**Staffordshire University**

**Faculty of Computing, Engineering and Technology**

**Advanced Programming Language Concepts**

**The Traveling Salesman Problem with Implementation and Critical Review of Java and Haskell Solutions**

**By**

**Harry Clewlow (C012952A)**

**Contents**

1. Introduction Page 3
2. The Traveling Salesman Problem (TSP) Page 3
3. Programming Paradigms Function & Imperative Page 4
4. Evaluating Haskell Solution against Java Solution Page 4 - 8
5. Conclusion Page 8
6. Appendix A
   1. Java Class Diagrams Page 9
7. Appendix B
   1. Example Results Page 10 - 11
8. Appendix C
   1. Commented Code Listing Page 12 - 35
9. Appendix D
   1. References Page 36

**Introduction**

The aim of this assignment was to first produce two working artefacts, one will be implemented in Haskell and the other will be implemented in Java. These artefacts are aimed at solving the traveling salesman problem (TSP). Secondly critically evaluate and compare the strengths and weaknesses of each paradigm, if they are suitable for solving the assignment, code design elements, test files and fully commented source code.

**Traveling Salesman Problem**

The traveling salesman problem (TSP) focuses on finding the shortest, most efficient route possible when given a list of locations to visit and return back to the starting location this can be applied to any method of transport. To calculate an optimal route a set of data to represent each location is needed this can be several coordinates or the distance between each location. When traveling along a route the salesman cannot visit the same location twice apart returning to the start city after all the other locations are visited. [1]

Due to the nature of the TSP, there is currently no known algorithm which is both optimal and efficient when generating a route for the TSP; this makes it an NP-Hard problem meaning that the problem can be separated into smaller problems which are just as hard as the original problem if not harder. Even with the TSP first being defined in the 1800s, it can be applied to many modern day applications in its purest form logistics which is becoming ever more important with the expansion of online marketplaces such as Amazon that deliver millions of packages each day.

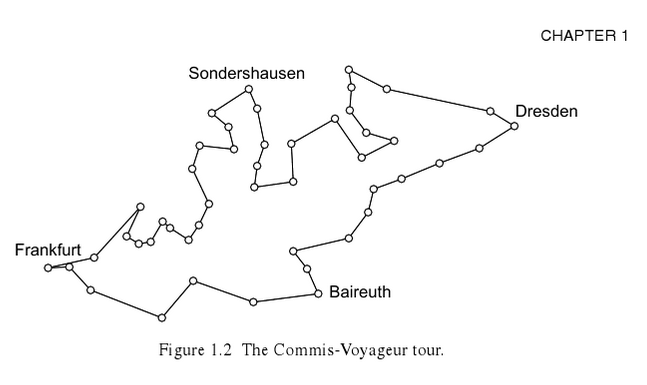


Figure Example of Travelling Salesman Problem [1]

There are several approaches when tackling the TSP each varying in complexity and each comprises either efficiency or producing an optimal solution, example methods include exhaustive, approximation and heuristics. In this report I will be focusing on three different algorithms they are a brute force algorithm, a nearest neighbour algorithm and a repetitive nearest neighbour algorithm.

A brute force algorithm (BFA) solution is inefficient due to having to calculate all the permutations possible (Hamilton circuits); in this case, a brute force algorithm would produce (10-1)! = 362,880 different circuits. Once all the circuits have been calculated along with their individual cost e.g. the total distance travelled when using each circuit, the route with the smallest cost circuit is the most optimal route to take. [2] The benefit is that the route generated will the most optimal because it has been compared against all other possible routes, this also means that is it very resource heavy and not practical in many situations where more than eleven different locations are used due to the calculations carried out growing disproportionately.

A greedy algorithm also called a nearest neighbour algorithm (NNA),produces a solution that is efficient, but the route it produces may not be optimal due to it generating a route by visiting the nearest city that hasn’t been visited before from its current locations, once all other cities have been visited it will return to the starting city. [2]The advantage to using the nearest neighbour algorithm is that it is more efficient than using a brute force algorithm, this means that is has compromised the most optimal route for speed because of this the optimal route isn’t always found and in special cases a route may not even if one exists.

An improvement on the nearest neighbour algorithm is called repetitive nearest neighbour algorithm (RNNA) where you can use any city as the starting location, then apply the nearest neighbour algorithm to get a route and weight; this is done for each of the different location with regards to the assignment it would be applied ten times each time at a different start city, once all the routes and weights have been calculated, the route with the best Hamilton circuit is chosen and transposed so that the starting location is traveling salesman’s home. [2] The advantage of using the repetitive nearest neighbour is that is more optimal than the standard nearest neighbour algorithm but has the same disadvantages as well.

**Programming Paradigms Function & Imperative**

The two different solutions are implemented in languages that follow different programming paradigms; one is implemented in the imperative paradigm which focuses on commands which affect a program's state where the order of the commands are also important in altering a program's state. The Imperative paradigm consists of the following characteristics; discipline, incremental and control structures, [3] examples languages are C, C, Ada, Perl and Java which I have chosen to implement one of my solutions in.

The other solution is implemented following a functional paradigm which is considered to be a cleaner and simpler paradigm because it focuses on mathematical function theory, where expressions are evaluated and the outputs used by other functions. The functional paradigm consists of the following characteristics; discipline, functions are like variables, computations are calculated by calling other functions and values are non-mutable mean they cannot be changed, [3] example languages are Curry, Idris and Haskell which the other solutions will be implemented in.

**Evaluating Haskell Solution against Java Solution**

To evaluate each solution, I will be comparing the development time, number of lines of code, language specific features, execution speed, resource usage, number of operations and processes used, whether the language is compiled or interpreted and the scalability of the solution when applied to larger problems from this a conclusion will be made on which solution is best suited to solving the traveling salesman problem and the advantages and disadvantages of using each different programming paradigm.

**Brute Force Algorithm**

When implementing the solution in both Haskell and Java, the development time was shorter in Java due to not having that same experience and being unfamiliar with the programming concepts and syntax involved with a functional language like Haskell. When it comes to the number of lines of code, Haskell has 49 lines and the Java code has 91 which is near double than the Haskell solutions, this can be explained by the fact that Haskell uses recursion for producing the permutations instead of for loops that are used in the Java code, also because the Haskell code does not have to initialise as many variables as the Java.

**Haskell**

Lines of Code: 49 lines

Development Time 2 days

Execution Time 5.54 seconds

CPU Usage 27%

**Java**

Lines of Code: 91 lines

Development Time 1 day

Execution Time 0.015 seconds

CPU Usage 19%

The difference in execution can be explained by the different approaches in creating the list of permutations, where the Haskell solutions generated by splitting all the elements in the list into a tree like structure that consists of each different variant and is verified by Java using 8% less CPU resources, but this might be due to my lack of experience with Haskell, which has led to the Java solution being more efficient.

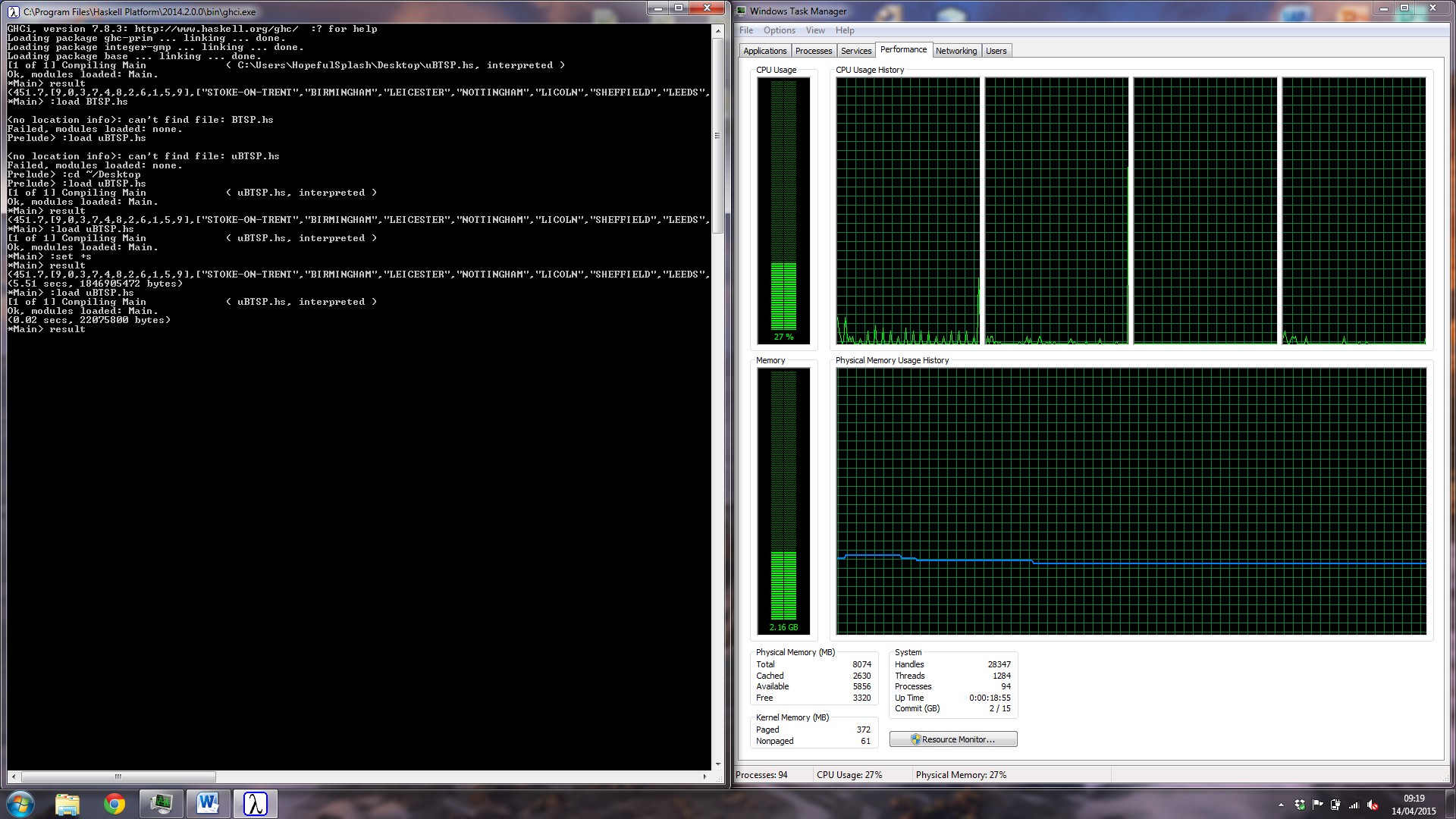
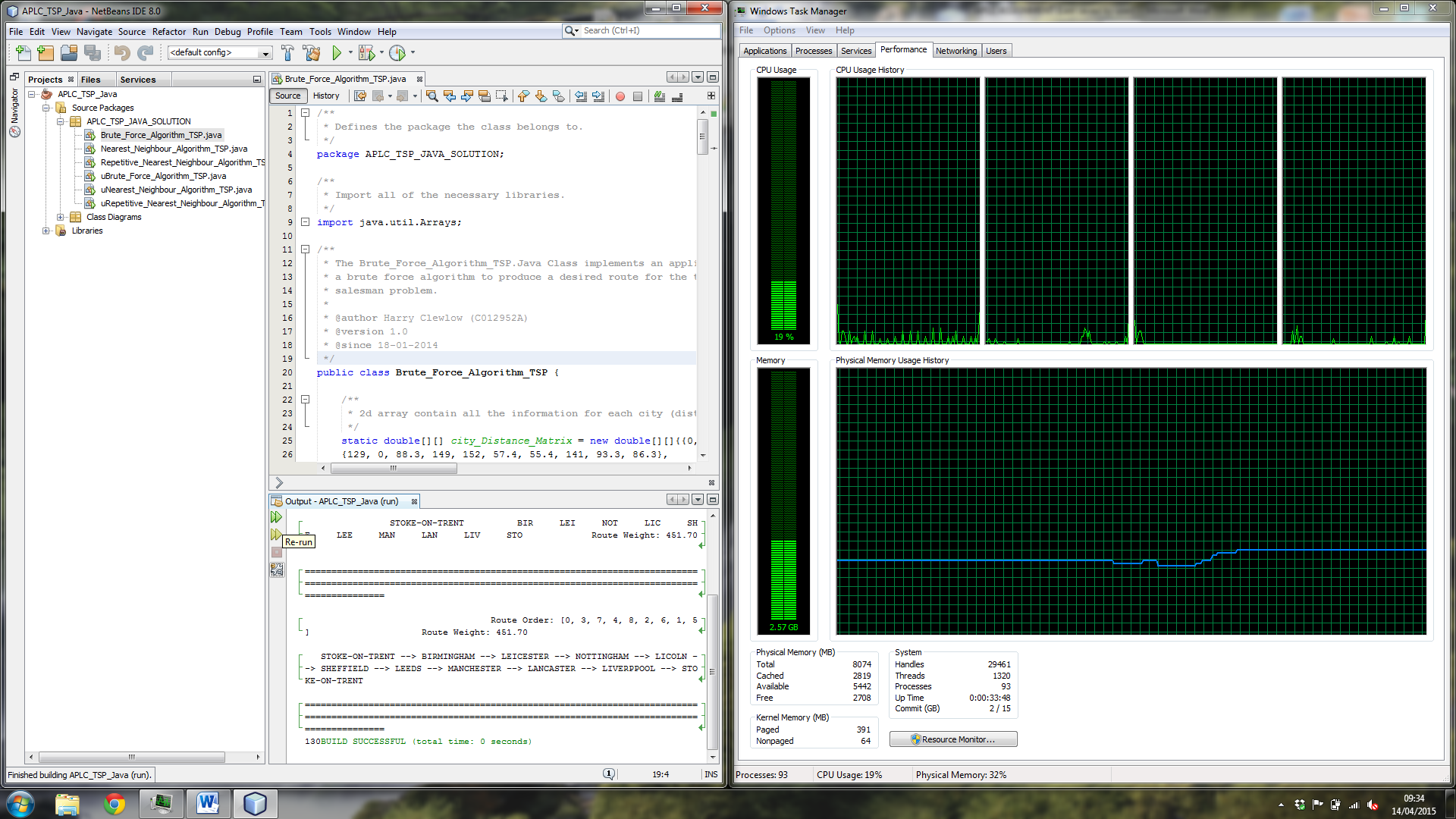
 

Figure Screenshots of CPU usage Haskell (Right) Java (Left)

Due to the brute force algorithm, calculating every possible different route you can take given a list of cities makes it not suitable for solving the traveling salesman problem, although using it for small routes consisting of ten or less cities computes the permutations relativity fast which is shown in the above solutions where the Haskell implementation was executed in 5.54 seconds, which had to generate 362880 permutations when dealing with nine other cities to visit but when you apply this to eleven or greater different cities you are having to compute 39,916,800 permutations which will be very demanding on resources and time consuming with means the algorithm has very poor scalability.

