QÜESTIIÓ, Vol 20, 1 pp. 111-119, 1996

AN ANALYSIS OF SELECTION SORT USING RECURRENCE RELATIONS

FRANCESC J. FERRI, JESÚS ALBERT*

Universitat de València

This paper presents a method for obtaining the expected number of data movements executed by the well-known Selection sort algorithm along with its corresponding variance. The approach presented here requires hardly any specific mathematical background. In particular, the average-case cost and variance are represented using recurrence relations whose solutions lead to the desired results. Even though this method is not applicable in general, it serves to conveniently present average-case algorithm analysis in depth in an elementary course on Algorithms.

1. INTRODUCTION

This paper is aimed at complementing the contents of an elementary course on Algorithm Analysis. It presents an exercise in which an algorithm is analyzed in depth assuming a limited mathematical background. Traditionally, similar topics are presented assuming a considerable fluency in Programming, Recursive Algorithms as well as knowledge about certain mathematical tools such as generating functions (Graham *et al.* 1989). In particular, generating functions can be used to obtain the very same results presented here (Knuth 1973a, page 99; Knuth 1973b, page 141). The implementation of the algorithm using a high-level programming language and the measures of its performance will be used to understand the average behavior of the algorithm.

*Francesc J. Ferri, Jesús Albert. Dept. Informàtica i Electrònica. Universitat de València. Doctor Moliner, 50. 46100 Burjassot (València). SPAIN.

- Article rebut el maig de 1995.

- Acceptat el febrer de 1996.

The paper is organized as follows. In the following section, the algorithm is presented and the complexity parameters to be obtained are specified. Section 3 develops the analysis of the algorithm which leads to its average-case cost. A similar analysis to obtain the variance of the cost is presented in Section 4. An overview of possible laboratory exercises to check the theoretically obtained results is included in Section 5. Finally, the main conclusions of the work presented are outlined in Section 6.

2. SELECTION SORT

Selection sort constitutes a simple sorting algorithm that is very interesting because of its relation to other sorting algorithms and because it is a very good example of average-case analysis. Several different implementations of this algorithm exist and they can be found in any book about programming (Sedgewick 1988; Wirth 1976; Knuth 1973b). It should be noted that the particular version used here is not the optimal one, but is the most interesting from the point of view of studying its behavior in the average case.

The method can be outlined as follows. The algorithm starts by selecting the minimum of all the elements and allocates this element to the first position. Then, it does the same with the minimum of the remaining $n-1$ elements, allocating it to the second position. At any iteration i, the first $i-1$ elements are already ordered and the algorithm looks for the minimum among the remaining $n-i+1$ elements and allocates it to the ith position. A possible implementation of this algorithm is shown in Figure 1.

The running time of Selection Sort is clearly quadratic. The number of comparisons made by the algorithm is given in any case by

$$T_c(n) = \sum_{i=1}^{n-1} n-i = \frac{n}{2}(n-1) = O(n^2).$$

What is really interesting to analyze in this algorithm from our point of view is the number of data moves in the array A. As there are assignments involving array elements inside and outside the internal `for` loop, we can take as a critical instruction (Brassard and Bradley 1987) the move, $x \leftarrow A[j]$, provided that the average number of times it is executed is more than lineal. (The number of moves outside this loop is, in any case, $3(n-1)$.)

```
Selection Sort
Input/Output:
        A:  vector[1..n] of Type
Begin
    for i ← 1 to n − 1 do
        x ← A[i]
        k ← i
        for j ← i + 1 to n do
            if A[j] < x then
                x ← A[j]
                k ← j
            endif
        endfor
        A[k] ← A[i]
        A[i] ← x
    endfor
End
```

Figure 1

Pseudocode of the Selection Sort algorithm. *Type* can be any ordered data type. The algorithm gives the corresponding elements in increasing order.

Let $T(n)$ be the number of times the critical instruction is executed. This is a random variable taking values for all possible configuration of the input vector. In the best and worst case its value equals 0 and $T_c(n)$, respectively, and the problem we want to solve consists of obtaining the expected value of $T(n)$. It is equally interesting to obtain an expression for the variance of $T(n)$.

