Queensland University Of Technology

Empirical Comparison of Two Algorithms

Daniel Huffer n9493671
Mitchell Bourne n9456040
May 25, 2017

1 Summary

This report details the theoretical and experimental analysis of two different algorithms designed to perform the same function. The function of which is to determine the smallest distance between any two elements within an array of integers. Using resulting data obtained from algorithm analysis techniques, potential differences in efficiency are discussed.

Contents

List of Tables

2 Description of the Algorithms

Both algorithms perform the same function, taking in an array of numbers as an input argument and returning the minimum distance. Where, minimum distance is defined as the the numeric difference between the two closest elements within the array. Despite having the same objective, there are discrepancies within the logic making each algorithm unique. The first of the two algorithms, MinDistance can be seen below in Figure 2.1.

2.1 MinDistance

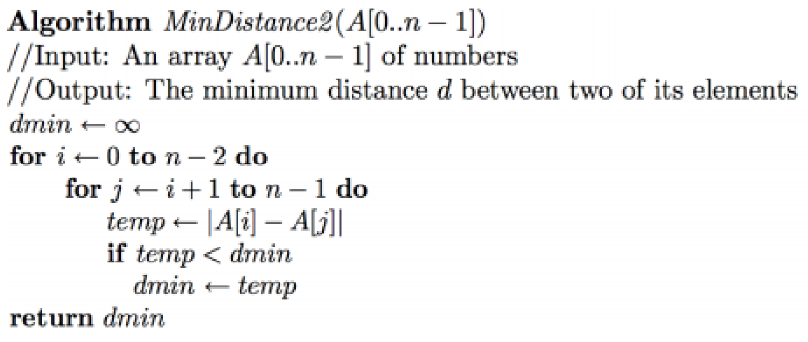
```
**Algorithm** $MinDistance(A[0..n-1])$
//Input: Array $A[0..n-1]$ of numbers
//Output: Minimum distance between two of its elements
$dmin \leftarrow \infty$
**for** $i \leftarrow 0$ **to** $n-1$ **do**
    **for** $j \leftarrow 0$ **to** $n-1$ **do**
        **if** $i \neq j$ **and** $|A[i] - A[j]| < dmin$
            $dmin \leftarrow |A[i] - A[j]|$
**return** $dmin$
```

Figure 2.1: Pseudocode for Algorithm One

In the pseudocode MinDistance takes a single input argument, an array of numbers and contains one local variable $dmin$ which is initialised to the value infinity. The algorithm then enters a nested loop set-up where the outer loop variable "i" iterates for the entire length of the array. Similarly, the inner loop variable "j" also iterates the entire length of the array, where within the inner loop the Boolean expression "(if i != j and |A[i] - A[j]| < dmin)" is evaluated each iteration. Assuming the array size is greater than one, this expression will always initially hold true on the second iteration of the inner loop as $dmin$ is set to infinity and "i ≠ j". Upon evaluating true, $dmin$ is reassigned to the value of |A[i] - A[j]| and the inner loop proceeds to continue checking for a smaller minimum distance. Once the inner loop exits the outer loop will iterate to the next element and start the inner loops search again for a smaller distance, this process will repeat itself until the outer loop has iterated the entire length of the array.

2.2 MinDistance2

The second of the two algorithms, MinDistance2 can be seen below in Figure 2.2.



```
**Algorithm** $MinDistance2(A[0..n-1])$
//Input: An array $A[0..n-1]$ of numbers
//Output: The minimum distance $d$ between two of its elements
$dmin \leftarrow \infty$
**for** $i \leftarrow 0$ **to** $n-2$ **do**
    **for** $j \leftarrow i+1$ **to** $n-1$ **do**
        $temp \leftarrow |A[i] - A[j]|$
        **if** $temp < dmin$
            $dmin \leftarrow temp$
**return** $dmin$
```

Figure 2.2: Pseudocode for Algorithm Two

MinDistance2 has the same input and output of its related algorithm MinDistance and also contains just a singular local variable $dmin$ initialised to infinity. The core functionality of the algorithm occurs within the nested loop set up, the outer loop starts at the first element and iterates to one element before the end of the array (n-2). The inner loop iterates from the current outer loop index plus one (i+1) to the very last element of the array (n-1). Therefore as the outer loops variable iterator increases, the inner loops variable iterator j starts to perform less iterations. Within the inner loop a temporary variable $temp$ is assigned the absolute value of A[i] - A[j] each iteration. If at any point the temporary variables value is less than the local variable $dmins$ value then $dmin$ is reassigned to the value of the temporary variable $temp$. This process continues until the outer loops variable iterator reaches the end of the array (n - 2).