**Nearest Neighbour Algorithm**

After implementing the first solution I got to grips with Haskell and Java and had several functions that I could reuse for the nearest neighbour algorithm, but Haskell still took a longer to program due to having to learn advanced recursion and list comprehension. The nearest neighbour algorithm implementation in Haskell has nearly half the amount of lines of code, then the Java implementation which is the same of the previous solution.

**Haskell**

Lines of Code: 68 lines

Development Time 2 days

Execution Time 0.0003452 seconds

CPU Usage 1%

**Java**

Lines of Code: 102 lines

Development Time 1 day

Execution Time 0.0002238 seconds

CPU Usage 15%

The execution time is very similar in each solution this is before they are preforming the same process of searching all the possible routes from the current city to the next and selecting the smallest weighted option each time and then repeating the process until no cities are left. The CPU usage is almost a reverse of the previous solution, whereas the Haskell only uses 1%, whereas the Java implementation uses 15%, this may be explained by Java being executed using the Java Virtual Machine which will takes more resources than the Haskell that is interpreted and executed directly thus using less recourse.

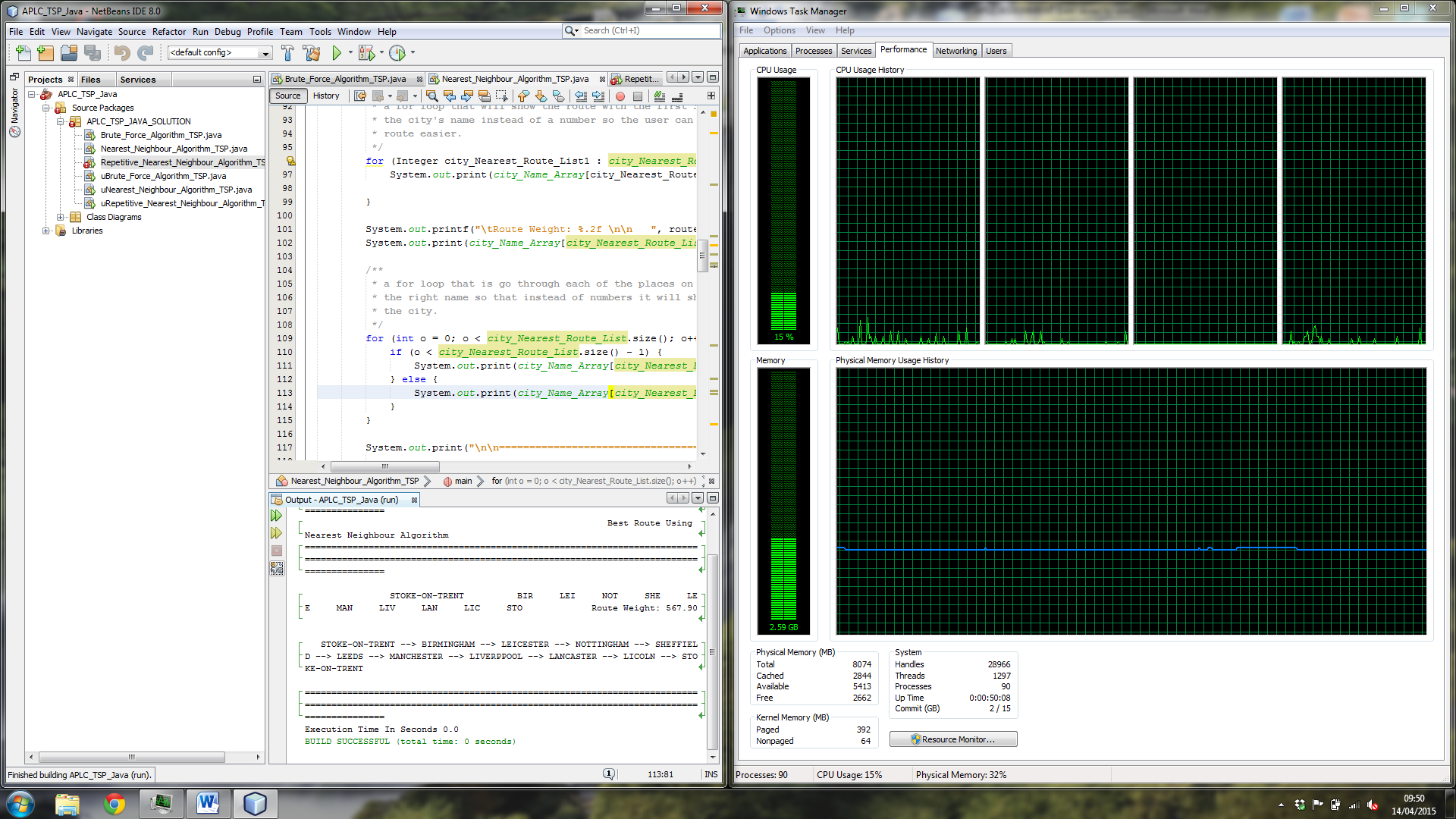
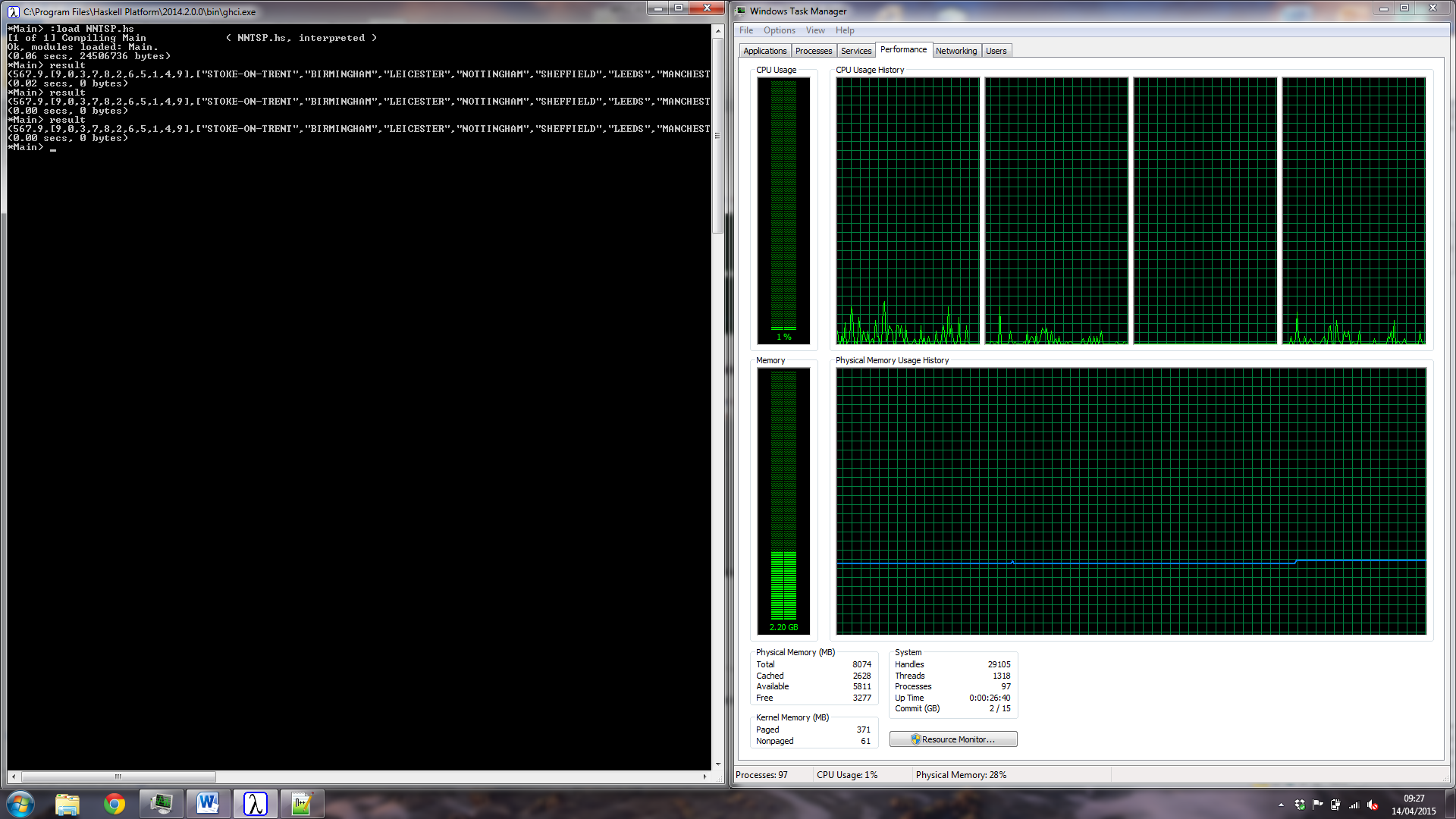


Figure 3 Screenshots of CPU usage Haskell (Right) Java (Left)

Unlike the brute force algorithm the number of possible routes the nearest neighbour algorithm has to calculate is significantly smaller, for ten cities you need to calculate 9+8+7+6+5+4+3+2+1 = 45 routes because of this the execution time is drastically smaller where each solution complete its executed in under a millisecond, this makes the solution very scalable due to the fact the number of possible routes you have to calculate is very small, where you find the closest city from the current location so when you apply it to a larger problem that has thirteen cities you only have 12+11+10+9+8+7+6+5+4+3+2+1 = 78, in comparison to the brute force solution, there are a lot less processes in producing a solving problem, but the result may not be as optimal meaning the route produced might have a higher cost than a brute force route because it comprising accuracy for speed.

**Repetitive Nearest Neighbour Algorithm**

The repetitive nearest neighbour algorithm is a slight improvement on the standard algorithm because of this, all of the previous code can be used and adding a couple of new functions to produce the new solution, once again the number of lines of code for the Haskell that of the Java solution where the Haskell has 82 lines and the Java has 161 lines due to the majority of the code being reused the development time for each solution was only one day.

**Haskell**

Lines of Code: 82 lines

Development Time 1 day

Execution Time 0.02783 seconds

CPU Usage 1%

**Java**

Lines of Code: 161 lines

Development Time 1 day

Execution Time 0.02726 seconds

CPU Usage 14%

The repetitive nearest neighbour algorithm is very similar to the nearest neighbour, but the process is repeated using each city as the starting point which means there are more possible routes to calculate because of this it takes more time than the standard algorithm, but this comprise results in a slight reduction in speed for an increase in accuracy where a more optimal route is found, the Haskell takes 0. 02783 seconds and the Java takes 0.02726 seconds so when using this solution for ten cities a small reduction in speed results in a more accurate route.

