Clarke-Wright vehicle routing algorithm
Implementation Report

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

1 Introduction

Realistic Real time physics simulation is highly sought after in interactive applications, especially games. Achieving high-accuracy while maintaining performance in often resource restricted environments(I.E a games console) requires the highest level of optimisations and often results in a trade-off with simulation speed against Accuracy. This project attempts to record and analyse the performance of various optimisations on a simulated scene. This will be taken further by applying the project to various different processing architectures. The scene that will be simulated is a large set of Bouncy balls, travelling down a hill. [Clarke and Wright 1964] [Lysgaard 1997]

Physics Engines Large and complex video games tend to use 3rd party physics solutions, this vastly cuts down on the project development man-hours, and the maintenance thereafter. Third party physics solutions have the benefit of being battle tested out in the wild beforehand, so internal reliability is usually a given. A further benefit is that being developed solely for the purpose of being a "a good physics engine" by people who are usually experts in the field, large optimisations are already implemented. The problems arise in the implementation, the coupling of a physics engine and the existing codebase. While they are usually well coded, they are not tailor made to each game.

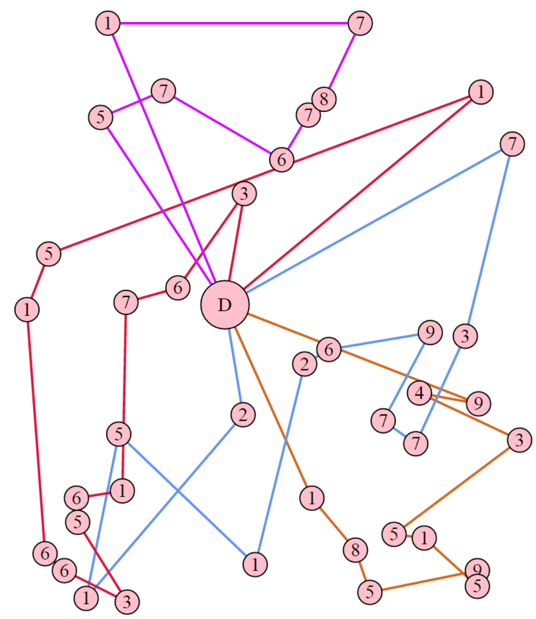

Figure 1: ***Bullet Physics PS3 Pipeline*** *- Requires Intermediate Data Swapping Between PPU and SPU*

2 Method

Optimising for Physics Engines Trying to regain performance from an external physics engine can be a hard task, diving into the source code requires expert knowledge of the inner-workings of the whole system. A common path is to shape the design of the game code to conform better to the demands of the physics engine and hope that the internal optimisations will be sufficient. Often enough, they are not.

3 Results

Optimising for Physics Engines Trying to regain performance from an external physics engine can be a hard task, diving into the source code requires expert knowledge of the inner-workings of the whole system. A common path is to shape the design of the game code to conform better to the demands of the physics engine and hope that the internal optimisations will be sufficient. Often enough, they are not.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio

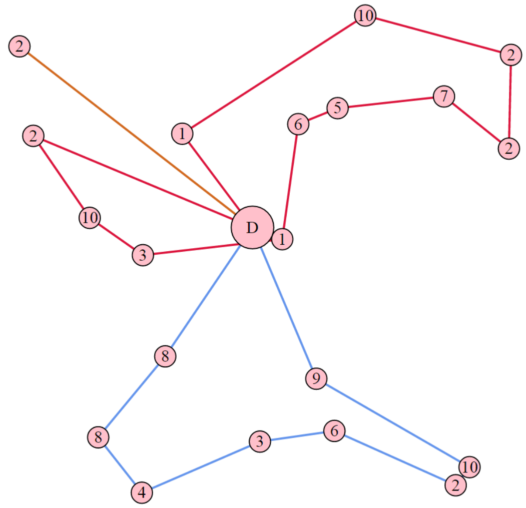

Figure 2: ***Bullet Physics PS3 Pipeline*** *- Requires Intermediate Data Swapping Between PPU and SPU*

placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

Fusce mauris. Vestibulum luctus nibh at lectus. Sed bibendum, nulla a faucibus semper, leo velit ultricies tellus, ac venenatis arcu wisi vel nisl. Vestibulum diam. Aliquam pellentesque, augue quis sagittis posuere, turpis lacus congue quam, in hendrerit risus eros eget felis. Maecenas eget erat in sapien mattis porttitor. Vestibulum porttitor. Nulla facilisi. Sed a turpis eu lacus commodo facilisis. Morbi fringilla, wisi in dignissim interdum, justo lectus sagittis dui, et vehicula libero dui cursus dui. Mauris tempor ligula sed lacus. Duis cursus enim ut augue. Cras ac magna. Cras nulla. Nulla egestas. Curabitur a leo. Quisque egestas wisi eget nunc. Nam feugiat lacus vel est. Curabitur consectetuer.

