Clarke-Wright vehicle routing algorithm Implementation Report

Sam Serrels
40082367@napier.ac.uk
Edinburgh Napier University
Algorithms and Data Structures (SET09117)

1 Introduction

Vehicle Routing Finding the optimal route for a delivery truck, delivering items of varying size to customers spread over a large area, is a complex and computationally intensive task. The problem is similar to the travelling salesman problem, where the task is to find the shortest route to visit every location within a set only once.
The Delivery truck problem extends this with the additional factors that the delivery trucks have a capacity limit to how much they can carry, and thus how many of the customers can be served by a single truck.

Multiple Trucks The capacity limitations means that multiple routes must be generated. If delivery time is a factor, then multiple trucks could be driving simultaneously, conversely a single truck could go round one route, resupply at the depot, then drive around another route. This report is not taking delivery time into into account, only the number of routes, and the combined cost of all the routes. The truck capacity is assumed to be the same for all trucks.

Multiple routes Generating the optimal path for multiple routes is no longer a simple optimisation task, multiple approaches can be taken and often the solution chosen is not the mathematically perfect, but chosen as a trade-off between efficiency and computation time. Assigning one customer to one route means taking into account the effects this will have for the other routes. A brute force method could be taken, to process all the possible route combinations, or a saving algorithm could be used.

G. Clarke and J. W. Wright's Saving Algorithm The savings algorithm implemented and tested in this report is the Clarke-Wright algorithm, first described in the 1962 paper "Scheduling of vehicles from a central depot to a number of delivery points" [Clarke and Wright 1962]. The algorithm finds a solution to the problem in a heuristic manor, so the result will not be the perfect solution, but should be reality close to perfect with substantially less computation time than a brute force attempt.

Customer Pairs The Algorithm first starts by calculating the "savings" of delivery to a single pair of customers, rather than each customers on two trips. This is done for every combination of customers, resulting in a list or every customer pair and the corresponding savings. This list is sorted by the savings, in descending order.

Assigning Pairs to routes The first pair forms the start of the first route. The sorted list of pairs is then iterated through, if a pair can be added to the start or end of the route, and the route does not contain both elements of the pair already, and the combined route would not be over capacity of a truck, then the pair is added to the route. If a pair is encountered that has no elements already in a route, then this would form the start of a new route. How this is handled in execution depends on which version of the Clarke-Wright algorithm is implemented.

Parallel and Sequential methods There are two ways to implement the Clarke-Wright algorithm, they both follow the same logic, but the order in which the task are carried out is different. This is described in the 1997 paper "Clarke and Wright's Savings Algorithm"[Lysgaard 1997].
The difference lies in the situation where a pair is encountered that could form a new route. In the sequential method, a new route is formed, but it is stored until all other pairs have been compared with the original route, then the remaining pairs are iterated though again but compared to the new route. This is repeated for any new route that is formed. For the parallel method, when a new route is created, the iteration continues through the list of pairs, but each one is checked against all routes in the same cycle of iteration.

Resulting difference The sequential method tends to generate an initial large route, and each subsequent route will be smaller. The Parallel method tends to have more larger routes and therefore less routes overall. Both methods are implemented and analysed in this report.

Problem summary including any limitations of your solution.

2 Method

Testing validity of algorithms Each solution produced by the algorithms was tested by looping through the routes, checking that each customer was visited and that the correct quantity was delivered. The total quantity of deliveries for each route was also calculated, assuring that no route was over the capacity of the truck.

Output The generated routes were exported as a .csv data file, and a visualisation of the routes were also generated and saved as a .svg vector image format. These images are included in the report and can bee seen in the results section.

Testing solution quality A basic solution in which a single truck was allocated to every customer was created, forming the most ineffective solution possible. This was used as the baseline set of data that the Clarke-Wright solutions were compared against. The solution cost and total number of routes for each algorithm were compared to each other and to the baseline numbers.

Testing solution computation time Test were carried to measure the relative computation time for each algorithm with an increasing set of customers. The tests were carried out fifty times and the results averaged to mitigate external factors.

Description of the experiment that you conducted.

