Circle Space Partitioning and Clarke-Wright

40111906 *
Edinburgh Napier University
Algorithms and Data Structures (SET09117)

1 Introduction

The aim of this project was to implement an algorithm that would solve the vehicle routing problem (VRP) on a set of supplied sample problems. Each problem consists of a depot surrounded by a number of nodes (known as customers in this scenario). The depot states the capacity that each path/route has and each customer has a requirement that must be fulfilled. In order for the solution to be valid, each customer must be included in one route and each route must not exceed its capacity (the sum of the customer requirements must remain lower than the capacity of the route). This report covers the two algorithms implemented in order to solve the problem along with how they compare to each other and the supplied sample solutions.

The two methods implemented were Clarke-Wright, which was researched [Toth and Vigo 2001] prior to implementation, and Circle Space Partitioning, which was developed specifically for this project but is by no means the first of its kind ([Alvarenga et al. 2007] as an example of prior work in the area). Both of these methods of solving the VRP return valid solutions but each is better and worse than the other in certain aspects. The Circle Space Partitioning algorithm (see subsection 2.1) splits the problem into multiple smaller problems until these problems are easy to manage, it then improves upon the ordering of the nodes within the problem to arrive at a solution. The Clarke-Wright algorithm (see subsection 2.2) takes all the nodes within the problem and finds savings between them. If there are savings to be made, the nodes are merged until a solution is found. These are just the basic workings of the algorithms and the entire process of each will be covered in more detail in section 2.

It is expected that the Circle Space Partitioning algorithm will out perform the Clarke-Wright algorithm in terms of speed due to how it simplifies the problem into multiple smaller problems but the cost will likely not be as good since it will find nodes that are near each other rather finding the optimal way to group them together.

2 Methods

This section will cover both of the algorithms implemented within the project in detail. In order to keep the results of each algorithm accurate and reliable, each follows a set of instruction and rules and there is no use of random number generation to create steps that may or may not run each time. This means that each time either algorithm runs, it runs the same as the last and the next time at least in terms of solution cost since the time to run each algorithm may vary depending on the cpu time the algorithm gets allocate.

2.1 Circle Space Partitioning

This solution to the vehicle routing problem was inspired by the broad phase stage of collision detection. The basic idea of broad phase collision detection is to repeatedly splice a group of objects into smaller sets until the sets only contain a small number of objects in comparison to the size of the overall problem. The process of the circle space partitioning algorithm is very similar to the process of the broad phase. To begin with, a large circle is created that encompasses all nodes Figure 1 (for this example, a simple group of nodes will be used).

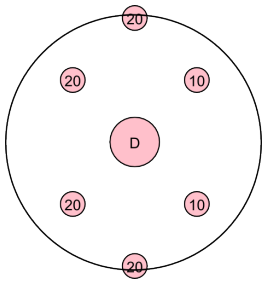

Figure 1: *Start with a large circle that encompasses all nodes. (D is the depot and the number in the other nodes is the requirement)*

The next step is iterative and is looped until all circles within the solution are under the capacity limit of the problem. For each circle currently within the solution, pick the farthest node away from the center of the current circle and create a new circle between the current one and the position of the selected node with a radius just large enough to reach the selected node. With this new circle, try to steal nodes from all other circles until the new circle cannot steal any more nodes without going over capacity or until there are no more to steal. Only steal a node from another circle if the new circle's center is closer to the node than the other circle's center is. After each loop through the circles, two other steps occur. Firstly, the circles are looped through to see if they can be merged with any other circles if the sum of both of their node's requirements is less than the capacity set by the problem and the two circles overlap. Secondly, the circles are looped through again, repositioned between their current position and the position of their farthest node and then shrunk down like before (see Figure 2 for the results of this problem after this iterative step). The entire step is then iterated over again if there are still circles over the capacity limit.

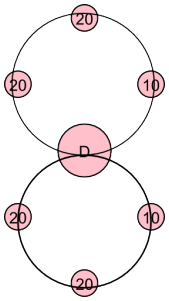

Figure 2: *New circle is created with the farthest node as the focus. All circles are repositioned and resized. (D is the depot and the number in the other nodes is the requirement)*

*e-mail:40111906@live.napier.ac.uk

Once all circles are under the capacity limit, a process similar to that of the Clarke-Wright method is ran over each circle in order to improve the order in which each node is stored within the circle. The process of the Clarke-Wright method is covered in detail in subsection 2.2 but the simplified version of it will be covered here since it is part of the Circle Space Partitioning method. Essentially, a list is created that represents the savings to the current circles solution by comparing each node within the circle to see if the overall distance travelled can be reduced by making a connection between them (Equation 1) then the saving is added to a savings list along with the nodes that create that saving. Once the list is created, it is then sorted into descending order so that the potential connections that will yield the largest savings, will be handled first.

$$s = c0ToDepot + c1ToDepot - c0Toc1 \quad (1)$$

s = the saving that can be made
c0ToDepot = the distance between customer 0 and the depot
c1ToDepot = the distance between customer 1 and the depot
c0Toc1 = the distance between customer 0 and customer 1

With the list of savings for each circle, a route is created for each and nodes are added to them as they appear in the savings list if they haven't already been added to the route. Once this process has completed, we have the final solution and each circle contains a route representing an efficient path between each node (customer) Figure 3.