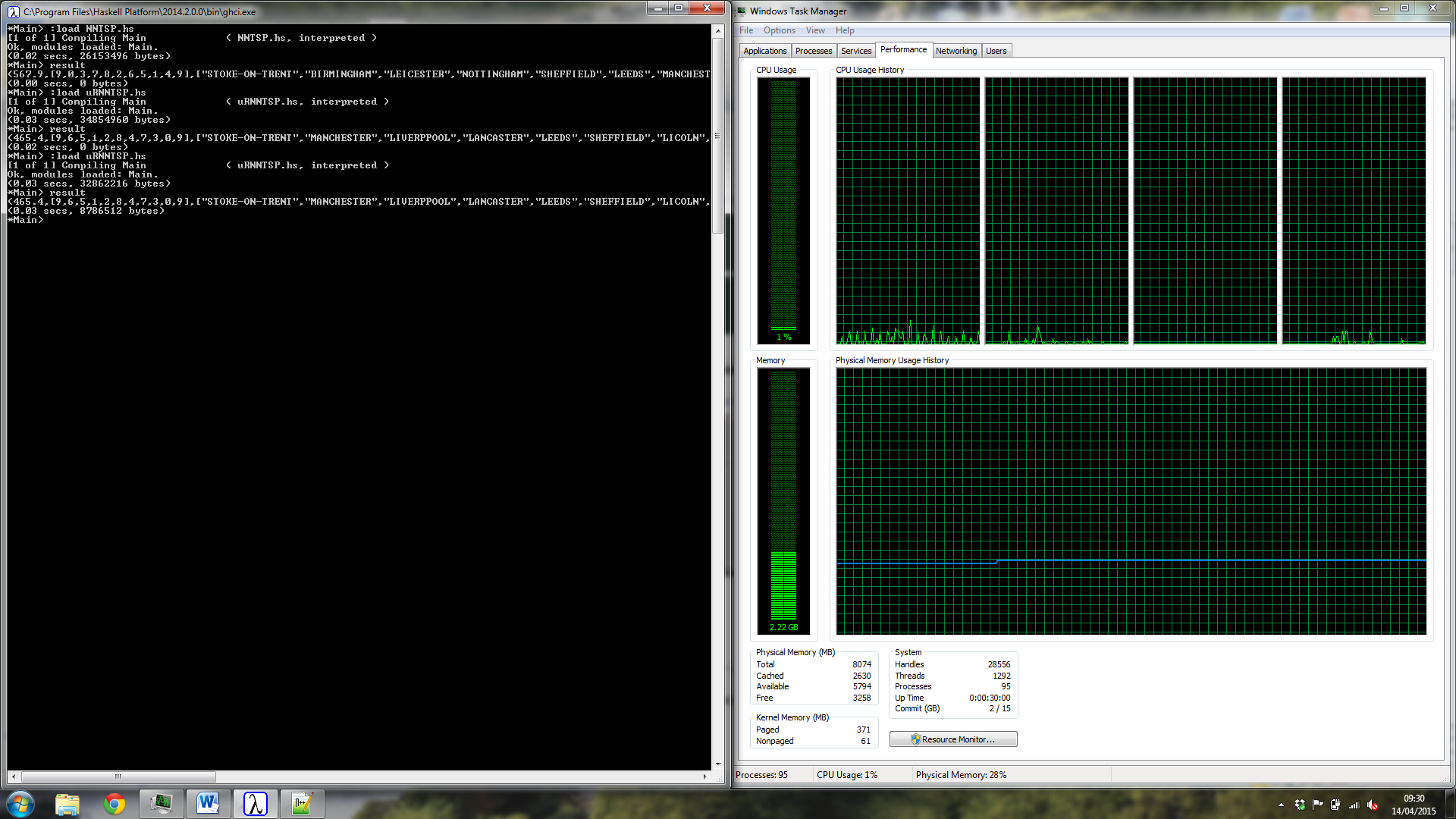
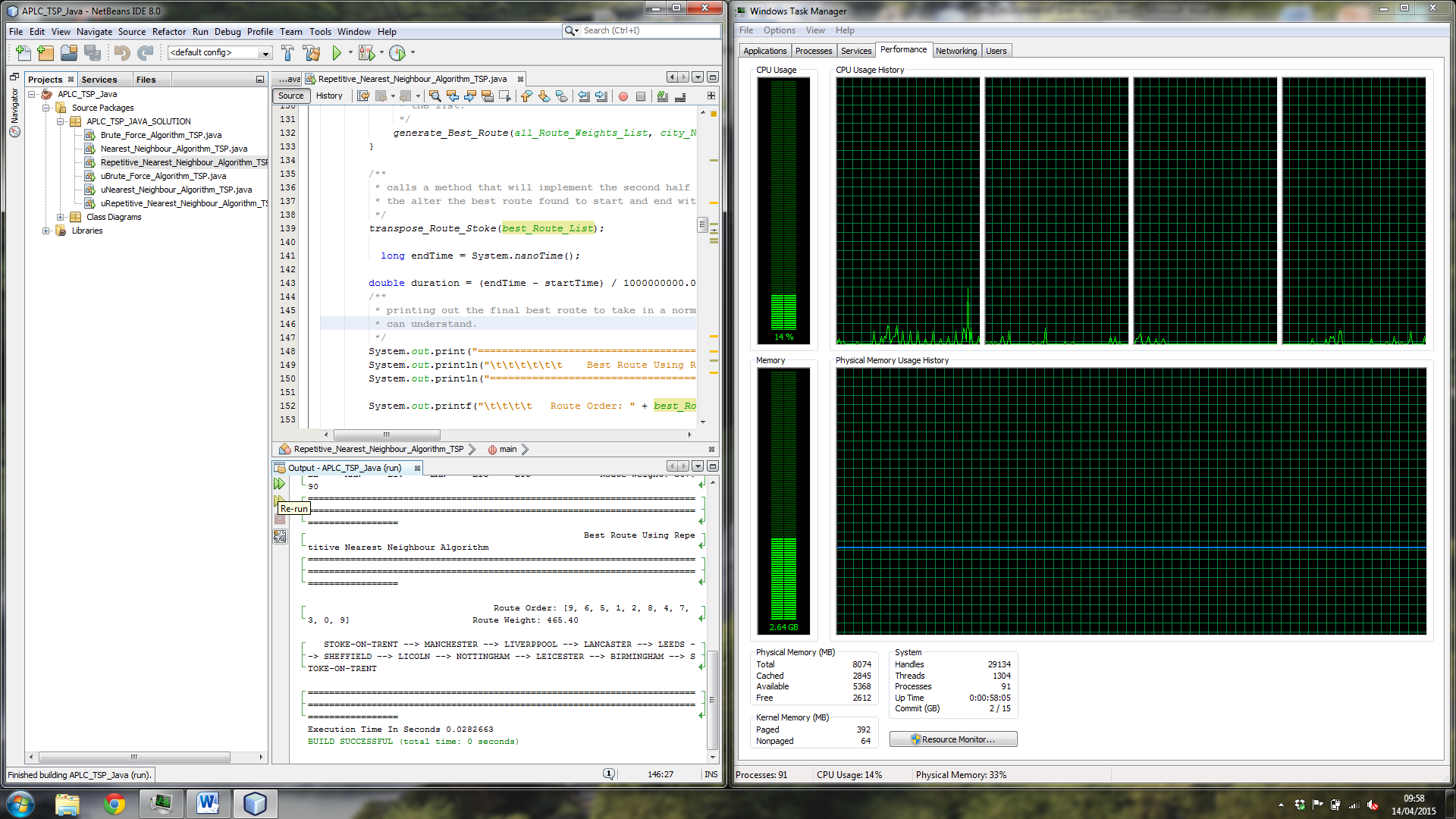
 

Figure 4 Screenshots of CPU usage Haskell (Right) Java (Left)

When you scale the solution from ten cities where you have a possibility of 45 x 10 = 450 different options to take, but if applied to a nationwide route you are dealing with hundreds, if not thousands of different locations, for example, take one hundred cities you would have 505,000 different options to check, this makes small reduction speed into a massive reduction in speed, so depending on what the solution are being used to generate you would been to choose accordingly whether it is the most efficient route or generating a route quickly.

The Java solution is both compiled and interpreted; it is first compiled into Java bytecode and then interpreted by the JVM (Java Virtual Machine) this is what gives Java its platform independence. [5] The Haskell solutions can be compiled or interpreted if you want to compile the code into an executable you can use GHC or you can use the interpreter GHCi there programs are interpreted and the functions can be called dynamically to monitor their output. [6]

The main difference between the two solutions is that Haskell utilizes lazy evaluation, which is deferring the evaluation of expressions until they are required by other expressions in the program, because the arguments remain unevaluated until their outputs are needed, this means Haskell can be more efficient because values that are not being computed if the program isn’t using them, this allows the use of infinite lists which makes Haskell perfect for solving mathematically and computation problems like the traveling salesman problem, unlike Java infinite lists are not possible. The disadvantage is that estimating the memory usage of a program is hard. [4]

**Conclusion**

The advantages of using functional programming languages are that mathematical functions are easily translated, which allows you to solve complex computation problems with concise code, lazy evaluation can be used to reduce the amount of resources used. On the other hand the disadvantages of using functional programming languages are that they can be less efficient when performing some task compared to other programming paradigms and due to the lack of variables and not being dependant on the sequence and large scale problems with many variables and sequential activities can be accomplished better in other paradigms such as imperative programing. [7]

The benefits of using imperative programming languages are that they are highly efficient when performing calculations; popular meaning that there is plenty of support available. The disadvantages to using an imperative language is that when compiling and executing errors can arise and makes debugging hard due to obscure error messages, the order in which operations are executed is vital which can be an issue when solving some problems. [7]

To conclude each programming paradigm has their benefits when tackling the traveling salesman problem, for example the Java implementations is faster for several of the algorithms, but also uses more resources when compared to the Haskell solutions because of this I would say Haskell is better suited to solving the TSP because when it is scaled for routes that may contain thousands of cities it would require less resource and still executed in the same amount of time a Java solution would.

To reflect on the assignment as a whole I found the Java a lot easier than the Haskell this is due to having more experience using Java, but when I programmed the assignment in Haskell it was more of a challenge and make me think about what I doing instead of going on auto pilot as I would when using Java. After completing all the algorithms in Haskell fee it has widened my understanding of the functional programming paradigm even though it was confusing sometimes and understand that there isn’t a perfect programming language where you cannot use the same language for every scenario.

**Appendix A**

**Class Diagrams**

**Brute\_Force\_Algorithm\_TSP.Java**

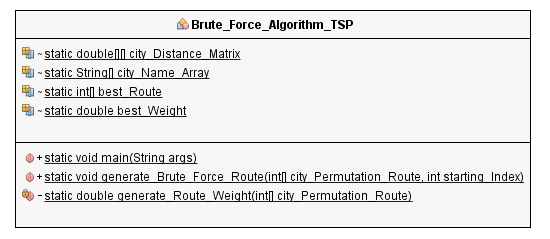


Figure Brute\_Force\_Algorithm\_TSP.Java Class Diagram

**Nearest\_Neighbour\_Algorithm\_TSP.Java**

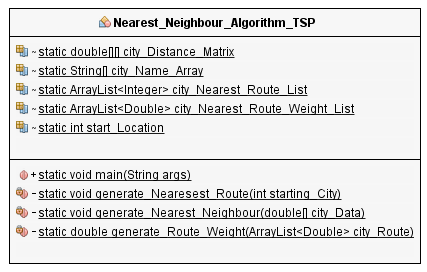


Figure Nearest\_Neighbour\_Algorithm\_TSP.Java Class Diagram

**Repetitive\_Nearest\_Neighbour\_Algorithm\_TSP.Java**

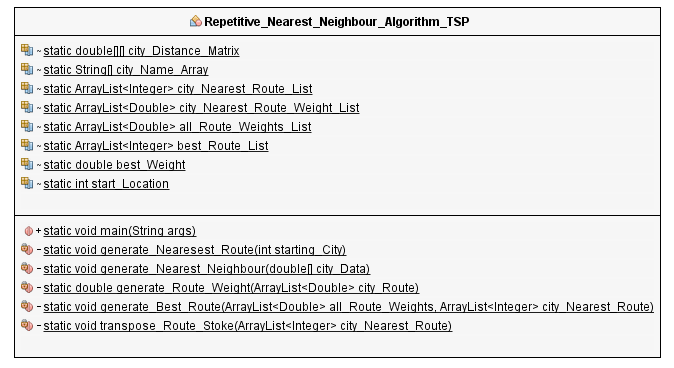
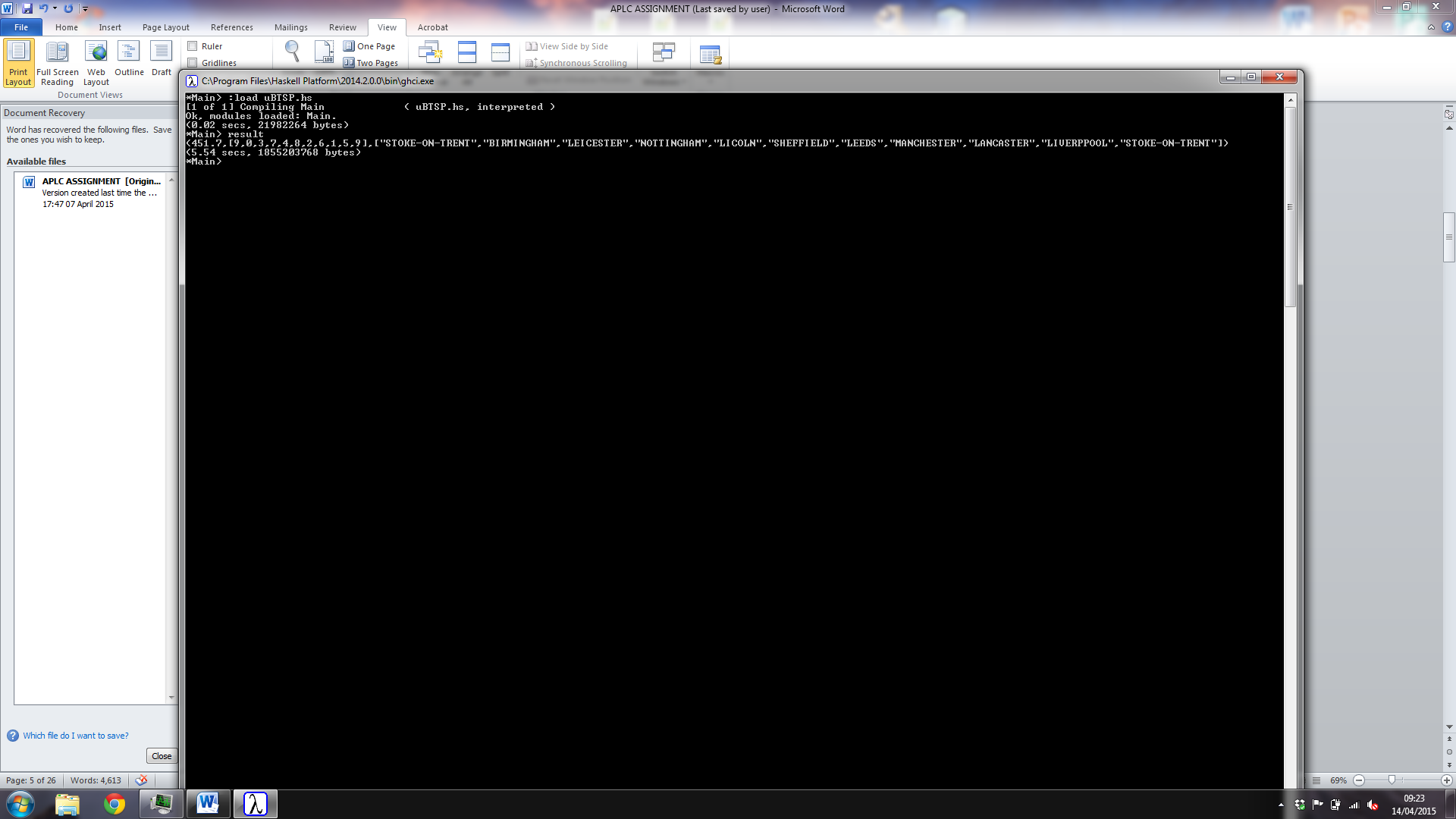