Suspendisse vel felis. Ut lorem lorem, interdum eu, tincidunt sit amet, laoreet vitae, arcu. Aenean faucibus pede eu ante. Praesent enim elit, rutrum at, molestie non, nonummy vel, nisl. Ut lectus eros, malesuada sit amet, fermentum eu, sodales cursus, magna. Donec eu purus. Quisque vehicula, urna sed ultricies auctor, pede lorem egestas dui, et convallis elit erat sed nulla. Donec luctus. Curabitur et nunc. Aliquam dolor odio, commodo pretium, ultricies non, pharetra in, velit. Integer arcu est, nonummy in, fermentum faucibus, egestas vel, odio.

Sed commodo posuere pede. Mauris ut est. Ut quis purus. Sed ac odio. Sed vehicula hendrerit sem. Duis non odio. Morbi ut dui. Sed accumsan risus eget odio. In hac habitasse platea dictumst. Pellentesque non elit. Fusce sed justo eu urna porta tincidunt. Mauris felis odio, sollicitudin sed, volutpat a, ornare ac, erat. Morbi quis dolor. Donec pellentesque, erat ac sagittis semper, nunc dui lobortis purus, quis congue purus metus ultricies tellus. Proin et quam. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Praesent sapien turpis, fermentum vel, eleifend faucibus, vehicula eu, lacus.

4 Conclusions

Optimising for Physics Engines Trying to regain performance from an external physics engine can be a hard task, diving into the source code requires expert knowledge of the inner-workings of the whole system. A common path is to shape the design of the game code to conform better to the demands of the physics engine and hope that the internal optimisations will be sufficient. Often enough, they are not.

5 Appendix

References

CLARKE, G., AND WRIGHT, J. 1964. 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*.

