CAB301 Algorithms and Complexity

Sara Alraddadi
N8779333

May 28, 2017

1 Description of the Algorithms

In this section we are analysing the algorithms *MinDistance* and *MinDistance2* given in the Assignment 2.

Both algorithms receive an Array of numbers as a parameter and both return the distance of the two closest elements in the array.

When we analyse in detail, we observe that the output result is being saved in the variable called *dmin*. At the beginning, it is initialised with the neutral value of the problem. As it is trying to find a minimum value, it needs to be initialised with the highest possible value (Infinite in the theoretical case). After this, the algorithms are slightly different.

The first algorithm given iterates with a double loop over the complete array, both loops going from the first element to the last one. Let's call the number of the position *index*. These loops allow every element to be compared with all the other elements. It is important to note that when both indices are the same, we are pointing to the same element.

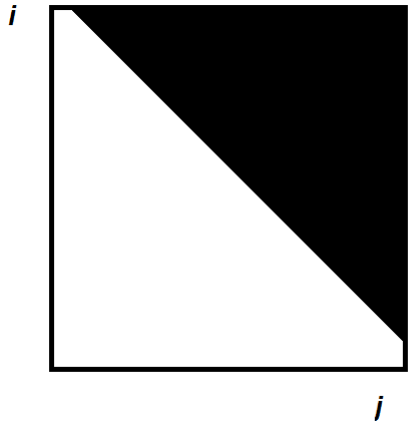
Inside both loops, we have a conditional *if*. The first part of this condition limits the comparison to only the elements in different positions (different indexes). The second checks that it is the lowest distance found at the moment. If the condition is *true*, then we save the value in *dmin*. At the end, the final saved value is returned.

Data: A[0..$n-1$]
Result: Minimum distance between two of its elements
Initialize dmin with the highest possible value;
for $i \leftarrow 0$ **to** $n-1$ **do**
  for $j \leftarrow 0$ **to** $n-1$ **do**
    if *Indexes are different and the distance is the lowest found* **then**
      Save it in dmin
  end
end
Result: dmin

Algorithm 1: MinDistance

In the second algorithm we observe that now the first loop goes from the first element to the penultimate element (instead of the last). And the inner loop goes now from a position next to the position of the outer loop until reaching the last element. In contrast with the algorithm 1, we are now comparing every element only once. We can represent the double loop as a square, being each side an index of the iteration (Figure 1). Positions inside the square can be then determined by the i and j values. This algorithm is comparing only the colour black area. Note that the other half would be comparing exactly the same elements.

Figure 1: Representation of the double loop as a geometry figure



Inside the inner loop, we have a conditional similar to the *MinDistance* algorithm. If the distance is the lowest until now, we save it in *dmin*. This time we are using a temporary variable, so we don't access to the array twice.

Data: A[0..$n-1$]
Result: Minimum distance between two of its elements
Initialize dmin with the highest possible value;
for $i \leftarrow 0$ **to** $n-2$ **do**
  for $j \leftarrow i+1$ **to** $n-1$ **do**
    if *Indexes are different and the distance is the lowest found* **then**
      Save it in dmin
  end
end
Result: dmin

Algorithm 2: MinDistance2

2 Theoretical Analysis of the Algorithms

2.1 Choice of the basic operation

Since we are trying to optimise the number of access to the elements of the Array A, we chose every expression that accesses the array as a basic operation. It is important to note that in the algorithm *MinDistance*, the conditional expression

has two Booleans. When the first one fails, the second is not evaluated. That means that it will not count as an operation.

2.2 Choice of Problem Size

We chose a problem size the amount of elements that the Array A contains. It is the only variable that makes a significant change in the amount of operations performed by both algorithms.

2.3 Average-Case Efficiency

The first algorithm has an outer loop that goes from 0 to $n-1$, and an inner loop from 0 to $n-1$. It can be seen that the amount of operations involved here are n^2. Inside the inner loop, the conditional will access to the array twice **only** when elements are different. It can be represented with the following expression: $n^2 - n$. That means that we have $n^2 - n$ basic operations without going inside the conditional.