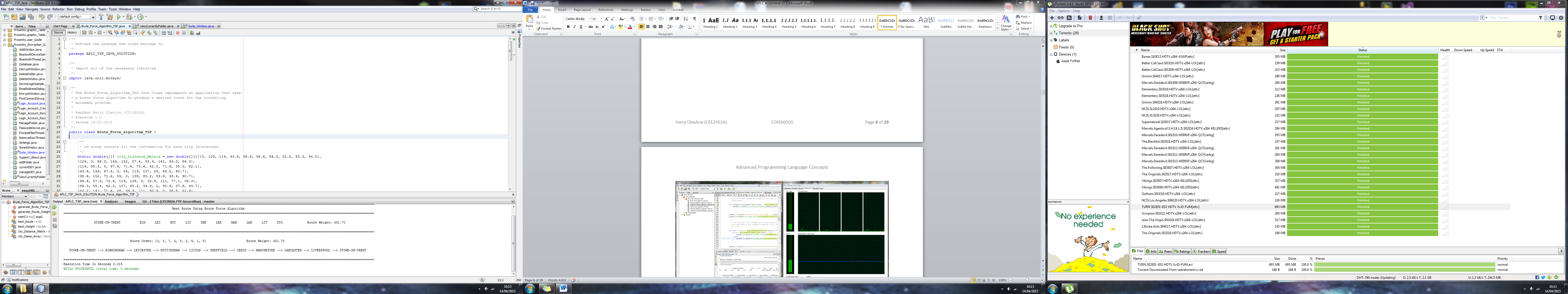
Figure Repetitive\_Nearest\_Neighbour\_Algorithm\_TSP.Java Class Diagram

