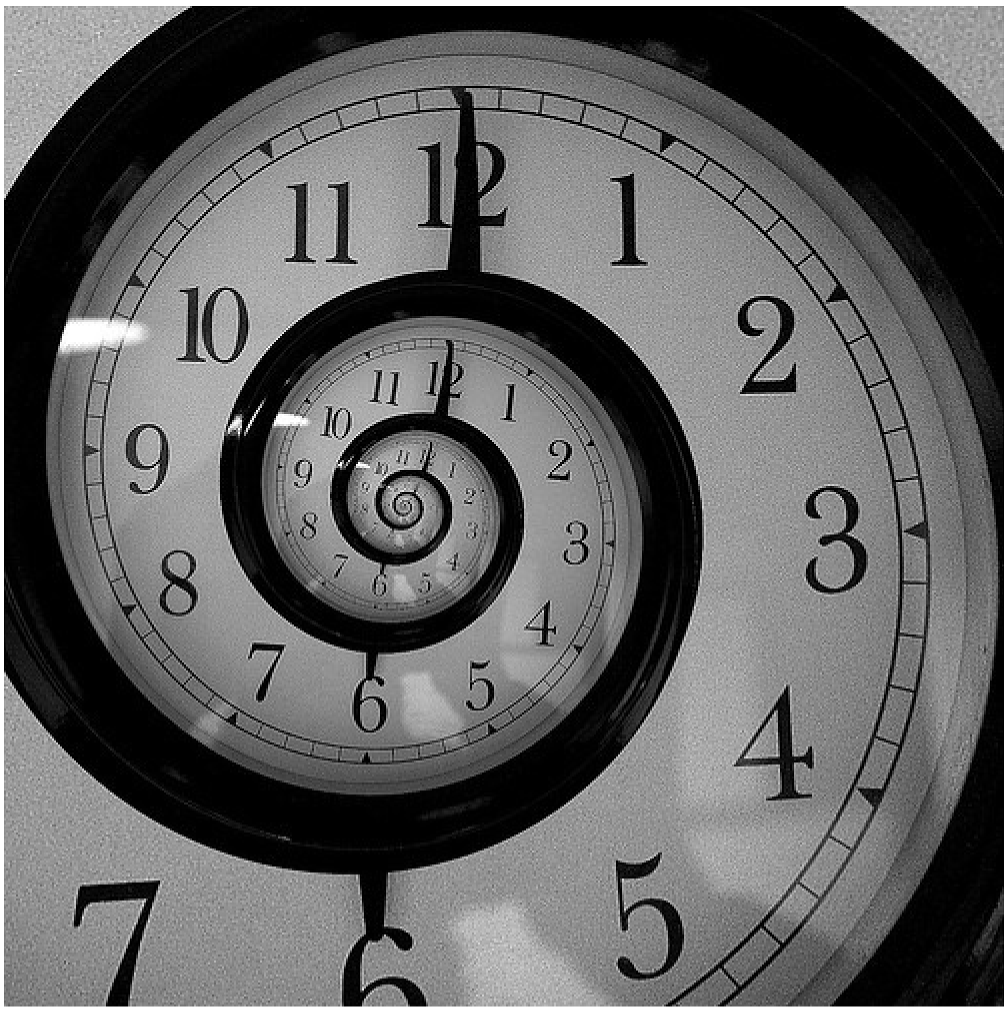
Fun with Time Travel:

Implementing Retroactive Data Structures

Chelsea Voss

6.851 Spring 2014



Introduction

Unlike normal data structures, which only allow operations to be carried out in the present, **retroactive data structures** allow operations to be inserted or deleted at any point in the past. In a **partially retroactive** data structure, queries may not be made into the past state of the data structure, but in a **fully retroactive** data structure, queries may be made at any point along the timeline and history of operations. Retroactive data structures were explored in a 2007 paper, "Retroactive Data Structures," by Demaine, Iacono, and Langerman.

The goal of this project is to turn known algorithms for various types of retroactive data structures (with runtimes summarized below) into *implementations*, developing a Python library that can be seamlessly imported into Python code and used to allow retroactive data structures to be used in code.

I surveyed "Retroactive Data Structures," looking for data structures that would be good candidates for implementation. Although I was not able to implement all of the data structures that the paper mentions, this was a productive project and I successfully implemented some of them: specifically, a partially-retroactive queue, which performs its operations in $O(1)$ time; a partially-retroactive data structure for searchable dynamic partial sums; and, most interestingly, two *general transformations*: one that converts any data structure into a partially-retroactive data structure, and one that converts any partially-retroactive data structure into a fully-retroactive data structure.

