Quickselect - Analysis of the expected number of comparisons

Alberto Parravicini

1 Abstract

The following report will analyze the number of comparisons performed by a **Rust** implementation of the **Quickselect** algorithm, a randomized algorithm used to find the $k-th$ smallest element in an unordered list.

The first section of the report presents the algorithm, and its **Rust implementation**.

The second section studies the number of **comparisons** between done by the algorithm, i.e. how many elements from the input list must be compared to each other to give the desired output. This numbers are then compared to the theoretical expected number of comparisons done by the algorithm.
The algorithm is tested against vectors with increasing size n and various ranges of k.

As addendum, an analysis of the **execution time** of *Quickselect* is reported, to show how the number of comparisons influences the running time of the algorithm.

2 Quickselect's algorithm implementation

2.1 *Theoretical background*

Given a list of $\mathbf{n}$ elements for which exists a total order under $\leqslant$, the **Quickselect** algorithm is used to find the $\mathbf{k-th}$ smallest element in the list (also referred as element of *rank k*), for $\mathbf{k} \in [\mathbf{1},\ \mathbf{n}]$.

To select the element of rank k (with $1 \leqslant k \leqslant n$), the algorithm works as follow:

```
quickselect(list, start, end):
    pick an element at random from the list,
        in the range [start, end], call it pivot;
    put all the elements < pivot on its left;
    put all the elements > pivot on its right;
    the pivot is now in its sorted position in the list:
        if position(pivot) == k, return it;
        if position(pivot) < k, return quickselect(start, position(pivot));
        if position(pivot) > k, return quickselect(position(pivot), end);
```

It can be seen how there is a close resemblance between the **Quickselect** and the **Quicksort** algorithms.
Indeed, **Quickselect** can be seen as a modified version of **Quicksort** in which the recursion is applied only to the portion of the list where the desired element of *rank k* is expected to be found.

2.2 *Rust implementation*

Rust was chosen as the programming language of the implementation as the inherent speed of the language, combined with the relative simplicity of the **Quickselect** algorithm, make it a good candidate for performing a large number of tests over many different parameter values.

What follows are snippets of code, to explain how the **Quickselect** algorithm was implemented:

- **Quickselect**

```rust
fn quickselect(vec: &mut Vec<usize>,
        start: usize, end: usize,
        k: usize, num_of_comparisons: u32) -> (usize, u32) {
  let mut num_of_comparisons: u32 = num_of_comparisons;

  if k < start || k > end {
    panic!("INVALID VALUE OF K: {}", k)
  }
  // If just one element is present, return it;
  if start >= end {
    return (vec[end], num_of_comparisons)
  }
  let pivot_index = rand::thread_rng().gen_range(start, end + 1);
  // Put the current pivot in its correct place, and return its position;
  let (pivot_index, temp_num_of_comparisons) = partition(vec, start, end, pivot_index);
  num_of_comparisons += temp_num_of_comparisons;

  // Apply the sorting only where the desired element could be,
  // or return it if it was found;
```

```
  if k == pivot_index {
    return (vec[k], num_of_comparisons)
  }
  else if k < pivot_index {
    let result = quickselect(vec, start, pivot_index, k, num_of_comparisons);
    return result
  }
  else {
    let result = quickselect(vec, pivot_index + 1, end, k, num_of_comparisons);
    return result
  }
}
```

- **Select a pivot and put it into its sorted position**

```
fn partition(vec: &mut Vec<usize>,
        start: usize, end: usize,
        pivot_index: usize) -> (usize, u32) {
  let pivot_value = vec[pivot_index];
  let mut temp_num_of_comparisons: u32 = 0;

  // Temporarily put the pivot at the end of the vector;
  vec.swap(pivot_index, end);
  let mut store_index = start;
  for i in start..end {
    temp_num_of_comparisons += 1;
    if vec[i] < pivot_value {
      // If a value lower than the pivot is found, put it in the left part of the vector;
      vec.swap(store_index, i);
      // store_index keeps track of how many values lower that the pivot exist,
      // and where the pivot should be placed at the end;
      store_index += 1;
    }
  }
  // Put the pivot in its correct place;
  vec.swap(end, store_index);

  // Return the sorted position of the pivot and the number of comparisons performed;
  (store_index, temp_num_of_comparisons)
}
```

