertex. Whenever a $-1$ is encountered, the start index is incremented more, thus removing that edge completely. When the start index becomes greater than the stop index for that vertex, the start index is changed to $-1$, so that we know every vertex adjacent to this vertex has been visited.

EmptyStructure handled the actual removal of already visited vertices, which were contained in the search structure, from the adjacency list. As mentioned above, "removing" an edge from the adjacency list actually is accomplished in EmptyStructure by overwriting the vertex number with a $-1$. No compaction is done by EmptyStructure. Since the search structure was actually just a linear unsorted list, removing edges consisted of reading a stripe from A, and for each vertex in the stripe, searching through the entire search structure to see if that vertex had already been visited. If so, a $-1$ was written into the stripe, and after the entire stripe was checked it was written back to disk.

### 3.1.4  Results

The first incarnation of dfs appeared to work fine, although it seemed that there was certainly room for optimizations. A way also had to be found to find disjoint trees, and dfs should be using logical stripes for it's I/O calls. But we did have enough to try and verify the I/O complexity given in the algorithm.

To do this, we must notice that despite the fact that dfs was run on a problem with $V$ vertices and $E$ edges, if there were disjoint trees dfs won't have visited all $V$ vertices, or processed all $E$ edges. In effect, what we are really dealing with is a smaller problem, one where $V$ is the number of vertices that the dfs tree produced by dfs contains, and where $E$ is the number of edges examined to create this tree.

Knowing this, we can use data from test runs to perform a regression to see how well the results fit the I/O complexity bounds given for the algorithm. For these test runs, we simulated a system where $B = 4$, $D = 1$, and $M = 1016$. Notice $M$ is not a power of 2. This is because some stripes are reserved for use, and the search structure uses the memory left over. The data gathered from the runs is shown in Table 1.

| Total I/O's | $E$ | $V$ | $E/BD$ | $VE/MBD$ | $V$ |
|---|---|---|---|---|---|
| 214 | 27 | 27 | 6.75 | 0.1794 | 27 |
| 14357 | 2681 | 954 | 670.25 | 629.34 | 954 |
| 45074 | 8321 | 2013 | 2080.25 | 4121.6 | 2013 |
| 167636 | 25349 | 4090 | 6337.25 | 25511.2 | 4090 |
| 641174 | 68086 | 8207 | 17021.5 | 137496 | 8207 |
| 2808373 | 212490 | 16415 | 53122.5 | 858273 | 16415 |

Table 1: Test run results for the original dfs implementation. The right three columns correspond to the variables in the expanded I/O complexity equation: $O(E/BD + VE/MBD + V)$

Performing a linear regression on this data showed a very good fit to the I/O complexity bound given. Specifically, $R^2 = 1.000$, and $prob(F) = 0.0001$, and the equation becomes:

$$numI/O's = 34.27E/BD + 1.52VE/MBD - 19.45V + 3065.52$$

### 3.1.5 Issues

Three major issues arose during and after the implementation. The first was described above, the realization that compaction was avoidable when emptying the search structure.

The second issue was somewhat more serious. After completing this version of dfs, we realized that the algorithm did not account for a graph in which not all the other vertices were reachable from the root. Theoretically, what depth-first search should do when not all other vertices are reachable is complete the search for the vertices it can reach, and then begin a seperate search from a vertex that has not been visited. This is repeated as many times as is necessary to visit all the vertices in the graph. A way to find disjoint trees was clearly necessary.

The third issue had to do with the ViC* environment. This first implementation attempted to read and write physical stripes instead of logical stripes. This also had to be remedied.

6

## 3.2 dfs with Disjoint Tree Searching and Hashing

After the first attempt at implementing external-memory depth-first search, it seemed clear that there was room for improvement in the original dfs program, as well as problems that should be fixed. A second, and currently final, version of dfs was made to fix problems found in the first and to try and optimize the algorithm further.

### 3.2.1 Memory

The way memory was divided up was changed to make way for various optimizations. Memory was divided into three areas: two stripes worth as workspace, two stripes worth for the stack to use as a cache, and some amount of the rest for the search structure. The reason why dfs only uses "some" of the rest will become clear in the next section. In this incarnation, dfs uses logical stripes instead of physical stripes.

### 3.2.2 Structures

No new structures were added, but the way inw which two of the structures work was changed. Instead of keeping the stack on disk all the time, which forces us to perform I/O's on every push and pop, we began to remember the top two stripes of the stack. If dfs pops past the last vertex from the bottom stripe, then read the next lower stripe into memory. If dfs pushes a vertex above the top of the upper stripe, then write the bottom stripe on to disk, and use the newly freed stripe in memory to write the new vertex in to. At worst then, we perform an I/O every number of vertices per stripe.