3 Results

Tables and charts Tables and charts that show the performance of your solution (how long does it take to run your algorithm). Demonstration that your solution is valid. Demonstration of the quality of your solution (what is the cost).

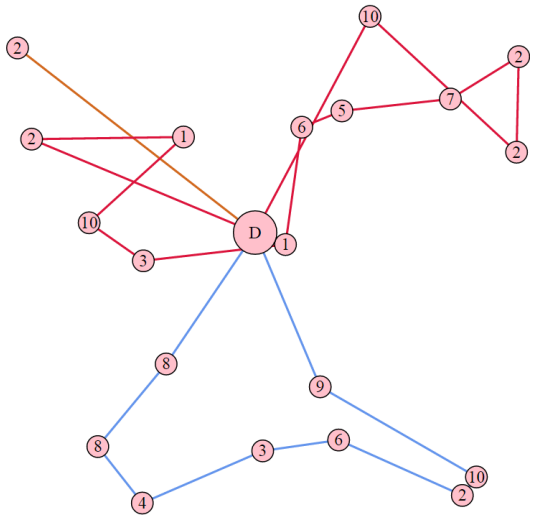

Figure 1: ***Parallel Algorithm with 20 Customers*** -

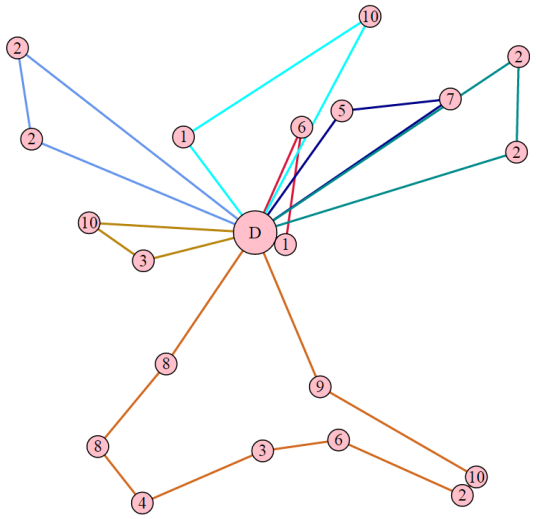

Figure 2: ***Sequential Algorithm with 20 Customers*** -

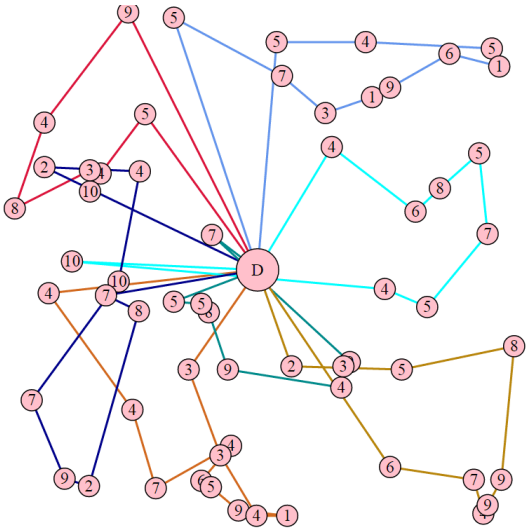

Figure 3: ***Parallel Algorithm with 50 Customers*** -

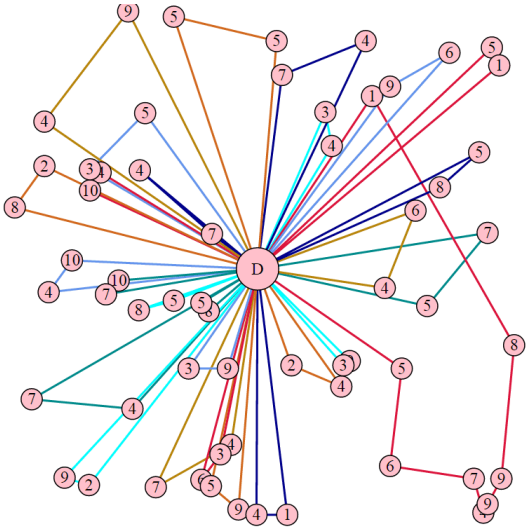

Figure 4: ***Sequential Algorithm with 50 Customers*** -

4 Conclusions

Both Implementations of the Clarke-Wright algorithms produced expected results, with the parallel version producing larger and fewer routes. As for the time taken to calculate, the performance is roughly equal.