3 Analysis of the expected number of comparisons

3.1 *Theoretical background*

For any two elements X_i, X_j of the list, with $j > i$,

$$X_{i,j} = \begin{cases} 1 \text{ if } X_i, X_j \text{ are compared during the execution of the algorithm.} \\ 0 \text{ else.} \end{cases}$$

The complexity of **Quickselect** is related to the total number of comparisons X performed by the algorithm.
It can be shown that the expected number of comparisons $E[X]$ is given by:

$$E[X] = E\Big[\sum_{i<j} X_{i,j}\Big] = \left(2n + 2n \cdot \ln\left(\frac{n}{n-k}\right) + 2k \cdot \ln\left(\frac{n-k}{k}\right)\right) \cdot (1 + o(1)) \qquad (*)$$

A number of observations can be done on this formula:

- If k is chosen to be proportional to n, the number of comparisons becomes linear with respect to n.
  As an example, if $k = \frac{n}{2}$, $E[X] = n \cdot (2 + 2 \cdot \ln(2)) \cdot (1 + o(1))$.

- The function $(*)$ is not defined for $k = n$; that said, it can be seen that for $k = n$ the value of $(*)$ is very close to its value in $k = 0$.
  As such, the function will be approximated in this way in the following analyses.

- In the interval $k \in [1, n)$, it can be seen that $(*)$ has a single local maximum, in $k = \frac{n}{2}$. In fact, by putting the derivative of $(*)$ w.r.t. k equal to 0:

  $$\frac{\partial}{\partial k}\left(2n + 2n \cdot \ln\left(\frac{n}{n-k}\right) + 2k \cdot \ln\left(\frac{n-k}{k}\right)\right) = \ln\left(\frac{n}{k} - 1\right) = 0$$

  From which one can find $k = \frac{n}{2}$, which is a maximum as the second derivative is negative in the interval $k \in [1, n)$. Moreover, the local minima of $(*)$ are at the extremes of the range of k.

3.2 ***First test:*** *increasing size of the list, select the median*

The first test consisted in running Karger's algorithm against connected graphs with *increasing number of vertices and edges*. The number of vertices goes from 20 to 100 (with a step size of 10 vertices), and the number of edges of each graph is equal to $4 \cdot num\_vertices$. For each graph, Karger's algorithm was run 50 times. For each graph size, 5 different graphs were built (thus, $9 \cdot 50 \cdot 5$ iterations of the algorithm have been performed in this test).

The success rate of the algorithm with respect to each graph size has been compared to the theoretical success rate baseline.

