UNIVERSITY OF TALCA
ENGINEERING FACULTY
CIVIL COMPUTER ENGINEERING SCHOOL

Experimental analysis of (I)IQS to fine-tune support for arrays with repeated elements

ERIK ANDRÉS REGLA TORRES

Supervisor: RODRIGO PAREDES

Thesis to apply for a Civil Computer Engineer degree

Curicó – Chile
mes, año

UNIVERSITY OF TALCA
ENGINEERING FACULTY
CIVIL COMPUTER ENGINEERING SCHOOL

Experimental analysis of (I)IQS to fine-tune support for arrays with repeated elements

ERIK ANDRÉS REGLA TORRES

Guide: RODRIGO PAREDES

Advisor: HUEÓN 1

Advisor: HUEÓN 2

Thesis to apply for a Civil Computer Engineer degree

This document was graded with a score of: ________

Curicó – Chile

mes, año

Dedicated to... someone ?

ACKNOWLEDGEMENTS

Agradecimientos a ... (how the fuck do I choose whom to acknowledge?)

CONTENT INDEX

Page

FIGURE INDEX

TABLE INDEX

SUMMARY

I'm gonna write the summary as the last part.

1. Introduction

Aquí va el texto del capítulo 1...

1.1 Context

Aquí va el texto de la primera sección del capítulo 1...

1.2 Application areas

Aquí va el texto de la primera subsección de la primera sección del capítulo 1...

1.3 Problem description

Aquí va el texto de la segunda subsección de la primera sección del capítulo 1...

1.4 Goals

Aquí va el texto de la segunda subsección de la primera sección del capítulo 1...

1.4.1 General goals

Aquí va el texto de la segunda subsección de la primera sección del capítulo 1...

1.4.2 Specific goals

Aquí va el texto de la segunda subsección de la primera sección del capítulo 1...

1.5 Document Structure

Aquí va el texto de la segunda subsección de la primera sección del capítulo 1...

1.6 Problem description

Aquí va el texto de la segunda subsección de la primera sección del capítulo 1...

2. Background

2.1 Sorting algorithms

One of the basic problems on algorithm design is the *sorting problem*, defined as for a given input sequence A of n numbers $\langle a_1, a_2, ..., a_n \rangle$ to find a permutation $A' = \langle a'_1, a'_2, ..., a'_n \rangle$ that yields $\forall a'_i \in A', a'_1 \leq a'_2 \leq ... \leq a'_n$.

Sorting algorithms are commonly used as intermediate steps for other processes, making them one of the most fundamental procedures to execute on computing problems, and strategies for solving this problem can vary depending on the input case constraints. For example, the number of repeated elements, their distribution, if there is some known info beforehand to accelerate the process, etc.

2.1.1 Types of sorting algorithms

The best reference on how to classify and understand which algorithm is best suitable for a given case is *A survey of adaptive sorting algorithms* by Vladimir Estivil[3] which gathers all the information at that time regarding *adaptive sorting algorithms* [4], *disorder measures* and *expected-case and worst-case* sorting.

A sorting algorithm is said to be adaptive if the time taken to solve the problem is a smooth growing function of the size and the measure of disorder of a given sequence. Note that the term array is not used on this definition as it extrapolates any generic sequence that is not bound to be contigous.

2.1.2 Measuring disorder

The concept of disorder measure is highly relative to the problem to be solved and as expected, not all measures work for all cases. One of the most common metrics used on partition-based algorithms is the *number of inversions* required to sort a given array. While this holds true for algorithms like *insertsort* which have their running time affected by how the elements are arranged in the sequence, it's not the case of *mergesort*, which is not an adaptive algorithm given that has a stable running time regardless on how the elements are distributed. Whilst the running time is a function of the size, it's not in function of the sequence. Estivil [3] on his survey describes ten functions that can be used to measure disorder on an array when used on adaptive sorting algorithms.

Expected case and Worst-case adaptive internal sorting

One of the classification of adaptive sorting algorithms is the *Expected-case adaptive (internal) sorting*, on which their design is driven by that worst cases are unlikely to happen in practice so, there is no harm on using it, in contrast of the definition of *Worst-case adaptive (internal) sorting*, which assumes a pessimistic view hence the design is driven to ensure a deterministic worst case running time and asymtotic complexity.