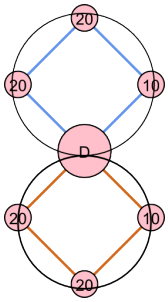

Figure 3: *Each circle has its nodes reordered to create a fast path between them, resulting in a solution. (D is the depot and the number in the other nodes is the requirement)*

2.2 Clarke-Wright

The process of the Clarke-Wright algorithm implemented was researched (for example [Toth and Vigo 2001] was a very useful resource for this algorithm) before hand to determine exactly how the algorithm works. To begin, you first have to take each node within the problem and put it into its own route (Figure 4). Once each node has its own route, a savings list needs to be calculated to determine where routes can be merged in order to reduce the total cost of the solution. This list is created by taking the start and end nodes (customers) of each route and calculating a saving using the saving equation (see Equation 1) and adding the saving along with the nodes that the saving is from to the list. This list is then sorted into descending order so that the highest savings are considered first.

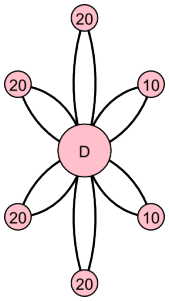

Figure 4: *Create a route for each node within the problem space. (D is the depot and the number in the other nodes is the requirement)*

With the savings list ordered, you can start to merge the routes. Loop through each saving within the list and check if one node is at the start of a route and the other node is at the end of a route. If that is true, the nodes aren't the same, the routes the nodes come from aren't the same and the sum of the current requirement of the two routes the nodes are from isn't greater than the capacity of the routes set by the problem then merge the routes (Figure 5).

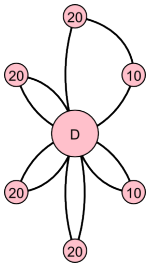

Figure 5: *The solution after one merge has occured. (D is the depot and the number in the other nodes is the requirement)*

After every saving in the list has been checked and merges have been executed where possible, the solution is found.

3 Results

All results were acquired by running the algorithm on a system with these specifications:

- CPU - AMD Phenom II x6 1090T (6 cores, 3.2Ghz)
- RAM - 16Gb
- Operating System - Windows 7

Problem Size	Partition Alg.	Clarke-Wright Alg.	Sample
10	1489	1472	1577
20	2485	2375	2699
30	3193	3251	3781
40	3424	3393	3727
50	5538	4688	5412
60	5785	5064	6015
70	5953	5050	5898
80	7192	6218	6883
90	8414	6910	8442
100	9217	7529	38704
200	15036	12968	76929
300	21643	16958	116481
400	27495	22645	153609
500	34313	27996	188740
600	40708	33416	228962
700	45299	37266	266675
800	52331	42246	311675
900	60033	48802	347699
1000	63143	51399	390783

Table 1: *Table Containing the cost of each algorithm across a range of problem sizes (how many nodes/customers there are in the problem).*

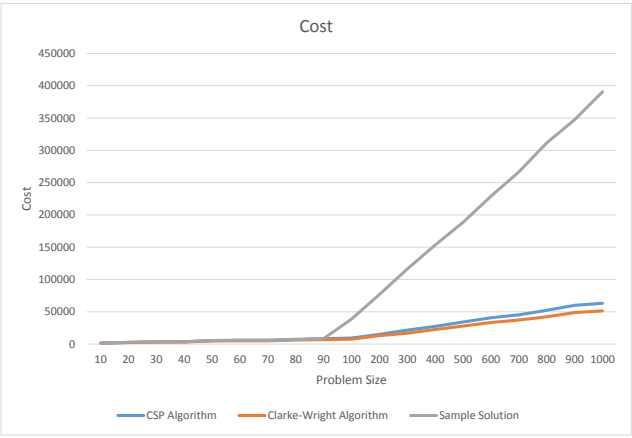

Figure 6: *A chart visually showing the difference in the cost for each algorithm's solution across a range of problem sizes*

Problem Size	Partition Alg. (ms)	Clarke-Wright Alg. (ms)
10	0.1	0.2
20	0.1	0.1
30	0.1	0.2
40	0.1	0.4
50	0.1	0.7
60	0.1	1
70	0.2	1.5
80	0.2	2.2
90	0.2	2.9
100	0.2	3.8
200	0.5	15.4
300	0.9	40.2
400	1.4	89.6
500	1.8	160.3
600	2.3	261.7
700	2.9	397.3
800	3.5	570
900	4.2	802
1000	5.1	1025.7

Table 2: *Table containing the time taken (averaged over 100 runs) in milliseconds to run the Circle Space Partitioning and Clarke-Wright algorithms on a range of problem sizes.*