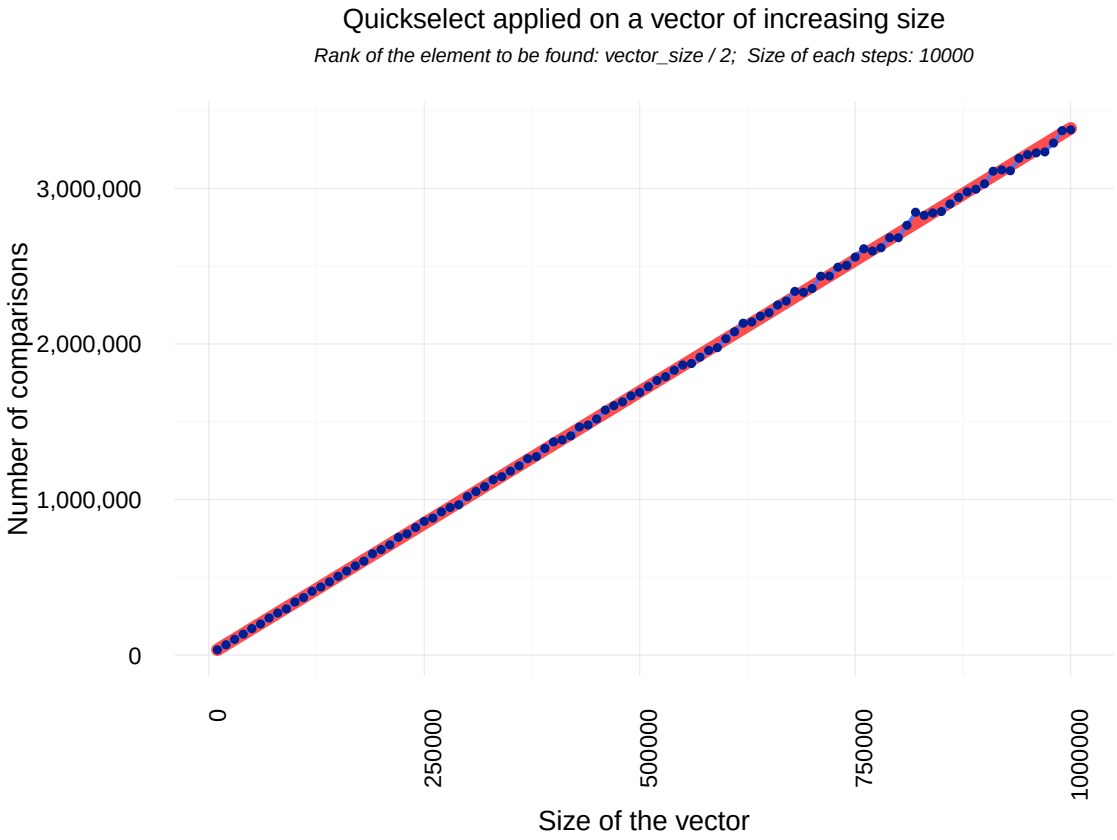

Figure 1: *In blue, the measured number of comparisons. In red, the theoretical number of comparisons.*

It can be seen that if the ratio between number of edges and vertices is kept constant, increasing the number of vertices doesn't hinder the success rate of Karger's algorithm. In all cases, the verified success rate is much higher than the theoretical lower bound, which could be related to the low value of the ratio between number of edges and vertices.

3.3 ***Second test:*** *increasing value of k, fixed list size*

From the previous test, it was empirically shown that as long as the ratio between number of edges and vertices is constant, Karger's algorithm success rate isn't affected by the size of the graph. On the other hand, increasing the number of edges while keeping the number of vertices fixed could have an impact on the algorithm success percentage.

To verify this hypothesis, Karger's algorithm was run on connected graphs with 50 vertices and a number of edges ranging from 50 to 500 (with a step size of 50 edges). For each number of edges, 5 different graphs were built. On each of them,

the algorithm was run 50 times, for a total of $10 \cdot 5 \cdot 50$ iterations of the algorithm. Once again, the results are compared to the theoretical success rate baseline.