The search structure was also changed. Although the change in this structure does not save any I/O's, it does save a significant amount of computation time. We decided to implement a doubly hashed, open addressing hash table. One characteristic of this type of hash table is that if the hash table size is relatively prime to the result of the second hash function, then every entry will be searched during a hash operation. Since we couldn't restrict the result of the second hash function, we instead made the search structure the size of the closest prime lower than the amount of memory actually available. This means that the value for M will be somewhat less than simply the amount of main memory available minus the number of reserved stripes.

### 3.2.3 Routines

No new major routines were added to dfs. However, EmptyStructure was changed in that it now makes use of a hash table instead of a linear, unsorted list. Also, the

7

| Total I/O's | $E$ | $V$ | $E/BD$ | $VE/MBD$ | $V$ |
|---|---|---|---|---|---|
| 5702 | 942 | 513 | 235.5 | 126.77 | 513 |
| 14878 | 2826 | 1026 | 706.5 | 760.62 | 1026 |
| 25990 | 8478 | 2052 | 2119.5 | 4563.71 | 2052 |
| 55519 | 25434 | 4104 | 6538.5 | 27382.2 | 4104 |
| 158133 | 76302 | 8208 | 19075.5 | 164293.5 | 8206 |
| 554211 | 228906 | 16416 | 57226.5 | 985761 | 16416 |

Table 2: Test run results for dfs with hashing and disjoint tree searching. The right three columns give the values for the variables in the expanded I/O complexity equation $O(E/BD + VE/MBD + V)$

main function was changed so that dfs could find disjoint search trees.

GetNextVertex marks a vertex as having had all its edges examined by changing the start index of that vertex to $-1$. When a depth-first search finishes building a tree by returning to the root and having no other edges to follow from there, it will have examined every edge from every vertex in the tree. Therefore, to find a vertex that has not been visited, all we need upon finishing a depth-first search tree is to scan through the start array to find an entry which is not $-1$. This vertex will be picked as the new root, and dfs will build a new depth-first search tree using this root. We can do this as many times as needed to visit every vertex in the graph.

### 3.2.4   Results

The most major change in dfs with respect to I/O's was the addition of disjoint tree searching. This should not change the I/O complexity bound, since really what is happening is we are running dfs on a set of smaller problems. To verify this, tests were run using the new version of dfs. The resulting data is in Table 2.

Again, a regression was performed to see verify if the bound still held for dfs. The results of the regression showed that dfs was still very close to the bound, since $R^2 = 0.9999$, and $prob(F) = 0.0001$. The equation becomes:

$$numI/O's = 1.829E/BD + 0.3153VE/MBD + 8.285V + 2755.78$$

### 3.2.5   Issues

After this version of dfs was finished, there were still small optimizations that could be added to further improve its performance. Among these was the possibility of having

GetNextVertex cache the stripes it uses against the chance of using those stripes in the next call, and finding a way to improve the efficiency of how disjoint trees are found. However, around this time the possibility of extending the dfs program by adding the ability to classify all the edges was suggested, and our time became devoted to adding this capability.

# 4   Implementation of dfs with Edge Classification

Edges in a graph that depth-first search is being run on can be classified in four ways:

**Tree Edge:** Runs from a vertex to its immediate descendant in the depth-first tree.

**Forward Edge:** Runs from a vertex to a non-immediate descendant.

**Back Edge:** Runs from a vertex to an ancestor in the tree.

**Cross Edge:** Runs from a vertex to another vertex which is not an ancestor or a descendant.

To classify a particular edge, dfs needs to keep track of more information than was previously necessary. In particular, dfs must specify discover (when the vertex was first encountered) and finish (when the vertex had all its edges processed) times. Clearly, changes were necessary to dfs to gain the ability to classify the edges in the graph.

## 4.1   dfs with Hashing, Disjoint Tree Searching, and Edge Classification

The obvious way to handle adding edge classification to the existing dfs program was to simply create structures to handle the classification process, and make small changes to GetNextVertex and the main function to use these structures.

### 4.1.1   Memory

The divisions of memory did not to need to be changed much to handle edge classification. Instead of reserving two stripes for workspace, dfs started reserving three, for use in GetNextVertex particularly. The size of the search structure is decreased accordingly.

### 4.1.2 Structures

Edge classification necessitated the creating of three new arrays. The first keeps track of the the classification of each edge (edges), the second is used to store discover times (discover), and the third is for finish times (finish). The times that discover and finish deal with are relative. Time goes by only when a vertex is first discovered or when it is finished.

### 4.1.3 Routines

EmptyStructure is much the same, except that instead of deleting edges by overwriting the destination, it overwrites vertex $v$ with the value $-(v+1)$. Since we need to classify all the edges, we need to know where each edge goes even if we have already visited the destination of the edge.

GetNextVertex handles the classification of edges. It does this by finding out what vertex the edge points to, and then accessing the discover and finish times of that vertex. If the vertex is a negative number, we know it has already been visited but GetNextVertex must still classify the edge. We reverse the operation that EmptyStructure executes by calculating $-v-1$ to find where the edge goes. Based on these times and the discover time of the current vertex, we can classify the edge. Let $u$ be the current vertex, and $v$ be the new one. If $v$ is undiscovered, the edge is a tree edge. If $v$ was discovered after $u$, then the edge is a forward edge. If $v$ was discovered before u, but has not been finished, then the edge is a back edge. If $v$ has been discovered and finished, then the edge is a cross edge.