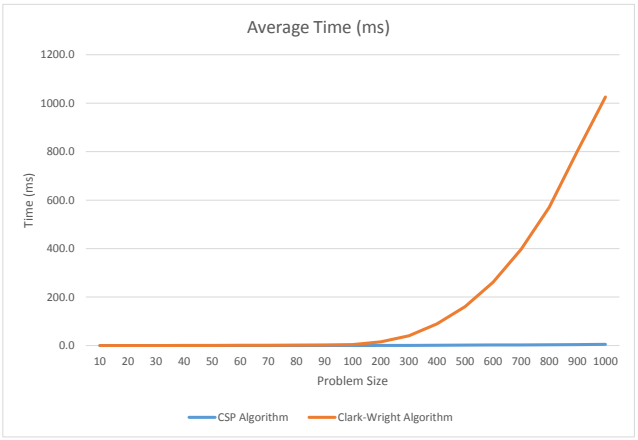

Figure 7: *A chart visually showing the difference in the time taken for the Circle Space Partitioning and Clarke-Wright algorithm to find a solution across a range of problem sizes*

4 Conclusions

From the results in section 3, specifically Figure 6, we can clearly see that both the Clarke-Wright and the Circle Space Partitioning (CSP) algorithm have greatly improved scores for their solutions in comparison to the sample solutions. This becomes even more apparent as the problem size grows larger. Figure 6 also shows that as the problem size increases, the Circle Space Partitioning algorithm gradually returns solutions with costs that are increasingly worse than that returned by the Clarke-Wright algorithm.

Although Clarke-Wright does return consistently better results in terms of cost compared to Circle Space Partitioning, if you look at Figure 7 it is clear that the Circle Space Partitioning method does have one major benefit. Since the main aspect of the CSP method is to iteratively split the problem into smaller, more manageable problems, a huge time saving is created. Because each problem is so small, it becomes a very quick and simple task to sort nodes so that their is an efficient path between them. This time saving is so large and the overall time to run the algorithm increases at such a low rate as the problem size grows that in comparison to the Clarke-Wright method, the CSP algorithm shows a promising use when dealing with very large problems.

When the time comparison in Figure 7 is compared with the cost comparison in Figure 6 it becomes apparent that in certain scenarios, the Circle Space Partitioning algorithm could be of more use than the Clarke-Wright algorithm. The likely scenarios that this would be the case would include when the cost difference between the two algorithms is so low (when the problem size is small) that it would be just as efficient to run either algorithm or when the problem size is so large that it would be very time consuming to run the Clarke-Wright algorithm and in that case, the increase in cost could be justified by the massive reduction in time taken.

The results for the Circle Space Partitioning and the Clarke-Wright algorithm were calculated by running each algorithm 100 times, taking the time taken to run the 100 iterations and then dividing the that time by the total number of iterations (100), giving the average run time for a problem. This was done to ensure that the results for each problem were accurate and reliable. The process was repeated for each problem to give a table of averages for each algorithm which can be seen in Table 2.

References

ALVARENGA, G. B., MATEUS, G. R., AND DE TOMI, G. 2007. A genetic and set partitioning two-phase approach for the vehicle routing problem with time windows. *Computers & Operations Research 34*, 6, 1561–1584. 1

TOTH, P., AND VIGO, D. 2001. *The vehicle routing problem*. Society for Industrial and Applied Mathematics. 1, 2

5 Appendix

The code, written in java, will be displayed in this order:

- Main 5.1
- VRPartitionSolution 5.2
- VRClarkeWrightSolution 5.3
- Circle 5.4
- SavingsNode 5.5
- Route 5.6
- Customer 5.7
- VRProblem 5.8

5.1 Main

```
package com.grant.smith;

import java.io.File;
import java.util.ArrayList;
import java.util.regex.Pattern;


public class Main {

  static String testDataFolder = "Test Data/";
  static String resultsFolder = "Results/";

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

      // Load all files in the test data folder that
          end with prob.csv and don't contain fail
      Pattern problemFilePattern =
          Pattern.compile("prob\\.csv");

      final File folder = new File(testDataFolder);
      ArrayList<File> files = new ArrayList<File>();
      for (final File f : folder.listFiles()) {
        if(problemFilePattern.matcher(f.getName()).find())
          if(!f.getName().contains("fail"))
            files.add(f);
      }
      // create a list to stores the times of each
          problem
      int[] times = new int[files.size()];

      int fileNo = 0;
      // For each file, run the VRP solver
      for(File f : files){
        // Solution classes:
        // VRClarkeWrightSolution
        // VRPartitionSolution
        VRPartitionSolution vrS = new
            VRPartitionSolution(new
            VRProblem(testDataFolder + f.getName()));

        double loops = 1;
        long startTime = System.nanoTime();
        for(int i = 0; i < loops; i++){
          vrS.solve();
        }
        // store the time taken for the total loops
        times[fileNo] = (int)((System.nanoTime() -
            startTime)/1000000.0);
```

```
        String fileName = f.getName().substring(0,
             f.getName().length() -
             "prob.csv".length());


        // output information about solution to the
             console
        System.out.println("-------------"+fileName+"-------------");
        System.out.println("Time taken for " + loops +
             " loops = " + times[fileNo] + "ms");
        System.out.println("Avg time " +
             (int)(times[fileNo]/loops) + "ms");
        System.out.println("Number of routes = " +
             vrS.soln.size());
        System.out.println("Cost = " + vrS.solnCost());
        System.out.println("vrS.verify() returned " +
             vrS.verify());
        System.out.println();

        // output the solution
        vrS.writeSVG(resultsFolder + fileName +
             "prob.svg", resultsFolder + fileName +
             "solu.svg");
        vrS.writeOut(resultsFolder + fileName +
             "solu.csv");
        fileNo++;

      }
    }catch(Exception e){
      e.printStackTrace();
    }

  }

}
```