Summary of Retroactive Datastructures

I attempted to implement many of the data structures mentioned in Demaine 2007. Some, I was unable to complete on time, because of the prerequisite of implementing a more basic ***building block.*** Building blocks included:

- *Doubly-linked list*: required for **Partially-Retroactive Queue** and for **Partially-Retroactive Priority Queue**. Implemented.
- *Binary search tree*: required for **Partially-Retroactive Priority Queue**. Implemented.
- *Link-cut trees*: required for **Fully-Retroactive Union-Find**. Not implemented. (Sleator and Tarjan, 1983)
- *Modified (a,b)-tree*: required for **Fully-Retroactive Deque** and for **Partially-Retroactive Priority Queue**. Not implemented. (Fleischer, 1996)

However, I implemented a good number of the datastructures described in "Retroactive Data Structures" – and the general transformations, especially, are enough to be useful to applications where retroactivity might be useful.

Below, I summarize the runtimes of the various retroactive datastructures to be implemented, along with their interrelationships and details about how the implementations were carried out.

For runtimes, *r* is a parameter describing how old retroactive operations are allowed to be, and *m* is the total number of retroactive updates that are performed on a data structure.

General Transformations

- Partial retroactivity, *O*(r) overhead: **Implemented!**

  This implementation uses the rollback method to implement retroactivity. It stores up to ***r*** *prior operations as well as the state of the data structure before those operations, so that these operations can be reversed. When an operation is removed or inserted, the current state of the data structure is "refreshed" from the past state by applying each operation in sequence.*

  Implementing this proved to be an entertaining exercise in abstraction: it needs to be able to wrap ***any*** *data structure, and allow* ***any*** *form of operation on that data structure. So, operations are represented – and passed as input – using Python functions: an operation is any function which takes in a data structure and returns a new data structure.*

- Full retroactivity, *O*(√m) overhead from partial: **Not implemented.** *Requires an implementation of persistence (Driscoll et al, 1989; Fiat and Kaplan, 2001).*

- Worse full retroactivity, *O*(m) overhead from partial: **Implemented!**

  I devised a simpler solution to create a general transformation for full retroactivity in Python. This implementation stores a list of partially-retroactive data structures, applying or deleting operations from those partially-retroactive data structures when relevant. When the fully-retroactive data structure is queried, we simply query the relevant partially-retroactive data structure. Although it does not achieve the known optimal overhead, this implementation is functional, and allows the same degree of abstraction: any data structure can be made fully retroactive.

Searchable, Dynamic Partial Sums (SDPS)

This data structure has commutative, invertible operations – allowing partial retroactivity to be achieved with no overhead.

- Partially retroactive solution, *O*(1): **Implemented!**

  Since operations on the "Searchable, Dynamic Partial Sums" data structure are commutative and invertible, we are able to convert it to a partially-retroactive data structure quite easily: to ***`insert`*** *an operation, perform it; to* ***`delete`*** *an operation, perform its inverse. The only effort that was required to implement this in Python was some special wrapping, specific to this data structure, which gives operations metadata that allows the inverse of any operation to be determined. If operations were passed in as simple functions, they would be impossible to invert; to fix this, the method* ***`PartiallyRetroactiveSDPS.update(i,c)`*** *was modified so that it returns a function to use as an operation to pass to* ***`insert`*** *and* ***`delete`****; this function remembers* ***`c`*** *as an attribute, enabling* ***`delete`*** *to create the inverse operation,* ***`update(i,-c)`****. It would be difficult to create a wrapper that works for all data structures with commutative, invertible operations, because of this same hurdle: computing the inverse of an arbitrary programming function is impossible. When the functions are defined beforehand, however, as with this implementation of partially-retroactive searchable dynamic sums, the task is more tractable.*

- Fully retroactive solution, *O*(√m): Implied by partially retroactive solution (above), using the general transformation from partial to full retroactivity.

Queue

- Partially retroactive solution, *O*(1): **Implemented!**

  Creating a partially retroactive queue involves storing a doubly-linked list of items, inserting whenever an enqueue is made, but not deleting any old data when a dequeue is made – in case that dequeue operation is deleted later, for example. Instead, two pointers, ***F*** *and* ***B****, point to the items at the front and back of the queue when t=0; these two pointers are moved according to the proper logic whenever an enqueue or dequeue is retroactively inserted or removed. Querying is as simple as looking up those two pointers.*

  Unlike the other retroactive data structures, this data structure has a few quirks. First, the time at which a retroactive operation is inserted or deleted cannot be specified by an integer number: it must instead by specified by a pointer to a reference operation which occurs immediately after the time when the operation is to be retroactively inserted. Inserting an operation must therefore return a pointer to that operation, in case the operation is used later as a time reference. The pointer is used to instantly navigate to the *relevant part of the doubly-linked list in O(1) time. One possibility is that the data structure could be converted to a form which uses integer times, with a binary tree on top of the doubly-linked list; to insert or remove an operation would then be O(log m), instead.*

*Another quirk is that it does not have a single **`query()`** method: instead, there are two possible queries that can be made – **`front()`** and **`back()`**.*

- Fully retroactive solution, *O*(log m): See **Deque**.
- Better fully retroactive solution, *O*(log m), but with *O*(1) for present-time operations: *Requires an implementation of order-statistic trees (Cormen et al, 2001).*

Stack

- Partially retroactive solution, *O*(log m): See **Deque**.
- Fully retroactive solution, *O*(log m): See **Deque**.