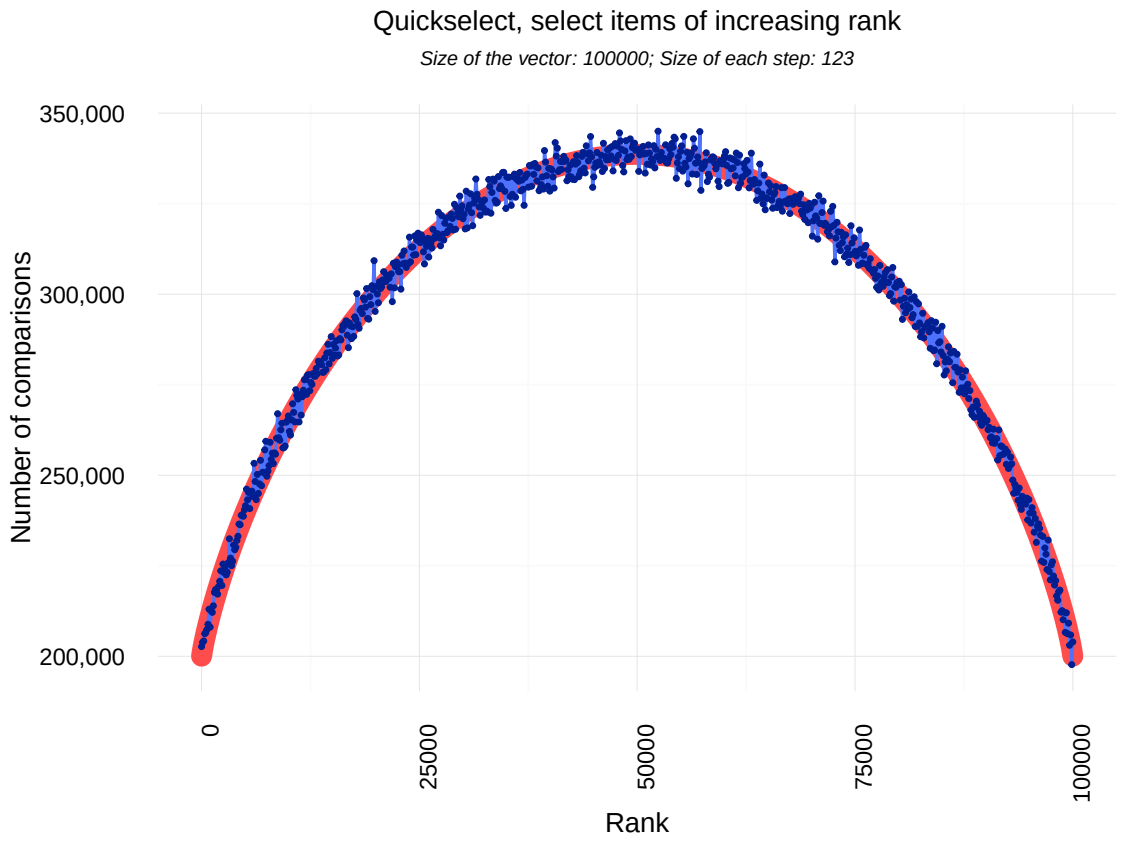

Figure 2: *In blue, the measured number of comparisons. In red, the theoretical number of comparisons.*

It can be seen that increasing the number of edges while keeping the number of vertices fixed dramatically reduces the algorithm success rate, to the point where the theoretical lower bound of the success rate is almost reached.

This can be intuitively explained as follows: Karger's algorithm will output the correct minimum cut size as long as no edge belonging to the minimum cut is contracted.
However, increasing the number of edges while keeping fixed the number of vertices implies that the graph will have many multi-edges. Clearly, if an edge between two vertices belongs to the minimum cut, all edges between these two edges will also belong to the minimum cut.
If we have a multi-edge between two nodes, and this multi-edge belongs to the minimum cut, the probability of contracting it (which would lead to a fail of Karger's algorithm) will be higher that the probability of selecting a single edge.

It can also be noted that if $|V| = |E|$, Karger's algorithm always give the correct minimum cut value (i.e. 1): this happens because the graph is a tree, and contracting the edges will never form a multi-edge. Hence, after $n - 2$ iterations, the graph will always have just 1 edge left.

4 Addendum

4.1 *Execution time analysis*

By making use of the *microbenchmark* library, it is possible to monitor the execution time of the implementation of Karger's algorithm.
Even though the implementation is not optimized for fast execution times, it is still interesting to study how the execution time scales with respect to the number of nodes and edges of a graph.

For both the previous tests, execution times were recorded and summarized in the following plots.

It emerges a linear scaling with respect to the number of vertices and edges.
This empirical results are coherent with the theoretical ones if we assume that the contraction operation happens in constant time.
Even without knowing the details of *igraph*, it is clear that contracting an edge requires a loop whose number of steps is equal to the size of the neighbourhood of a given vertex (i.e. at most $|V|$).
This would give a theoretical complexity of $O(|V|^2)$; however, if the graph is rather sparse, it might happen that the size of the neighbourhood of a given vertex is quite small, and thus the contraction operation happens quite fast in practice.

In the second plot, where the number of vertices is kept constant, this behaviour emerges even more: having a constant number of vertices would imply a constant execution time, but in practice it can be noticed once again a linear scaling; this happens because the graph becomes progressively less sparse, and thus the contraction operation becomes more and more time consuming.