The approach taken by such algorithms can be classified as *distributional*-in which a "natural distribution" of the sequence is expected to be solved- or *randomized* -on which their behaivour is not related on how the sequence is distributed at all-. There is a huge problem when dealing with distributional approaches as they tend to be very sensible to changes on the sequence distribution, making them suitable to highly constrained problems on specific-purpose algorithm.

On the other hand, randomized approaches have the benefit of generality and being rather simple to port to other implementations due to their nature.

By example, let's take as example the QuickSelect algorithm -which is the basis of IQS which will be explained in detail later- used to find the element that belongs to the $k-th$ position or a given sequence A. This searching algorithm can be classified as *partition-based*, given that the process in charge of preserving the invariant is the partition stage.

As it can be seen, the behaivour of $quickselect$ depends on how the element is selected in the *select* procedure. Then, we can implement two versions of *select*, namely $select_fixed$ and $select_random$ which yields different values in order to introduce randomization into $quickselect$.

Algorithm 1 QuickSelect definition

procedure $quickselect(A, i, j, k)$
  $pIdx \leftarrow select(i, j)$
  $pIdx \leftarrow partition(A, pIdx, i, j)$
  if $pIdx = k$ **then return** A_k
  if $pIdx < k$ **then return** $quickselect(A, k, j)$
  if $pIdx > k$ **then return** $quickselect(A, i, k)$

Algorithm 2 Fixed Selection

procedure $select\_fixed(i, j)$
  return $\frac{(i+j)}{2}$

In such cases, whilst the randomized version of QuickSelect will take average time of $n * log_2(n)$ to complete the task, we can see that for the fixed pivot version, it depends on the distribution of data, which can bias the pivot result. Now we have two versions of QuickSelect algorithm, with both distributional and randomized strategies.

2.2 Incremental Sorting

While sorting algorithms can be seen as a straightforward process, the definition of sorting can be extended as *partial sorting* and *incremental sorting*, as in practice, while sorting is used as the intermediate step of many procedures, it is not mandatory to always sort the entire array, rather than just sort a fragment of interest.

As partitions of a sequence can be seen as a equivalence relationship between the pivot and the leftmost and rightmost segments[2], then for a given sequence $A' \in A$, we can define a *partial order* if the relationship on the elements of A is reflexive, antisymmetric and transitive and then A' is called a *partially ordered sequence.*

Using this very same definition of partial order, if we retrieve the elements of a sequence and store them as A_s -a partially sorted sequence of A, if the elements are retrieved in a way that subsecuential pushes to the A_s is always ordered, then it is

Algorithm 3 Random selection

procedure $select\_random(i, j)$
  return $random\_between(i, j)$

said that A is being *incrementally sorted*.

A good example of the uses of this kind of sorting are the results given by a web search engine. When a user inputs a query, regardless of the size of the database, the search engine paginates the results and presents only the first page of results. It is not actually needed to sort all the results, rather to get the most relevants, then there is no need to waste time sorting all the elements for a query that can be executed only one time.

2.2.1 IQS

Incremental QuickSort (IQS) [5] is a variant of QuickSelect designed for usage on incremental sorting problems, intended to be a direct replacement of HeapSort on Kruskal's algorithm.

Algorithm overview

Algorithm 4 IncrementalQuickSort

```
procedure iqs(A, i, S)
    if i ≤ S.top() then
        S.pop() return A[i]
    pivot ← select(i, S.top() − 1)
    pivot′ ← partition(A, pivot, i, S.top() − 1)
    S.push(pivot′)
    return iqs(A, i, S)
```

As it can be seen, the execution of this algorithm is similar to sorting the array by executing sequentially QuickSelect for $1, 2, 3, ..., n$ in order, as it is yielding each one of those elements in A already ordered. The advantage of using IQS is that since the stack stores all the previous call results, in average all subsecuential calls are cheaper than the first one, hence the $n * log_2(n)$ running time.