Let's analyse now how many time we go inside the conditional. Let's say without losing generality that the closest element are in the position k, with $1 <= k <= kmax$. Let's calculate $kmax$ now.

If we take a look at the square given, we notice that if we find a minimal distance it can be **only** inside the black area. That means that counting the amount of elements in the black area tells us how many possibilities we have ($kmax$). We intuitively see that this number tends to zero while the input size grows, because while more and more numbers are compared, it is more probable to get a very small *dmin*. That means it will go inside the conditional fewer times when the input size grows. So, let's ignore this expression for the average case, because it doesn't add any meaningful cost to the experiment.

It gives a result that the average cost is:

$$avg \approx (n^2 - n) \tag{1}$$

The second algorithm accesses the outer for-loop $n-1$ times. The inner for-loop performs $n-1$ operations the first time, $n-2$ the second time, and so until reaching 1 operation at the end. We can calculated a closed expression for this, $(n-1)n/2$. It gives a result that we have the following amount of basic operations:

$$avg \approx \frac{(n-1)n}{2} \tag{2}$$

We clearly see that both are quadratic, however the first one is around twice as slow than the second one as intuition suggests. The second is only doing half of the comparisons. It can be also be inferred from the graphical point of view of the problem.

3 Methodology, Tools and Techniques

3.1 Programming Environment

· **Programming Language** We are using C++ Language to implement the algorithms using the GNU MinGW32 Compiler.

· **Hardware and Software** The experiments were performed on a Core i7 2600 processor, running Microsoft Windows 8.1 and CodeBlock 16.01 as IDE. Other details are going to be omitted because they are irrelevant to the experiment.

· **Report** It was created using the online compiler available in ShareLatex in LaTeX.

· **Graphics** They were created using the online computational knowledge engine Wolfram Alpha.

3.2 Implementations of Algorithms

Both implementations are pretty straightforward. We only needed to determinate what kind of number use. We decided to use *integers* for the experiment, since it doesn't change the amount of basic operations. In both implementations we added the operation counter passed as parameter, taking care of the conditional (See the commends in code below).

```
int minDistance(int a[], int n, long long & operations){
    operations = 0L;
    int dmin = LONG_MAX;  // Inf
    for ( int i = 0 ; i < n ; i++ ){
        for ( int j = 0 ; j < n ; j++ ){
            if ( i != j ){
                // We separated the condition to count the
                // access to the array ONLY when it really happens.
                operations ++;
                if ( abs(a[i] - a[j]) < dmin ){
                    operations ++;
                    dmin = abs(a[i] - a[j]);
                }
            }
        }
    }

    return dmin;
}
```

Very similar to the previous, we have here the second implementation.

```
int minDistance2(int a[], int n, long long & operations){
    operations = 0L;
    int dmin = LONG_MAX;  // Inf
    for ( int i = 0 ; i < n - 1 ; i++ ){
        for ( int j = i + 1 ; j < n ; j++ ){
            operations ++; // counter
            int temp = abs(a[i] - a[j]);
            if ( temp < dmin ){
```

```
                dmin = temp;
            }
        }
    }

    return dmin;
}
```

Since the algorithms are very simple, they were tested with a couple of handmade cases, called from the main function. We changed the numbers in the array and compared the results of the application with the handmade calculation. Here is the code used for testing the functions:

```
int a [] = {4,7,34,10,12,342,64,42,24,656};
int n = 10;
int operations = 0;
int result = minDistance2(a, n, operations);
cout << result << " " << operations << endl;
```

3.3 Generating Test Data and Running the Experiments

To generate the test data, we are using the random number generator to create the array while we change the size of the input set. We decided to use the following sizes: 5, 20, 50, 100, 300, 500, 1000, 5000, 10000 and 20000. It will create a reasonably clear tendency, that compared with the theoretical analysis, allows the estimation for any input size. We are saving and execution time and the amount of basic operations. Every test was run three times to have more stable execution times.