To use a notation similar to most books on algorithms we will write $T(n)$, $\mathbf{T}(n)$, and $\mathrm{Var}(T(n))$ for random variables, expected value, and variance respectively. The same apply for the other cost functions introduced.

3. AVERAGE-CASE COST

Let us suppose that there are no repeated elements and that every possible initial ordering of the n elements has the same probability. We can then assume without loss of generality that there are $n!$ possible cases that correspond to the permutations of the first n integers.

At each iteration, i, the algorithm searches for the minimum among the $n-i+1$ elements on the right hand side of the location i (including i).

Let us cal $R(m)$ the number of moves needed to select the minimum among m elements. Let $\mathbf{R}(m)$ be its expected value. By the particular form of the algorithm, it is possible to write the following relation

$$\mathbf{T}(n) = \sum_{i=1}^{n-1} \mathbf{R}(n-i+1), \tag{1}$$

where expectations are taken over the $n!$ and $(n-i+1)!$, $i = 1,2,\ldots,n-1$ possible cases, respectively.

Let us rename the elements $i, i+1, i+2, \ldots, n$ as $1,2,\ldots,m$ where $m = n-i+1$. As there are no repeated elements among them, we can assume that the possible cases for this subproblem correspond to the $m!$ permutations of the first m integers.

The average number of times the instruction is executed at a given iteration will be

$$\mathbf{R}(m) = \frac{S(m)}{m!}, \tag{2}$$

where $S(m)$ is the total number of times it is executed for all $m!$ possible permutations of m elements.

It is not easy to obtain an expression for $S(m)$ by summing the contributions of each single permutation. However, it is much easier to obtain a convenient relation between $S(m)$ and $S(m-1)$ which will lead to the desired expression.

Let us consider only the $(m-1)!$ cases in which the minimum is located in the last position. The number of times the instruction is executed for all these instances will be

$$S(m-1) + (m-1)!,$$

because the algorithm behaves as if there were only $m-1$ non-repeated values and, moreover, the instruction will be executed once for each one of the $(m-1)!$ cases because the minimum is effectively found in the last position.

For the $(m-1)(m-1)!$ remaining cases, it is obvious that the minimum has been selected while searching the first $m-1$ elements and then the corresponding number of times is only $S(m-1)$ for each group of $(m-1)!$ permutations, which gives

$$(m-1)S(m-1).$$

The total number of times for all possible cases will be the sum of these two expressions:

$$S(m) = m\ S(m-1) + (m-1)!.$$

This constitutes a recurrence relation for $S(m)$ which ends with $S(1) = 0$. If we write this recurrence for $\mathbf{R}(m)$ instead of $S(m)$ using Equation (2), we obtain

$$\mathbf{R}(m) = \begin{cases} 0 & \text{if } m \leq 1 \\ \mathbf{R}(m-1) + \frac{1}{m} & \text{if not} \end{cases},$$

which is a trivial recurrence, whose solution is reduced to the summation

$$\mathbf{R}(m) = \sum_{k=2}^{m} \frac{1}{k} = H_m - 1,$$

where H_m is the m-th harmonic number.

The average number of times the instruction is executed after n iterations can now be obtained from Equation (1).

$$\mathbf{T}(n) = \sum_{i=1}^{n-1} (H_{n-i+1} - 1) = \sum_{i=2}^{n} (H_i - 1).$$

Manipulating this summation (Graham *et al.* 1989) we have

$$\begin{aligned} \mathbf{T}(n) &= \sum_{i=2}^{n} \sum_{j=2}^{i} \frac{1}{k} = \\ &= (\frac{1}{2}) + (\frac{1}{2} + \frac{1}{3}) + \ldots + (\frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}) = \\ &= \sum_{k=2}^{n} (n-k+1)\frac{1}{k} = (n+1)\sum_{k=2}^{n} \frac{1}{k} - \sum_{k=2}^{n} 1 = \\ &= (n+1)(H_n - 1) - n + 1. \end{aligned}$$

Finally, we obtain

$$\mathbf{T}(n) = (n+1)H_n - 2n = O(n \lg n). \tag{3}$$

From this expression we observe that the asymptotic behavior of $\mathbf{T}(n)$ is not modified even if we had taken into account the data moves outside the internal `for` loop. Since the number of data moves in this case is $T(n) + 3(n-1)$, its expectation would be $(n+1)H_n + n - 3$.

