High-Performance Persistent Graphs

John Moody
Colorado College '16
john.moody@coloradocollege.edu

Benjamin Ylvisaker
Assistant Professor, Colorado College
ben.ylvisaker@coloradocollege.edu

Abstract

In the world of persistent data structures, there exist few high performance graph libraries. We propose in this paper a C library which stores an application-controlled persistent graph in a structure derived from a hash array mapped trie, using chunking and lazy copying to conserve memory and increase performance. We achieve -some stuff about time complexity that I haven't figured out yet-.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

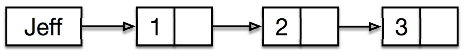
Keywords persistent data structures, graphs, hash array mapped trie

1. Introduction

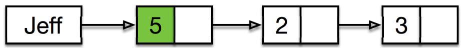
A graph is defined as a structure containing some number of nodes and some number of edges, which connect nodes together. The graph is a data structure with wide-ranging applications, from computational graphs to databases, networking and pathfinding. To get information about nodes or edges in a graph within a program we typically use an array or some manner of key-value store. The value associated with a node is usually a list of adjacent nodes, which are either predecessors to that node or antecedents. We will explore what it means for a graph to be stored in a persistent way, and then propose a structure with strong performance characteristics for various operations and optimized use of memory.

1.1 Persistence

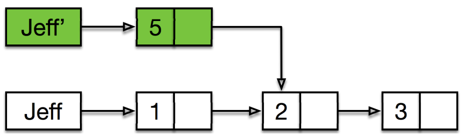
What does it mean for data to be stored in a persistent way? A piece of data is persistent if it does not change. Consider a linked list in memory, Jeff:



There are a number of ways to make an edit to this structure. If we wanted to change the frontal value of Jeff from 1 to 5, and we do not need the original any longer, we may simply change it:



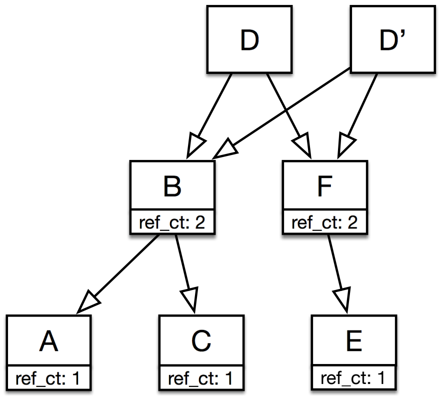
If we want this notion of persistence to apply to Jeff, however, Jeff cannot change. Instead, we need to come up with a way to change the front value of Jeff to 5 while keeping the original version of Jeff with 1 at the front intact. Enter "Jeff'":



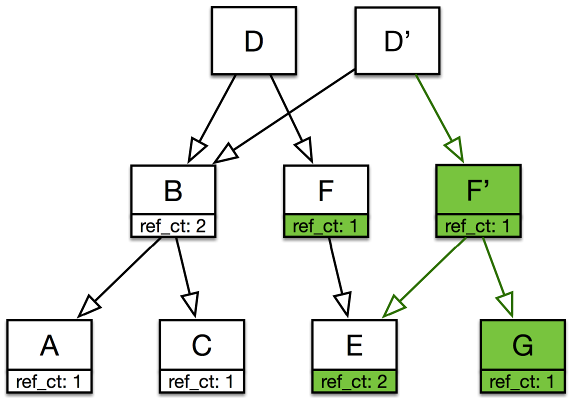
Jeff, we notice, has not changed. Jeff' preserves the parts of Jeff's structure that they have in common. Persistent data structures, then, refer to structures like Jeff and Jeff', which, after being created, will always remain the same.

1.1.1 Trees and Reference Counting

Let us now consider an example using a simple binary search tree, where we have D, and a separate copy D':



For us to be able to make edits to D' without changing D, we must introduce the concept of reference counting. A reference count keeps track of how many objects point to a given node. Here, since B and F have reference counts greater than one, we know that we can't modify those nodes without changing another version of the data structure. Therefore, when we insert G into D', we will copy any nodes that have reference counts greater than one and adjust the tree as necessary:



2. Tries

How do we represent a graph persistently?

A. Appendix Title

This is the text of the appendix, if you need one.

Acknowledgments

Acknowledgments, if needed.

References

P. Q. Smith, and X. Y. Jones. ...reference text...