EDINBURGH NAPIER UNIVERSITY

SET09117 - Algorithms and Data Structures Coursework 2014

Vehicle Routing

40056161

11/17/2014

Contents

Introduction

This assignment requires the implementation of an algorithm to solve the vehicle routing problem.

The vehicle routing problem consists of a set of vehicles, a depot from where they originate and a set of customers each with a unique location. Each vehicle leaves the depot and visits a number of customers ensuring that all customers have received a visit. Each customer has a weight (requirement), and the 'vehicle' has a set capacity. When the capacity is full the vehicle must return to the depot.

The algorithm implemented by the author creates a 'master' route (Grand Tour) travelling to all customers disregarding capacity to begin with Fig 1Figure 1. The order in which the route takes is decided by the closest customer who has not already been visited. This approach was inspired by the Travelling Salesman problem, taking a 'nearest neighbour' approach[1]. One the route is concluded at the depot, the route is then iterated upon taking capacity into account. Once capacity cannot be added to any longer the route ends at the depot and a new route is created resuming the original 'master' routeFig2.

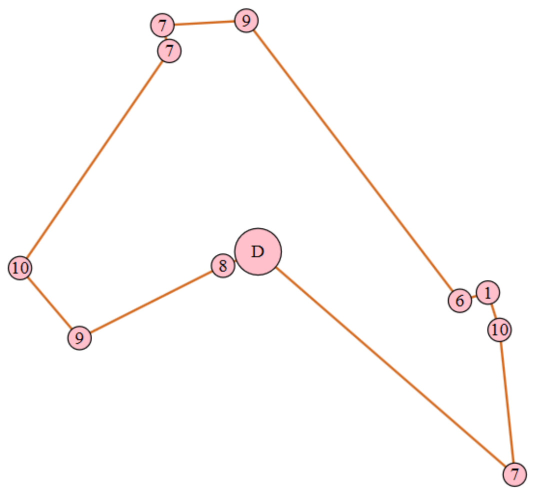

Figure 1-Route finding disregarding capacity.

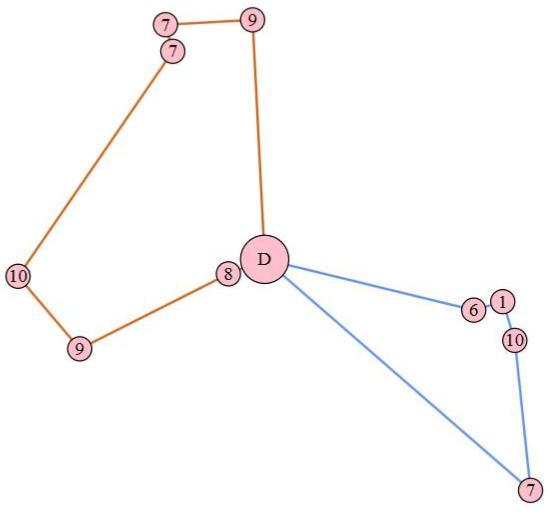

Figure 2-Route finding accounting for capacity.

This solution is simple in terms of complexity yet efficient. As each customer is passed it is actually removed from the list, meaning that the customer will no longer be accessed if it has already been seen. The customers list is then re-instated at the end of each iteration. The author expects the algorithm to have a balance between the time taken and the cost of each solution run. For problems with a larger size, it is expected that some routes will appear to be inefficient in some routes due to the nature of repeating the master route and then returning to depot when full. Unlike other algorithms, the author's never needs to remove or merge routes thus the number of routes created is kept to a minimum.

[1] Nearest Neighbour – Travelling Salesman Wikipedia

Method

After implementing a vehicle routing algorithm, it will be evaluated and the results will be recorded. The performance of the algorithm will be measured using provided data sets of varying sizes, ranging from a problem with 10 customers, to a problem with 100. The results will be compared against a ‘dumb’ algorithm which assigns every customer one route each.

The algorithm is run on the selected problems 50 times to establish sensible averages. The performance of the algorithm is measured using the built in Java function ‘System.nanoTime’ to record the time taken for the algorithm to be completed. The author expects the time to increase exponentially in regards to the size of the problem. All tests were carried out on the same hardware to ensure accuracy.