3 Comparison of Algorithms

3.1 Basic Operation

The basic operation of an algorithm is defined as the operation which is most frequently executed resulting in a majority of the algorithm's execution time (Algorithm Analysis, 2017). For the two given algorithms, in order to effectively compare the efficiency; a common bias for comparison must be determined. In order to find this, a basic operation in which is applicable to both algorithms must be selected. MinDistance2 aims to increase efficiency by reducing the number of times the array is indexed by using a temporary variable to store the resulting operation of "|A[I] - A[j]|". MinDistance2 also cuts back on indexing the array by avoiding already calculated differences, e.g once "|A[I] - A[j]|" is performed "|A[j] - A[i]|" will not be. These optimisations were the basis of the efficiency increases of MinDistance2. In saying this, the operation is also the most frequently executed in both algorithms, resulting in an expected execution time difference for the optimized algorithm. Therefore, this was determined to be the basic operation of both algorithms.

3.2 Problem Size

In order to accurately test the efficiency of each algorithm, multiple array input sizes were considered. For testing basic operations it was decided that there would be 20 data points ranging from 0 to 2000. This range of data allows for a nice visualization of the two algorithms behavior as input size increases. For execution times the problem size was increased by a scale of 1000. This was due to the fact that if the problem size is too small the program may execute too rapidly resulting in data that doesn't accurately reflect the algorithms rate of growth. Also worth mentioning is that at a scale of 1000 in respect to the basic operation problem size, the execution run time for both algorithms can be tested thoroughly without execution occurring too quickly or taking extremely long amounts of time.

3.3 Theoretical Efficiency

This section covers the predicted theoretical rate of growth for each of the two algorithms.

3.3.1 MinDistance

For MinDistance the average case efficiency will vary slightly as one instance of the basic operation occurs inside the conditional statement "if i $\neq$ j and |A[i] - A[j]| < dmin". On randomly generated data sets this statement can hold true a varying number of times resulting in the inability to calculate an exact mathematical average case efficiency. However, the average case class efficiency can be determined as quadratic with a constant multiplier, this is due to the nested for loops. It should also be noted that the instance of the basic operation in the boolean expression will not be evaluated when "i == j". The Mathematical function and asymptotic notation for the algorithm can be seen below:

$$C_{avg_{(n)}} = ((\sum_{i=0}^{n-1} 1)(\sum_{j=0}^{n-1} 1)) + (x - n)$$

$$= n^2 + x - n$$
$$\in \theta n^2$$

3.3.2 MinDistance2

Unlike MinDistance, the basic operation in MinDistance2 is not dependant on any conditional statements and is simply executed each iteration of the inner for loop. Therefore a complete mathematical equation can represent the basic operation count for the algorithm. As with MinDistance this algorithms average case efficiency class also falls into that of a quadratic with a constant multiplier. The Mathematical function and asymptotic notation for the algorithm can be seen below:

$$C_{avg_{(n)}} = (\sum_{i=0}^{n-2} 1)((\sum_{j=0}^{n-1} 1)/2)$$
$$= 1/2(n-1)n$$
$$= \frac{n^2}{2} - \frac{n}{2}$$
$$\in \theta n^2$$

In the equation above, the first summation shows the outer loop iterating the length of the array minus one. The second summation shows the inner loop iterating the length of the array, however the end result is divided by two. This is because of the way MinDistance2 works where the inner loop variable iterator starts at the outer loops variable iterator plus one, which results in a linear decrease of loop iterations for the inner loop.

4 Tools, Methodology and Techniques

This section details the computing environment and programs used for all experiments conducted.