5.2 VRPartitionSolution

```
package com.grant.smith;

import java.util.*;
import java.awt.geom.Point2D;
import java.io.*;


public class VRPartitionSolution {

  // Stores the information about the problem
  public VRProblem prob;
  // Stores the solution after solve() is ran
  public List<Route> soln;
  // Stores the circles used to create the solution
  ArrayList<Circle> circles;

  public VRPartitionSolution(VRProblem problem){
    this.prob = problem;
  }

  //Students should implement another solution
  public void solve(){

    // Create a circle that encompasses all nodes
    Circle root = new Circle(prob.depot.x,
         prob.depot.y, 0.0);
    for(Customer c : prob.customers){
      root.add(c);
    }
    // Shrink the circle so that it just covers all of
         the nodes within it
    root.shrink();

    // Create the list to contain all off the circles
    circles = new ArrayList<Circle>();
    // Add the first circle to the list
    circles.add(root);

    boolean overCap = true;
    while(overCap){
      overCap = false;
      for(int i = 0; i < circles.size(); i++){
        // Get the circle for this iteration
        Circle c = circles.get(i);
        // Check if this circle should be kept (if is
             contains any nodes)
        if(!c.keepAlive()) { circles.remove(i);
             continue;} // If this circle shouldn't be
             kept, remove it
        if(c.demand > prob.depot.c){
          // Get the farthest node in this circle
          Customer farthest = c.farthestNode();

          // Create a new circle between that node and
               this circle's centre
           double mx = (farthest.x + c.pos.x)/2;
           double my = (farthest.y + c.pos.y)/2;
          Circle newC = new Circle( new
               Point2D.Double(mx,my), c.radius/2);

          // Loop through all the circles and try to
               steal nodes
          for(Circle others : circles){
            newC.steal(others);
            if(others.demand <= prob.depot.c)
              others.canBeStolenFrom = false;
          }

          // Add the new circle to the list
          circles.add(newC);

          // Check if we need to keep looping the
               'while'
          if(c.demand > prob.depot.c || newC.demand >
               prob.depot.c)
            overCap = true;

        }

      }

      // Merge all circles if they cross over and
           merging them wouldn't put them over capacity
      boolean merged = true;
      while(merged){ // If there was a merge, check
           again to see if there are any more
        merged = false;
        // Loop through every circle for each circle
        for(int i = 0; i < circles.size()-1; i++){
          for(int j = i+1; j < circles.size(); j++){ //
               Don't loop through circles that have
               already checked this circle
            // Get circle ci and circle cj
            Circle ci = circles.get(i);
            Circle cj = circles.get(j);
            // Check if they should be merged
            if((ci.demand + cj.demand <= prob.depot.c)
                 && (ci.pos.distance(cj.pos) <=
                 ci.radius*2 + cj.radius*2)){
              // merge the circles
              circles.remove(cj);
              ci.mergeRoutes(cj);
              // Shrink the circle so that it just
                   covers all of the nodes within it
              ci.shrink();
              // state that there was a merge
              merged = true;
```

```
            }
          }
        }
      }

      // Resize and reposition all of the circles
           depending on the farthest node within them
           and their position
      for(int i = 0; i < circles.size(); i++){
        Circle c = circles.get(i);
        Customer farthest = c.farthestNode();
        if(farthest == null){ circles.remove(i);
             continue;}
        // Get the midpoint between the circle's
             position and the farthest node's position
         double mx = (farthest.x + c.pos.x)/2;
         double my = (farthest.y + c.pos.y)/2;
         // Set the circle's position to the calculated
              midpoint
        c.pos.setLocation(mx, my);
        // Shrink the circle so that it just covers all
             of the nodes within it
        c.shrink();

      }
    }


    for(Circle c : circles){
      if(c.size() > 1){
        c.improve(prob.depot);
      }
    }

    soln = new LinkedList<Route>();

    soln.addAll(circles);

    //Remove the circles so that they don't render
    circles.clear();

  }

  //Calculate the total journey
  public double solnCost(){
    double cost = 0;
    for(List<Customer>route:soln){
      Customer prev = this.prob.depot;
      for (Customer c:route){
        cost += prev.distance(c);
        prev = c;
      }
      //Add the cost of returning to the depot
      cost += prev.distance(this.prob.depot);
    }
    return cost;
  }

  public Boolean verify(){
    //Check that no route exceeds capacity
