Optimal Fully Persistent Pointer Machine Data Structures

Fadel Kassab
American University of Beirut

May 13, 2024

Abstract

This paper shows that optimal fully persistent pointer machine data structures exist after offering a formal setting to the open problem of finding an algorithm to transform ephemeral data structures to optimal fully persistent ones. It also shows a nice way to view the environment of the problem along with a literature review and applications.

1 Introduction

Since it is still an open problem if there is an algorithm that transforms your regular pointer machine data structure to an optimal fully persistent one this paper tries to tackle this problem. The problem was first introduced by the paper by James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan [1]. In their paper they offer in a sense the first view of persistence as we see it today - even though there was some papers on that but not in the same way. This paper is for a research in an advanced data structures course offered by Prof. Amer Mouawad.

As well it is worth mentioning that it is my first research paper on a theoretical computer science topic.

2 Preliminary

In order to create a formal environment for the problem, we will start by defining our terminology. It is inspired from the one in [1].

2.1 Definitions

In order to offer a formal setting for the upcoming problems, consider the following definitions:

2.1.1 Pointer Machine Data Structure

A Pointer Machine Data Structure is a 4-tuple (N, E, Q, U) where:

- N: finite set of nodes.
- E: fixed set of entry pointers.
- Q: finite set of query operations.
- U: finite set of update operations.

2.1.2 Node

A node is a 2-tuple (Label, Fields) where:

- Label: A string representing the type of the node.
- Fields: A finite tuple containing a number of fields.

2.1.3 Field

A field can be information, a pointer, or null.

2.1.4 Ephemeral Data Structure

A pointer machine data structure is said to be ephemeral if it is not persistent.

2.1.5 Persistent Data Structure

A pointer machine data structure is said to be persistent if it is partially, fully, confluently, or functionally persistent.

2.1.6 Fully Persistent Data Structure

A pointer machine data structure is said to be fully persistent if it supports both queries and updates in any version of the data structure.

2.1.7 Data Structure Version

The i-th version of a data structure is the version produced by the i-th update operation in a sequence of update operations.

This definition works best with partial persistence due to linear ordering. We will see what to do for full persistence with this in mind.

2.1.8 Optimal Fully Persistent Data Structure

A pointer machine structure is said to be optimal if all operations of the ephemeral version of that data structure have corresponding operations with O(1) overhead worst-case both space & time in the fully persistent version.

2.1.9 The Relation f

F is a relation containing 2-tuples (e, p) the ephemeral data structure and its corresponding optimal fully persistent version. We say p = f(e) in case such a p exists.

2.1.10 The Set E

$$E = \{e \mid e \text{ is an ephemeral pointer machine data structure}\}$$

2.1.11 The Set P_f

$$P_f = \{p \mid p = f(e) \text{ for some } e \in E\}$$

3 Main Problems

The paper [1] asks if there is a way to transform an ephemeral pointer machine data structure to an optimal fully persistent data structure which is the main problem that we are approaching. Therefore, in order to gain a better understanding of what is going on, I propose to consider the following three problems with the third one being the one stated in [1].

3.1 The Existence Problem

We define the existence problem to inquire about the most fundamental question: "Is there such a thing as an optimal fully persistent data structure?"

More formally the problem can be stated as follows:

What is the truth value of:

$$\exists e \in E \, \exists p \in P \, (p = f(e))$$

We will offer a solution to this problem in the following section, which means that there is such a thing as an optimal fully persistent pointer machine data structure (i.e P_f is non-empty).

3.2 The Universal Existence Problem

Now we can take a step further since we know that P_f is non-empty and ask the ourselves the following: "Is it the case that every ephemeral data structure will have a corresponding optimal fully persistent one?"

More formally the problem can be stated as follows:

What is the truth value of:

$$\forall e \in E \, \exists p \in P \, (p = f(e))$$

The power of this problem is that if one proves that there is some ephemeral data e structure with no relating f(e) then it will imply that the procedural problem is solved, since if there is no correspondence in the first place then there won't be an algorithm for every pointer machine data structure.

