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

Page

TABLE INDEX

Page

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...

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[7] which gathers all the information at that time regarding *adaptive sorting algorithms* [12], *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 [7] 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

```
1: procedure quickselect(A, i, j, k)
2:     pIdx ← select(i, j)
3:     pIdx ← partition(A, pIdx, i, j)
4:     if pIdx = k then return A_k
5:     if pIdx < k then return quickselect(A, k, j)
6:     if pIdx > k then return quickselect(A, i, k)
```

Algorithm 2 Fixed Selection

```
1: procedure select_fixed(i, j)
2:     return (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.

Estivil [7] on his survey lists 10 different *metrics of disorder* to be used when studing adaptive sorting algorithms. For the sake understanding, we will asume that S is any sequence of numbers in any order and W_1, W_2 are instances of S yielded as:

$$W_1 = 6, 2, 4, 7, 3, 1, 9, 5, 10, 8$$

$$W_2 = 6, 5, 8, 7, 10, 9, 12, 11, 4, 3, 2$$

The definitions are as folows:

Maximum inversion distance (Dis)

Defined as the largest distance determined by an inversion of a pair of elements in a given sequence[6]. By example, in W_1, 5 and 6 are the elements which require an inversion in order to be locally sorted whom are the most furthest apart in the sequence, hence $Dis(W_1) = 7$.

Algorithm 3 Random selection

```
1: procedure select_random(i, j)
2:     return random_between(i, j)
```

Maximum sorting distance (Max)

This metric considers that local disorder is not as important as global disorder, under the premise that when indexing objects if they are grouped in some way, then it is easy to find similar elements, on the other hand if there is an element belonging to a group and it is on another place far away in the sequence then it is hard to find such element. Then Max is defined as the largest distance that a an element of the sequence needs to travel in order to be in its sorted position[6]. By example, $Max(W_1) = 5$ given that 1 requires to move 5 positions to the left in order to be sorted.

Minimum number of exchanges (Exc)

Based on the premise that the number of performed operations is important to evaluate a sorting algorithm, a simple operation to measure is the minimum number of swaps between indices involved on a given sorting operation, then Exc is defined as the minimum number of exchanges required to sort the entire sequence[10]. By example, as it is impossible to sort W_1 in fewer than 7 exchange operations, then $Exc(W_1) = 7$.

Minimum elements to be removed (Rem)

Another definition of disorder is as a phenomena produced by the incorrect insertion of elements in a sequence[8]. In this fashion, we can define the minimum amount of elements to be removed in order to the the longest sorted sequence. By example, by removing 5 elements from W_1 we can obtain a sorted sequence, then $Rem(W_1) = 5$.

Minimum number of ascending portions (Runs)

Driven by the definition of partial sorting[5], any sorted subsequence of S implies that locally has a minimum amount of ascending runs as 1 to be sorted, then another measure is the minimum number of ascending runs that can be found on any sequence, given that the elements that compose the sequence must be in the same order as found in the original sequence. In this case, W_1 has 5 ascending runs, hence $Runs(W_1) = 5$. Knuth defined this phenomena as *step-downs*[8].

Minimum number of shuffled subsequences (SUS)

A generalization of *Runs* but ignoring the fact that the elements but be in the same sequential order as found in the original sequence by removing elements from it. Then SUS is defined as the minimum number of ascending subsequences in which we can partition a sequence[4]. In this case W_2 has 7 ascending runs, then, $SUS(W_2) = 5$.

Minimum number of shuffled monotone subsequences (SMS.SUS)

A generalization of SUS now the elements can be grouped as a subsequence as long as they are sorted in any way generating a monotone subsequence. For W_2 we can get 2 ascending shuffled sequences[4] and 1 descending shuffled sequence, hence $SMS(W_2) = 3$.

Sorted lists constructed by Melsort (Enc)

Skiena's Melsort [16] takes another approach at presortnedness by treating sequences as a set of enroached lists, which is similar to mergesort but chunks are generated not by the recursive call itself but rather by a series of deque operations[2]. Then the number of enroached lists generated by Melsort are a measure of disorder which Skiena denoted as Enc. For W_1 the number of enroached lists generated is 2, hence $Enc(W1) = 2$.

Oscilation of elements in a sequence (Osc)

Defined by Levcopoulos as a metric of presortnedness for heapsort, Osc is defined for each element as the number of intersections for a given element over the cartesian tree of a sequence[9], motivated by the geometric interpretation of the sequence itself. In the case of W_1 as the cartesian tree manifests 5 crosses between its elements, $Osc(W_1) = 5$.

Regional insertion sort (Reg)

Based on the internal workings of *regional insertion sort*[1] which is a historcial sorting algorithm. Then Reg is the value of the time dimension required to sort a certain sequence.

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[5], 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 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) [14] 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

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.

Algorithm 4 IncrementalQuickSort

```
1: procedure iqs(A, i, S)
2:     if i ≤ S.top() then
3:         S.pop() return A[i]
4:     pivot ← select(i, S.top() − 1)
5:     pivot′ ← partition(A, pivot, i, S.top() − 1)
6:     S.push(pivot′)
7:     return iqs(A, i, S)
```

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[15].

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*[3], 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

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.

Algorithm 5 Introspective IncrementalQuickSort