Here we have the testing code used:

```
srand((unsigned)time(0));
int a [100000];
int sizes [] = {5, 20, 50, 100, 300, 500, 1000, 5000, 10000,
    20000};
int sizesn = 10;

for (int i = 0 ; i < sizesn ; i++ ){
    for ( int j = 0 ; j < sizes[i] ; j++ ){
        a[j] = (rand()%1000000);
    }
    long long operations = 0;
    long long operations2 = 0;

    int time1start = 0;
    int time1end = 0;
    int delta1 = 0;
    int time2start = 0;
    int time2end = 0;
    int delta2 = 0;

    for ( int j = 0 ; j < 100 ; j++ ){
        time1start = clock();
        minDistance(a, sizes[i], operations);
        time1end = clock();
```

```
            time2start = clock();
25          minDistance2(a, sizes[i], operations2);
            time2end = clock();
27          delta1 += (time1end - time1start);
            delta2 += (time2end - time2start);
29      }


31
        cout << "Size: " << sizes[i];
33      cout << ", Operations 1: " << operations;
        cout << ", Operations 2: " << operations2;
35      cout << " Time 1: " << (delta1 / 100.0);
        cout << " Time 2: " << (delta2 / 100.0) << endl;
37  }
```

We are sending to both algorithms the same array, so we are certain that the comparison is fair.

3.4 Basic Operations

As already mentioned in the assignment, we are improving the algorithm *MinDistance* to reduce the amount of **expressions that access elements of Array A**. We used this principle to choose the basic operation. In the first algorithm, we have two expressions that access the array. One in the conditional and the other in the next line. That is why we decided to use it as our basic operation.

As it can be seen in the source code, the operation counter is the variable called *operations*. We took special care in the first algorithm with the conditional evaluation. Since the array is only accessed when the indices are different.

For the first algorithm, we count one basic operation in the line number 9, because it has two array accesses in the line number 10. Very similar with the line number 11 and 12.

For the second algorithm, we only count the access in the line 6.

3.5 Time measures

We used the *clock* function that the *ctime* library contains. As it can be seen in the source code in the Generating Data section, we are saving the time when the minDistance starts, and then we save it again when it finishes. We are running all the experiments 100 times, and we are averaging the times to have useful results. After that, we print the results to standard output. Exactly the same happens with *minDistance*.

Table 1: Experimental Basic Operations in the MinDistance Algorithm

Input Size	Operations
5	24
20	388
50	2456
100	9911
300	89714
500	249509
1000	999011
5000	24995009
10000	99990010
20000	399980011

4 Experimental Results

4.1 Functional Testing

To confirm that the application is running as expected, we ran several small cases with the debugger step by step.

4.2 Number of Basic Operations

From the theoretical asymptotic analysis, we can estimated that both experiment were going to have a quadratic behaviour when the input set size is change. It was confirmed with the empirical comparison.

Let's analyse first the number of basic operations in the algorithm *MinDistance* (Table 1). We are using ten different input sizes to compare the results. Using the Computational Knowledge Engine of Wolfram Alpha to do the quadratic regression, we get the following formula (Figure 2). It was very similar to the theoretical prediction

Figure 2: Least Squares fit for the Experimental Basic Operations of MinDistance

Least-squares best fit:

$$1.\,x^2 - 0.999827\,x + 8.84351$$

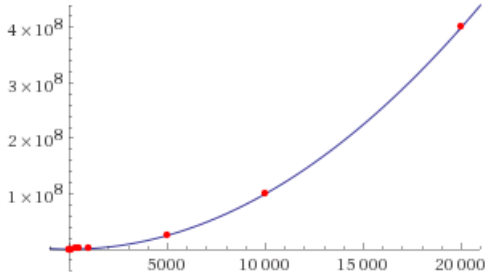
After plotting it in Wolfram Alpha, we have as result of the Figure 3 [1].