Computation time The discrepancies shown in Figure 5 when the amount of customers increases beyond 800 is possibly due to optimisations carried out by the Java virtual machine. The total operations carried out is roughly the same for each algorithm, however the arrays are accessed and modified at different times, this is a possible cause for the difference in processing time.

Solution Quality The Parallel solution produced a large saving of up to a 600% increase against the baseline cost, shown in Figure 6. The Sequential solution produced a constant saving of around 200%. These results are also shown in Figure 7, showing the number of routes.

Summary of your results. Summary of your results. Reflection on your performance on this assessment.

5 Appendix

References

CLARKE, G., AND WRIGHT, J. 1962. Scheduling of vehicles from a central depot to a number of delivery points. *Operations Research 12*, 4, 568–581.

LYSGAARD, J. 1997. Clarke and wright's savings algorithm http://pure.au.dk/portal-asb-student/files/36025757/bilag_e_savingsnote.pdf. *Department of Management Science and Logistics, The Aarhus School of Business*.

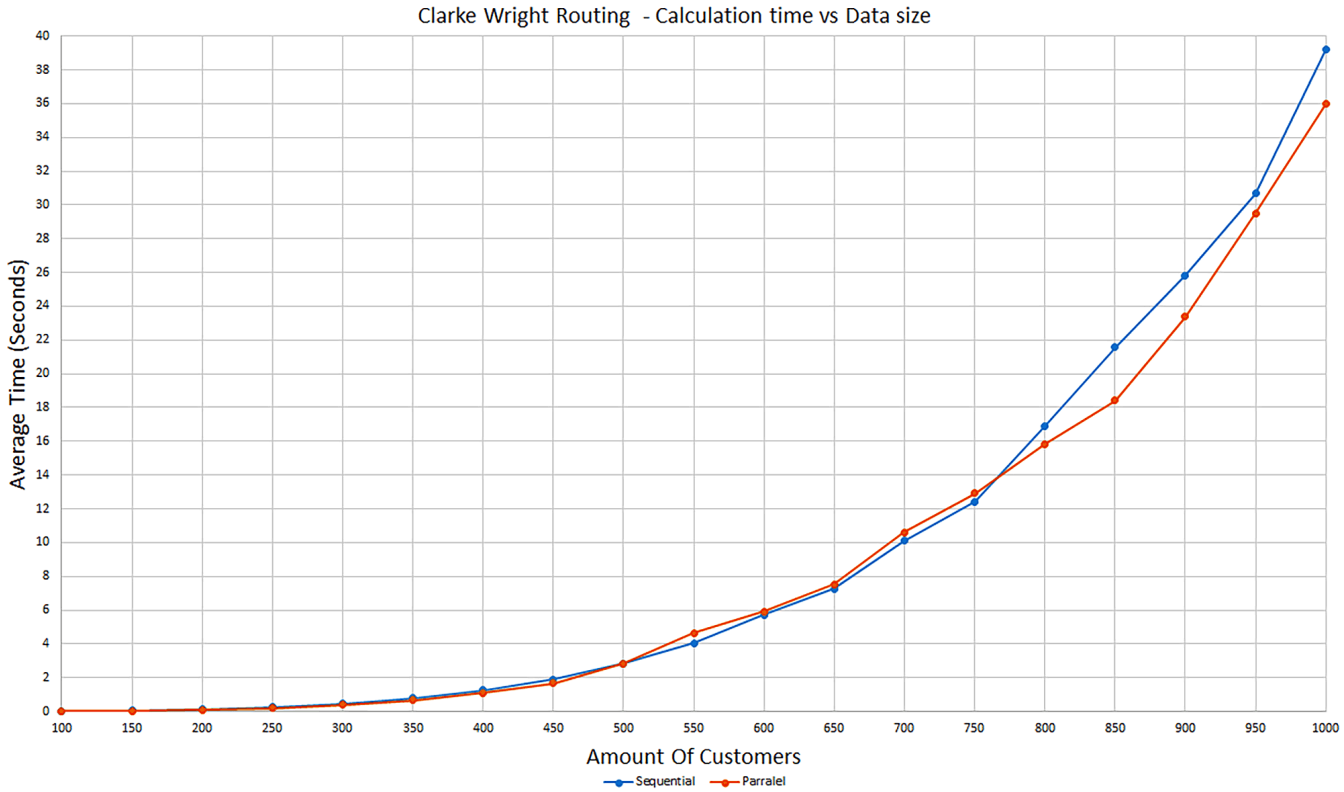

Figure 5: ***Clark Wright implementation results*** *- Average time to process each algorithm with an increasing amount of customers*

Number of Customers	Dumb solution cost	Sequential Cost	Parralel Cost	Sequential Routes	Parralel Routes	Sequential Time (Seconds)	Parralel Time (Seconds)
10	3781	1955	2027	2	2	0.000909157	0.001013164
20	7931	3749	3019	7	3	0.001404089	0.001417152
30	10302	5576	4717	11	5	0.000896818	0.000497884
40	15775	7271	4562	15	5	0.000736218	0.000602604
50	20073	9746	5753	21	7	0.001405751	0.00112024
60	23194	11918	6183	27	7	0.002299655	0.001860301
70	25925	12783	7293	31	8	0.003710384	0.002752865
80	28757	14238	7813	35	10	0.005782794	0.004625466
90	34580	17543	9202	41	11	0.011383535	0.008565139
100	38704	19375	9854	46	13	0.012820712	0.011544076
150	58323	29787	14632	71	18	0.042564373	0.035033029
200	76929	38920	17809	96	23	0.112553256	0.087635926
250	94218	47177	19586	119	28	0.236658078	0.19809618
300	116481	58912	22549	145	32	0.458011761	0.3729932
350	132529	67566	25634	171	37	0.795852936	0.655195371
400	153609	78127	31412	196	46	1.242856913	1.10756995
450	174708	88128	32206	220	51	1.909523115	1.674798974
500	188740	96343	35368	246	58	2.837620963	2.831905538
550	208973	105802	37923	270	63	4.026630142	4.657466973
600	228962	116788	43113	296	71	5.720755786	5.927701178
650	246924	126412	42089	322	72	7.251799771	7.534708523
700	266675	135277	44760	345	79	10.09592432	10.61137241
750	283723	143957	48740	370	82	12.4131069	12.93193556
800	311675	158686	51313	396	88	16.92304974	15.81973182
850	329742	168032	55694	422	97	21.56937135	18.40950886
900	347699	177212	59699	446	106	25.80634481	23.36173086
950	361993	184769	59529	472	105	30.69257549	29.51002702
1000	390783	199258	61701	497	109	39.20094816	35.98109894

Table 1: *Results of all tests carried out on both versions of the Clarke Wright algorithm*