    Boolean okSoFar = true;
    for(Route route : soln){
      //Start the spare capacity at
      int total = 0;
      for(Customer c:route)
        total += c.c;
      if (total>prob.depot.c){
        System.out.printf("********FAIL Route starting
             %s is over capacity %d\n",
            route.get(0),
            total
            );
        okSoFar = false;
      }
    }
    //Check that we keep the customer satisfied
    //Check that every customer is visited and the
         correct amount is picked up
    Map<String,Integer> reqd = new
         HashMap<String,Integer>();
    for(Customer c:this.prob.customers){
      String address = String.format("%fx%f", c.x,c.y);
      reqd.put(address, c.c);
    }
    for(Route route:this.soln){
      for(Customer c:route){
        String address = String.format("%fx%f",
             c.x,c.y);
        if (reqd.containsKey(address))
          reqd.put(address, reqd.get(address)-c.c);
        else
          System.out.printf("********FAIL no customer
               at %s\n",address);
      }
    }
    for(String address:reqd.keySet())
      if (reqd.get(address)!=0){
        System.out.printf("********FAIL Customer at %s
             has %d left
             over\n",address,reqd.get(address));
        okSoFar = false;
      }
    return okSoFar;
  }

  public void readIn(String filename) throws Exception{
    BufferedReader br = new BufferedReader(new
         FileReader(filename));
    String s;
    this.soln = new ArrayList<Route>();
    while((s=br.readLine())!=null){
      Route route = new Route();
      String [] xycTriple = s.split(",");
      for(int i=0;i<xycTriple.length;i+=3)
        route.add(new Customer(
             (int)Double.parseDouble(xycTriple[i]),
             (int)Double.parseDouble(xycTriple[i+1]),
             (int)Double.parseDouble(xycTriple[i+2])));
      soln.add(route);
    }
    br.close();
  }

  public void writeSVG(String probFilename,String
       solnFilename) throws Exception{
    String[] colors = "chocolate cornflowerblue crimson
         cyan darkblue darkcyan darkgoldenrod".split("
         ");
    int colIndex = 0;
    String hdr =
        "<?xml version='1.0'?>\n"+
            "<!DOCTYPE svg PUBLIC '-//W3C//DTD SVG
                 1.1//EN' '../../svg11-flat.dtd'>\n"+
            "<svg width='8cm' height='8cm' viewBox='0
                 0 500 500'
                 xmlns='http://www.w3.org/2000/svg'
                 version='1.1'>\n";
    String ftr = "</svg>";
    StringBuffer psb = new StringBuffer();
    StringBuffer ssb = new StringBuffer();
    psb.append(hdr);
    ssb.append(hdr);
    for(List<Customer> route:this.soln){
      ssb.append(String.format("<path d='M%s %s
           ",this.prob.depot.x,this.prob.depot.y));
      for(Customer c:route)
        ssb.append(String.format("L%s %s",c.x,c.y));
```

```
      ssb.append(String.format("z' stroke='%s'
           fill='none' stroke-width='2'/>\n",
           colors[colIndex++ % colors.length]));
    }
    for(Customer c:this.prob.customers){
      String disk = String.format(
           "<g transform='translate(%.0f,%.0f)'>"+
                "<circle cx='0' cy='0' r='%d'
                     fill='pink' stroke='black'
                     stroke-width='1'/>" +
                "<text text-anchor='middle'
                     y='5'>%d</text>"+
                "</g>\n",
                c.x,c.y,10,c.c);
      psb.append(disk);
      ssb.append(disk);
    }
    for(Circle c : this.circles){
      String disk = String.format(
           "<g transform='translate(%.0f,%.0f)'>"+
                "<circle cx='0' cy='0' r='%d'
                     fill='none' stroke='black'
                     stroke-width='1'/>" +
                "<text text-anchor='middle'
                     y='5'>%s</text>"+
                "</g>\n",
                c.pos.x, c.pos.y, (int)c.radius, "");
      psb.append(disk);
      ssb.append(disk);
    }
    String disk = String.format("<g
        transform='translate(%.0f,%.0f)'>"+
        "<circle cx='0' cy='0' r='%d' fill='pink'
             stroke='black' stroke-width='1'/>" +
        "<text text-anchor='middle' y='5'>%s</text>"+
        "</g>\n",
             this.prob.depot.x,this.prob.depot.y,20,"D");
    psb.append(disk);
    ssb.append(disk);
    psb.append(ftr);
    ssb.append(ftr);
    PrintStream ppw = new PrintStream(new
         FileOutputStream(probFilename));
    PrintStream spw = new PrintStream(new
         FileOutputStream(solnFilename));
    ppw.append(psb);
    spw.append(ssb);
    ppw.close();
    spw.close();
  }
  public void writeOut(String filename) throws Exception{
    PrintStream ps = new PrintStream(filename);
    for(List<Customer> route:this.soln){
      boolean firstOne = true;
      for(Customer c:route){
        if (!firstOne)
          ps.print(",");
        firstOne = false;
        ps.printf("%f,%f,%d",c.x,c.y,c.c);
      }
      ps.println();
    }
    ps.close();
  }
}
```