3.3 The Procedural Problem

And now if we take another step further we may ask the question that is stated in [1], which is: "Is there a way to make an ephemeral data structure optimal fully persistent?".

More formally the problem is:

$$\exists a \in A \, \forall e \in E \, (a(e) = f(e))$$

with A being the set of algorithms (Deciders if you will) and a(e) = f(e) that the algorithm will return f(e) on e.

4 Solving The Existence Problem

4.1 Theorem: They Exist

It is the case that:

$$\exists e \in E \, \exists p \in P \, (p = f(e))$$

4.2 Proof:

In the following we will offer a solution to the problem that we defined in the previous section which is the existence problem by constructing a pointer machine data structure that by definition belongs to P_f.

Consider the "CounterBox" ephemeral data structure which we define as follows:

- N will have only one node which will have one field "count".
- E will have only one entry pointer.
- Q will have only one operation: "matches".
- U will have only one operation: "increaseCount".

Algorithm 1 increaseCount

```
1: procedure INCREASECOUNT
2:     Enter the node
3:     count := count + 1
4: end procedure
```

Algorithm 2 matches

```
1: procedure MATCHES(SOMEVALUE)
2:     Enter the node
3:     if someValue == node.counter then
4:         return True
5:     end if
6:     return False
7: end procedure
```

Those are both constant time operations.

Now consider the following construction of the data structure which we will see if it happens to be an optimal fully persistent version of our CounterBox, call it "PersistentCounterBox":

- N will contain contain two types of nodes: "size" and "log". It will have only one size node which will contain one field size, and it will initially contain only one log node which will have one field count.
- E will have two entry pointers: sizePointer, and countPointer. sizeField will let us access the size field in the size node. and countPointer will give us access to the "linked list" of counts.
- Q contains one operation: persistentMatches.
- U contains one operation: persistentIncreaseCount.

Algorithm 3 persistentIncreaseCount

```
1: procedure PERSISTENTINCREASECOUNT(VERSIONINDEX)
2:     if versionIndex == sizeNode.size + 1 then
3:         Add a new countNode with count := versionIndex + 1
4:         sizeNode.size := sizeNode.size + 1
5:     end if
6: end procedure
```

Algorithm 4 persistentMatches

```
1: procedure PERSISTENTMATCHES(VERSIONINDEX, SOMEVALUE)
2:     if versionIndex > sizeNode.size then
3:         return NON-EXISTING-VERSION
4:     end if
5:     if versionIndex == someValue then
6:         return True
7:     end if
8:     return False
9: end procedure
```

The runtime of these two operations is indeed within a O(1) overhead of that of each corresponding algorithm in the ephemeral CounterBox.

$$\therefore \text{PersistentCounterBox} = f(\text{CounterBox})$$

and so

$$\therefore \exists e \in E \, \exists p \in P \, (p = f(e)) \square$$

The trick was here is that we can have a linear ordering of versions, so that all branches in the "time tree" of the data structure can be omitted and the tree can be just a "linked list" which is the longest path in the "time tree" in order to index versions.

And as an exercise try to find a simpler optimal fully persistent data structure for our CounterBox to prove that these are not unique.

5 An Approach To The Problem

Now that we know that they exist from the proof above, we may also take a step further and try to tackle the universal existence problem that we defined above which remains an open problem.

This problem is important because of the following fact:

NOT UNIVERSAL EXISTENCE $\implies$ NOT PROCEDURAL

More formally:

$$\neg(\forall\ e \in E \quad \exists\ p \in P \quad (p = f(e))) \implies \neg(\exists\ a \in A \quad \forall\ e \in E \quad (a(e) = f(e)))$$

Therefore it is sufficient to find one ephemeral data structure that does not have a related optimal fully persistent data structure to prove that there is no such algorithm.