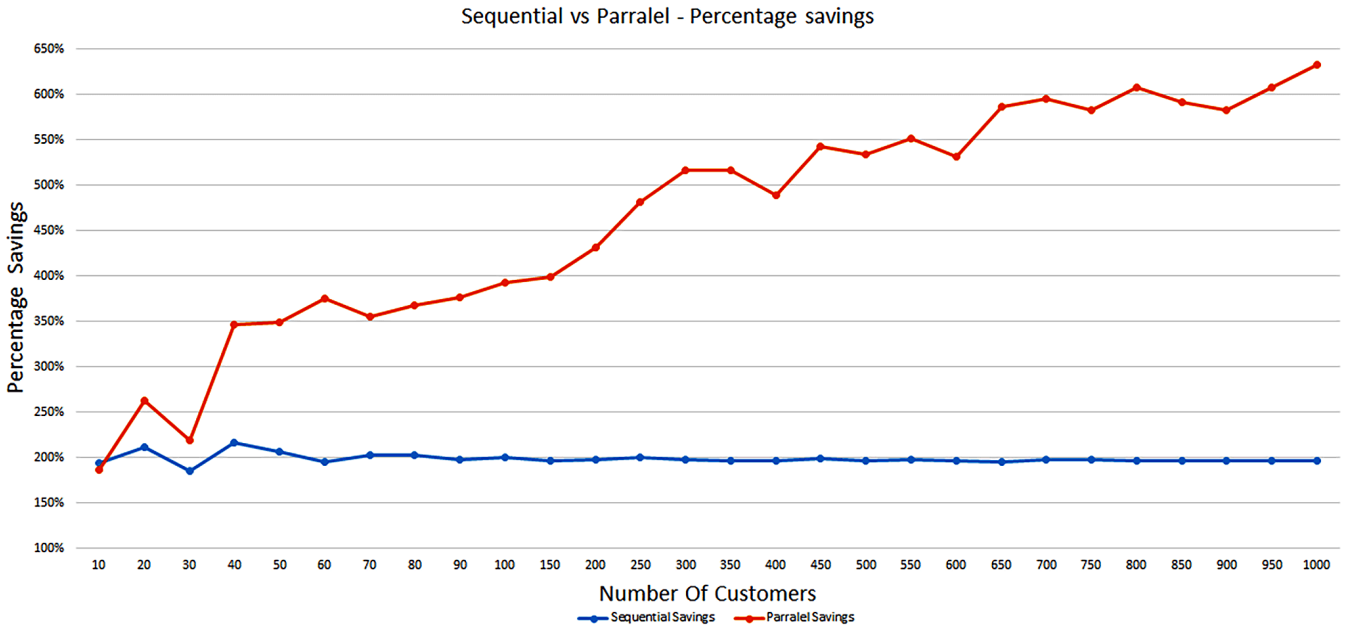

Figure 6: ***Clark Wright Percentage Savings*** *- The Percentage distance cost saved by each algorithm, based from sending a single truck to each customer*

