Columbia University
IN THE CITY OF NEW YORK

Research Project

COMS 4901

Prof. Stephen Edwards and John Hui

Rust implementation of solutions to the Order Maintenance Problem

Alexis Gadonneix (ag4625)

December 2023

1 Tests

Blabla

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

maths :

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

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 [DS87] and the second by Bender et al. in 2002 [Ben+02]. 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 them
- Benchmark, compare, and optimize them

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 [DS87], 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 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

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:

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

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)`: $O(1)$

todo: talk about indirection

3.3 Bender et al. [Ben+02]

4 Implementation

5 Testing, Benchmarking, and results

5.1 Unit tests, Integration tests, and Quickcheck

5.2 Bugs found and fixed

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

References

[DS87] P. Dietz and D. Sleator. "Two Algorithms for Maintaining Order in a List". In: *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*. STOC '87. New York, New York, USA: Association for Computing Machinery, 1987, pp. 365–372. ISBN: 0897912217. DOI: `10.1145/28395.28434`. URL: `https://doi.org/10.1145/28395.28434`.

[Ben+02] Michael A. Bender et al. "Two Simplified Algorithms for Maintaining Order in a List". In: *Proceedings of the 10th Annual European Symposium on Algorithms*. ESA '02. Berlin, Heidelberg: Springer-Verlag, 2002, pp. 152–164. ISBN: 3540441808.