Deque

- Partially retroactive solution, *O*(log m): Implied by fully retroactive solution (below).
- Fully retroactive solution, *O*(log m): **Not implemented.** *Requires an implementation of modified (a,b)-trees.*

Union-Find

- Partially retroactive solution, *O*(log m): Implied by fully retroactive solution (below).
- Fully retroactive solution, *O*(log m): **Not implemented.** *Requires an implementation of link-cut trees.*

Priority Queue

- Partially retroactive solution, *O*(log m): **Not implemented.** *Requires an implementation of modified (a,b)-trees.*
- Fully retroactive solution, *O*(√m log m): Implied by partially retroactive solution (above), using the general transformation from partial to full retroactivity.

Source Code

The source code for this project can be located on GitHub:

<https://github.com/csvoss/retroactive>.

It is split among five Python files, each of which has a special functionality.

`partial_retroactivity.py` – Implementations for partially-retroactive data structures.

`GeneralPartiallyRetroactive`: Adds partial retroactivity to any data structure.

`PartiallyRetroactiveSDPS`: Creates a partially-retroactive version of a searchable dynamic partial sums data structure.

`PartiallyRetroactiveQueue`: Creates a partially-retroactive version of a simple front-to-back queue.

`PartiallyRetroactivePriorityQueue` (incomplete): Creates a partially-retroactive version of a priority queue.

`full_retroactivity.py` – Implementations for fully-retroactive data structures.

`GeneralFullyRetroactive`: Adds full retroactivity to any data structure.

`FullyRetroactivePriorityQueue`: Using the general transformation and the partially-retroactive priority queue, creates a fully-retroactive priority queue.

`RetroactiveDeque` (incomplete): Creates a fully-retroactive version of a deque – enabling retroactive stacks and queues, as well.

`RetroactiveUnionFind` (incomplete): Creates a fully-retroactive version of the Union-Find data structure.

`basic.py` – Basic building blocks that were used in the implementations of specific retroactive data structures.

`DLLNodeForPRQ, DLLNodeForPRPQ`: Doubly-linked list nodes, to use in implementing the partially-retroactive queue and the partially-retroactive priority queue.

`BSTNode`: Binary search tree.

`examples.py` – Tests and example usage for each retroactive data structure.

`utils.py` – Miscellaneous utility functions.

Example Usage

These are a few examples of how the retroactive data structures can be used in a regular Python context.

The general transformations, in particular, were designed with usability in mind: these transformations should be able to take any sort of data structure and any sort of operation on that data structure, and produce a partially or fully retroactive version efficiently and without error.

General Partial Retroactivity

To use one of these general transformations, simply initialize a Python class:

```
>>> x = GeneralPartiallyRetroactive([1,2,3,4,5])
```

This creates a partially-retroactive list, initialized to [1,2,3,4,5]. We can add or remove operations in the present:

```
>>> def appendOne(lst):
    return lst + [1]
>>> x.insertAgo(appendOne, tminus=0)
>>> x.insertAgo(appendOne, tminus=0)
>>> x.insertAgo(appendOne, tminus=0)
>>> x.query()
[1, 2, 3, 4, 5, 1, 1, 1]   ##Three appendOnes!
```

...and we can add or remove operations from the past:

```
>>> def appendSix(lst):
    return lst + [6]
>>> x.insertAgo(appendSix, tminus=2)   ##Insert *two* operations ago
>>> x.query()
[1, 2, 3, 4, 5, 1, 6, 1, 1]
>>> x.deleteAgo(tminus=3)   ##Delete the first appendOne
>>> x.query()
[1, 2, 3, 4, 5, 6, 1, 1]
```

General Full Retroactivity

Creating a fully-retroactive data structure is similar, but allows querying into the past instead of just the present. Let us create a fully-retroactive list:

```
>>> y = GeneralFullyRetroactive([1,2,3])
>>> y.insertAgo(appendOne, tminus=0)
>>> y.insertAgo(appendSix, tminus=0)   ##This one should come last
>>> y.insertAgo(appendTen, tminus=2)   ##This one should come first
>>> y.query()
[1, 2, 3, 10, 1, 6]    ##The current state of the data structure
>>> y.query(1)
[1, 2, 3, 10, 1]
>>> y.query(2)
[1, 2, 3, 10]
>>> y.query(3)
[1, 2, 3]    ##The state of the data structure way back in the past
```

Looking back in time at the state of the data structure, we can see that the retroactive operations are taking place in the right order.