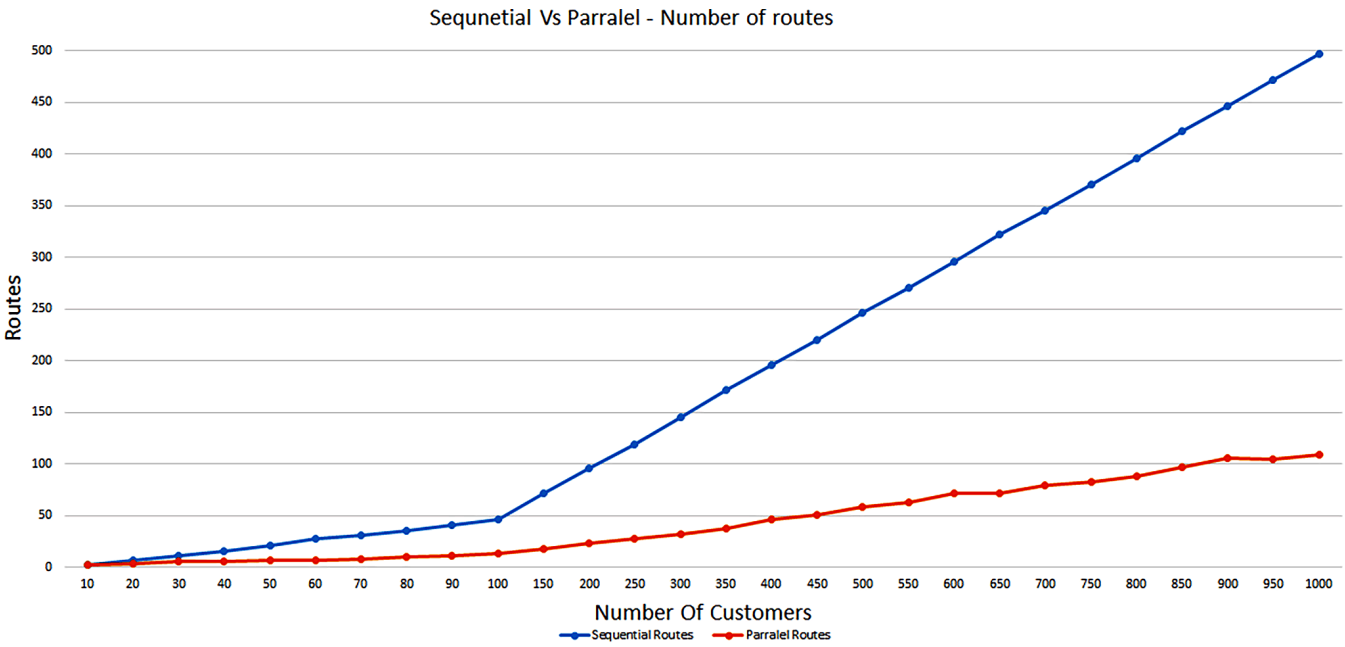

Figure 7: ***Clark Wright Routes*** *- Number of routes required by each algorithm. X-axis is not a uniform scale.*

6 Code

6.1 ClarkeWright.java

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.HashSet;
import java.util.List;

class Route implements Comparable<Route>
{
   private int _capacity;
   private int _weight;
   private double _cost;
   private double _savings;
   public ArrayList<Customer> customers;

   private void calculateSavings(){
      double originalCost = 0;
      double newCost = 0;
      double tempcost =0;
      Customer prev = null;

      //Foreach customer in the route:
      for(Customer c:customers){
         // Distance from Depot
         tempcost = Math.sqrt((c.x*c.x)+(c.y*c.y));
         originalCost += (2.0*tempcost);

         if(prev != null){
            // Distance from previous customer to this customer
            double x = (prev.x - c.x);
            double y = (prev.y - c.y);
            newCost += Math.sqrt((x*x)+(y*y));
         }else{
            //If this is the first customer in the route, no change
            newCost += tempcost;
         }
         prev = c;
      }
      newCost += tempcost;
      _cost = newCost;
      _savings = originalCost - newCost;
   }

   public Route(int capacity){
      _capacity = capacity;
      customers = new ArrayList<Customer>();
      _weight =0;
      _cost= 0;
      _savings =0;
   }

   public void addCustomer(Customer c, boolean order){
      //Add customer to the start or end of the route?
      if(order){
         customers.add(0,c);
      }else{
         customers.add(c);
      }

      if(c.c > _capacity){
         System.out.println("Customer order too large");
      }

      _weight += c.c;

      if(_weight > _capacity){
         System.out.println("Route Overloaded");
      }

      calculateSavings();
   }

   public double getSavings(){
      return _savings;
   }
   public double getCost(){
      return _cost;
   }
   public int getWeight(){
      return _weight;
   }
   public int compareTo(Route r) {
      return Double.compare(r.getSavings(), this._savings);
   }

}

//
//##Sequential solver##
//

public class ClarkeWright
{
   public static int truckCapacity = 0;