4. VARIANCE OF $T(n)$

The approach followed in this section to obtain the variance of $T(n)$ is basically the same presented in the previous section. We first calculate the variance of the cost at each iteration i, $\mathrm{Var}(R(n-i+1))$. From the definition of variance of $R(m)$, we can write

$$\mathrm{Var}(R(m)) = \frac{1}{m!}\sum_{i=1}^{m!} r_i^2 - \mathbf{R}(m)^2.$$

where r_i, $i = 1,\ldots,m!$ are the costs for each possible permutation of m elements.

The problem will be now to establish a recurrence relation for the quantity $U(m)$ defined as

$$U(m) = \sum_{i=1}^{m!} r_i^2,$$

In a way similar to the one in the previous section, there are $(m-1)(m-1)!$ permutations which contribute to $U(m)$ with

$$(m-1)\sum_{i=1}^{(m-1)!} r_i^2 = (m-1)U(m-1).$$

On the other hand, the $(m-1)!$ permutations with the minimum at the end, will now sum to

$$\sum_{i=1}^{(m-1)!}(r_i+1)^2 = U(m-1) + 2\sum_{i=1}^{(m-1)!} r_i + (m-1)!,$$

because each one of them will produce one more iteration to get the minimum element.

Summing up the two last expressions we obtain

$$U(m) = mU(m-1) + 2(m-1)!\mathbf{R}(m-1) + (m-1)!.$$

It is possible to simplify this recurrence relation in the following manner. Dividing by $m!$ and subtracting $\mathbf{R}(m)^2$ at both sides of the equation, we have

$$\frac{U(m)}{m!} - \mathbf{R}(m)^2 = \frac{U(m-1)}{(m-1)!} + \frac{2}{m}\mathbf{R}(m-1) + \frac{1}{m} - \mathbf{R}(m)^2.$$

Given that

$$\mathbf{R}(m)^2 = (\mathbf{R}(m-1) + \frac{1}{m})^2 = \mathbf{R}(m-1)^2 + \frac{2}{m}\mathbf{R}(m-1) + \frac{1}{m^2},$$

and using the fact that

$$\mathrm{Var}(T(n)) = \frac{U(m)}{m!} - \mathbf{R}(m)^2,$$

we obtain the following recurrence relation

$$\mathrm{Var}(R(m)) = \begin{cases} 0 & \text{if } m \leq 0 \\ \mathrm{Var}(R(m-1)) + \frac{1}{m} - \frac{1}{m^2} & \text{if not} \end{cases},$$

where we have taken into account that $\mathrm{Var}(R(0)) = 0$. This constitutes a trivial recurrence whose solution is

$$\mathrm{Var}(R(m)) = \sum_{k=1}^{m} \frac{1}{k} - \sum_{k=1}^{m} \frac{1}{k^2} = H_m - H_m^{(2)},$$

where $H_m^{(2)}$ is the m-harmonic number of order 2 (Knuth 1973a).

To approximately calculate $\mathrm{Var}(T(n))$ we can assume that the number of iterations of the internal loop, $R(n-i+1)$, of the Selection Sort algorithm is independent at each iteration of the external loop. With this simplification we have

$$\mathrm{Var}(T(n)) \approx \sum_{i=1}^{n-1} \mathrm{Var}(R(n-i+1)) = \sum_{i=2}^{n} \mathrm{Var}(R(i)) = \sum_{i=2}^{n} H_i - \sum_{i=2}^{n} H_i^{(2)},$$

from which we obtain an approximate expression for the variance of $T(n)$

$$\mathrm{Var}(T(n)) = (n+2)H_n - (n+1)H_n^{(2)} - n = O(n \lg n).$$

5. EMPIRICAL TESTS

The material presented in the previous sections is aimed at being used for teaching purposes either in an elementary course or in a more advanced one for graduate students. Specially in the first case, attention must be paid to the practical implications of the obtained results.

In particular, it is quite interesting to write a simple program and empirically measure the cost of the algorithm by counting the number of times the corresponding instruction is executed. This has to be repeated for a number of different random permutations to obtain the corresponding expected value.

As discussed in Section 2, there is no point in measuring the cost in seconds for this algorithm because it is dominated by the number of comparisons which is $O(n^2)$.

