CAB301 Algorithms and Complexity Assignment 2

Empirical Comparison of Two Algorithms

# 1 Description of the Algorithms:

## 1.1 Minimum Distance 1

A. Levitin proposed an algorithm for finding the distance between the two closest elements in an array of numbers (see Figure 1).

# 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).

TODO Basic Operations and Execution Time

# 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. 2. Cplusplus.com (n.d) time. Retrieved May 13, 2017, from http://www.cplusplus.com/reference/ctime/clock/

## 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).