CAB301 Algorithms and Complexity Assignment 2

Empirical Comparison of Two Algorithms

# Description of the Algorithms:

In his textbook, ‘Introduction to The Design and Analysis of Algorithms’, A. Levitin proposes an algorithm *MinDistance* (see Figure 1). The idea of the algorithm is to find the minimum distance between any pair of two elements in a given array of numbers. Levitin’s proposed algorithm is given as an exercise for the reader, the task being to modify or write a new algorithm that has a better efficiency than his. This report will compare and contrast a second algorithm, *MinDistance2*, in terms of both time taken, and the number of basic operations performed.

### Minimum Distance 1:

Levitin’s algorithm takes in an array ***A*** of numbers, with length ***n***. The output of the algorithm is then the minimum distance between two of its elements. The output variable ***dmin*** is set (line b), to a value of infinity, so that no integer distance between two elements could possibly be smaller. The algorithm contains 2 nested ***for*** loops (line c and d), that run loop counters from ***0*** to ***n-1***, and are used to compare the distance between the current two loop values, and ***dmin***. Inside the second ***for*** loop is an ***if*** statement (line e), this checks that the compared elements are not the same, and whether the distance between the elements, ***A[i]*** and ***A[j]*** is less than the current ***dmin*** value. If both conditions are true, the new smallest distance is stored in the ***dmin*** value on line f. This algorithm contains no early exit statement and will always run through both loops entirely. Finally, line g contains the ***return*** statement in which the smallest value is returned.

### Minimum Distance 2:

The proposed solution to Levitin’s problem is *MinDistance2*. Much like the above, this algorithm also takes in an array ***A*** of numbers, with length ***n***. The output of the algorithm is then the minimum distance between two of its elements. Again, the integer variable ***dmin*** is set on line b. *MinDistance2* contains two nested ***for*** loops. The first ***for*** loop runs loop variable ***i*** from ***0*** to ***n-1***. The nested ***for*** loop however, runs the loop variable ***j*** from the current value of ***i*** to ***n-1***. In this algorithm, the distance between the two loop values (***A[i]*** - ***A[j]***) is stored in a temporary variable, ***temp*** (line e). In this case the ***if*** statement (line f) simply checks the stored ***temp*** value against ***dmin***. If it is lower the ***temp*** value is stored in ***dmin*** on line g. Finally, the ***dmin*** value is returned in line h.

# Theoretical Analysis of the Algorithm

This section describes both algorithm’s predicted average-case efficiency, with respect to their basic operation.

### 2.1) Basic Operation of the Algoithms

Analysis of an algorithm can be separated into two separate categories, space and time analysis. This choice determines the basic operation. In this case, we are interested in time analysis, which makes the basic operation one which has the greatest influence on the total running time of the algorithm (Levitin, 2012, p. 44). In this case, the basic operation can then be said to be any operation which calculates the distance between two array elements. These operations, seen as ***A[i] - A[j]*** in Figure 1, exist on lines e and f. In line e, the distance calculation exists as part of an ***if*** statement. Due to the fact that the calculation is not stored in a variable, it must then also be calculated again when being assigned to the minimum distance variable ***dmin*** online f. Although the calculation on line f is not performed in every single loop it was still determined as a basic operation due to the significant difference in clock cycles between the minimum distance operation, and all other operations in the algorithm.

Much the same as Levitin’s proposed algorithm, the basic operation for the given potential solution, *MinDIstance2,* is any operation which calculates the absolute value between any two elements in the given array. This operation can be seen on line e, where the calculated value is stored in the ***temp*** value for further use in the algorithm.

### 2.2) Choice of Problem Size

The choice of problem size for the algorithms is quite simple. In both cases the problem size is the size of the given array, ***n.*** For testing purposes, ***n*** was varied from 100 through to 10,000. Increments of 100 were used as test cases and provided sufficient data to show the growth of both algorithms. ***MAY ADD MORE HERE. TUTOR SAID MINIMAL INFO NEEDED.***

### 2.3) Average-Case Efficiency for MinDistance

For Levitin’s algorithm, both cases of the basic operation exist inside the nested ***for*** loops. Both loops will run through ***n*** times, as neither have an early exit condition. However, as we are interested in the basic operation, we can see that due to the loop counter check, ***i≠j***, the basic operation will then occur ***n-1*** times for the inner loop. With large enough data pools, the data in the arrays can be said to have a standard distribution. This then means that the basic operation on line f will occur ***n/2*** times. From this we can show the average case efficiency is as follows:

This gives us an efficiency class of , as the growth of the quadratic is much greater than the value, it can be overlooked.