4.1 Tools

4.1.1 Programs

- CLion
  - A powerful cross platform C and C++ IDE
- Excel
  - A Microsoft electronic spreadsheet program.
- MacTeX
  - A type setting environment which is a redistribution of TeX Live

4.1.2 Environment

- All experiments were conducted on the following machine:
  - Model Name: MacBook Pro
  - Model Identifier: MacBookPro13,1
  - Operating System: macOS Sierra (version 10.12)
  - Processor Name: Intel Core i5
  - Processor Speed: 2 GHz
  - Number of Processors: 1
  - Total Number of Cores: 2
  - L2 Cache (per Core): 256 KB
  - L3 Cache: 4 MB
  - Memory: 8 GB

4.2 Methodology and Techniques of Test Experiment

To understand the comparative efficiency of the two algorithms and how they compare to the derived theoretical analysis, a basic operation count experiment is conducted. The experiment is designed to help visualise the discrepancies in the amount of "work" performed by each algorithm. To do this, random data sets of varying sizes ranging from 100-2000 are produced and tested on each of the two algorithms.

5 Code Implementation

For the implementation of the pseudocode into C++ the input and output data was defined as type integer. To replicate assigning the local variable $dmin$ to the value of infinity the numeric_limits class template was used, where a call to max is made to return the largest possible value for type integer. Additionally "n" was added as an input parameter as opposed to having it declared in the global scope. Aside from these slight discrepancies the rest of the implementation for each of the two algorithms strictly followed the pseudocode. The coded implementations can be seen below in Listing 1: MinDistance Implementation and Listing 2: MinDistance2 Implementation.

Listing 1: MinDistance Implementation

```
static int MinDistance(int A[], int n){
int dmin = std::numeric_limits<int>::max();
for(int i = 0; i < n; i++){
        for(int j = 0; j < n; j++){
            if((i != j) && ((abs(A[i]-A[j])) < dmin)){
               dmin = abs(A[i] - A[j]);
               }
        }
}
    return dmin;
}
```

Listing 2: MinDistance2 Implementation

```
static int MinDistance2(int A[], int n){
int dmin = std::numeric_limits<int>::max();
for(int i = 0; i < n-1; i++){
        for(int j = i + 1f; j < n; j++){
                int temp = (abs(A[i]-A[j]));
            if(temp < dmin){
                dmin = temp;
            }
               }
        }
    return dmin;
}
```

5.1 TESTING OF CODE IMPLEMENTATION

In order to test that the implementation of the algorithms into C++ was completed correctly, various sample test cases were generated. For test cases 1 - 5, basic algorithm functionality was tested, these ranged from sequential data sets to mixed ranging from small distances to large. Test cases 6 and 7 test that the absolute value functionality was working as intended. Test cases 8 and 9 test arrays in which have all equal data sets, checking 0 gap functionality. With this, the output data collected in shown in the table below. These tests check the absolute value of the algorithm, by applying negative values to the algorithm.

Table 5.1: Test Cases for Algorithm Implementation

Input Array	Expected Output	MinDistance Output	MinDistance2 Output
1, 2, 3, 4	1	1	1
1, 2, 4, 7	1	1	1
5, 8, 3, 8	0	0	0
3, 8, 25, 46	5	5	5
4, 46, 74643, 35346, 3453	42	42	42
-1, -2, 4, 6, 90	1	1	1
-53, 9, 123, 5, 3	2	2	2
0, 0, 0, 0, 0	0	0	0
4, 4, 4, 4, 4	0	0	0

As can be seen, the expected outputs of the test data above is equivalent; the code implementation of the algorithms is working as expected.

6 Basic Operation Counts

This section details the implementation, findings and results of the basic operation experiment. To count the basic operations for each of the two algorithms, a local integer variable count was used. The count variable is initialized to zero outside of the for loops then incremented by one each iteration of the inner loop. Once the for loop statements have finished executing the count variable is then returned by the algorithm. The coded implementation can be seen in Appendix A - Basic Operations.

6.1 Basic Operation Experiment

Testing for the basic operation experiment involved testing array sizes ranging from 100 to 2000 in increments of 100, a total of 20 data points. For each varying test size the array was repopulated with pseudo random generated numbers ranging from 0 to the length of the current array test size. Once the random data set was generated, both of the algorithms were passed the data set and the resulting basic operation count for the current test size was then streamed to a text file on the local disk for plotting purposes. The results of the basic operation experiment can be seen below in Figure 6.1.