The cost of the algorithm will be measured a number of ways. Firstly against the ‘dumb’ algorithm to show that it is more efficient. The cost of a solution is the total distance travelled when completing the problem. The tests carried out run each algorithm over a number of solutions and outputs the results. Provided with the assignment is a set of test data which contain the results of a Clarke-Wright algorithm used to solve the problem. Clarke-Wright works differently from the author’s algorithm by calculating the savings between customers and then merging nodes to find the most efficient route[2]. The results of the algorithm will be used as a benchmark to compare the author’s algorithm. Although Clarke-Wright is by no means an optimal solution, it is a good target to aim for in terms of cost.

To check the validity of the implemented algorithm, a test is run to ensure that the algorithm never exceeds capacity and that no customer is missed out. Just to ensure that the tests operate as expected; invalid problems are included that will always fail. One problem is over capacity while the other misses a customer. These invalid tests will be failed whilst the other problems should all pass.

[2] Clarke-Wright Algorithm – Jens Lysgaard

Results

The performance time of the algorithm increases as the cost increases. As seen in the graph Fig 3, the performance appears to grow exponentially for the first few problems. However, at problem 800 the line deviates from this trend. Despite a few dips in the graph, the amount of time taken shows a linear pattern as the problem size increases. A problem with these results; it was not exactly clear how the algorithm would behave with larger problems.

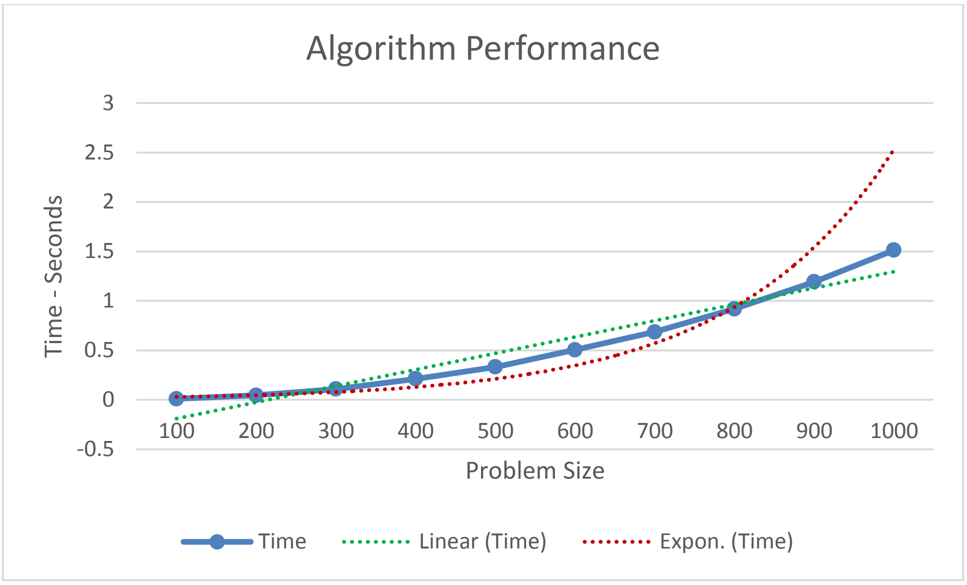

Figure 3 – Algorithm Performance.

The graph above does not clearly forecast the nature of the time vs problem size. To circumnavigate this issue, the author created an additional 5 problems and recorded the results. Below you can see that the performance is indeed linear in nature Fig 4. A problem of size 1000 takes 1.5 seconds and 3.5 seconds for a size of 1500.

