COLUMBIA UNIVERSITY
IN THE CITY OF NEW YORK

RESEARCH PROJECT

COMS 4901

PROF. STEPHEN EDWARDS AND JOHN HUI

Implementation of two solutions to the Order Maintenance Problem

Alexis GADONNEIX (ag4625)

December 2023

1 Tests

Blabla

```
        fn test() {
                code = working;
        }
```

maths:

$$\begin{aligned} \text{minimize} \quad & \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij} \\ \text{subject to} \quad & \sum_{i=1}^{n} x_{ij} = 1, \quad j = 1, \ldots, n, \\ & \sum_{j=1}^{n} x_{ij} = 1, \quad i = 1, \ldots, n, \\ & x_{ij} \in \{0, 1\}, \quad i, j = 1, \ldots, n. \end{aligned} \tag{1}$$

2 Introduction

The order maintenance problem is a well-known problem in computer science. It is defined as follows: given a set of elements, we want to maintain a data structure that supports the following operations:

- `insert(x)`: insert a new element right after `x`
- `delete(x)`: delete `x`
- `compare(x, y)`: which element of `x` and `y` has higher priority?

A first very crude solution to this problem is to use a vector. But all operations are linear in the size of the vector. We can do better. A better idea would be to use a linked list. But then, the `compare` operation is linear in the size of the list...

This data structure has several applications in computer science such as process scheduling and graph algorithms. We will focus on two data structures to solve this problem: The firstwas proposed by Dietz & Sleator in 1987 [**10.1145/28395.28434**] and the second by Bender et al. in 2002 [**10.5555/647912.740822**]. The two solutions have very similar theoretical bounds, but the second one is more practical.

The main goals of this project are:

- Implement both solutions in Rust
- Test and debug
- Benchmark, compare, and optimize

3 Algorithms

3.1 Naive

Before diving into the two solutions, let's look at a naive solution to the problem. We can store our priorities in a binary tree that grows deeper and deeper as we insert elements. To avoid having to maintain an actual tree, we can simply use the labels of the nodes:

- The first priority is labeled 0
- When we do an `insert(x)`, we relabel `x` to be $2 \times$ `label(x)` and the new element has label $2 \times$ `label(x)` $+ 1$
- The comparison between two elements is simply a comparison of their labels

TODO: insert image

This solution is easy to implement and operations are fast, but it has a major drawback: the labels grow exponentially with the number of insertions and will quickly overflow.

3.2 Dietz & Sleator

In the solution proposed by Dietz & Sleator in 1987 [**10.1145/28395.28434**], we think of the set of possible labels (e.g. 0 to 2^{64}) as a circular list that we will fill in as we insert elements. The priorities are connected together by a circular doubly-linked list. The general idea for inserting a new priority after a priority `x` is the following (see algorithm TODO for more details):

- We iterate through the successors of `x` and compute a weight for each of them based on the distance between the elements
- We stop when we reach an element whose weight is smaller than a threshold (intuitively, this means that the element is "far enough")
- We then relabel those elements evenly
- Finally, we choose a label in between the labels of `x` its successor (in the middle) and we create a new priority with this label

We call this algorithm **list-range relabeling**.

TODO: insert image

Note that to avoid overflowing, we will simply loop back to 0 when we reach the maximum label (2^{64}). But this means that in order to compare two elements, we have to keep track of a "base" label starting at 0 and shifting if necessary. Then, the comparison operation is:

$$(\texttt{label(x)} - \texttt{label(base)}) \mod 2^{64} < (\texttt{label(y)} - \texttt{label(base)}) \mod 2^{64}$$

Deleting an element of the data structure is as simple as removing it from the linked list.

The paper proves the following theoretical complexity bounds (where n is the number of elements in the data structure):

- `insert(x)`: $O(\log n)$ (amortized)
- `delete(x)`: $O(1)$
- `compare(x, y)`: $O(1)$

It is possible to get an $O(1)$ amortized complexity for `insert(x)` by adding a level of indirection using a technique by Tsakalidis [**10.1007/BFb0036494**]. We chose not to implement this extra-layer for the sake of simplicity and because we thought that the complexity gains would be too small in comparison to the cost of the extra lookup.

This data structure theoretically only supports $\sqrt{M}$ insertions where M is the number of possible labels (2^32 or 2^{64}). It is a significant limitation on a 32-bits machine as you can insert at most 65K priorities.

3.3 Bender et al.

In 2002, Bender et al. proposed a new solution to the problem in [**10.5555/647912.740822**]. The main goal of this approach is not to provide better bounds, but to get an algorithm and a proof that are more intuitive. In practice, we will see that our implementation is in fact faster than the first algorithm for some operations.