**Appendix B**

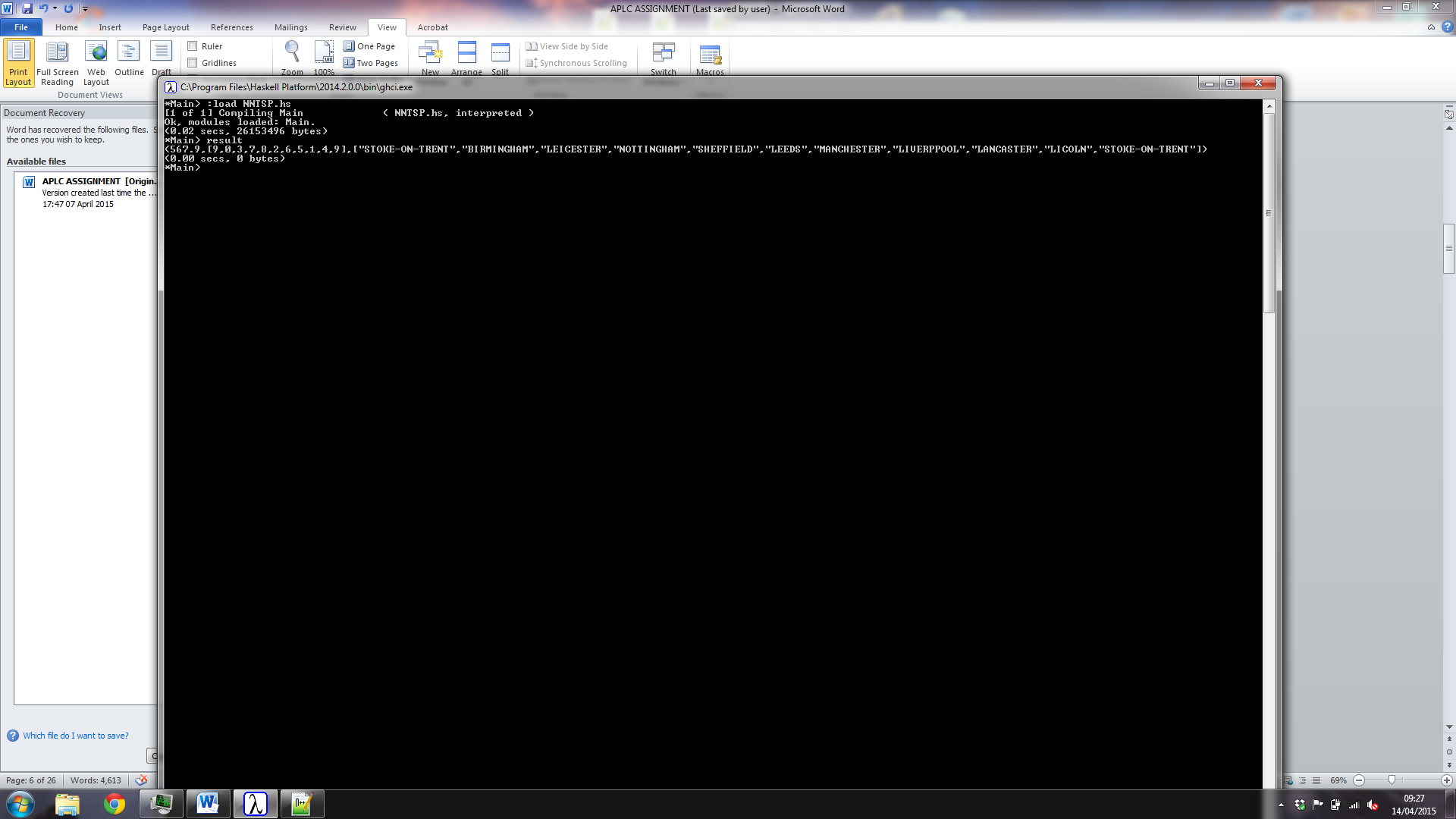
**Example Results**

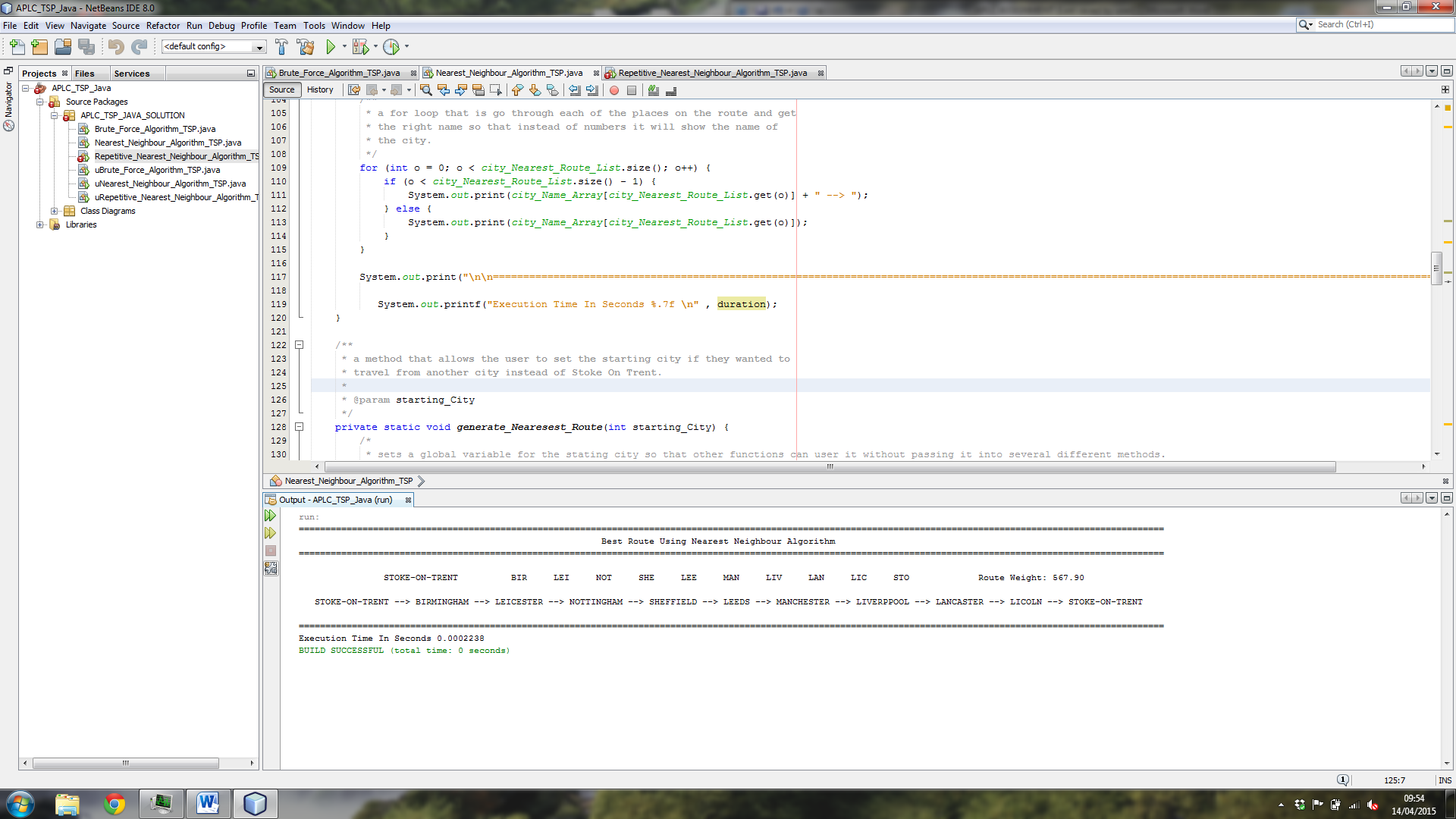
**Brute force Algorithm**



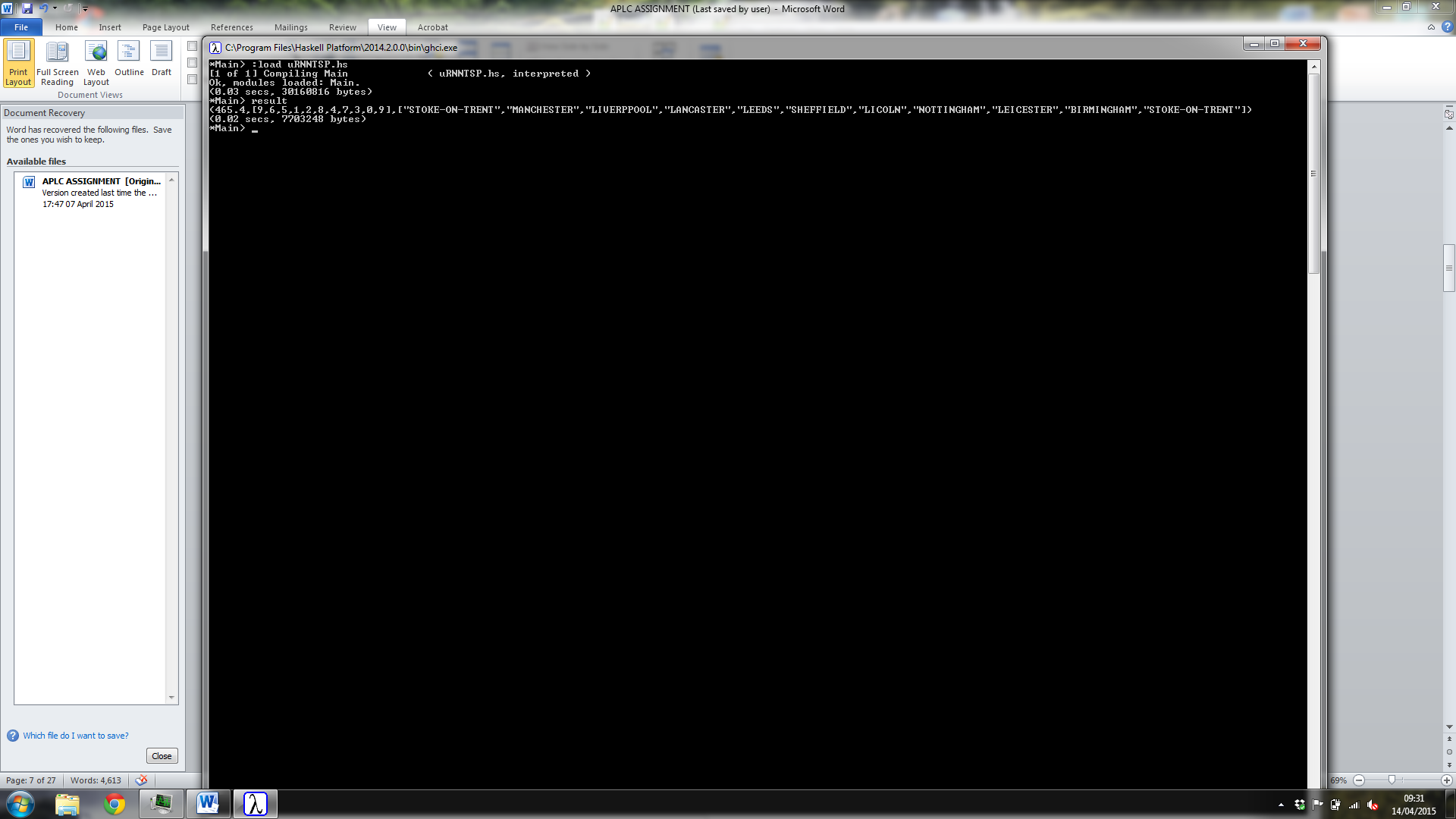


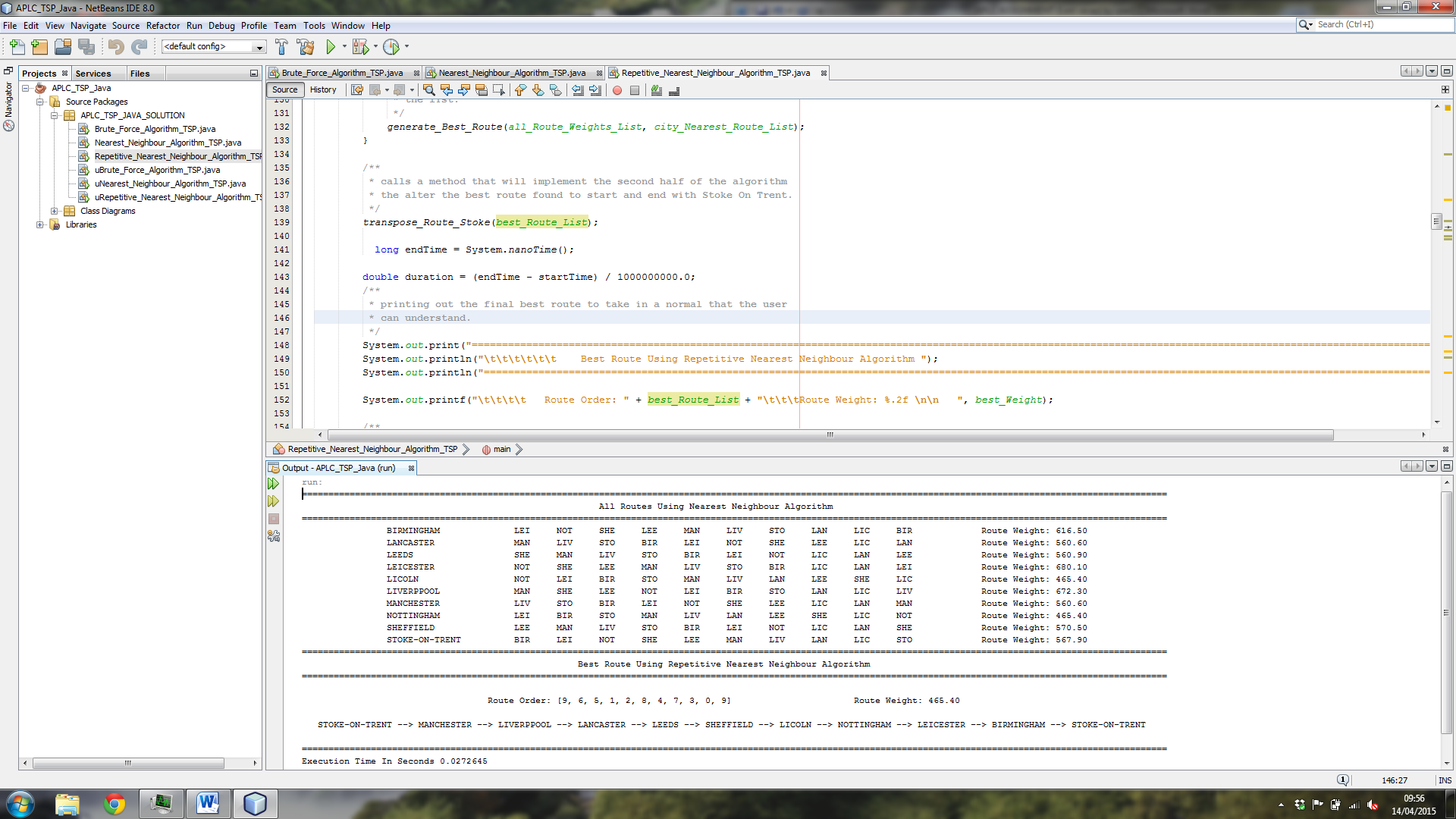
**Nearest Neighbour Algorithm**





**Repetitive Nearest Neighbour Algorithm**





**Appendix C**

**Commented Code Listing**

**Brute\_Force\_Algorithm\_TSP.hs**

{-

The Brute\_Force\_Algorithm\_TSP.Hs implements an application using a

brute force algorithm to produce a route for the travelling salesman problem.

@author Harry Clewlow (C012952A)

@version 1.0

@since 18-03-2014

-}

{- The list of lists used for storing the information about each city as a double (distance from one city to another)-}

distance\_Matrix = [[0, 129, 119, 43.6, 98.6, 98.6, 86.3, 52.2, 85.3, 44.5],

[129, 0, 88.3, 149, 152, 57.4, 55.4, 141, 93.3, 86.3],

[119, 88.3, 0, 97.4, 71.6, 72.6, 42.5, 71.6, 35.5, 92.1],

[43.6, 149, 97.4, 0, 54, 119, 107, 28, 64.2, 60.7],

[98.6, 152, 71.6, 54, 0, 138, 85.2, 39.9, 48.6, 90.7],

[98.6, 57.4, 72.6, 119, 138, 0, 34.9, 111, 77.1, 56.3],

[86.3, 55.4, 42.5, 107, 85.2, 34.9, 0, 80.9, 37.9, 44.7],

[52.2, 141, 71.6, 28, 39.9, 111, 80.9, 0, 38.8, 52.4],

[85.3, 93.3, 35.5, 64.2, 48.6, 77.1, 37.9, 38.8, 0, 47.4],

[44.5, 86.3, 92.1, 60.7, 90.7, 56.3, 44.7, 52.4, 47.4, 0]] :: [[Double]];

{- The names of each city in the same order as they are in the distance\_Matrix so they can be used to output a route so a user can understand it -}

name\_Array = ["BIRMINGHAM", "LANCASTER", "LEEDS", "LEICESTER", "LICOLN", "LIVERPPOOL", "MANCHESTER", "NOTTINGHAM", "SHEFFIELD", "STOKE-ON-TRENT"];

{-

A list of chars that represent the cities a travelling salesman can visit not including STOKE-ON-TRENT i have hard coded this because the

assignment states that you are starting from and returning to STOKE-ON-TRENT

-}

city\_Index = [0 .. 8] :: [Int];

{- A series of functions that will get all the different possible permutations of the numbers 0-8 each all the different routes you can take from STOKE-ON-TRENT

by inserting and joining each element in the list in all possible ways.

-}

permutate\_City :: [Int] -> [[Int]]

permutate\_City [] = [[]]

permutate\_City (x:xs) = concat [insert x ys | ys <- permutate\_City xs]

insert :: Int -> [Int] -> [[Int]]

insert x [] = [[x]]

insert x (y:ys) = (x:y:ys) : consall y (insert x ys)

consall y [] = []

consall y (xs:xss) = (y:xs) : consall y xss

{-

A function that will take the list of permutations for the different routes you can take from STOKE-ON-TRENT and then return back to it, then it then it will add

9 to the start of each element so that they can be modified into co-ordinates to be used on the distance\_Matrix.

-}

get\_Permutations :: [[Int]] -> [(Double, [Int] )]

get\_Permutations [[]] = error "Empty List"

get\_Permutations xs = [(get\_Weight ( 9:x++[9]) , (9:x++[9])) | x <- xs ]

{-

A function that will get the take a list of Ints and zip them together so that they can be used as co-ordinates for use on the distance\_Matrix

to get the value that represents the distance from x to y (one city to another).

-)

get\_Weight :: [Int] -> Double

get\_Weight [] = error "Empty List"

get\_Weight (x:xs) = generate\_Weight $ zipWith (\c1 c2 -> distance\_Matrix !! c1 !! c2) (x:xs)(xs)

{-

A function that take a list of double values and add them all up to give you a total, in this case it will add all the different stops

on a route up to give you the total weight for that specific route.

-}

generate\_Weight :: [Double] -> Double

generate\_Weight [] = 0

generate\_Weight (x:[]) = x

generate\_Weight (x:xs) = x + generate\_Weight xs

{-

A function that is used to et the route with the smallest weight (best) it does this by compare one element against the next using an if

statement that will return the smallest route once it has compared all the values in the object passed into it.

-}

get\_Min\_Route :: (Ord a) => [a] -> a

get\_Min\_Route [] = error "Empty List"

get\_Min\_Route [x] = x

get\_Min\_Route (x:y:xs) = if x < y then get\_Min\_Route (x:xs)

else get\_Min\_Route (y:xs)

{-

A function that is covert a list of Int type into the names of each city so that it is more user friendly by taking a tuple and adding a

list of list of chars (String) that will contain the names this is done by using the Ints as an index for name\_Array.

-}

get\_Names :: (Double, [Int]) -> (Double, [Int], [[Char]])

get\_Names (nil , []) = error "Empty List"

get\_Names data\_Tuple = (fst data\_Tuple , snd data\_Tuple , [name\_Array !! n | n <- (snd data\_Tuple)])

{- A variable to store the result of all the above functions so that it can be called easily in the CMD interface to show the user the route that has been generated.

-}

result = get\_Names (get\_Min\_Route (get\_Permutations (permutate\_City city\_Index)))

**Nearest\_Neighbour\_Algorithm\_TSP.hs**

{-

The Nearest\_Neighbour\_Algorithm\_TSP.Hs implements an application using a

nearest neighbour algorithm to produce a route for the travelling salesman problem.

@author Harry Clewlow (C012952A)

@version 1.0

@since 18-03-2014

-}

{- The list of lists used for storing the information about each city as a double (distance from one city to another)-}

distance\_Matrix = [[0, 129, 119, 43.6, 98.6, 98.6, 86.3, 52.2, 85.3, 44.5],

[129, 0, 88.3, 149, 152, 57.4, 55.4, 141, 93.3, 86.3],

[119, 88.3, 0, 97.4, 71.6, 72.6, 42.5, 71.6, 35.5, 92.1],

[43.6, 149, 97.4, 0, 54, 119, 107, 28, 64.2, 60.7],

[98.6, 152, 71.6, 54, 0, 138, 85.2, 39.9, 48.6, 90.7],

[98.6, 57.4, 72.6, 119, 138, 0, 34.9, 111, 77.1, 56.3],

[86.3, 55.4, 42.5, 107, 85.2, 34.9, 0, 80.9, 37.9, 44.7],

[52.2, 141, 71.6, 28, 39.9, 111, 80.9, 0, 38.8, 52.4],

[85.3, 93.3, 35.5, 64.2, 48.6, 77.1, 37.9, 38.8, 0, 47.4],

[44.5, 86.3, 92.1, 60.7, 90.7, 56.3, 44.7, 52.4, 47.4, 0]] :: [[Double]];

{- The names of each city in the same order as they are in the distance\_Matrix so they can be used to output a route so a user can understand it -}

name\_Array = ["BIRMINGHAM", "LANCASTER", "LEEDS", "LEICESTER", "LICOLN", "LIVERPPOOL", "MANCHESTER", "NOTTINGHAM", "SHEFFIELD", "STOKE-ON-TRENT"];

{-

A list of chars that represent the cities a travelling salesman can visit not including STOKE-ON-TRENT i have hard coded this because the

assignment states that you are starting from and returning to STOKE-ON-TRENT

-}

city\_Index = [0 .. 9] :: [Int];

{-

A function that will take the list of permutations for the different routes you can take from STOKE-ON-TRENT and then return back to it, then it then it will add

9 to the start of each element so that they can be modified into co-ordinates to be used on the distance\_Matrix.

-}

get\_Permutations :: [[Int]] -> [(Double, [Int] )]

get\_Permutations [[]] = [( 0 , [])]

get\_Permutations xs = [(get\_Weight ( ( x)) , ( x)) | x <- xs ]

{-

A function that will get the take a list of Ints and zip them together so that they can be used as co-ordinates for use on the distance\_Matrix

to get the value that represents the distance from x to y (one city to another).

-}

get\_Weight :: [Int] -> Double

get\_Weight [] = error "Empty List"

get\_Weight (x:xs) = generate\_Weight $ zipWith (\c1 c2 -> distance\_Matrix !! c1 !! c2) (x:xs)(xs)

{- A function that will add a Int representing a city in front of all the different elements in a list so that they can be used as co-ordinates as reference

for the distance\_Matrix

-}

add\_City :: Int -> [Int] -> [[Int]]

add\_City i [] = []

add\_City i (x:xs) = [i,x] : add\_City i xs

{-

A function that take a list of double values and add them all up to give you a total, in this case it will add all the different stops

on a route up to give you the total weight for that specific route.

-}

generate\_Weight :: [Double] -> Double

generate\_Weight [] = 0

generate\_Weight (x:[]) = x

generate\_Weight (x:xs) = x + generate\_Weight xs

{- A function that will remove all the co-ordinates that return 0 eg (9,9 so the same city to itself).-}

getlist :: [(Double, [Int])] -> [(Double, [Int])]

getlist [(**\_**, [])] = [(0,[])]

getlist xs = filter condition xs

**where** condition (n,**\_**) = n > 0

{- A function that will get the next city to visit by getting the last element in the list of Int eg p[9,0] it will get 0 back so the next city

would be BIRMINGHAM.

-}

get\_Next\_City :: (Double, [Int]) -> Int

get\_Next\_City (**\_**, []) = error "List is empty"

get\_Next\_City place = last (snd place)

{- A function that will remove an element from a list eg if the route have just visited number 0 it will remove 0 from the list-}

remove\_Item :: Int -> [Int] -> [Int]

remove\_Item **\_** [] = []

remove\_Item i xs = filter condition xs

**where** condition (n) = n /= i

{-

A function that is used to et the route with the smallest weight (best) it does this by compare one element against the next using an if

statement that will return the smallest route once it has compared all the values in the object passed into it.