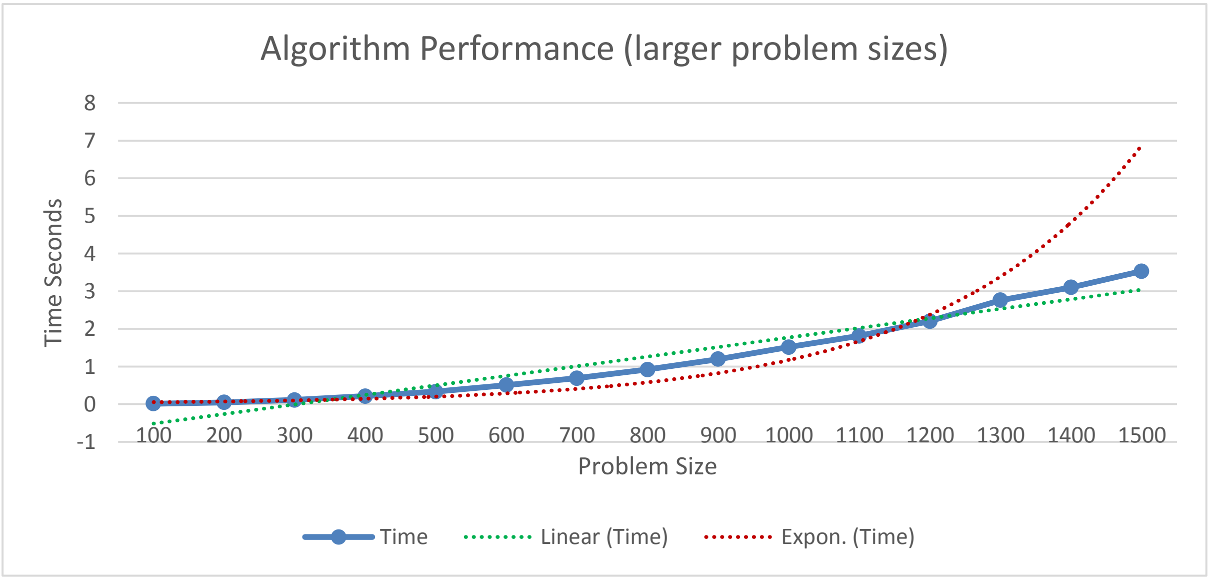

Figure 4 - Algorithm Performance using larger problem sizes

Since the 'dumb' algorithm assigns a separate route for each customer the result are linear with the cost increasing relative to the size of the problem. The author's algorithm cost is much lower than the 'dumb' algorithm. The minimum and maximum cost for each algorithm are shown below Fig 5.

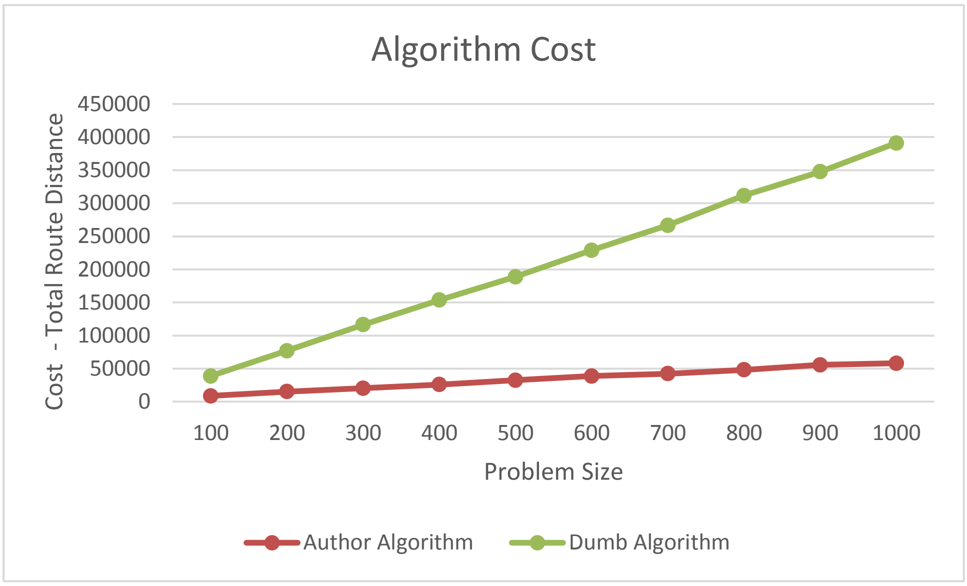

Figure 5 – Smallest and Largest problem size shown with the cost from each algorithm.

As you can see the 'dumb' algorithm increases proportionally to size of the problem. This is easily observed when displaying the maximum algorithm cost and the minimum Fig 6.