This time, we look at the set of possible labels (0 to 2^{64}) as the leaves of a binary tree whose root is the empty string, and each node n has two children $n:0 = 2 \times n$ and $n:1 = 2 \times n + 1$ (where : is the concatenation to the binary representation). We don't have to maintain an explicit tree because we can easily compute the parent or the children of a node with bit operations. And once again, the priorities (which are leaves of the implicit tree) are connected together by a doubly-linked list (this one doesn't need to be circular).

The algorithm for `insert(x)` is pretty straightforward (ref algo TODO), and the key steps are:

- if there is space (`label(next(x))` $>$ `label(x)` $+ 1$), then just insert the new element somewhere in between.
- if there is no space available, we have to relabel (we call this algorithm **tag-range relabeling**).
  - To do so, we simply have to go up the tree and find the first sub-tree containing `x` that is not "too dense".
  - Once we find this sub-tree, we simply have to spread evenly all the elements it contains over the range of labels the sub-tree. Note that once again, all these operations can be done with simple bit operations over the labels.

TODO: Insert image

The paper proves that this algorithm has the same complexity bounds as the previous one:

- `insert(x)`: $O(\log n)$
- `delete(x)`: $O(1)$
- `compare(x, y)`: $O(1)$

The same indirection trick can be applied to reduce the insertion complexity, but we chose not to implement it for the same reasons as before.

Deleting an element of the data structure is as simple as removing it from the linked list. And a small but important difference is that, in order to compare two elements, we simply have to compare their labels (and we don't have to compute their position relative to a base as in the previous solution).

The previous data structure was limited to $\sqrt{M}$ insertions. Here, the maximum number of elements you can insert depends on the parameter T. If T is close to 2, the insertion will be faster but you can only insert a small amount of priorities before the root overflows. If T is close to 1, the algorithm is slower but you can insert more priorities (close to M). The issue is that it's difficult to find a good trade-off as the user doesn't usually know up-front how many priorities he will need. But the good news is that T doesn't have to be fixed and you can compute the best T at each insertion. This computation is not free though, and we will see later (TODO ref to relevant section) how we made this process faster.

4 Implementation

In this section, I will present the main data structures and design choices of our implementations, as well as some tricks we used to improve the speed.

4.1 Interface

The project was built as a Rust library (a `cargo crate`). In order to have a common interface for multiple implementations, as well as shared tests and benchmarks, we leveraged Rust's `Trait` system. The interface for a priority is minimal:

- `new: () -> Priority`: create a new priority, unrelated to any other priority
- `insert: Priority -> Priority`: insert a new priority right after the argument

The comparison is made possible through the implementation of the `PartialCmp` trait which lets us use the common comparison operators (`>=`, `<`, etc.).

The deletion is implemented through the `Drop` trait, which acts as a destructor. A priority is deleted from the data structure when it goes out of scope.

4.2 Common data structures

I won't go too much into the technical details, but we used a couple of data structures to implement both solutions ([**10.5555/647912.740822**] and [**10.1145/28395.28434**]).

- `PriorityInner` is a node of a circular doubly-linked list. It can be seen as the private interface of a `Priority`. It stores the following fields: `prev`, `next`, `label`, `ref_count`. The first two are (smart) pointers to the neighbors, the third is the label as a `usize = u64` (on a 64-bit machine), and the last one is used for the manual reference counting. Those fields are actually wrapped in smart pointers called `RefCell`, mutable memory locations. It is necessary to abide by Rust's borrow checker when trying to mutate a field of an object that is not owned or borrowed as mutable (this is called the interior mutability pattern).
- `Arena` is the internal store of priorities containing all the `PriorityInner`. In our case, it's a `Slab`. It is indexed by `PriorityKey`, an alias for `usize`. The implementation of the `Arena` contains all the methods in charge of creating, deleting, and connecting the `PriorityInner`, the nodes of the linked list.

The end-user has access to a `Priority`, which contains (privately) its `PriorityKey` and a reference to the `Arena`.

4.3 Pre-compute the density thresholds

5 Testing, Benchmarking, and results

5.1 Unit tests, Integration tests, and Quickcheck

5.2 Bugs found and fixed

Overflow issue DnS

min and max labels overflow Bender

5.3 Benchmarks, Profiling, and Optimization

6 Conclusion and Future Work

- Test and benchmark on a microcontroller
- Try different allocation strategies
- Implement a "naive start" for small use cases