Generally, it is important to discuss both the worst-case and best-case efficiency when describing the average-case. However, due to the fact that the algorithm has no early exit condition and that the basic operation will always run the same number of times in this case, all three efficiency cases will be the same. The proposed efficiency solution has a dramatically better performance. Again, the basic operation exists inside the two nested ***for*** loops. There are two changes which effect the average case efficiency, these being the choice of loop counter in the inner ***for*** loop, and the variable being used to hold the distance calculation. The outer for loop in this case, runs ***n-1*** times. The inner loop however, will run ***n-2*** iterations on the first iteration, ***n-3*** iterations for the second, and so on. This occurs until final iteration where it will run ***n-(n-1)*** times. Inside these loops the basic operation is performed once on line e, when storing the distance in the ***temp*** variable. Thus, we have an average case efficiency as follows:

The efficiency class for *MinDistance2* is still however, it can be seen that the average case efficiency is much lower.

# 3 Methodology, Tools and Techniques

## 3.1 Program Environment

The experiments were executed on an i5-2500k desktop with 8gb of ram and a 7200rpm HDD running windows 7 service pack 1. C++’s pseudorandom number generator [1] found in the stdlib library was used to produce test data and the clock library [2] was used to measure execution times. Windows 7 was chosen over newer operating systems because it has much less background processes that run so therefore reduced random interference between tests. All other processes were also closed during testing to further reduce interference between tests.

Graphs were then produced using Microsoft Excel. This was done by programming the test applications to write project results to a ‘.csv’ file which was then imported to Excel.

## 3.2 Program Implementations

The two separate algorithms were implemented using methods in separate static classes to allow for clean code layout instead of lumping everything into one file. The methods that represent the algorithms can be found in appendix A and B respectively. The translation from pseudocode to C++ was rather straightforward so when comparing them the layout should be very consistent with the pseudocode.

## 3.3 Generating Test Data and Running the Experiments

To test the correctness of both the algorithm’s implementations a small test program was written for each which ran the implementation with given variables and compared the results to the expected input. The data supplied by this process was then printed as string stating if it succeeded or failed and gave context (Appendix C).

To count the number of occurrences of the basic operation preformed in each algorithm, modified versions of the algorithms were created so that they increment a counter each time a basic operation is performed and return the count (Appendix D). A function was then created to generate datasets of increasing size and test them on both modified versions of the algorithms making sure to use the same dataset on both algorithms. This is to ensure that the comparisons that can be made between them is accurate (Appendix E).

To measure the time efficiency between both algorithms a function was generated (Appendix F) that uses the original versions of both algorithms to count the time taken to preform increasing sizes of sets. Since the algorithms in small set sizes executed very quickly, the tests were repeated 10 times with different sets of the same size for each repetition to produce a total execution time. The total time was then divided by the number of repetitions to get an average execution time per set size. Again, like basic operation count, care was taken to use the same sets of data for both algorithms to produce meaningful and accurate comparisons.

# 4 Experimental Results

## 4.1 Functional Testing

To test the correctness of both implementations the approach described in Appendix C was used. An example of this is testing an array with the items 2,7,16,30 using either algorithm 1 or 2, expecting a minimum distance of 5 and getting the following output:

Input: {2,7,16,30}

Expected Output: 5 Observed Output: 5 (TEST PASSED)

This confirms that when given an ordered list of varying gaps it correctly identifies the correct distance.

In Another test with the inputs -1, -5, -7, -10, -15, when expecting the output 2, using either of the algorithms the following output was produced:

Input: {-1,-5,-7,-10,-15}

Expected Output: 2 Observed Output: 2 (TEST PASSED)

This shows that when given negative numbers both algorithms can still correctly identify the shortest distance.

On the lower extreme inputting an array of just 1s such as 1, 1, 1, with the expected result of 0, either one of the algorithms outputted the following data:

Input: {1,1,1}

Expected Output: 0 Observed Output: 0 (TEST PASSED)

Confirming that when given an array with items of zero distance the items are not ignored and instead correctly returns zero.

Another extreme case is using an unsorted list such as 5, -5, 0, -2, 10, 15, with the expected result being 2. When testing this on either algorithm the following result is outputted:

Input: {5,-5,0,-2,10,15}

Expected Output: 2 Observed Output: 2 (TEST PASSED)

This confirms that the algorithm works even when given the extreme case of unordered input. The tests shown here and other tests that were done on the algorithms demonstrate that each algorithm outputs the desired outputs.

## 4.2 Average-Case Number of Basic Operations

TODO

## 4.3 Average Case Execution Time

TODO

# References

1. Cplusplus.com (n.d) rand. Retrieved May 13, 2017, from <http://www.cplusplus.com/reference/cstdlib/rand/>
2. Cplusplus.com (n.d) time. Retrieved May 13, 2017, from <http://www.cplusplus.com/reference/ctime/clock/>
3. Levitin, A. (2012). Introduction to The Design and Analysis of Algorithms (3rd ed.). Pearson Education Inc.