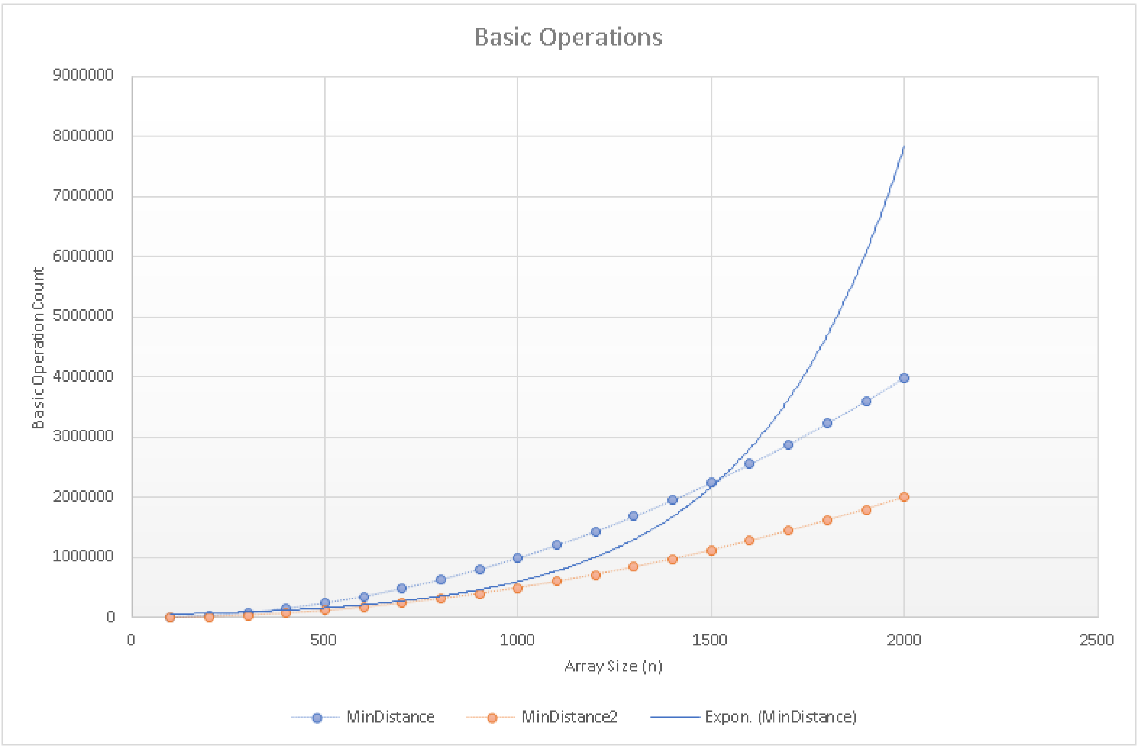


Figure 6.1: Basic operation count for varying array sizes

Looking at the results of the basic operation experiment in Figure 6.1 it can be seen with the assistance of an exponential trend line with respect to MinDistance that both of the algorithms are following the predicted θn^2 rate of growth. For the maximum array size of 2000 elements, MinDistance accumulated a total of 3,998,007 basic operations. Whereas, MinDistance2 accumulated a total of 1,999,000. Dividing the total count of basic operations for MinDistance by the total count for MinDistance2. It was found that theoretically, MinDistance has an execution time exactly two times slower for any given input size (n-1).

7 Execution Time Comparison

Testing execution times for the two given algorithms was conducted in a similar fashion to the basic operation experiment. This was done to generate results that provided a better comparison between experimental and theoretical results. Problem size was increased by a scale of 1000 and array sizes from 1000 to 20,000 in increments of 1000 were tested, resulting in a total of 20 data points. For each data point 50 execution time tests were completed, each test the array was repopulated with pseudo random generated numbers ranging from 0 to the length of the current array size been tested. The total execution time for each algorithm was averaged out and recorded for each data point. Results for each data point were recorded into text files on the local disk, where a plot was generated representing the data for both algorithms; this can be seen below in Figure 7.1. Code for the implementation of the execution times can be viewed in Appendix C - Execution Times.