5.3 VRClarkeWrightSolution

```
package com.grant.smith;

import java.util.*;
import java.io.*;


public class VRClarkeWrightSolution {

  // Stores the information about the problem
  public VRProblem prob;
  // Stores the solution after solve() is ran
  public List<Route> soln;

  public VRClarkeWrightSolution(VRProblem problem){
    this.prob = problem;
  }

  //The dumb solver adds one route per customer
  public void oneRoutePerCustomerSolution(){
    this.soln = new ArrayList<Route>();
    for(Customer c:prob.customers){
      Route route = new Route();
      route.add(c);
      soln.add(route);
    }
  }

  // Clarke Wright solver function
  public void solve(){
    // Create a route for each customer
    oneRoutePerCustomerSolution();

    // Calculate the savings list for the problem
    List<SavingsNode> savings = getSavings();

    // Loop through each saving
    for (SavingsNode savingsNode : savings) {
      // Get the route where ci is the first customer
      Route route0 =
           routeWhereCustomerIsLast(savingsNode.ci);
      if(route0 != null){ // check if we found a route
           for route0
        // Get the route where cj is the first customer
        Route route1 =
             routeWhereCustomerIsFirst(savingsNode.cj);
        if(route1 != null){ // check if we found a
             route for route1
          if(route0 == route1){ continue;} // if route0
               and route1 are the same, do nothing
          if (route0.demand + route1.demand <=
               prob.depot.c) { // if merge is feasible
            // Merge the two routes
            soln.remove(route1);
            route0.mergeRoutes(route1);
          }
        }
      }
    }

  }

  // returns a route where c is the last element in a
       route
  private Route routeWhereCustomerIsLast(Customer c){
    for(Route r : soln){
      if(r.getEnd() == c) return r;
    }
    return null;
  }
```

```
// returns a route where c is the first element in a
     route
private Route routeWhereCustomerIsFirst(Customer c){
  for(Route r : soln){
    if(r.getStart() == c) return r;
  }
  return null;
}

// Calculates the savings list
public List<SavingsNode> getSavings(){
  // Create a list of customers for the savings list
       to use
  List<Customer> customers = new
       ArrayList<Customer>();
  // loop through every route
  for(Route r : soln){
    if(r.demand < prob.depot.c){
      // add the start of the route to the list
      customers.add(r.getStart());
      // if the route is larger than just 1 element,
           add the end as well
      if(r.size() > 1)
        customers.add(r.getEnd());
    }
  }

  // Create the savings list
  List<SavingsNode> savings = new
       ArrayList<SavingsNode>();
  // Loop through each node and add the relative
       savings to the savings list if there is a
       saving to be made
  for(int i = 0; i < customers.size(); i++){
    for(int j = i; j < customers.size(); j++){
      if(i == j) continue;
      Customer ci = customers.get(i);
      Customer cj = customers.get(j);
      double saving = (prob.depot.distance(ci) +
           prob.depot.distance(cj)) - ci.distance(cj)
           ;
      if(saving > 0)
        savings.add(new SavingsNode(ci,cj,saving));
    }
  }
  // Sort the list into descending order
  Collections.sort(savings);
  return savings;
}

//Calculate the total journey
public double solnCost(){
  double cost = 0;
  for(List<Customer>route:soln){
    Customer prev = this.prob.depot;
    for (Customer c:route){
      cost += prev.distance(c);
      prev = c;
    }
    //Add the cost of returning to the depot
    cost += prev.distance(this.prob.depot);
  }
  return cost;
}

public Boolean verify(){
  //Check that no route exceeds capacity
  Boolean okSoFar = true;
  for(Route route : soln){
    //Start the spare capacity at
    int total = 0;
    for(Customer c:route)
      total += c.c;
    if (total>prob.depot.c){
      System.out.printf("********FAIL Route starting
           %s is over capacity %d\n",
           route.get(0),
           total
           );
      okSoFar = false;
    }
  }
  //Check that we keep the customer satisfied
  //Check that every customer is visited and the
       correct amount is picked up
  Map<String,Integer> reqd = new
       HashMap<String,Integer>();
  for(Customer c:this.prob.customers){
    String address = String.format("%fx%f", c.x,c.y);
    reqd.put(address, c.c);
  }
  for(Route route:this.soln){
    for(Customer c:route){
      String address = String.format("%fx%f",
           c.x,c.y);
      if (reqd.containsKey(address))
        reqd.put(address, reqd.get(address)-c.c);
      else
        System.out.printf("********FAIL no customer
             at %s\n",address);
    }
  }
  for(String address:reqd.keySet())
    if (reqd.get(address)!=0){
      System.out.printf("********FAIL Customer at %s
           has %d left
           over\n",address,reqd.get(address));
      okSoFar = false;
    }
  return okSoFar;
}

public void readIn(String filename) throws Exception{
  BufferedReader br = new BufferedReader(new
       FileReader(filename));
  String s;
  this.soln = new ArrayList<Route>();
  while((s=br.readLine())!=null){
    Route route = new Route();
    String [] xycTriple = s.split(",");
    for(int i=0;i<xycTriple.length;i+=3)
      route.add(new Customer(
           (int)Double.parseDouble(xycTriple[i]),
           (int)Double.parseDouble(xycTriple[i+1]),
           (int)Double.parseDouble(xycTriple[i+2])));
    soln.add(route);
  }
  br.close();
}

public void writeSVG(String probFilename,String
     solnFilename) throws Exception{
  String[] colors = "chocolate cornflowerblue crimson
       cyan darkblue darkcyan darkgoldenrod".split("
       ");
  int colIndex = 0;
  String hdr =
       "<?xml version='1.0'?>\n"+
            "<!DOCTYPE svg PUBLIC '-//W3C//DTD SVG
                 1.1//EN' '../../svg11-flat.dtd'>\n"+
            "<svg width='8cm' height='8cm' viewBox='0
                 0 500 500'
                 xmlns='http://www.w3.org/2000/svg'
                 version='1.1'>\n";
  String ftr = "</svg>";
  StringBuffer psb = new StringBuffer();
  StringBuffer ssb = new StringBuffer();
  psb.append(hdr);
```

```
    ssb.append(hdr);
    for(List<Customer> route:this.soln){
      ssb.append(String.format("<path d='M%s %s
           ",this.prob.depot.x,this.prob.depot.y));
      for(Customer c:route)
        ssb.append(String.format("L%s %s",c.x,c.y));
      ssb.append(String.format("z' stroke='%s'
           fill='none' stroke-width='2'/>\n",
           colors[colIndex++ % colors.length]));
    }
    for(Customer c:this.prob.customers){
      String disk = String.format(
           "<g transform='translate(%.0f,%.0f)'>"+
                "<circle cx='0' cy='0' r='%d'
                     fill='pink' stroke='black'
                     stroke-width='1'/>" +
                "<text text-anchor='middle'
                     y='5'>%d</text>"+
                "</g>\n",
                c.x,c.y,10,c.c);
      psb.append(disk);
      ssb.append(disk);
    }
    String disk = String.format("<g
         transform='translate(%.0f,%.0f)'>"+
         "<circle cx='0' cy='0' r='%d' fill='pink'
              stroke='black' stroke-width='1'/>" +
         "<text text-anchor='middle' y='5'>%s</text>"+
         "</g>\n",
              this.prob.depot.x,this.prob.depot.y,20,"D");
    psb.append(disk);
    ssb.append(disk);
    psb.append(ftr);
    ssb.append(ftr);
    PrintStream ppw = new PrintStream(new
         FileOutputStream(probFilename));
    PrintStream spw = new PrintStream(new
         FileOutputStream(solnFilename));
    ppw.append(psb);
    spw.append(ssb);
    ppw.close();
    spw.close();
  }
  public void writeOut(String filename) throws Exception{
    PrintStream ps = new PrintStream(filename);
    for(List<Customer> route:this.soln){
      boolean firstOne = true;
      for(Customer c:route){
        if (!firstOne)
          ps.print(",");
        firstOne = false;
        ps.printf("%f,%f,%d",c.x,c.y,c.c);
      }
      ps.println();
    }
    ps.close();
  }
}
```