6 Literature Review

So far we only know how to do O(1) amortized space time for transforming an ephemeral data structure to a fully persistent one

6.1 Approaches to persistence

6.1.1 Brute Force

The most primitive yet general method of achieving persistence is brute force by literally storing every version of the ephemeral data structure which takes $\Omega(n)$ space and time. This was first introduced in [2].

6.1.2 Storing Update Sequences

This approach stores the entire update sequence on an ephemeral data structure and reconstructs accordingly. It is as the previous solution a general one. This approach takes O(m) space on a sequence of m operations if each update operation takes O(1) space. Also queries in Ω(i) time to access the i-th version even if an update takes O(1) time. This approach was mentioned in [1].

6.1.3 A 6.1.1 & 6.1.2 Hybrid

A hybrid of the two previous methods was mentioned in [1]. It invloves not only storing the entire update sequence, but also every k-th version, for some chosen k, Therefore accessing the i-th version now requires rebuilding it from version k*floor(i/k) by performing the appropriate sequence of updates. This hybrid approach will have a space time trade-off depending ultimately on k and the running times of the ephemeral operations. This also offers a general solution.

6.1.4 Dynamization Techniques of Bentley and Saxe

This method was introduced in [3]. It offers not a general solution but rather one to so-called "decomposable search problems". It assumes that your ephemeral data structure has only insertion in its update operations. Making a persistent data structure will be within a logarithmic factor.

6.1.5 Fat Node Approach

This approach was introduced in [1]. With it you may achieve O(1) space per update operation, and O(logm) time.

6.1.6 Node Copying

Also presented in [1], this approach follows the one above. It solves the problem with the fat node approach by not allowing nodes to become arbitrarily fat but rather we allow a fixed number of pointers per node. This method achieves O(1) amortized space time.

6.1.7 Brodal's Results

Building on the work of [1] Brodal solved the problem of finding a way to make an ephemeral data structure in a sense optimally partially persistent [4]. By viewing the problem as a two-player pebble game on dynamic graphs Brodal achieves O(1) worst case space time for the problem but for partial persistence.

7 Applications

- **Version Control Systems:** Systems like Git use persistent data structures to manage different versions of code, allowing for efficient storage and retrieval of any version of a project without duplicating entire repositories.
- **Functional Programming Languages:** Languages like Haskell and Clojure use persistent data structures due to their immutability, which matches the functional programming paradigm that emphasizes pure functions and immutable data.
- **Undo Features in Applications:** Software applications like text editors or graphic design tools use persistent data structures to handle undo operations efficiently, enabling users to revert to any previous state easily.
- **Database Systems:** Databases may employ persistent data structures to handle transactions and rollback capabilities, enabling quick reversion to previous states in case of errors or system failures.
- **Concurrent Data Access:** In multithreaded applications, persistent data structures provide a way to manage concurrent access to data, minimizing the risk of race conditions and making them ideal for concurrent programming.
- **Blockchain and Cryptocurrencies:** Blockchain technology uses principles similar to persistent data structures, where each block contains a state of the ledger that builds upon the previous one, ensuring data integrity and traceability.

- **Snapshot Systems:** Systems that require snapshots, such as virtual machine environments or backup systems, benefit from persistent data structures to efficiently capture and store state at different points in time.
- **Game Development:** Games can use persistent data structures to manage game states, facilitating easy implementation of features like save and load game functionality, or to rewind gameplay.

8 References

[1] James R. Driscoll, Neil Sarnak, Daniel Dominic Sleator, Robert Endre Tarjan: Making Data Structures Persistent. J. Comput. Syst. Sci. 38(1): 86-124 (1989)

[2] M.H. Overmars, "Searching in the past II: general transforms," Technical Report RUU-CS-81-9, Department of Computer Science, University of Utrecht, Utrecht, The Netherlands, 1981.

[3] J.L. Bentley and J. B. Saxe, "Decompositional searching problems I: static-to-dynamic transformations," J. Algorithms 1 (1980), 301-358.

[4] Gerth Stlting Brodal: Partially Persistent Data Structures of Bounded Degree with Constant Update Time. Nord. J. Comput. 3(3): 238-255 (1996)