-}

get\_Min\_Route :: (Ord a) => [a] -> a

get\_Min\_Route [] = error "List is empty"

get\_Min\_Route [x] = x

get\_Min\_Route (x:y:xs) = **if** x < y **then** get\_Min\_Route(x:xs)

**else** get\_Min\_Route(y:xs)

{-

A function that is covert a list of Int type into the names of each city so that it is more user friendly by taking a tuple and adding a

list of list of chars (String) that will contain the names this is done by using the Ints as an index for name\_Array.

-}

get\_Names :: (Double, [Int]) -> (Double, [Int],[[Char]])

get\_Names ( nil , []) = error "List is empty"

get\_Names xs = (fst xs , snd xs, [name\_Array !! p| p<-(snd xs)] )

{- A function that will generate the best route when a user enters the starting location and the list of other cities you can visit-}

generate\_Best\_Route :: Int -> [Int] -> [Int]

generate\_Best\_Route **\_** [] = []

generate\_Best\_Route strt citylist = get\_Next\_City(get\_Min\_Route(getlist (get\_Permutations (add\_City strt citylist)))) : generate\_Best\_Route (get\_Next\_City(get\_Min\_Route(getlist (get\_Permutations (add\_City strt citylist))))) (remove\_Item (get\_Next\_City(get\_Min\_Route(getlist (get\_Permutations (add\_City strt citylist))))) citylist)

{- A function that will remove the a value that is passed into the method eg the start city from the list of cities you can visit. -}

remove\_Duplicate :: Int -> [Int] -> [Int]

remove\_Duplicate **\_** [] = []

remove\_Duplicate str citylist = generate\_Best\_Route str (remove\_Item str citylist)

{- A function that gets the total weight of the final route so that it an be displayed to the user. -}

get\_Nearest\_Neighbour :: [Int] -> (Double, [Int])

get\_Nearest\_Neighbour [] = (0 , [])

get\_Nearest\_Neighbour route = (get\_Weight route , route)

{- A variable to store the result of all the above functions so that it can be called easily in the CMD interface to show the user the route that has been generated.-}

result = get\_Names (get\_Nearest\_Neighbour (9:(generate\_Best\_Route 9 (remove\_Duplicate 9 city\_Index))++[9]))

**Repetitive\_Nearest\_Neighbour\_Algorithm\_TSP.hs**

{-

The Repetitive\_Nearest\_Neighbour\_Algorithm\_TSP.Hs implements an application using a

repetitive nearest neighbour algorithm to produce a route for the travelling salesman problem.

@author Harry Clewlow (C012952A)

@version 1.0

@since 18-03-2014

-}

{- The list of lists used for storing the information about each city as a double (distance from one city to another)-}

distance\_Matrix = [[0, 129, 119, 43.6, 98.6, 98.6, 86.3, 52.2, 85.3, 44.5],

[129, 0, 88.3, 149, 152, 57.4, 55.4, 141, 93.3, 86.3],

[119, 88.3, 0, 97.4, 71.6, 72.6, 42.5, 71.6, 35.5, 92.1],

[43.6, 149, 97.4, 0, 54, 119, 107, 28, 64.2, 60.7],

[98.6, 152, 71.6, 54, 0, 138, 85.2, 39.9, 48.6, 90.7],

[98.6, 57.4, 72.6, 119, 138, 0, 34.9, 111, 77.1, 56.3],

[86.3, 55.4, 42.5, 107, 85.2, 34.9, 0, 80.9, 37.9, 44.7],

[52.2, 141, 71.6, 28, 39.9, 111, 80.9, 0, 38.8, 52.4],

[85.3, 93.3, 35.5, 64.2, 48.6, 77.1, 37.9, 38.8, 0, 47.4],

[44.5, 86.3, 92.1, 60.7, 90.7, 56.3, 44.7, 52.4, 47.4, 0]] :: [[Double]];

{- The names of each city in the same order as they are in the distance\_Matrix so they can be used to output a route so a user can understand it -}

name\_Array = ["BIRMINGHAM", "LANCASTER", "LEEDS", "LEICESTER", "LICOLN", "LIVERPPOOL", "MANCHESTER", "NOTTINGHAM", "SHEFFIELD", "STOKE-ON-TRENT"];

{-

A list of chars that represent the cities a travelling salesman can visit not including STOKE-ON-TRENT i have hard coded this because the

assignment states that you are starting from and returning to STOKE-ON-TRENT

-}

city\_Index = [0 .. 9] :: [Int];

{-

A function that will take the list of permutations for the different routes you can take from STOKE-ON-TRENT and then return back to it, then it then it will add

9 to the start of each element so that they can be modified into co-ordinates to be used on the distance\_Matrix.

-}

get\_Permutations :: [[Int]] -> [(Double, [Int] )]

get\_Permutations [[]] = [( 0 , [])]

get\_Permutations xs = [(get\_Weight ( ( x)) , ( x)) | x <- xs ]

{-

A function that will get the take a list of Ints and zip them together so that they can be used as co-ordinates for use on the distance\_Matrix

to get the value that represents the distance from x to y (one city to another).

-}

get\_Weight :: [Int] -> Double

get\_Weight [] = error "Empty List"

get\_Weight (x:xs) = generate\_Weight $ zipWith (\c1 c2 -> distance\_Matrix !! c1 !! c2) (x:xs)(xs)

{- A function that will add a Int representing a city in front of all the different elements in a list so that they can be used as co-ordinates as reference

for the distance\_Matrix

-}

add\_City :: Int -> [Int] -> [[Int]]

add\_City i [] = []

add\_City i (x:xs) = [i,x] : add\_City i xs

{-

A function that take a list of double values and add them all up to give you a total, in this case it will add all the different stops

on a route up to give you the total weight for that specific route.

-}

generate\_Weight :: [Double] -> Double

generate\_Weight [] = 0

generate\_Weight (x:[]) = x

generate\_Weight (x:xs) = x + generate\_Weight xs

{- A function that will remove all the co-ordinates that return 0 eg (9,9 so the same city to itself).-}

getlist :: [(Double, [Int])] -> [(Double, [Int])]

getlist [(**\_**, [])] = [(0,[])]

getlist xs = filter condition xs

**where** condition (n,**\_**) = n > 0

{- A function that will get the next city to visit by getting the last element in the list of Int eg p[9,0] it will get 0 back so the next city

would be BIRMINGHAM.

-}

get\_Next\_City :: (Double, [Int]) -> Int

get\_Next\_City (**\_**, []) = error "List is empty"

get\_Next\_City place = last (snd place)

{- A function that will remove an element from a list eg if the route have just visited number 0 it will remove 0 from the list-}

remove\_Item :: Int -> [Int] -> [Int]

remove\_Item **\_** [] = []

remove\_Item i xs = filter condition xs

**where** condition (n) = n /= i

{-

A function that is used to et the route with the smallest weight (best) it does this by compare one element against the next using an if

statement that will return the smallest route once it has compared all the values in the object passed into it.

-}

get\_Min\_Route :: (Ord a) => [a] -> a

get\_Min\_Route [] = error "List is empty"

get\_Min\_Route [x] = x

get\_Min\_Route (x:y:xs) = **if** x < y **then** get\_Min\_Route(x:xs)

**else** get\_Min\_Route(y:xs)

{-

A function that is covert a list of Int type into the names of each city so that it is more user friendly by taking a tuple and adding a

list of list of chars (String) that will contain the names this is done by using the Ints as an index for name\_Array.

-}

get\_Names :: (Double, [Int]) -> (Double, [Int],[[Char]])

get\_Names ( nil , []) = error "List is empty"

get\_Names xs = (fst xs , snd xs, [name\_Array !! p| p<-(snd xs)] )

{- A function that will generate the best route when a user enters the starting location and the list of other cities you can visit-}

generate\_Best\_Route :: Int -> [Int] -> [Int]

generate\_Best\_Route **\_** [] = []

generate\_Best\_Route strt citylist = get\_Next\_City(get\_Min\_Route(getlist (get\_Permutations (add\_City strt citylist)))) : generate\_Best\_Route (get\_Next\_City(get\_Min\_Route(getlist (get\_Permutations (add\_City strt citylist))))) (remove\_Item (get\_Next\_City(get\_Min\_Route(getlist (get\_Permutations (add\_City strt citylist))))) citylist)

{- A function that will remove the a value that is passed into the method eg the start city from the list of cities you can visit. -}

remove\_Duplicate :: Int -> [Int] -> [Int]

remove\_Duplicate **\_** [] = []

remove\_Duplicate str citylist = generate\_Best\_Route str (remove\_Item str citylist)

{- A function that gets the total weight of the final route so that it an be displayed to the user. -}

get\_Nearest\_Neighbour :: [Int] -> (Double, [Int])

get\_Nearest\_Neighbour [] = (0 , [])

get\_Nearest\_Neighbour route = (get\_Weight route , route)

{- A function that is get the best route from running the nearest neighbour algorithm from each different city as the starting point-}

get\_Repetitive\_Nearest\_Neighbour :: [Int] -> (Double, [Int])

get\_Repetitive\_Nearest\_Neighbour [] = (0 , [])

get\_Repetitive\_Nearest\_Neighbour list1 = get\_Min\_Route ([get\_Nearest\_Neighbour (x:(generate\_Best\_Route x (remove\_Duplicate x list1)++[x])) | x<-[0,1,2,3,4,5,6,7,8,9]])

{- A function that will transpose the the best route so that it starts and returns to the city entered as the input.-}

transpose\_Route :: Int -> (Double, [Int]) -> (Double, [Int])

transpose\_Route **\_** (**\_**, []) = (0 , [])

transpose\_Route i (weight, route) = (weight, get\_New\_Route i (tail route))

{- A function that will move the starting location of the route to the one that have been entered as an input -}

get\_New\_Route :: Int -> [Int] -> [Int]

get\_New\_Route **\_** [] = []

get\_New\_Route i (x:xs) = **if** x == i **then** x:xs++[i]

**else** get\_New\_Route i (xs ++ [x])

{- A variable to store the result of all the above functions so that it can be called easily in the CMD interface to show the user the route that has been generated.-}

result = get\_Names(transpose\_Route 9 (get\_Repetitive\_Nearest\_Neighbour city\_Index))

**Brute\_Force\_Algorithm\_TSP.Java**

/\*\*

\* **Defines the package the class belongs to.**

\*/

package APLC\_TSP\_JAVA\_SOLUTION;

/\*\*

\* **Import all of the necessary libraries.**

\*/

import java.util.Arrays;

/\*\*

\* **The Brute\_Force\_Algorithm\_TSP.Java Class implements an application uses a**

\* **brute force algorithm to produce a desired route for the travelling salesman**

\* **problem.**

\*

\* **@author** Harry Clewlow (C012952A)

\* **@version** 1.0

\* **@since** 18-01-2014

\*/

public class **Brute\_Force\_Algorithm\_TSP** {

/\*\*

\* **2d array contain all the information for each city (distances).**

\*/

static double[][] *city\_Distance\_Matrix* = new double[][]{{0, 129, 119, 43.6, 98.6, 98.6, 86.3, 52.2, 85.3, 44.5},

{129, 0, 88.3, 149, 152, 57.4, 55.4, 141, 93.3, 86.3},

{119, 88.3, 0, 97.4, 71.6, 72.6, 42.5, 71.6, 35.5, 92.1},

{43.6, 149, 97.4, 0, 54, 119, 107, 28, 64.2, 60.7},

{98.6, 152, 71.6, 54, 0, 138, 85.2, 39.9, 48.6, 90.7},

{98.6, 57.4, 72.6, 119, 138, 0, 34.9, 111, 77.1, 56.3},

{86.3, 55.4, 42.5, 107, 85.2, 34.9, 0, 80.9, 37.9, 44.7},

{52.2, 141, 71.6, 28, 39.9, 111, 80.9, 0, 38.8, 52.4},

{85.3, 93.3, 35.5, 64.2, 48.6, 77.1, 37.9, 38.8, 0, 47.4},

{44.5, 86.3, 92.1, 60.7, 90.7, 56.3, 44.7, 52.4, 47.4, 0}

};

/\*\*

\* **an array containing all the cities names in order so it can be used to**

\* **display the route.**

\*/

static String[] *city\_Name\_Array* = {"BIRMINGHAM", "LANCASTER", "LEEDS", "LEICESTER", "LICOLN", "LIVERPPOOL", "MANCHESTER", "NOTTINGHAM", "SHEFFIELD", "STOKE-ON-TRENT"};

/\*\*

\* **creating objects to hold the different elements used to calculate the**

\* **route (the best route and weight).**

\*/

static int[] *best\_Route*;

static double *best\_Weight* = Double.*MAX\_VALUE*;

/\*\*

\* **the main method used to generate and display the resulting route that the**

\* **algorithm produces.**

\*

\* **@param** args

\*/

public static void ***main***(String args[]) {

/\*\*

\* **creates an array of the cities to visit in number form 9 is not**

\* **included due to it being Stoke On Trent.**

\*/

int[] a = {0, 1, 2, 3, 4, 5, 6, 7, 8};

/\*\*

\* **calls the method which generates the route that will be used later.**

\*/

*generate\_Brute\_Force\_Route*(a, 0);

/\*\*

\* **printing out display information to make the results understandable.**

\*/

System.*out*.print("===================================================================================================================================================================**\n**");

System.*out*.println("**\t\t\t\t\t\t\t** Best Route Using Brute Force Algorithm ");

System.*out*.print("===================================================================================================================================================================**\n\n**");

System.*out*.print("**\t\t**" + *city\_Name\_Array*[9] + "**\t\t**");