GetNextVertex cycles through the adjacency list of the current vertex until it finds a tree edge and returns that vertex, or until it exhausts the adjacency list.

The main function is mostly unchanged, except that as the search pops vertices from the stack, the main function writes finish times for vertices that have had all their edges processed.

### 4.1.4 Issues

As this version of dfs approached completion, it became increasingly clear that having both an edge classification routine and a search structure to keep track of visited vertices was redundant. When GetNextVertex checks the discover time of the possible next vertex, it essentially finds out whether or not the vertex has already been visited. This obviates the need for the search structure, as well as EmptyStructure. No test runs were run for this version of dfs, since by the time it was ready to be tested, it was clear it was not going to be the final product.

## 4.2 dfs without Hashing

Changing the first attempt at dfs with classification to remove the redundancy described above was not difficult. All that needed to be done was to remove the search structure and rely on GetNextVertex to either return a valid vertex or to have processed all the vertices from the current vertex.

### 4.2.1 Memory

Now that there is no search structure, all that we need from main memory is three stripes for GetNextVertex to work with, and two stripes for the stack.

### 4.2.2 Structures

Except for the lack of the search structure, the structures didn't change.

### 4.2.3 Routines

EmptyStructure and its associated hashing functions can be removed.

As mentioned above, the main function now can rely on GetNextVertex to return either a valid, unvisited vertex, or to process all remaining edges leading from the current vertex. Therefore this function has no need to search for a new vertex in the search structure to see if it has already been visited.

GetNextVertex itself does not need to be changed much. There will no longer be negative numbers in the adjacency list, since the only time an edge is discarded is when GetNextVertex classifies it, and increments start to point to the next edge.

### 4.2.4 Results

Examining the theorem in which the I/O complexity for the external-memory depth-first search is detailed, we see that the $V/M$ term is no longer necessary. This term dealt with how often the algorithm needs to scan through the adjacency list in the process of the search structure. Since there is no longer a search structure, this term can be removed, resulting in a complexity bound of

$$O(E/BD + V)$$

Once again, dfs is put thru test runs, and the data, which can be seen in Table 3, is put into a regression to see how true to the complexity bound it is.

| Total I/O's | E | V | E/BD | V |
|---|---|---|---|---|
| 9978 | 942 | 513 | 235.5 | 513 |
| 24222 | 2826 | 1026 | 706.5 | 1026 |
| 54712 | 8478 | 2052 | 2119.5 | 2052 |
| 127691 | 25434 | 4104 | 6538.5 | 4104 |
| 311495 | 76302 | 8208 | 19075.5 | 8206 |
| 791398 | 228906 | 16416 | 57226.5 | 16416 |

Table 3: Test run results for dfs with edge classification and disjoint tree searching. The right two columns give the values for the variables in the I/O complexity equation $O(E/BD + V)$

The regression results clearly show that this implementation of dfs is very true to the theoretical bound. The results are $R^2 = 1.0000$, and $prob(F) = 0.0001$, and the equation with coefficeients is:

$$numI/O's = 8.776E/BD + 17.64V - 495.393$$

### 4.2.5   Issues

The same issues from the final version of dfs without classification are still present in the final version of dfs with classification. Since dfs now only uses five stripes of main memory, a caching scheme for GetNextVertex could be relatively easy to implement. Also, since dfs has so much memory left over from the lack of a search structure, small optimizations could be added to GetNextVertex to reduce the number of I/O's per call.

# 5   Final Results

After this rather lengthy process of attempting to improve the dfs algorithm, we essentially ended up with two programs. One of these programs provides edge classification, and the other does not. dfs without classification uses significantly fewer I/O's than the newer version, but edge classification can be important in some applications. It is also possible that the number of I/O's performed in the newer version of dfs could be reduced by coming up with a caching scheme to use the now empty space in memory.

At any rate, we have added useful new functionality to the original algorithm without raising the basic I/O complexity bound, as shown by the regressions performed on the test results.

# 6    Conclusion

The process detailed in this paper of implementing and attempting to improve the given DFS algorithm was clearly at least partially successful. Although we were unable to actually improve the I/O complexity of the algorithm by more than a constant factor, we were able to make the algorithm more complete, by adding disjoint tree search, and more useful, by adding edge classification.

Further improvements could be made to both of the resulting programs, although we don't foresee any optimizations that could improve the number of I/O's performed by more than a constant factor. However, considering how expensive I/O's are, every little bit may help.

# References

[CGG$^+$95] Yi-Jen Chiang, Michael T. Goodrich, Edward F. Grove, Roberto Tamassia, Darren Erik Vengroff, and Jeffrey Scott Vitter. External-memory graph algorithms. In *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 139–149, January 1995.