Problem Size	'Dumb' Algorithm	Author's Algorithm
100	38703.90	8809.50
1000	390783.40	58062.76

Figure 6 – Algorithm Cost.

When compared against Clarke-Wright, the author's algorithm cost was only slightly higher. Some problem sizes were better solved by the author's algorithm but on a whole the Clarke-Wright implementation was more successful. A comparison of costs can be seen below [Fig 7].

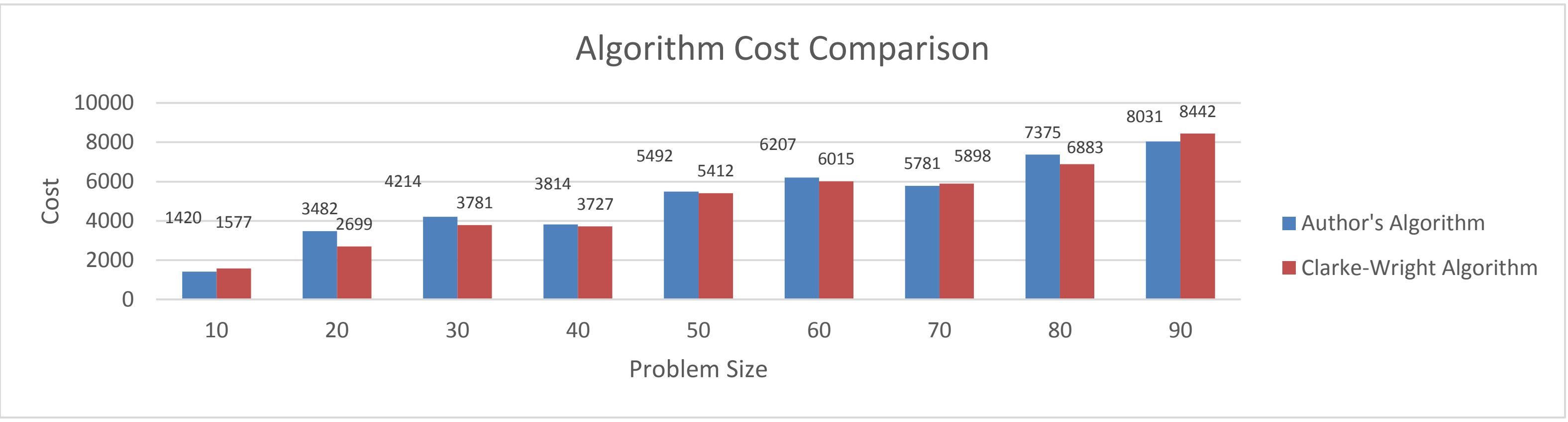

Figure 7 – Algorithm Cost Comparison

The tests passed as expected and the invalid problems were failed [Fig 8]. The reason for the customer failing was printed to the output log.

Problem	Soln	Size	Cost	Valid
rand00010	MyAlg	10	1420	TRUE
rand00020	MyAlg	20	3482	TRUE
rand00030	MyAlg	30	4214	TRUE
rand00040	MyAlg	40	3814	TRUE
rand00050	MyAlg	50	5492	TRUE
rand00060	MyAlg	60	6207	TRUE
rand00070	MyAlg	70	5781	TRUE
rand00080	MyAlg	80	7375	TRUE
rand00090	MyAlg	90	8031	TRUE
rand00100	MyAlg	100	8850	TRUE
rand00200	MyAlg	200	15120	TRUE
rand00300	MyAlg	300	20534	TRUE
rand00400	MyAlg	400	26027	TRUE
rand00500	MyAlg	500	32972	TRUE
rand00600	MyAlg	600	38733	TRUE
rand00700	MyAlg	700	42204	TRUE
rand00800	MyAlg	800	48211	TRUE
rand00900	MyAlg	900	55408	TRUE
FAIL Route starting Point2D.Double[200.0, 200.0] is over capacity 70	MyAlg	6	666	FALSE
FAIL Customer at 300.000000x200.000000 has 10 left over	MyAlg	6	524	FALSE

Figure 8 – Testing the Author's Algorithm. Invalid problems failed the test with appropriate error messages.