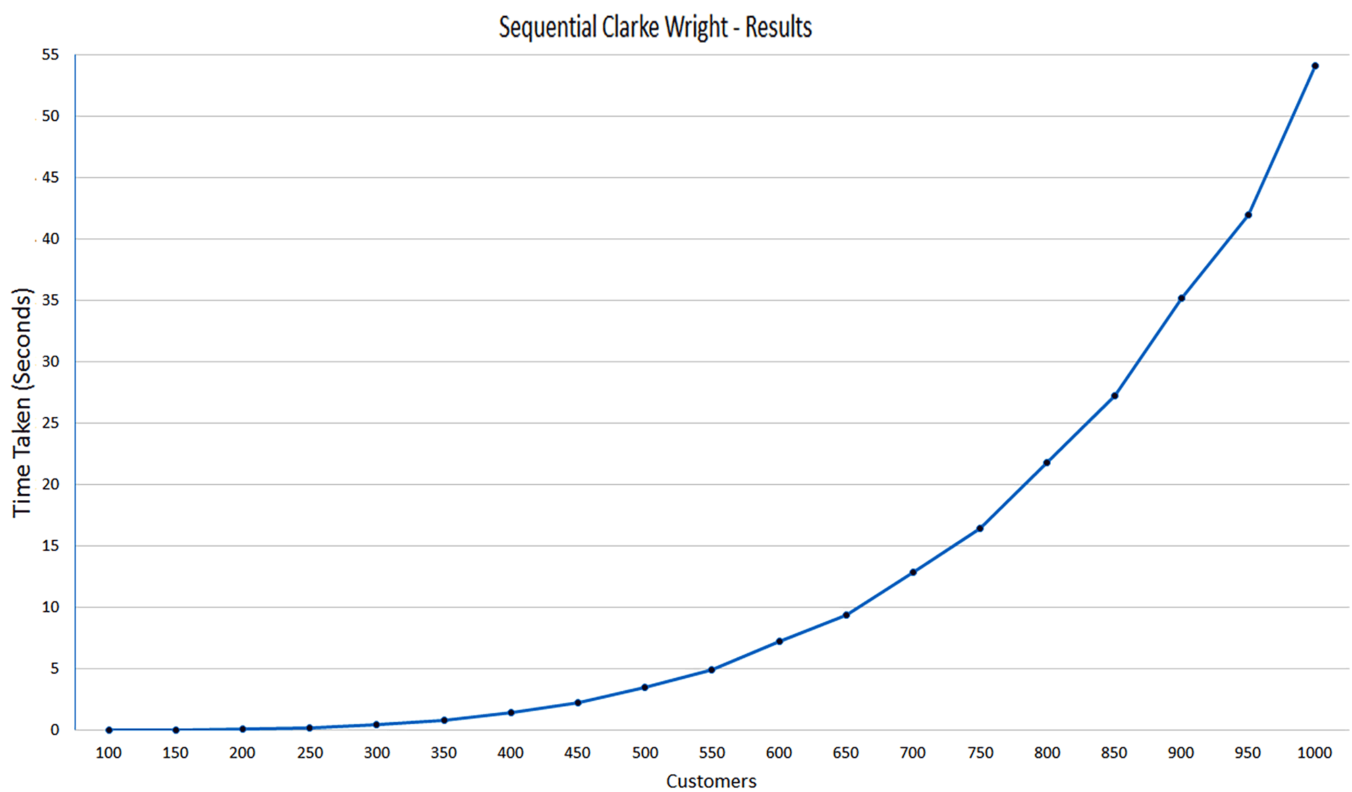

Figure 3: ***Sequential Clark Wright implementation results*** *- Requires Intermediate Data Swapping Between PPU and SPU*

6 Code

6.1 ClarkeWright.java

```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);
    }

}

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);

        //start combining pairs into routes
        for(int i=0; i<pairs.size(); i++)
        {
            Route ro = pairs.get(i);

            for(int j=i+1; j<pairs.size(); j++){
                Route r = pairs.get(j);
                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);

                //do they have any common nodes?
                if(c1 == cr1){
                    //could we combine these based on weight?
                    if(c2.c + ro.getWeight() <= truckCapacity){
                        //Does the route already contain BOTH these nodes already?
                        if(!ro.customers.contains(c2)){
                            ro.addCustomer(c2, true);
                        }
                    }
                }else if (c1 == cr2){
                    if(c2.c + ro.getWeight() <= truckCapacity){
                        if(!ro.customers.contains(c2)){
                            ro.addCustomer(c2, false);
                        }
                    }
                }else if (c2 == cr1){
                    if(c1.c + ro.getWeight() <= truckCapacity){
                        if(!ro.customers.contains(c1)){
                            ro.addCustomer(c1, true);
                        }
                    }
                }else if (c2 ==cr2){
                    if(c1.c + ro.getWeight() <= truckCapacity){
                        if(!ro.customers.contains(c1)){
                            ro.addCustomer(c1, false);
                        }
                    }
                }
            }
```

```

            //Remove any pairs that have visited customers
            // Also keep a tab on any customers we remove
            for(int j=i+1; j<pairs.size(); j++){
                Route r = pairs.get(j);
                Customer c1 = r.customers.get(0);
                Customer c2 = r.customers.get(1);
                byte a = 0;
                if(ro.customers.contains(c1)){
                    a+=1;
                }
                if(ro.customers.contains(c2)){
                    a+=2;
                }
                if(a>0){
                    if(a == 1){
                        abandoned.add(c2);
                    }else if(a == 2){
                        abandoned.add(c1);
                    }else if(a == 3){
                        abandoned.remove(c1);
                        abandoned.remove(c2);
                    }
                    pairs.remove(r);
                    j--;
                }

            }

        }

        //Edge case: A single Customer can be left out of all routes due to capacity constraints
        // abandoned keeps track of all customers not attached to a route
        for(Customer C:abandoned){
            //we could tack this onto the end of a route if it would fit
            //or just create a new route just for it. As per the Algorithm
            ArrayList<Customer> l = new ArrayList<Customer>();
            l.add(C);
            solution.add(l);
        }

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

6.2 VRSolution.java

Lines 20 to 28

```

    //Students should implement another solution
    public void clarkeWrightSolution(){
        ClarkeWright cw = new ClarkeWright();
        cw.truckCapacity = prob.depot.c;
        this.soln = cw.solve(prob.customers);
    }
```

6.3 Experiment.java

```
import java.util.*;
public class Experiment {

    public static void main(String[] args)throws Exception{
        String outdir = "output/";
        String problemdir = "tests/";
        String [] probs = {
                "rand00100",
                "rand00150",
                "rand00200",
                "rand00250",
                "rand00300",
                "rand00350",
                "rand00400",
                "rand00450",
                "rand00500",
                "rand00550",
                "rand00600",
                "rand00650",
                "rand00700",
                "rand00750",
                "rand00800",
                "rand00850",
                "rand00900",
                "rand00950",
                "rand01000"
                };
        for (String f:probs){
            ArrayList<Long> timing = new ArrayList<Long>();
            VRProblem vrp = new VRProblem(problemdir+f+"prob.csv");
            VRSolution vrs = new VRSolution(vrp);
            System.out.printf("%s, %d\n",f,vrp.size());
            for(int i=0;i<50;i++){
                long start = System.nanoTime();
                vrs.clarkeWrightSolution();
                long delta = System.nanoTime()-start;
                timing.add(delta);
                System.out.print(delta+", ");
            }
            System.out.print("\n\n");
            vrs.writeOut(outdir+f+"CWsn.csv");
        }

    }
}
```