/\*\*

\* **a for loop that will show the route with the first 3 characters of**

\* **the city's name instead of a number so the user can understand the**

\* **route easier.**

\*/

for (Integer city\_Nearest\_Route\_List1 : *best\_Route*) {

System.*out*.print(*city\_Name\_Array*[city\_Nearest\_Route\_List1].substring(0, 3) + "**\t**");

}

System.*out*.print(*city\_Name\_Array*[9].substring(0, 3) + "**\t**");

System.*out*.printf("**\t**Route Wieght: %.2f **\n\n**", *best\_Weight*);

System.*out*.print("===================================================================================================================================================================**\n\n**");

System.*out*.printf("**\t\t\t\t** Route Order: " + Arrays.*toString*(*best\_Route*) + "**\t\t\t**Route Wieght: %.2f **\n\n** ", *best\_Weight*);

System.*out*.print(*city\_Name\_Array*[9] + " --> ");

/\*\*

\* **a for loop that is go through each of the places on the route and get**

\* **the right name so that instead of numbers it will show the name of**

\* **the city.**

\*/

for (int o = 0; o < *best\_Route*.length; o++) {

System.*out*.print(*city\_Name\_Array*[*best\_Route*[o]] + " --> ");

}

System.*out*.print(*city\_Name\_Array*[9]);

System.*out*.print("**\n\n**===================================================================================================================================================================**\n**");

}

/\*\*

\* **a method that implements a brute force algorithm when the data for a city**

\* **is entered (the 1d array from the 2d array containing all the distance**

\* **values) and a start location (0 so that is knows where to start).**

\*

\* **@param** city\_Permutation\_Route

\* **@param** starting\_Index

\*/

public static void ***generate\_Brute\_Force\_Route***(int[] city\_Permutation\_Route, int starting\_Index) {

/\*\*

\* **creating variables that will be used in the calculations in**

\* **determining the route.**

\*/

double route\_Distance;

/\*\*

\* **checks if the permutation of the array has completed a cycle so it**

\* **can check the weight of the route.**

\*/

if (city\_Permutation\_Route.length == starting\_Index) {

/\*\*

\* **generates the weight for the route.**

\*/

route\_Distance = *generate\_Route\_Weight*(city\_Permutation\_Route);

/\*\*

\* **checks if the weight is better than the best weight.**

\*/

if (route\_Distance < *best\_Weight*) {

/\*\*

\* **stores the weight and route so it can be printed out to the**

\* **user later.**

\*/

*best\_Weight* = route\_Distance;

*best\_Route* = city\_Permutation\_Route;

}

} else {

/\*\*

\* a for loop does several cycles of permutation until the

\* starting\_Index is equal to i:

\*/

for (int i = starting\_Index; i < city\_Permutation\_Route.length; i++) {

/\*\*

\* **setting temp variables to save the data to while getting the**

\* **permutations.**

\*/

int[] city\_Route = city\_Permutation\_Route.clone();

int temp\_Route = city\_Route[i];

/\*\*

\* **swapping the position of the elements.**

\*/

city\_Route[i] = city\_Route[starting\_Index];

city\_Route[starting\_Index] = temp\_Route;

/\*\*

\* **goes through the method again to recursively get the**

\* **permutations.**

\*/

*generate\_Brute\_Force\_Route*(city\_Route, starting\_Index + 1);

}

}

}

/\*\*

\* **a method that will generate the weight (the cost) for the best route that**

\* **has been chosen.**

\*

\* **@param** city\_Route

\* **@return**

\*/

private static double ***generate\_Route\_Weight***(int[] city\_Permutation\_Route) {

/\*\*

\* **a variable to store the weight while doing the calculations.**

\*/

double route\_Distance;

/\*\*

\* **adds the distance from stoke to the first stop to the total weight.**

\*/

route\_Distance = *city\_Distance\_Matrix*[9][city\_Permutation\_Route[0]];

/\*\*

\* **a for loop that will add each of the values for each stop to get the**

\* **total weight of the route.**

\*/

for (int i = 0; i < city\_Permutation\_Route.length - 1; i++) {

route\_Distance = route\_Distance + *city\_Distance\_Matrix*[city\_Permutation\_Route[i]][city\_Permutation\_Route[i + 1]];

}

/\*\*

\* **adds the distance to get from the last stop back to Stoke On Trent to**

\* **the weight.**

\*/

route\_Distance = route\_Distance + *city\_Distance\_Matrix*[city\_Permutation\_Route[8]][9];

return route\_Distance;

}

}

**Nearest\_Neighbour\_Algorithm\_TSP.Java**

/\*\*

\* **Defines the package the class belongs to.**

\*/

package APLC\_TSP\_JAVA\_SOLUTION;

/\*\*

\* **Import all of the necessary libraries.**

\*/

import java.util.ArrayList;

/\*\*

\* **The Nearest\_Neighbour\_Algorithm\_TSP.Java Class implements an application uses**

\* **the Nearest Neighbour algorithm to produce a desired route for the travelling**

\* **salesman problem.**

\*

\* **@author** Harry Clewlow (C012952A)

\* **@version** 1.0

\* **@since** 18-01-2014

\*/

public class **Nearest\_Neighbour\_Algorithm\_TSP** {

/\*\*

\* **2d array contain all the information for each city (distances).**

\*/

static double[][] *city\_Distance\_Matrix* = new double[][]{{0, 129, 119, 43.6, 98.6, 98.6, 86.3, 52.2, 85.3, 44.5},

{129, 0, 88.3, 149, 152, 57.4, 55.4, 141, 93.3, 86.3},

{119, 88.3, 0, 97.4, 71.6, 72.6, 42.5, 71.6, 35.5, 92.1},

{43.6, 149, 97.4, 0, 54, 119, 107, 28, 64.2, 60.7},

{98.6, 152, 71.6, 54, 0, 138, 85.2, 39.9, 48.6, 90.7},

{98.6, 57.4, 72.6, 119, 138, 0, 34.9, 111, 77.1, 56.3},

{86.3, 55.4, 42.5, 107, 85.2, 34.9, 0, 80.9, 37.9, 44.7},

{52.2, 141, 71.6, 28, 39.9, 111, 80.9, 0, 38.8, 52.4},

{85.3, 93.3, 35.5, 64.2, 48.6, 77.1, 37.9, 38.8, 0, 47.4},

{44.5, 86.3, 92.1, 60.7, 90.7, 56.3, 44.7, 52.4, 47.4, 0}

};

/\*\*

\* **an array containing all the cities names in order so it can be used to**

\* **display the route.**

\*/

static String[] *city\_Name\_Array* = {"BIRMINGHAM", "LANCASTER", "LEEDS", "LEICESTER", "LICOLN", "LIVERPPOOL", "MANCHESTER", "NOTTINGHAM", "SHEFFIELD", "STOKE-ON-TRENT"};

/\*\*

\* **creating objects to hold the different elements used to calculate the**

\* **route (lists: the best route and weight and the starting location.**

\*/

static ArrayList<Integer> *city\_Nearest\_Route\_List* = new ArrayList<>();

static ArrayList<Double> *city\_Nearest\_Route\_Weight\_List* = new ArrayList<>();

static int *start\_Location*;

/\*\*

\* **the main method used to generate and display the resulting route that the**

\* **algorithm produces.**

\*

\* **@param** args

\*/

public static void ***main***(String args[]) {

/\*\*

\* **variable to hold the routes weight.**

\*/

double route\_Weight;

/\*\*

\* **calls the method which generates the route that will be used later.**

\*/

*generate\_Nearesest\_Route*(9);

/\*\*

\* **generates the routes weight and stores it so it can be compared to**

\* **the other routes later.**

\*/

route\_Weight = *generate\_Route\_Weight*(*city\_Nearest\_Route\_Weight\_List*);

/\*\*

\* **printing out display information to make the results understandable.**

\*/

System.*out*.print("===================================================================================================================================================================**\n**");

System.*out*.println("**\t\t\t\t\t\t\t** Best Route Using Nearest Neighbour Algorithm ");

System.*out*.print("===================================================================================================================================================================**\n\n**");

System.*out*.print("**\t\t**" + *city\_Name\_Array*[*city\_Nearest\_Route\_List*.get(9)] + "**\t\t**");

/\*\*

\* **a for loop that will show the route with the first 3 characters of**

\* **the city's name instead of a number so the user can understand the**

\* **route easier.**

\*/

for (Integer city\_Nearest\_Route\_List1 : *city\_Nearest\_Route\_List*) {

System.*out*.print(*city\_Name\_Array*[city\_Nearest\_Route\_List1].substring(0, 3) + "**\t**");

}

System.*out*.printf("**\t**Route Wieght: %.2f **\n\n** ", route\_Weight);

System.*out*.print(*city\_Name\_Array*[*city\_Nearest\_Route\_List*.get(9)] + " --> ");

/\*\*

\* **a for loop that is go through each of the places on the route and get**

\* **the right name so that instead of numbers it will show the name of**

\* **the city.**

\*/

for (int o = 0; o < *city\_Nearest\_Route\_List*.size(); o++) {

if (o < *city\_Nearest\_Route\_List*.size() - 1) {

System.*out*.print(*city\_Name\_Array*[*city\_Nearest\_Route\_List*.get(o)] + " --> ");

} else {

System.*out*.print(*city\_Name\_Array*[*city\_Nearest\_Route\_List*.get(o)]);

}

}

System.*out*.print("**\n\n**===================================================================================================================================================================**\n**");

}

/\*\*

\* **a method that allows the user to set the starting city if they wanted to**

\* **travel from another city instead of Stoke On Trent.**

\*

\* **@param** starting\_City

\*/

private static void ***generate\_Nearesest\_Route***(int starting\_City) {

/\*

\* sets a global variable for the stating city so that other functions can user it without passing it into several different methods.

\*/

*start\_Location* = starting\_City;

/\*\*

\* **calls a method that will generate the route using the nearest**

\* **neighbour algorithm using the city related to the value of the**

\* **variable starting\_City.**

\*/

*generate\_Nearest\_Neighbour*(*city\_Distance\_Matrix*[starting\_City]);

}

/\*\*

\* **a method that implements the nearest neighbour algorithm when the data**

\* **for a city is entered (the 1d array from the 2d array containing all the**

\* **distance values).**

\*

\* **@param** city\_Data

\*/

private static void ***generate\_Nearest\_Neighbour***(double[] city\_Data) {

/\*\*

\* **creating variables that will be used in the calculations in**

\* **determining the route.**

\*/

double min\_Value = Double.*MAX\_VALUE*;

int next\_City = 0;

/\*\*

\* **a for loop that will run through all the different distances store in**

\* **the 1d array called city\_Data.**

\*/

for (int i = 0; i < city\_Data.length; i++) {

/\*\*

\* **checks if the value that is pulled from the array is not equal to**

\* **0 so that it doesn't use the value that is for a cities distance**

\* **from itself.**

\*/

if (city\_Data[i] != 0) {

/\*\*

\* **checks if the city that is not already in the route and is**

\* **not the starting city.**

\*/

if (!*city\_Nearest\_Route\_List*.contains(i) && i != *start\_Location*) {

/\*\*

\* **check is distance is lower than the min\_Value so get the**

\* **next city to visit.**

\*/

if (city\_Data[i] <= min\_Value) {

/\*\*

\* sets the min\_Value so that is can be compared to the

\* next value/s

\*/

min\_Value = city\_Data[i];

/\*\*

\* **sets the value of the next city to visit.**

\*/

next\_City = i;

}

}

}

}

/\*\*

\* **after getting the nearest route from the for loop above the values**

\* **are stored in the relevant lists.**

\*/

*city\_Nearest\_Route\_List*.add(next\_City);

*city\_Nearest\_Route\_Weight\_List*.add(min\_Value);

/\*\*

\* **checks if all the possible routes have been generated if they have**

\* **then it will add the values to the list and exit the method if they**

\* **have not then it will use recursion to generate the rest.**

\*/

if (*city\_Nearest\_Route\_List*.size() != 9) {

*generate\_Nearest\_Neighbour*(*city\_Distance\_Matrix*[next\_City]);

} else {

*city\_Nearest\_Route\_List*.add(*start\_Location*);

*city\_Nearest\_Route\_Weight\_List*.add(*city\_Distance\_Matrix*[*city\_Nearest\_Route\_List*.get(8)][*start\_Location*]);

}

}

/\*\*

\* **a method that will generate the weight (the cost) of each route so that**

\* **the best route can be chosen.**

\*

\* **@param** city\_Route

\* **@return**

\*/

private static double ***generate\_Route\_Weight***(ArrayList<Double> city\_Route) {

/\*\*

\* **a variable to store the weight while doing the calculations.**

\*/

double route\_Weight = 0;

/\*\*

\* **a for loop that will add each of the values for a route to get the**

\* **total weight of the route.**

\*/

for (Double city\_Route1 : city\_Route) {

route\_Weight += city\_Route1;

}

return route\_Weight;

}

}

**Repetitive\_Nearest\_Neighbour\_Algorithm\_TSP.Java**

/\*\*

\* **Defines the package the class belongs to.**

\*/

package APLC\_TSP\_JAVA\_SOLUTION;

/\*\*

\* **Import all of the necessary libraries.**

\*/

import java.util.ArrayList;

/\*\*

\* **The Nearest\_Neighbour\_Algorithm\_TSP.Java Class implements an application uses**

\* **a Repetitive Nearest Neighbour algorithm to produce a desired route for the**

\* **travelling salesman problem.**

\*

\* **@author** Harry Clewlow (C012952A)