5.4 Circle

```
package com.grant.smith;

import java.awt.geom.Point2D;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;

public class Circle extends Route{

  // stores the radius of this circle
  double radius;
  // stores the position of this circle
  Point2D.Double pos;
  // stores the stealable state (whether another circle
       can steal from it or not)
  boolean canBeStolenFrom = true;


  // Constructor for a circle
  Circle(double x, double y, double radius){
    this(new Point2D.Double(x,y), radius);
  }

  // Constructor for a circle
  Circle(Point2D.Double pos, double radius){
    this.pos = pos;
    this.radius = radius;
  }

  // Shrink the circle so that it just covers all of the
       nodes within it
  public void shrink(){
    if(size() > 0)
      radius = farthestNode().distance(pos);
  }

  // Should this circle be kept (false if it has no
       children)
  public boolean keepAlive(){
    if(size() == 0)
      return false;

    return true;
  }

  // Add all of the nodes from one circle to this one
  public void mergeRoutes(Circle other){
    super.mergeRoutes(other);
    // Get the midpoint between the circle's position
         and the farthest node's position
     double mx = (other.pos.x + pos.x)/2;
     double my = (other.pos.y + pos.y)/2;
     // Set the circle's position to the calculated
          midpoint
    pos.setLocation(mx, my);
    // Shrink this circle so that it just covers all of
         the nodes within it
    shrink();
  }

  // Attempt to steal nodes from another circle until
       full or out of nodes to steal
  public void steal(Circle other){
    if(!other.canBeStolenFrom) return; // If the other
         doesn't allow stealing, return
    Iterator<Customer> it = other.iterator();
    // For each node in the other circle
    while(it.hasNext()){
      Customer c = it.next();
```

```
      // Check which circle the node is closer to
      if(c.distance(pos) < c.distance(other.pos)){
        // If the node is closer to this circle then
            steal it
        other.demand -= c.c;
        add(c);
        it.remove();
      }
    }
    // Shrink this circle so that it just covers all of
        the nodes within it
    shrink();
    // Shrink the other circle so that it just covers
        all of the nodes within it
    other.shrink();
  }

  // Improves the pathing between the nodes contained
      withing this circle
  public void improve(Point2D.Double depot){
    // Create a list of savings
    List<SavingsNode> savings = new
        ArrayList<SavingsNode>();
    // Loop through each node and add the relative
        savings to the savings list if there is a
        saving to be made
    for(int i = 0; i < size(); i++){
      for(int j = 0; j < size(); j++){
        if(i == j) continue;
        Customer ci = get(i);
        Customer cj = get(j);
        double saving = depot.distance(ci) +
            depot.distance(cj) - ci.distance(cj);
        if(saving > 0)
          savings.add(new SavingsNode(ci,cj,saving));
      }
    }

    // Sort the list into descending order
    Collections.sort(savings);

    // Create an ordered list
    List<Customer> ordered = new LinkedList<Customer>();
    // Add the highest saving nodes and remove them
        from the savings list
    ordered.add(savings.get(0).ci);
    ordered.add(savings.remove(0).cj);

    // Loop through the savings list
    for(SavingsNode s : savings){
      // if the first ordered node is ci and ordered
          doesn't contain cj then add cj
      if(ordered.get(0) == s.ci &&
          !ordered.contains(s.cj)){
        ordered.add(0,s.cj);
      // else if the lat ordered node is ci and ordered
          doesn't contain cj then add cj
      }else if(ordered.get(ordered.size()-1) == s.ci &&
          !ordered.contains(s.cj)){
        ordered.add(s.cj);
      // else add cj to the side that has the largest
          saving
      }else{
        Customer first = ordered.get(0);
        Customer last = ordered.get(ordered.size()-1);

        double savingF = (depot.distance(first) +
            depot.distance(s.cj)) -
            first.distance(s.cj);
        double savingL = (depot.distance(last) +
            depot.distance(s.cj)) -
            last.distance(s.cj);
        if(savingF >= savingL &&
            !ordered.contains(s.cj))
          ordered.add(0,s.cj);
        else if(!ordered.contains(s.cj))
          ordered.add(s.cj);
      }
    }
    // clear this list
    clear();
    // set the demand to 0
    demand = 0.0f;
    // add all of the ordered nodes to this list
    addAll(ordered);
  }

  // Return the farthest node from the center of this
      circle that this circle owns
  public Customer farthestNode(){
    // Set far to a very low number
    double far = Double.MIN_VALUE;
    Customer farthest = null;
    // Loop through each customer
    for(Customer c : this){
      // If the distance to c is larger than far then
          set the farthest to c
      if(pos.distance(c) > far){
        far = pos.distance(c);
        farthest = c;
      }
    }
    // return the farthest node
    return farthest;
  }

}
```