   public static ArrayList<List<Customer>> solve(ArrayList<Customer> customers){
      ArrayList<List<Customer>> solution = new ArrayList<List<Customer>>();
      HashSet<Customer> abandoned = new HashSet<Customer>();

      //calculate the savings of all the pairs
      ArrayList<Route> pairs = new ArrayList<Route>();

      for(int i=0; i<customers.size(); i++){
         for(int j=i+1; j<customers.size(); j++){
            Route r = new Route(truckCapacity);
            r.addCustomer(customers.get(i),false);
            r.addCustomer(customers.get(j),false);
            pairs.add(r);
         }
      }
      //order pairs by savings
      Collections.sort(pairs);

      HashSet<Route> routes = new HashSet<Route>();
      routes.add(pairs.get(0));
      pairs.remove(0);

      //start combining pairs into routes
      for(Route ro :routes)
      {
         outerloop: for(int i=0; i<pairs.size(); i++){
            Route r = pairs.get(i);
            Customer c1 = r.customers.get(0);
            Customer c2 = r.customers.get(r.customers.size()-1);
            Customer cr1 = ro.customers.get(0);
            Customer cr2 = ro.customers.get(ro.customers.size()-1);

            boolean edge = false;
            for(int a=0; a<2;a++)
            {
               edge = !edge;
               Customer e1 = (!edge) ? c1 : c2;
               Customer e2 = (edge) ? c1 : c2;
               //do they have any common nodes?
               if(e1 == cr1 ||e1 == cr2){
                  //could we combine these based on weight?
                  if(e2.c + ro.getWeight() <= truckCapacity){
                     //Does route already contain BOTH these nodes?
                     if(!ro.customers.contains(e2)){
                        //no, but is it in another route already?
                        boolean istaken = false;
                        for(Route rr :routes)
                        {
                           if(rr.customers.contains(e2)){
                              istaken = true;
                              break;
                           }
                        }
                        if(!istaken){
                           //No other route have this, add to route
                           if(e1 == cr1){
                              ro.addCustomer(e2, true);
```