\* **@version** 1.0

\* **@since** 18-01-2014

\*/

public class **Repetitive\_Nearest\_Neighbour\_Algorithm\_TSP** {

/\*\*

\* **2d array contain all the information for each city (distances).**

\*/

static double[][] *city\_Distance\_Matrix* = new double[][]{{0, 129, 119, 43.6, 98.6, 98.6, 86.3, 52.2, 85.3, 44.5},

{129, 0, 88.3, 149, 152, 57.4, 55.4, 141, 93.3, 86.3},

{119, 88.3, 0, 97.4, 71.6, 72.6, 42.5, 71.6, 35.5, 92.1},

{43.6, 149, 97.4, 0, 54, 119, 107, 28, 64.2, 60.7},

{98.6, 152, 71.6, 54, 0, 138, 85.2, 39.9, 48.6, 90.7},

{98.6, 57.4, 72.6, 119, 138, 0, 34.9, 111, 77.1, 56.3},

{86.3, 55.4, 42.5, 107, 85.2, 34.9, 0, 80.9, 37.9, 44.7},

{52.2, 141, 71.6, 28, 39.9, 111, 80.9, 0, 38.8, 52.4},

{85.3, 93.3, 35.5, 64.2, 48.6, 77.1, 37.9, 38.8, 0, 47.4},

{44.5, 86.3, 92.1, 60.7, 90.7, 56.3, 44.7, 52.4, 47.4, 0}

};

/\*\*

\* **an array containing all the cities names in order so it can be used to**

\* **display the route.**

\*/

static String[] *city\_Name\_Array* = {"BIRMINGHAM", "LANCASTER", "LEEDS", "LEICESTER", "LICOLN", "LIVERPPOOL", "MANCHESTER", "NOTTINGHAM", "SHEFFIELD", "STOKE-ON-TRENT"};

/\*\*

\* **creating objects to hold the different elements used to calculate the**

\* **route (lists: all the routes and weights, the best route and weight and**

\* **the starting location.**

\*/

static ArrayList<Integer> *city\_Nearest\_Route\_List* = new ArrayList<>();

static ArrayList<Double> *city\_Nearest\_Route\_Weight\_List* = new ArrayList<>();

static ArrayList<Double> *all\_Route\_Weights\_List* = new ArrayList<>();

static ArrayList<Integer> *best\_Route\_List* = new ArrayList<>();

static double *best\_Weight* = Double.*MAX\_VALUE*;

static int *start\_Location*;

/\*\*

\* **the main method used to generate and display the resulting route that the**

\* **algorithm produces.**

\*

\* **@param** args

\*/

public static void ***main***(String args[]) {

/\*\*

\* **variable to hold the routes weight.**

\*/

double route\_Weight;

/\*\*

\* **printing out display information to make the results understandable.**

\*/

System.*out*.print("===================================================================================================================================================================**\n**");

System.*out*.println("**\t\t\t\t\t\t\t**All Routes Using Nearest Neighbour Algorithm ");

System.*out*.print("===================================================================================================================================================================**\n**");

/\*\*

\* a for loop that will generate the a route using the algorithm for

\* each city

\*/

for (int i = 0; i < *city\_Distance\_Matrix*[0].length; i++) {

/\*\*

\* **clearing the lists so that the data for the next city can be**

\* **entered into them.**

\*/

*city\_Nearest\_Route\_List*.clear();

*city\_Nearest\_Route\_Weight\_List*.clear();

/\*\*

\* **calls the method which generates the route that will be used**

\* **later.**

\*/

*generate\_Nearesest\_Route*(i);

/\*\*

\* **printing out the results so that a user can see all the different**

\* **routes so the user can understand the data being used to generate**

\* **the best possible route.**

\*/

if (i == 2 || i == 4) {

System.*out*.print("**\t\t**" + *city\_Name\_Array*[*start\_Location*] + "**\t\t\t**");

} else {

System.*out*.print("**\t\t**" + *city\_Name\_Array*[*start\_Location*] + "**\t\t**");

}

/\*\*

\* **a for loop that will show the route with the first 3 characters**

\* **of the city's name instead of a number so the user can understand**

\* **the route easier.**

\*/

for (Integer city\_Nearest\_Route\_List1 : *city\_Nearest\_Route\_List*) {

System.*out*.print(*city\_Name\_Array*[city\_Nearest\_Route\_List1].substring(0, 3) + "**\t**");

}

/\*\*

\* **generates the routes weight and stores it so it can be compared**

\* **to the other routes later.**

\*/

route\_Weight = *generate\_Route\_Weight*(*city\_Nearest\_Route\_Weight\_List*);

System.*out*.printf("**\t**Route Wieght: %.2f **\n**", route\_Weight);

/\*\*

\* **adds the current routes weight to the list of the other routes**

\* **weights so it can be compared.**

\*/

*all\_Route\_Weights\_List*.add(route\_Weight);

/\*\*

\* **each time a route is added it will check if that route is the**

\* **best route against the other routes that are currently store in**

\* **the list.**

\*/

*generate\_Best\_Route*(*all\_Route\_Weights\_List*, *city\_Nearest\_Route\_List*);

}

/\*\*

\* **calls a method that will implement the second half of the algorithm**

\* **the alter the best route found to start and end with Stoke On Trent.**

\*/

*transpose\_Route\_Stoke*(*best\_Route\_List*);

/\*\*

\* **printing out the final best route to take in a normal that the user**

\* **can understand.**

\*/

System.*out*.print("===================================================================================================================================================================**\n**");

System.*out*.println("**\t\t\t\t\t\t** Best Route Using Repetitive Nearest Neighbour Algorithm ");

System.*out*.println("===================================================================================================================================================================**\n**");

System.*out*.printf("**\t\t\t\t** Route Order: " + *best\_Route\_List* + "**\t\t\t**Route Wieght: %.2f **\n\n** ", *best\_Weight*);

/\*\*

\* **a for loop that is go through each of the places on the route and get**

\* **the right name so that instead of numbers it will show the name of**

\* **the city.**

\*/

for (int o = 0; o < *best\_Route\_List*.size(); o++) {

if (o < *best\_Route\_List*.size() - 1) {

System.*out*.print(*city\_Name\_Array*[*best\_Route\_List*.get(o)] + " --> ");

} else {

System.*out*.print(*city\_Name\_Array*[*best\_Route\_List*.get(o)]);

}

}

System.*out*.println("**\n\n**===================================================================================================================================================================");

}

/\*\*

\* **a method that allows the user to set the starting city if they wanted to**

\* **travel from another city instead of Stoke On Trent.**

\*

\* **@param** starting\_City

\*/

private static void ***generate\_Nearesest\_Route***(int starting\_City) {

/\*

\* sets a global variable for the stating city so that other functions can user it without passing it into several different methods.

\*/

*start\_Location* = starting\_City;

/\*\*

\* **calls a method that will generate the route using the nearest**

\* **neighbour algorithm using the city related to the value of the**

\* **variable starting\_City.**

\*/

*generate\_Nearest\_Neighbour*(*city\_Distance\_Matrix*[starting\_City]);

}

/\*\*

\* **a method that implements the nearest neighbour algorithm when the data**

\* **for a city is entered (the 1d array from the 2d array containing all the**

\* **distance values).**

\*

\* **@param** city\_Data

\*/

private static void ***generate\_Nearest\_Neighbour***(double[] city\_Data) {

/\*\*

\* **creating variables that will be used in the calculations in**

\* **determining the route.**

\*/

double min\_Value = Double.*MAX\_VALUE*;

int next\_City = 0;

/\*\*

\* **a for loop that will run through all the different distances store in**

\* **the 1d array called city\_Data.**

\*/

for (int i = 0; i < city\_Data.length; i++) {

/\*\*

\* **checks if the value that is pulled from the array is not equal to**

\* **0 so that it doesn't use the value that is for a cities distance**

\* **from itself.**

\*/

if (city\_Data[i] != 0) {

/\*\*

\* **checks if the city that is not already in the route and is**

\* **not the starting city.**

\*/

if (!*city\_Nearest\_Route\_List*.contains(i) && i != *start\_Location*) {

/\*\*

\* **check is distance is lower than the min\_Value so get the**

\* **next city to visit.**

\*/

if (city\_Data[i] <= min\_Value) {

/\*\*

\* sets the min\_Value so that is can be compared to the

\* next value/s

\*/

min\_Value = city\_Data[i];

/\*\*

\* **sets the value of the next city to visit.**

\*/

next\_City = i;

}

}

}

}

/\*\*

\* **after getting the nearest route from the for loop above the values**

\* **are stored in the relevant lists.**

\*/

*city\_Nearest\_Route\_List*.add(next\_City);

*city\_Nearest\_Route\_Weight\_List*.add(min\_Value);

/\*\*

\* **checks if all the possible routes have been generated if they have**

\* **then it will add the values to the list and exit the method if they**

\* **have not then it will use recursion to generate the rest.**

\*/

if (*city\_Nearest\_Route\_List*.size() != 9) {

*generate\_Nearest\_Neighbour*(*city\_Distance\_Matrix*[next\_City]);

} else {

*city\_Nearest\_Route\_List*.add(*start\_Location*);

*city\_Nearest\_Route\_Weight\_List*.add(*city\_Distance\_Matrix*[*city\_Nearest\_Route\_List*.get(8)][*start\_Location*]);

}

}

/\*\*

\* **a method that will generate the weight (the cost) of each route so that**

\* **the best route can be chosen.**

\*

\* **@param** city\_Route

\* **@return**

\*/

private static double ***generate\_Route\_Weight***(ArrayList<Double> city\_Route) {

/\*\*

\* **a variable to store the weight while doing the calculations.**

\*/

double route\_Weight = 0;

/\*\*

\* **a for loop that will add each of the values for a route to get the**

\* **total weight of the route.**

\*/

for (Double city\_Route1 : city\_Route) {

route\_Weight += city\_Route1;

}

return route\_Weight;

}

/\*\*

\* **a method that will generate the best route by comparing all the best**

\* **routes when you start a different city and store the result in the global**

\* **variables.**

\*

\* **@param** all\_Route\_Weights

\* **@param** city\_Nearest\_Route

\*/

private static void ***generate\_Best\_Route***(ArrayList<Double> all\_Route\_Weights, ArrayList<Integer> city\_Nearest\_Route) {

/\*\*

\* **a for loop that is go trough all the elements of the**

\* **all\_Route\_Weights list.**

\*/

for (Double all\_Route\_Weight : all\_Route\_Weights) {

/\*\*

\* checks if the value is better than the current best weight if so

\* it will then assign it to the variable and get the route it came

\* from and save that in another variable

\*

\*/

if (all\_Route\_Weight < *best\_Weight*) {

*best\_Weight* = all\_Route\_Weight;

*best\_Route\_List* = new ArrayList<>(city\_Nearest\_Route);

}

}

}

/\*\*

\* **a method that will convert the best route from another city into a route**

\* **where you start and end at Stoke On Trent.**

\*

\* **@param** city\_Nearest\_Route

\*/

private static void ***transpose\_Route\_Stoke***(ArrayList<Integer> city\_Nearest\_Route) {

/\*\*

\* **a temp variable to store a routes city in while it is moved.**

\*/

int temp\_City = 0;

/\*\*

\* **a while loop that will check is the route has been transposed so that**

\* **9 (Stoke On Trent) is at the front.**

\*/

while (city\_Nearest\_Route.get(0) != 9) {

/\*\*

\* **puts the first element in the list into temp.**

\*/

temp\_City = city\_Nearest\_Route.get(0);

/\*\*

\* **a for loop that is go through all the elements in the list.**

\*/

for (int i = 0; i < city\_Nearest\_Route.size(); i++) {

/\*\*

\* **check if the there is any elements left to change.**

\*/

if (i == city\_Nearest\_Route.size() - 1) {

/\*\*

\* **sets the value of i in the city\_Nearest\_Route to the**

\* **value of temp\_city.**

\*/

city\_Nearest\_Route.set(i, temp\_City);

} else {

/\*\*

\* **sets the value of i in the city\_Nearest\_Route to value of**

\* **i+1 so that it swaps the position.**

\*/

city\_Nearest\_Route.set(i, city\_Nearest\_Route.get(i + 1));

}

}

}

/\*\*

\* **adds 9 to the end of the list so that you can see the route returning**

\* **to Stoke On Trent.**

\*/

city\_Nearest\_Route.add(9);

}

}

**Appendix D**

**References**

1. Applegate, David L. Bixby, Robert E. Chvatal, Vasek (September 2011). Princeton Series in Applied Mathematics: Traveling Salesman Problem: A Computational Study. Princeton: Princeton University Press. p1-p55.
2. http://www.people.vcu.edu/. (September 2011). The Tale of a Traveling Sales Person. Available: http://www.people.vcu.edu/~gasmerom/MAT131/hamilton.html. Last accessed 7th Apr 2014.
3. people.cs.aau.dk. (July 2013). Overview of the four main programming paradigms. Available: http://people.cs.aau.dk/~normark/prog3-03/html/notes/paradigms\_themes-paradigm-overview-section.html. Last accessed 7th Apr 2014.
4. wiki.haskell.org. (March 2015). *Lazy evaluation.* Available: https://wiki.haskell.org/Lazy\_evaluation. Last accessed 7th Apr 2014..
5. Mehrdad Afshari. (Aug 2009). Is Java a Compiled or an interpreted programming language?. Available: http://stackoverflow.com/questions/1326071/is-java-a-compiled-or-an-interpreted-programming-language. Last accessed 7th Apr 2014.
6. wiki.haskell.org. (). GHC / GHCi. Available: https://wiki.haskell.org/GHC/GHCi. Last accessed 7th Apr 2014.
7. eecs.ucf.edu. (May 1993). Major Programming Paradigms. Available: http://www.eecs.ucf.edu/~leavens/ComS541Fall97/hw-pages/paradigms/major.html. Last accessed 7th Apr 2014..