5.5 SavingsNode

```
package com.grant.smith;


public class SavingsNode implements
    Comparable<SavingsNode> {

  // Saving to be made
  public final double saving;
  // Customers that make the saving
  public final Customer ci, cj;

  public SavingsNode(Customer ci, Customer cj, double
      saving){
    this.ci = ci;
    this.cj = cj;
    this.saving = saving;
  }

  // Comparator for sorting
  public int compareTo(SavingsNode sj) {
    return saving < sj.saving ? 1 : saving == sj.saving
        ? 0 : -1;
  }

}
```

5.6 Route

```
package com.grant.smith;

import java.util.ArrayList;


public class Route extends ArrayList<Customer>{

  // stores the demand of this route
  double demand = 0.0f;

  // stores the start and end to save time fetching them
      from the array
  private Customer start;
  private Customer end;

  // add a customer to the route
  @Override
  public boolean add(Customer c) {
    // increment the demand by the customer's
        requirement
    demand += c.c;

    // if this list is empty, cache c as the start
    if(size() == 0)
      start = c;

    // cache c as the end
    end = c;

    // add c to this list
    return super.add(c);
  }


  // Add all of the nodes from one circle to this one
  public Route mergeRoutes(Route route2){
    // increment the demand by the other route's
        requirement
    demand += route2.demand;
    end = route2.end;
    addAll(route2);
    return this;
  }

  // returns the start node
  public Customer getStart(){
    // return the cached start
    return start;
  }

  // returns the end node
  public Customer getEnd(){
    // return the cached end
    return end;
  }


}
```

5.7 Customer

```
package com.grant.smith;

import java.awt.geom.Point2D;


public class Customer extends Point2D.Double{

  // Requirements of the customer (number to be
      delivered)
  public int c;
  public Customer(int x, int y, int requirement){
    this.x = x;
    this.y = y;
    this.c = requirement;
  }

}
```

5.8 VRProblem

```
package com.grant.smith;

import java.util.*;
import java.io.*;
public class VRProblem {
  public String id;
  public Customer depot;
  ArrayList<Customer> customers;
  public VRProblem(String filename) throws Exception{
    this.id = filename;
    BufferedReader br = new BufferedReader(new
        FileReader(filename));
    //Details of the depot and the truck capacity are
        stored in the first line
    String s = br.readLine();
    String dpt [] = s.split(",");
    depot = new Customer(
        Integer.parseInt(dpt[0]),
        Integer.parseInt(dpt[1]),
        Integer.parseInt(dpt[2]));
    customers = new ArrayList<Customer>();
    //Every customer is stored on a comma separated line
    while ((s=br.readLine())!=null){
      String wrd [] = s.split(",");
      customers.add(new Customer(
          Integer.parseInt(wrd[0]),
          Integer.parseInt(wrd[1]),
          Integer.parseInt(wrd[2])));
    }
    br.close();
  }
  public int size(){
    return this.customers.size();
  }
}
```