```
1: procedure IIQS(A, S, k)
2:     while k < S.top() do
3:         pidx ← random(k, S.top() − 1)
4:         pidx ← partition(A_{k,S.top()−1}, pidx)
5:         m ← S.top() − k
6:         α ← 0.3
7:         r ← −1
8:         if pidx < k + αm then
9:             r ← pidx
10:            pidx ← pick(A_{r+1,S.top()−1})
11:            pidx ← partition(A_{r+1,S.top()−1}, pidx)
12:        else if pidx > S.top() − αm then
13:            r ← pidx
14:            pidx ← pick(A_{k,pidx})
15:            pidx ← partition(A_{k,r}, pidx)
16:            r ← −1
17:        S.push(pidx)
18:        if r > −1 then
19:            S.push(r)
20:    S.pop()
21:    return A_k
```

2.3 Experimental algorithmics

Experimental algorithmics can be seen as a spin-off of *Design of experiments* which is a statistic technique applied to understand how a process is affected by the relationships between its variables and external factors, by sistematically studying and explaining the variation of information provided by altering the environment of the same process[17].

Mostly in algorithm design, pure mathematical and theoretical approaches are taken in order to devise how to create, develop and optimize new algorithms, but the gap between theoretical analysis and the actual implementation is still huge with rise of new platforms and compute architectures nowadays.

In the case of computer science, computational experiments are by any means are not to replace theoretical analysis but rather to complement and speed up the discovery process by guiding via results their research, tuning and implementation.

The most complete recopilation of techniques and recommendations on how to apply design of experiments to algorithm design is currently made by Catherine McGeoch on her book *A guide to experimental Algorithmics*[11], con which she introduces a comphrensive guide on how to apply this method by introducing a fair amount of guidelines for computer scientists.

2.3.1 Methodology overview

On experimental algorithmics the first step is to plan the involved experiments by following the following steps in a cyclic way:

Planning

- Formulate a question.
- Assemble or build the test environment.
- Experiment design to address the question.

At this stage we don't analyse any data yet as we only design the process to study at a later stage. Given that the experimental setup can alter the question at hand, this process tends to repeat until the experiment is fully assembled.

In order to determine if a given experiment is viable -namely the *workhorse experiment*, the practitioner must perform a series of pilot experiments beforehand, in order to understand the environment, implementation challenges and the problem itself. Pilot experiments are expected to be small experiments which answer a highly contrained and specific question that drives towards the contruction of the workhorse experiment, as workhorse experiments are expected to be complex in both setup, execution and analysis.

At this step it is expected that the practitioner develops metrics, indicators and setups which leads to a deeper understanding of the original question proposed.

Execution

Whilst this step is taken locally as a sequential process, execution step is required during the planning stage for the pilot experiments as well as the execution state of the workhorse experiment by executing the experiment and gathering data by running tests and then analysing the data in order to glean information and insight. If the question at hand is answered then the process is finished, otherwise the process starts again from the planning stage taking the results from this stage as input.

2.3.2 Result driven development

As results from experiment execution and analysis yield data from all the levels of algorithm design, those results are used to tune up existing algorithms in order to optimize their execution for certain cases, specific architectures or given contraints.

One of the key benefits of this strategy is that whilst a pure theoretical approach can be difficult or unfeasible in some cases, a systematic experimental approach can help to both guide theoretical analysis by giving insight and validating theoretical results.

This approach is widely used on metaheuristics tunning, as experiments are used to fill the gap by simplyfing asumptions necessary to theory and the realistic use case, caracterize and profile best, average and worst cases, suggest new theorems and proof strategies and to exten theroretical analyses to realistic inputs.

A example of it is the building of *LZ-index* [13], a compressed data structure designed to support indexing and fast lookup. Navarro used experiments to guide the choices of during the implementation process and compare the finished product

against current competing strategies.

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

4. Experiments

4.1 Experimental setup

4.2 Experimental results

4.3 Metrics and indicators

4.3.1 Local entropy decay

4.3.2 BFPRT executions

4.3.3 Partitioner executions

4.4 Tunning

4.4.1 Data generation and execution control

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] *Recent Issues in Pattern Analysis and Recognition*. Springer Berlin Heidelberg, 1989.

[2] *Computer Science*. Springer US, 1992.

[3] 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.

[4] Svante Carlsson, Christos Levcopoulos, and Ola Petersson. Sublinear merging and natural mergesort. *Algorithmica*, 9(6):629–648, Jun 1993.

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

[6] Vladimir Estivill-Castro and Derick Wood. A new measure of presortedness. *Information and Computation*, 83(1):111–119, Oct 1989.

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

[8] Donald E. Knuth. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., USA, 1997.

[9] C. Levcopoulos and O. Petersson. Adaptive heapsort. *Journal of Algorithms*, 14(3):395–413, May 1993.

[10] Heikki Mannila. Measures of presortedness and optimal sorting algorithms. *IEEE Transactions on Computers*, C-34(4):318–325, Apr 1985.

[11] Catherine C. McGeoch. *A Guide to Experimental Algorithmics*. Cambridge University Press, USA, 1st edition, 2012.

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

[13] Gonzalo Navarro. Implementing the lz-index. *Journal of Experimental Algorithmics*, 13:1.2, Feb 2009.

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

[15] 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.

[16] Steven S. Skiena. Encroaching lists as a measure of presortedness. *BIT*, 28(4):775–784, Dec 1988.

[17] Jr. Wagner, John R., III Mount, Eldridge M., and Jr. Giles, Harold F. *Design of Experiments*, page 291–308. Elsevier, 2014.

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...