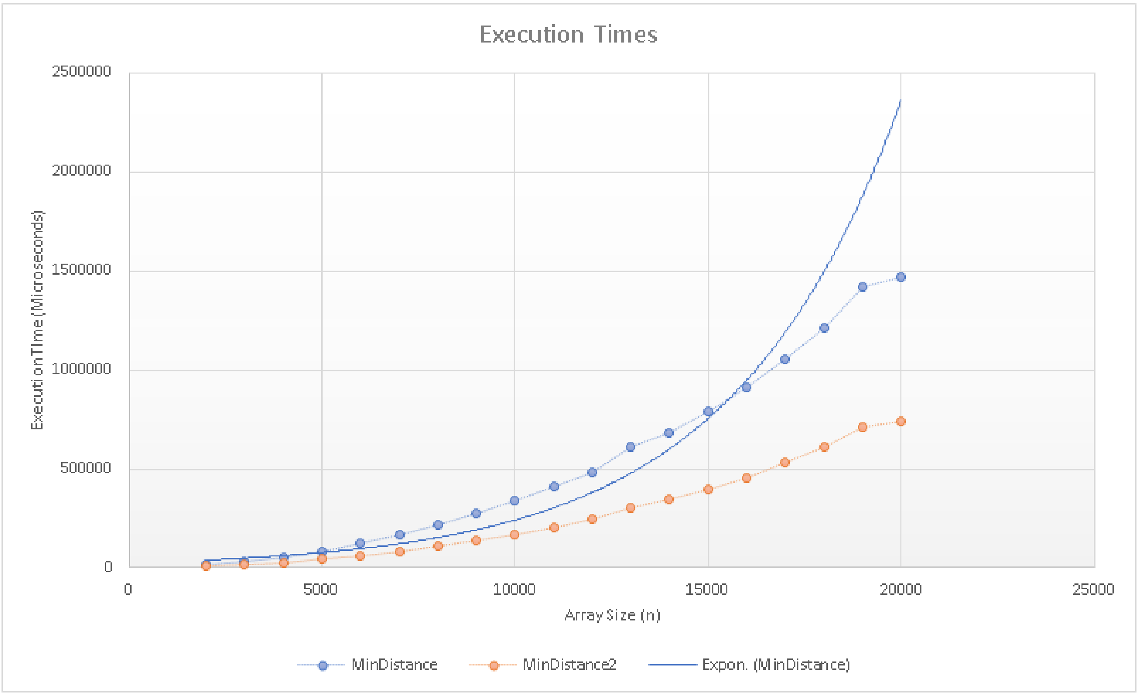


Figure 7.1: Execution Time Comparison

It can be seen in Figure 7.1 that the difference in efficiency between the two algorithms remains consistent with the difference depicted in the previous basic operation experiment. Looking at the execution time data for the maximum array size tested (20,000), the resulting times were 1.468883 seconds for MinDistance and 0.73885 seconds for MinDistance2. Finding the ratio by dividing MinDistance by MinDistance2 shows that MinDistance executed on average 1.988 times slower than that of MinDistance2. Referring back to the basic operation experiment data to compare the results, it can be seen that MinDistance theoretically should have run exactly two times slower than MinDistance2. The results of the execution time show that MinDistance ran 1.988 times slower, almost exactly as predicted. The nature and similarity of these results show that the choice of basic operation is correct and can very accurately model execution trends for the two algorithms. It also shows how effective optimizations in which reduce array access can be. In this case, these optimizations have reduced execution times of MinDistance2 by almost half.

8 REFERENCES

Algorithm Analysis. (2017). Web.cs.wpi.edu. Retrieved 25 May 2017, from http:// web.cs.wpi.edu/ mebal-azs/cs507/slides02/slides.html