To check the accuracy of the previous analysis, especially for the last approximation of the variance, programs for random and exhaustive (recursive) permutation generation can be used to feed the algorithm and collect the results. Then, the expected value of $T(n)$ can be empirically obtained (approximately for large n). The standard deviation can be also computed from the different runs of the algorithm for each n.

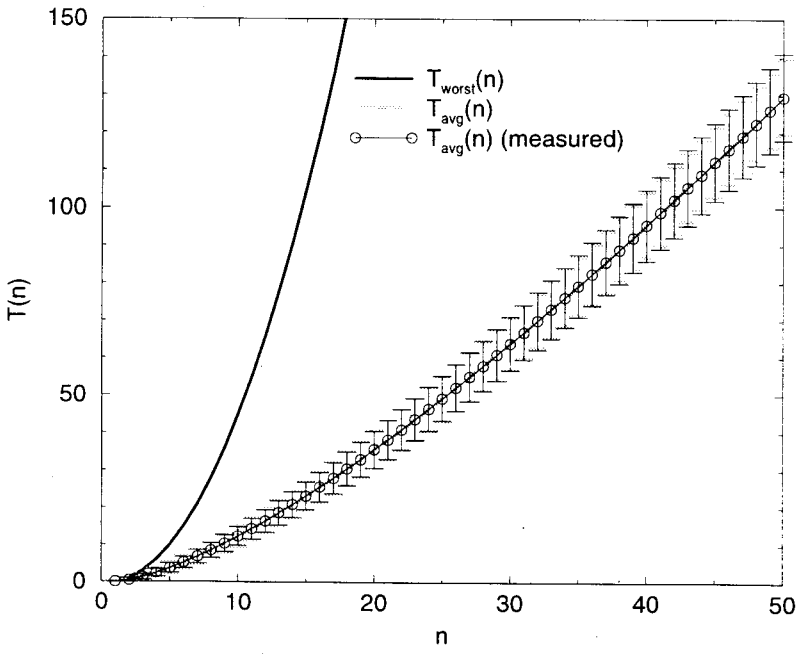

Figure 2

Average number of data moves in the Selection Sort algorithm with the standard deviation accumulated both empirically and computed from the obtained expressions.

Figure 2 shows the expressions obtained in previous sections for the expected value and variance (standard deviation) of $T(n)$ along with the average number of iterations and the standard deviation (which is shown accumulated to $\mathbf{T}(n)$ in both cases) obtained from the corresponding program. The worst-case cost is also shown for illustration purposes.

6. CONCLUDING REMARKS

A convenient exercise for average-case algorithm analysis has been presented with both theoretical and practical aspects. The main interest of the paper lies in the way this particular example is presented which is significantly different from the usual way this analysis is done. The algorithm is analyzed in depth while keeping the mathematical requirements to a minimum by using recurrence relations that can be easily solved in this case.

The importance of empirically checking the accuracy of the theoretical analysis is emphasized in order to achieve a better understanding of the behavior of the algorithm.

Some other simple algorithms (as, for example, binary search or palindrome string checking) can be analyzed in a similar way with a similar level of difficulty. This could constitute the basis for a practically oriented course or seminar on Basic Algorithm Analysis.

REFERENCES

[1] **Brassard, G.** and **Bradley** (1987). *Algoritmique: Conception et Analyse.* Masson.

[2] **Graham, R., D. Knuth** and **O. Patashnik** (1989). *Concrete Mathematics: A Foundation for Computer Science.* Addison-Wesley.

[3] **Knuth, D.E.** (2nd ed. 1973a). *The Art of Computer Programming I: Fundamental Algorithms.* Addisson-Wesley.

[4] **Knuth, D.E.** (1973b). *The Art of Computer Programming III: Sorting and Searching.* Addisson-Wesley.

[5] **Knuth, D.E.** (1980). *El Arte de Programar Ordenadores*, **1,3**. Reverté.

[6] **Sedgewick, R.** (1988). *Algorithms.* Addison-Wesley.

[7] **Wirth, N.** (1976). *Algorithms + Data Structures = Programs.* Prentice Hall.

[8] **Wirth, N.** (1980). *Algoritmos + Estructuras de Datos = Programas.* Ediciones del Castillo.