```
                        }else{
                            ro.addCustomer(e2, false);
                        }
                    }
                }
                abandoned.remove(e2);
                pairs.remove(r);
                i--;
                continue outerloop;
            }
        }
    }

    //If we reach here, the pair hasn't been added to any routes
    boolean a = false;
    boolean b = false;
    for(Route rr :routes){
        if(rr.customers.contains(c1)){
            a = true;
        }
        if(rr.customers.contains(c2)){
            b = true;
        }
    }
    if(!(a||b)){
        //no routes have any of these customers, make new route
        abandoned.remove(c1);
        abandoned.remove(c2);
        routes.add(r);
    }else{
        //Some routes have some of these customers already
        if(!a){
            abandoned.add(c1);
        }
        if(!b){
            abandoned.add(c2);
        }
    }
    pairs.remove(r);
    i--;
  }

}

//A Customer can be left over due to capacity constraints
outerloop:for(Customer C:abandoned){
  //we could tack this onto the end of a route if it would fit
  for(Route r:routes){
    if(r.getWeight() + C.c < truckCapacity)
    {
      //would this be more efficient than sending a new truck?
      Customer[] cca={r.customers.get(r.customers.size()-1),
      r.customers.get(0)};
      for(Customer cc:cca){
        double X = C.x - cc.x;
        double Y = C.y - cc.y;
        if(Math.sqrt((X*X)+(Y*Y))
        < Math.sqrt((C.x*C.x)+(C.y*C.y)))
        {
          r.addCustomer(C, false);
          break outerloop;
        }
      }
    }
  }

  //Send a new truck, just for this Customer
  ArrayList<Customer> l = new ArrayList<Customer>();
  l.add(C);
  solution.add(l);
}

//output
for(Route r:routes){
  ArrayList<Customer> l = new ArrayList<Customer>();
  l.addAll(r.customers);
  solution.add(l);
}
return solution;
}

//
//##Parallel solver##
//

public static ArrayList<List<Customer>> solveP(ArrayList<Customer> customers){
  ArrayList<List<Customer>> solution = new ArrayList<List<Customer>>();
  HashSet<Customer> abandoned = new HashSet<Customer>();

  //calculate the savings of all the pairs
  ArrayList<Route> pairs = new ArrayList<Route>();

  for(int i=0; i<customers.size(); i++){
    for(int j=i+1; j<customers.size(); j++){
      Route r = new Route(truckCapacity);
      r.addCustomer(customers.get(i),false);
      r.addCustomer(customers.get(j),false);
      pairs.add(r);
    }
  }
  //order pairs by savings
  Collections.sort(pairs);

  HashSet<Route> routes = new HashSet<Route>();
  routes.add(pairs.get(0));
  pairs.remove(0);

  //start combining pairs into routes
  outerloop: for(int j=0; j<pairs.size(); j++){
    Route r = pairs.get(j);
    Customer c1 = r.customers.get(0);
    Customer c2 = r.customers.get(r.customers.size()-1);

    for(Route ro :routes)
    {
      Customer cr1 = ro.customers.get(0);
      Customer cr2 = ro.customers.get(ro.customers.size()-1);
      boolean edge = false;
      for(int a=0; a<2;a++)
      {
        edge = !edge;
        Customer e1 = (!edge) ? c1 : c2;
        Customer e2 = (edge) ? c1 : c2;
        //do they have any common nodes?
        if(e1 == cr1 || e1 == cr2){
          //could we combine these based on weight?
          if(e2.c + ro.getWeight() <= truckCapacity){
            //Does route already contain BOTH these nodes?
            if(!ro.customers.contains(e2)){
              //no, but is it in another route already?
              boolean istaken = false;
              for(Route rr :routes){
                if(rr.customers.contains(e2)){
                  istaken = true;
                  break;
                }
              }
              if(!istaken){
                //No other route have this, add to route.
                if(c1 == cr1){
                  ro.addCustomer(e2, true);
                }else{
                  ro.addCustomer(e2, false);
                }
              }
            }
            abandoned.remove(e2);
            pairs.remove(r);
            j--;
            continue outerloop;
          }
        }
      }
    }