APPENDIX A - BASIC OPERATIONS

Listing 3: Basic Operation Counts

```
 1  int MinDistance(int A[], int n){
 2  int operationCount = 0;
 3  int dmin = std::numeric_limits<int>::max();
 4  for(int i = 0; i < n; i++){
 5          for(int j = 0; j < n; j++){
 6                  operationCount++; // Basic operation count
 7              if((i != j) && ((abs(A[i]-A[j])) < dmin)){
 8                 dmin = abs(A[i] - A[j]);
 9                 }
10          }
11  }
12      return operationCount;
13  }
14
15  int MinDistance2(int A[], int n){
16  int operationCount = 0;
17  int dmin = std::numeric_limits<int>::max();
18  for(int i = 0; i < n-1; i++){
19          for(int j = i + 1; j < n; j++){
20              operationCount++; // Basic operation count
21              int temp = (abs(A[i]-A[j]));
22              if(temp < dmin){
23                  dmin = temp;
24              }
25                 }
26          }
27      return operationCount;
28  }
```

As explained in section 6, the implementation of the counter can be seen on lines 2, 6 and 12 for MinDistance and lines 16, 20, 27 for MinDistance2.

APPENDIX B - BASIC OPERATION EXPERIMENT

Listing 4: Basic Operation Experiment

```
1  // Constants
2  int ARRAY_CAPACITY = 2000;
3  int STEP_SIZE = 100;
4  int NUM_TESTS = 50;
5
6  // Function Prototypes
7  int MinDistance(int A[], int n);
8  int MinDistance2(int A[], int n);
9
10 int main()
11 {
12     std::ofstream min_distance("MinDistance.txt"); // Output file stream MinDistance
13     std::ofstream min_distance2("MinDistance2.txt"); // Output file stream MinDistance2
14     srand(time(NULL)); // Seed random
15     int A[ARRAY_CAPACITY]; // Init array
16     for(int i = STEP_SIZE; i <= ARRAY_CAPACITY; i+=STEP_SIZE){
17         long average = 0;
18         for(int p = 0; p < NUM_TESTS; p++ ){
19             for(int k = 0; k < i; k++){
20                 A[k] = rand() % i; // Random numbers as great as current array size
21             }
22             average += (MinDistance(A,i)-i); // Average total for each test size,
                   Subtract i to account for i!=j
23
24         }
25         min_distance << i << "," << average/NUM_TESTS << " \n";
26         min_distance2 << i << "," << MinDistance2(A,i) << " \n";
27     }
28     min_distance.close(); // Close Stream
29     min_distance2.close(); // Close Stream
30 }
```

As the basic operation situated inside the conditional statement of MinDistance occurs a variable amount of times, it was tested tested 50 times with new data each time for each and averaged out for each incrimenting array size. The implimentation of this can be seen on lines 18,19,20,21,22.

APPENDIX C - EXECUTION TIMES

Listing 5: Execution Times Code

```cpp
// Constants
int ARRAY_CAPACITY = 20000;
int STEP_SIZE = 1000;
int NUM_TESTS = 50;

int main()
{
    std::ofstream min_distance("MinDistance_execution.txt"); // Output file stream
    std::ofstream min_distance2("MinDistance2_execution.txt"); // Output file stream
    srand(time(NULL)); // Seed random
    int A[ARRAY_CAPACITY]; // Init array
    for(int i = STEP_SIZE; i <= ARRAY_CAPACITY; i+=STEP_SIZE){
        long avg_mindistance = 0, avg_mindistance2 = 0;
        for(int p = 0; p < NUM_TESTS; p++){
            for(int k = 0; k < i; k++){
                A[k] = rand() % i;
            }
            high_resolution_clock::time_point t1 = high_resolution_clock::now(); // Start
                timer
            MinDistance(A,i); // Run algorithm
            high_resolution_clock::time_point t2 = high_resolution_clock::now(); // End
                timer
            auto duration = duration_cast<microseconds>( t2 - t1 ).count();
            avg_mindistance += duration; // Add totals for md
            t1 = high_resolution_clock::now();
            MinDistance2(A,i); // Run algorithm
            t2 = high_resolution_clock::now();
            auto duration_2 = duration_cast<microseconds>( t2 - t1 ).count();
            avg_mindistance2 += duration_2; // Add totals for md2
        }
        min_distance << i << "," << avg_mindistance/NUM_TESTS << " \n"; // Average total
            and send to txt file
        min_distance2 << i << "," << avg_mindistance2/NUM_TESTS << " \n";
    }
    min_distance.close(); // Close Stream
    min_distance2.close(); // Close Stream
}
```