Worst case

A way to force a worst case execution is to force the pivot selection to choose each time a pivot that makes a whole partition of the array and leaves it at the end. To force this we use a sequence of elements ordered in a decreasing way and we force the pivot selection to always select the first element of the sequence.

2.2.2 IIQS

A slightly more complex version of IQS, intented to avoid the worst case running time of IQS by changing the pivot selection strategy on function of how many recursive calls has executed so far[6].

The partition algorithm uses the information of the relative position of the given pivot ar the partition stage to determine if the pivot obtained can be refined or not by using another pivot selection technique, in this case the used algorithm is the *median of medians*[1], which guarantees that the median selected will belong to $P_{70} \cap P_{30}$.

If the median returned by *select* does not belong to that segment, then median of medians is executed in order to guarantee a decrease of the search space for the next call.

Algorithm overview

Algorithm 5 Introspective IncrementalQuickSort

```
procedure IIQS(A, S, k)
    while k < S.top() do
        pidx ← random(k, S.top() − 1)
        pidx ← partition(A_{k,S.top()−1}, pidx)
        m ← S.top() − k
        α ← 0.3
        r ← −1
        if pidx < k + αm then
            r ← pidx
            pidx ← pick(A_{r+1,S.top()−1})
            pidx ← partition(A_{r+1,S.top()−1}, pidx)
        else if pidx > S.top() − αm then
            r ← pidx
            pidx ← pick(A_{k,pidx})
            pidx ← partition(A_{k,r}, pidx)
            r ← −1
        S.push(pidx)
        if r > −1 then
            S.push(r)
    S.pop()
    return A_k
```

The reason behind why use median of medians is that has $O(n)$ complexity, same as partition, preventing the asymtotic complexity to increase if such algorithm is used.

2.3 Design of experiments

3. Methodology

3.1 Experimental design and goals

3.1.1 Instances of test cases

Ordered unique

Random unique

Distributed repeated

Distributed repeated with random unique portion

3.1.2 Pilot experiments

Incremental version of BFPRT

Introspective step rule changes

Three-way partition pivot location bias

Three-way partition pivot store

Change rules to store pivots

3.2 Metrics

Number of inversions

Local entropy decay

Execution time (rsu)

BFPRT executions

Partitioner executions

3.3 Tunning

3.3.1 Data generation and execution control

4. Experiment stage

4.1 Experimental setup

4.2 Experimental results

4.3 Metrics and indicators

5. Segundo Capítulo

Glossary

El primer término: Este es el significado del primer término, realmente no se bien lo que significa pero podría haberlo averiguado si hubiese tenido un poco mas de tiempo.

El segundo término: Este si se lo que significa pero me da lata escribirlo...

Bibliography

[1] Manuel Blum, Robert W. Floyd, Vaughan Pratt, Ronald L. Rivest, and Robert E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7(4):448–461, Aug 1973.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[3] Vladmir Estivill-Castro and Derick Wood. A survey of adaptive sorting algorithms. *ACM Comput. Surv.*, 24(4):441–476, December 1992.

[4] Kurt Mehlhorn. *Data Structures and Algorithms 1*. Springer Berlin Heidelberg, 1984.

[5] Gonzalo Navarro and Rodrigo Paredes. On sorting, heaps, and minimum spanning trees. *Algorithmica*, 57(4):585–620, Mar 2010.

[6] E. Regla and R. Paredes. Worst-case optimal incremental sorting. In *2015 34th International Conference of the Chilean Computer Science Society (SCCC)*, pages 1–6, 2015.

ANNEX

A. El Primer Anexo

Aquí va el texto del primer anexo...

A.1 La primera sección del primer anexo

Aquí va el texto de la primera sección del primer anexo...

A.2 La segunda sección del primer anexo

Aquí va el texto de la segunda sección del primer anexo...

A.2.1 La primera subsección de la segunda sección del primer anexo

B. El segundo Anexo

Aquí va el texto del segundo anexo...

B.1 La primera sección del segundo anexo

Aquí va el texto de la primera sección del segundo anexo...