Analyzing now the number of basic operations in the algorithm *MinDistance2* we have the results showed in the Table 2. [2]. According to the theoretical prediction, the amount of basic operations are around half of the algorithm

[1] To replicate this results, the following expression has to be used: quadratic fit {5,24},{20,388},{50,2456},{100,9911},{300,89714},{500,249509},{1000,999011},{5000, 24995009},{10000, 99990010},{20000, 399980011}

[2] Remember that both algorithms had exactly the same input array

Figure 3: Square Fit of the Experimental Basic Operations of MinDistance



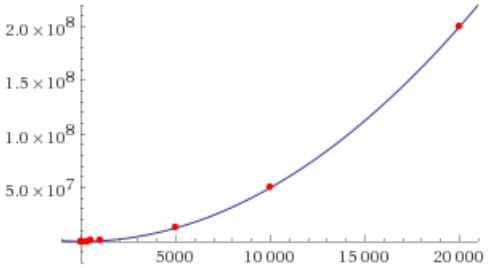
MinDistance2. Using Wolfram Alpha, we get the following closed formula for the best Square Fit (Figure 4). It is very close to the theoretical prediction. And exploring the plot, we have the Figure 5.

Figure 4: Least Squares of the Experimental Basic Operations of MinDistance

Least-squares best fit:

$$0.5\,x^2 - 0.5\,x - 6.89125 \times 10^{-8}$$

Figure 5: Square Fit of the Experimental Basic Operations of MinDistance2



4.3 Execution Times

As we are studying a very simple algorithm in detail, we decided to compile it using the optimisation flag -O3. It gives special speed optimisation that can be interesting for our study [3]. All the time measures are in milliseconds.

Amazingly, the speed different was huge between both algorithms. Our conclusion is that the compiler optimisation created that difference. If we take a close look in the table 3, second column, we see that it has a quadratic behaviour. It can be confirmed in the Figure 6 with the Least Square formula and its plot in the Figure 7.

[3]For more details, please visit https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

Table 2: Experimental Basic Operations in the MinDistance Algorithm

Input Size	Operations
5	10
20	190
50	1225
100	4950
300	44850
500	124750
1000	499500
5000	12497500
10000	49995000
20000	199990000

Table 3: Experimental Basic Operations in the MinDistance Algorithm

Input Size	MinDistance	MinDistance2
5	0	0
20	0	0
50	0	0
100	0.01	0
300	0.12	0
500	0.34	0
1000	1.28	0
5000	31.69	0.1
10000	126.15	0.02
20000	500.73	0.03

Figure 6: Comparison of both functions

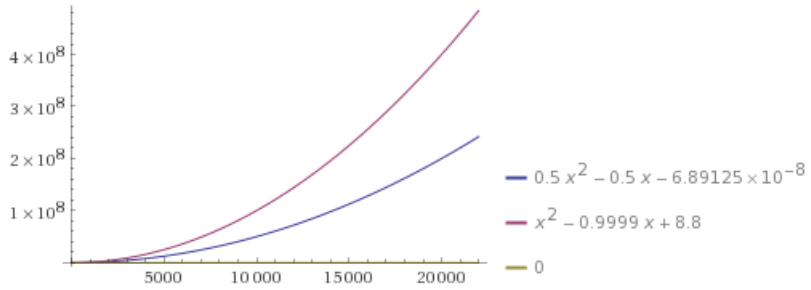


Figure 7: Least Square for the time taken by the MinDistance algorithm

Least-squares best fit:

$$1.24365 \times 10^{-6}\, x^2 + 0.000167228\, x - 0.0460911$$

However, the second algorithm took constant time for all the input cases considered. We were curious about it, and we changed the input size up to 20.000.000, and it took only 20 milliseconds. Change done for this was to increase one of the sizes showed in the code before [4].

5 Conclusions

After doing this empirical comparison and theoretical analysis, we realised that both algorithms belong to the same efficiency class, which is quadratic. They both offer similar efficiency after changing the input size, however the change offered by the improvement in the algorithm *MinDistance2* gave a large performance boost. It reduced the time and amount of steps performed to half.

We also experimented with the optimisation flags, which showed to have a large performance boost to the applications, with an apparent reduction of efficiency. It was inferred due the tests performed in the Execution Times section. However, more studies are needed to confirm that.

[4] Code showed in the subsection Generating Test Data

Figure 8: Quadratic fit for the time taken by the MinDistance algorithm