Partially Retroactive Queue

In contrast, the partially retroactive queue is used somewhat differently from either general transformation, because of its quirk of specifying times using pointers.

```
>>> z = PartiallyRetroactiveQueue()
>>> q = z.insertEnqueue(42)   ## INSERT enqueueings of some things
>>> r = z.insertEnqueue(43)
>>> s = z.insertEnqueue(44)
>>> print z    ## Manually query the entire state.
               ## This is NOT the standard O(1) query of front/back.
Front=42, Back=44, State=[44, 43, 42]
>>> t = z.insertEnqueue(1, r)    ## INSERT an enqueue of 1
                                 ## BEFORE the enqueueing of 43
>>> print z
Front=42, Back=44, State=[44, 43, 1, 42]
>>> u = z.insertDequeue(s)       ## INSERT a dequeue
                                 ## BEFORE the enqueueing of 44
>>> print z
Front=1, Back=44, State=[44, 43, 1]
>>> v = z.delete(u)              ## DELETE that dequeue
>>> print z
Front=42, Back=44, State=[44, 43, 1, 42]
>>> w = z.delete(q)              ## DELETE the enqueue of 42
>>> print z
Front=1, Back=44, State=[44, 43, 1]
```

On Abstraction

Initially, I wanted to be able to unify all retroactive data structures under a single abstraction (e.g. **`PartiallyRetroactive({'pi':1, 'foo':4, 'bar':55})`** or **`FullyRetroactive(PriorityQueue())`**), so that the interface for using these data structures would be simpler. However, this endeavor proved to be something which would add unnecessary layers of complication, instead of simplicity as I would have liked. There are several reasons for this:

- Most data structures only allow one exactly one sort of query, accessed via **`query()`**, but some must allow more (for example, the Partially Retroactive Queue allows querying the elements at the front *and* back of the queue)
- Most data structures require a number, *tminus*, specifying how long ago to insert/delete an operation, but some data structures (again, Partially Retroactive Queue) require specifying this information with a pointer, instead.
- Most data structures require operations to be functions which take a datastructure as input and return a datastructure as output, but some (for example, Partially Retroactive Searchable, Dynamic Partial Sums) require a more complicated paradigm.

For these reasons, unifying the implementations under a single abstraction would have required adding extra options and layers, which would just make the implementations more complicated and therefore difficult for a user to use and understand. Instead, I choose to just present the implementations as classes, as-is. It is suggested that a user who wishes to use the implementation of a particular retroactive data structure should read that data structure's documentation, learning how to use each feature it provides and how to use it correctly, then go from there.

Future Developments

1. Not all data structures are implemented yet! "Building blocks" were an unexpected barrier to completing many of these implementations in a timely fashion. It would be cool to see these finished eventually.
2. There likely remain hidden errors within this implementation – since the code is open-source and located on Github, any errors which are detected by others can be communicated and possibly corrected.

Conclusion

Through examining “Retroactive Data Structures,” Demaine 2007, I implemented a number of retroactive data structures using Python, from special cases that get improved efficiency for particular data structures to general transformations that work for any series of operations on any data structures and can easily be utilized within a Python application.

References

Demaine, Erik D, John Iacono, and Stefan Langerman, “Retroactive Data Structures”, ACM Transactions on Algorithms, vol. 3, no. 2 (May 2007). [link]

Demaine, Erik D. 6.851 lecture notes, [link]

Driscoll, James R, Neil Sarnak, Daniel D Sleator, and Robert E Tarjan. Journal of Computer and System Sciences, 38(1): 86-124 (Feb 1989). [link]

Fiat, Amos, and Haim Kaplan. “Making Data Structures Confluently Persistent”, Journal of Algorithms, 48(1): 16-58 (2003). [link]

Fleischer, Rudolf. “A Simple Balanced Search Tree with *O*(1) Worst-Case Update Time.” International Journal of Foundations of Computer Science. 7(2): 137-150 (1996). [link]

Sleator, Daniel D, and Robert E Tarjan. “A Data Structure for Dynamic Trees”, Journal of Computer and System Sciences, 26(3): 362-391 (June 1983). [link]

Cormen, Thomas H, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. “Introduction to Algorithms.” The MIT Press (2001). Section 14.1: Dynamic order statistics, 302-307.