## Figures

1. //Input: Array of numbers  
   //Output: Minimum distance between two of its elements

Figure 1: A. Levitin’s given algorithm – MinDistance

1. //Input: Array of numbers  
   //Output: Minimum distance *d* between two of its elements

Figure 2: Proposed solution of greater efficiency – MinDsitance2

# Appendix A: Implementation of Algorithm 1

The following appendix presents the C++ implementation that is used in this report for the pseudocode that is described in Figure 1. It consists of two for loops (lines 3 and 4) which match the two in Figure 1 (Conditions C and D). these are used to compare each item against each item in the supplied array. The application then uses this comparison setup to generate the difference between each item to see if the difference between them is less than the current minimum (line 5). This corresponds to the check in Figure 1 (Condition E). When true the current value is made the new minimum (Line 6) which compares to the value set in Figure 1 (Condition F). The value is then returned at the end of the loops (line 10) which is the same as Figure 1 (Condition G).

****

# Appendix B: Implementation of Algorithm 2

The following appendix presents the C++ implementation that is used in this report for the pseudocode that is described in Figure 2. It consists of two for loops (lines 3 and 4) which match the two in Figure 2 (Conditions C and D). these are used to compare each item against each item against the items after it in the current array. The application then uses this comparison setup to generate the difference between each item to see if the difference between them is less than the current minimum (line 6). This corresponds to the check in Figure 2 (Condition F). When true the current value is made the new minimum (Line 7) which compares to the value set in Figure 2 (Condition G). The value is then returned at the end of the loops (line 11) which is the same as Figure 2 (Condition H).

# 

# Appendix C: Code for Functional Testing

To test the correctness of the implementations the following function was written to run the implementation with given variables and compare the results to the expected output. Near-identical versions were created for the other algorithm.

Firstly, the output is initialised and filled with the contents of the input array (lines 2-9). The computed distance is then created and inserted to the output stream (lines 10 and 11). The computed output is then compared to the expected output and a pass or fail flag is appended to the end of the stream (lines 12-16).

****

By varying the input of the function to test for various cases, it was confirmed that the two implementations worked correctly and returned the expected results every time.

# Appendix D: Code Modifications for Counting Basic Operations

To measure the number of basic operations (Section 2.1) performed by both algorithms they had to be modified to count the number of times the basic operation was executed. This was done by adding a counter that incremented that was then returned instead of the distance.

The following version of the first algorithm (Appendix A) was created to count the number of basic operations. Underlined code is new or modified.

****

Firstly, the counter variable *operations* is initialised (line 2). The counter is then incremented each time the basic operation of getting the absolute variable of the comparator minus the compared is encountered (lines 7-8 and 9-10), This required some heavy modifying by splitting the if statement (line 6) so that a counter can be added for when the second half is executed. The resulting count is then returned instead of the distance (line 13).

Similar modifications were made to the second algorithm which follows.

****

the counter variable *operations* is initialised (line 2). The counter is then incremented each time the basic operation of getting the absolute variable of the comparator minus the compared is encountered (lines 6-7). The resulting count is then returned instead of the distance (line 13).

# Appendix E: Code for Counting the Number of Basic Operations

The following code in this appendix presents the function that was written to count the number of basic operations for both algorithms across growing array lengths. The following function was used to generate Figure **TODO**.

****

The code first creates a file stream and sets up the randomiser (lines 2-4) then for each increment of *testJump* from zero to *setSize* does the following. Firstly, it sets up a test array with a length that is the current increment of the loop and places random numbers into each position (lines 6-10). With this array, the count for both algorithms are then calculated by running the modified versions of the algorithms and outputted to a csv file (lines 12-14).

# Appendix F: Code to Measure Execution Times

The following appendix presents the function that was written to measure the execution time for both algorithms for sets of incrementing sizes. The following code counts the time taken to run a set number of each algorithm for incrementing sizes of arrays to generate Figure **TODO**.

****

The code first creates a file stream and sets up the randomiser (lines 2-4) then for each increment of *testJump* from zero to *setSize* does the following. Firstly, it sets up a test array with a length that is the number of repeats and a depth of the current increment of the loop and then places random numbers into each position (lines 6-12). With this data, the time taken to execute all repeats is done for both algorithms using a time taken at the start and a time taken at the end separately for both (lines 13-22). The starting and finish times are then subtracted from each other respectively and divided by the number of repeats which is then written to a .csv file (line 24).