  }
```

```
306
307         //If we reach here, the pair hasn't been added to any routes
308         boolean a = false;
309         boolean b = false;
310         for(Route ro :routes){
311            if(ro.customers.contains(c1)){
312               a = true;
313            }
314            if(ro.customers.contains(c2)){
315               b = true;
316            }
317         }
318         if(!(a||b)){
319            //no routes have any of these customers, make new route
320            abandoned.remove(c1);
321            abandoned.remove(c2);
322            routes.add(r);
323         }else{
324            //Some routes have some of these customers already
325            if(!a){
326               abandoned.add(c1);
327            }
328            if(!b){
329               abandoned.add(c2);
330            }
331         }
332         pairs.remove(r);
333         j--;
334
335      }
336
337      //A Customer can be left over due to capacity constraints
338      outerloop:for(Customer C:abandoned){
339         //we could tack this onto the end of a route if it would fit
340         for(Route r:routes){
341            if(r.getWeight() + C.c < truckCapacity)
342            {
343               //would this be more efficient than sending a new truck?
344               Customer[] cca={r.customers.get(r.customers.size()-1),
345               r.customers.get(0)};
346               for(Customer cc:cca){
347                  double X = C.x - cc.x;
348                  double Y = C.y - cc.y;
349                  if(Math.sqrt((X*X)+(Y*Y))
350                  < Math.sqrt((C.x*C.x)+(C.y*C.y)))
351                  {
352                     r.addCustomer(C, false);
353                     break outerloop;
354                  }
355               }
356            }
357         }
358
359         //Send a new truck, just for this Customer
360         ArrayList<Customer> l = new ArrayList<Customer>();
361         l.add(C);
362         solution.add(l);
363      }
364
365      //output
366      for(Route r:routes){
367         ArrayList<Customer> l = new ArrayList<Customer>();
368         l.addAll(r.customers);
369         solution.add(l);
370      }
371      return solution;
372   }
373 }
```

6.2 VRSolution.java

Lines 20 to 28

```
1
2    //Students should implement another solution
3    public void clarkeWrightSolution(boolean b){
4       ClarkeWright cw = new ClarkeWright();
5       cw.truckCapacity = prob.depot.c;
6       if(b){
7          this.soln = cw.solveP(prob.customers);
8       }else{
```

6.3 Experiment.java

```
1 import java.util.*;
2 public class Experiment {
3
4    public static void main(String[] args)throws Exception{
5       String outdir = "output/";
6       String problemdir = "tests/";
7       String [] probs = {
8             "rand00010",
9             "rand00020",
10            "rand00030",
11            "rand00040",
12            "rand00050",
13            "rand00060",
14            "rand00070",
15            "rand00080",
16            "rand00090",
17            "rand00100",
18            "rand00150",
19            "rand00200",
20            "rand00250",
21            "rand00300",
22            "rand00350",
23            "rand00400",
24            "rand00450",
25            "rand00500",
26            "rand00550",
27            "rand00600",
28            "rand00650",
29            "rand00700",
30            "rand00750",
31            "rand00800",
32            "rand00850",
33            "rand00900",
34            "rand00950",
35            "rand01000",
36            "fail00002",
37            "fail00004"
38            };
39       for (String f:probs){
40          VRProblem vrp = new VRProblem(problemdir+f+"prob.csv"↩
          );
41          VRSolution vrs = new VRSolution(vrp);
42          System.out.print(vrp.size()+",");
43          for(int i=0;i<50;i++){
44             long start = System.nanoTime();
45             vrs.clarkeWrightSolution(true);
46             long delta = System.nanoTime()-start;
47             System.out.print(delta+",");
48          }
49          System.out.print("\n");
50       }
51
52    }
53 }
```