Conclusion

Even though the implemented algorithm is a relatively simple solution, it is shown to have a good balance between performance (time) and the overall result (total cost). The results showed that the algorithm's performance was good with a problem with a size of 1500 being completed in 3.5 seconds. The algorithm cost is close to the benchmark Clarke-Wright results provided.

The author believes that the algorithm could be refined to increase performance. After visually inspecting the created SVG images it can be seen that some routes are vastly inefficient.

The Clarke-Wright test data only included problems up to a size of 90, the author would be interested to compare results on larger problems. It is expected that the Clarke-Wright would have an overall lower cost than the author's implementation, although the author believes that his solution would have a significantly better performance time.

The author initially began to implement his own Clarke-Wright algorithm. They found this difficult and opted for a solution which would eliminate the need to create and then delete routes (Clarke-Wright merges routes). The first step was first establishing a Grand Tour approach to act as a master route After this had been done successfully, the author could then go on to complete the algorithm.

Appendix

References

Travelling Salesman – Nearest Neighbour

http://en.wikipedia.org/wiki/Travelling_salesman_problem

Clarke-Wright

http://pure.au.dk/portal-asb-student/files/36025757/Bilag_E_SAVINGSNOTE.pdf

Result Data

The data used to create the graphs can be found in the attached Excel workbook – 40056161.xlsx

Source Code

mySolver() - method

```
// Author's Solution - 40056161
    public void mySolver() {

        this.route = new ArrayList<Customer>(); // Master route
        this.newRoute = new ArrayList<Customer>();
        this.soln = new ArrayList<List<Customer>>();

        ArrayList<Customer> temp_customers = new ArrayList<Customer>(prob.customers);
        ArrayList<CustomerDistance> distances = new ArrayList<CustomerDistance>();

        // Sort Customers by distance from depot, smallest distance first.
        // Used to establish the first customer to visit
        for (Customer c : prob.customers) {
            CustomerDistance cd = new CustomerDistance(c,c.distance(prob.depot));
            distances.add(cd);
        }
        //Sorts customers by smallest distance first
        Collections.sort(distances);

        // Adds the customer to the route
        route.add(distances.get(0).customer);
        // findRoute() - recursive function to establish the 'master route'
        findRoute(distances.get(0).customer);

        //Iterates through the master route accounting for capacity
        for (Customer c : route) {
            if (c.c + count <= prob.depot.c) {
                newRoute.add(c);
                count += c.c;
            } else if (c.c + count > prob.depot.c) {
                // Finish existing route
                soln.add(newRoute);
                // Reset the capacity count and route
                count = 0;
                count += c.c;
                newRoute = new ArrayList<Customer>();
            }
            if (!(newRoute.contains(c))) {
                newRoute.add(c);
            }
        }
        soln.add(newRoute);

        // Reset list of customers
        prob.customers = temp_customers;

    }
```

findRoute() - method

```
public void findRoute(Customer source) {
        // Used to create the initial 'master' route
        // Finds the nearest customer to the source customer
        ArrayList<CustomerDistance> distances = new ArrayList<CustomerDistance>();
        for (Customer c : prob.customers) {
            CustomerDistance cd = new CustomerDistance(c, c.distance(source));
            distances.add(cd);
        }
        Collections.sort(distances);
        // Adds nearest customer to the route then removes the source customer
        //(temp_customers is used to reset the customer list after problem is solved)
        route.add(distances.get(1).customer);
        prob.customers.remove(distances.get(0).customer);
        if (prob.customers.size() > 1) {
            findRoute(distances.get(1).customer);
        }
    }
```

CustomerDistance - class

```
package vrp;

public class CustomerDistance implements Comparable {
    // Each customer has a list of the distances between itself
    // and every other customer
    public Customer customer;
    public double distance;

    public CustomerDistance(Customer c, double d) {
        customer = c;
        distance = d;
    }

    @Override
    // Comparator used to return the smallest distance
    public int compareTo(Object arg0) {
        if (distance > ((CustomerDistance) arg0).distance) {
            return 1;
        } else if (((CustomerDistance) arg0).distance > distance) {
            return -1;
        }
        return 0;
    }
}
```