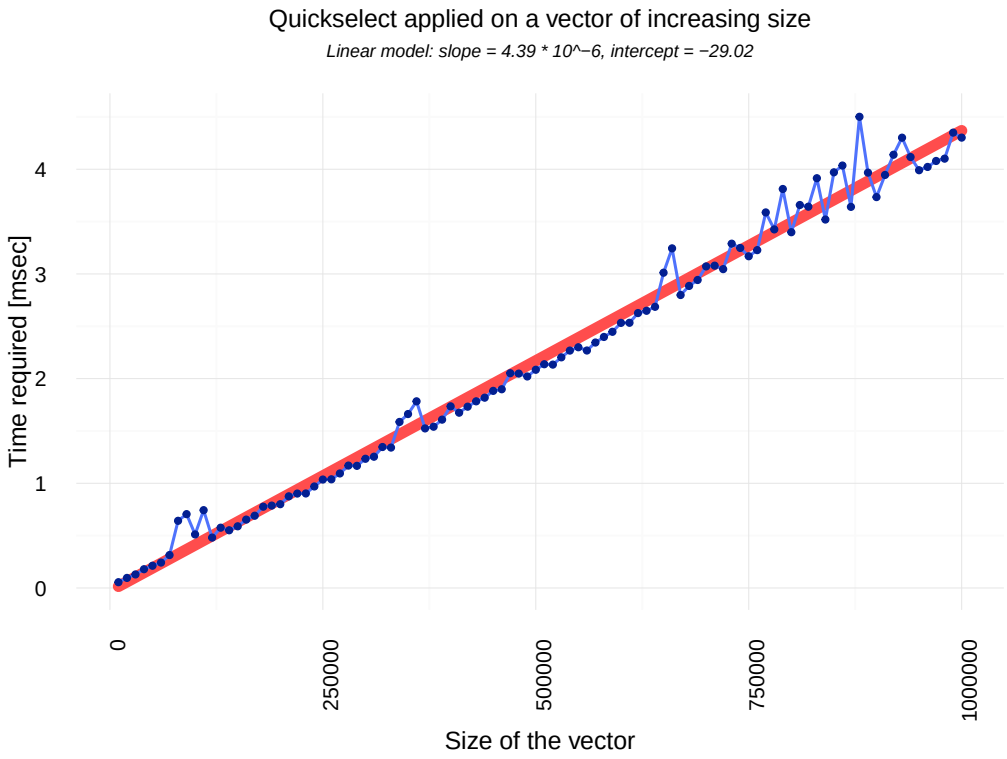

Figure 3: *In blue, the measured execution time. In red, the linear regression of the execution time.*

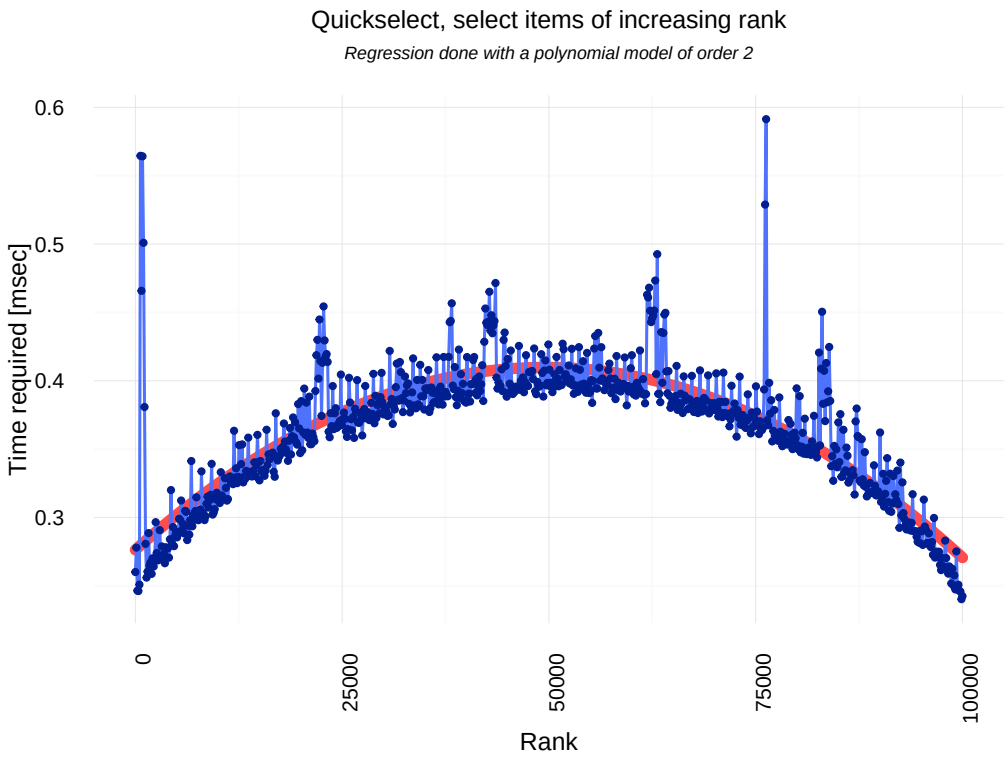

Figure 4: *In blue, the measured execution time. In red, the order 2 polynomial regression of the execution time.*

• **Full code (Rust implementations + R profiling) available at**
https://github.com/AlbertoParravicini/data_structures_and_algorithms