**Productivity Apex, Inc.**

Drayage Route Optimizer  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Projects Included In Solution**

|  |  |
| --- | --- |
| **ConsoleDemo** | Executable - Sample Application Demonstrating Usage. Includes Ninject open source package for Dependency Injection |
| **Packages** | Open source and NuGet packages used in creating the Drayage Optimization Library [Moq.4.2.1402.2112, Ninject.3.0.1.10, Ninject.3.2.2.0, ninject.extensions.conventions.3.2.0.0, Ninject.MockingKernel.3.2.0.0, Ninject.MockingKernel.Moq.3.2.0.0, NUnit.2.6.3] |
| **PAI.Core** | Provides basic core objects and their implementation that are accessed across the other projects of the class library |
| **PAI.CTIP.Optimization** | Optimization Algorithm  Provides all the supporting objects, methods, and services necessary to transform the supplied list of drivers and jobs into an optimized route schedule. |
| **Tests** | An NUnit Unit Test project that confirms the optimization algorithm is operating as expected and is producing the correct output. |

**Summary**

The Drayage Optimizer will produce a solution set that represents the best determined sequence of stops based upon **Locations**, **Drivers**, **Jobs** and **Routes** provided to the algorithm.

The solution is built leveraging v4.5 of the .NET Framework. The optimization class library has a small footprint, and should run successfully on any computer capable of supporting .NET v4.5.

**Dependencies**

This solution has no external dependencies aside from the utilization of a Dependency Injector (ex: Ninject, Unity, Castle Windsor) to facilitate Inversion of Control. This process has been demonstrated in the included sample executable with open-source solution “Ninject”.

**Database Schema**

The Optimization Algorithm does not require a database in order to properly build solutions. As demonstrated in the sample executable, object can be manually initialized in order to produce a solution without the dependency of a database. If desired, object persistence can easily be achieved using any .NET supported database type and ORM, based upon project requirements and desired implementations.

**Overview of Key Objects**

**(PAI.CTIP.Optimization.Model.Location)  
Location:** a representation of latitude and longitude points that relate to a given location. The Location object is used to represent the address of each of the route stops. The location is identified by the DisplayName property, and the Latitude and Longitude properties are used to represent the coordinates. Each Driver is also assigned a StartingLocation.

**(PAI.CTIP.Optimization.Model.Orders.Driver)  
Driver:** represents a driver that is able to complete a Job. Essential properties are DisplayName (name/nickname), AvailableDrivingHours (maximum TimeSpan for driving in a given day), AvailableDutyHours (maximum TimeSpan for allowable work hours within a day), EarliestStartTime (TimeSpan representing the earliest a Driver can work), and StartingLocation (representing the driver’s starting point).

**(PAI.CTIP.Optimization.Model.Orders.Job)  
Job:** represents a series of load / unloads to be performed by a Driver to satisfy the delivery requirements for a client order. DisplayName identifies the Job, while RouteStops is a collection representing each “stop” of the job (minimum of 2 RouteStops must be provided for each job [pickup and drop-off]).

**(PAI.CTIP.Optimization.Model.Orders.RouteStop)  
RouteStop:** represents each stop / leg of a Job. Complemented by a StopAction which determines the specific action to be performed at each stop. RouteStop objects also have properties for Location (the coordinates of the load/unload), StopDelay (optional – representing the maximum anticipated duration of the stop while on-location), WindowStart and WindowEnd (the earliest and latest time window that a load/unload can be performed at this location).

**(PAI.CTIP.Optimization.Model.Orders.StopAction)  
StopAction:** represents the load/unload action that is to be performed at each stop. Available actions are: PickupChassis, DropOffChassis, PickupEmpty, DropOffEmpty, PickupEmptyWithChsassis, DropOffEmptyWithChassis, PickupLoaded, DropOffLoaded, PickupLoadedWithChassis, DropOffLoadedWithChassis, LiveLoading, LiveUnloading. The sequence of these actions must logically be valid in order for the Job to be processed (ex: if RouteStop 1’s action is PickUpLoadedWithChasis, RouteStop 2 cannot be DropOffEmpty, as the truck is already loaded and is not in the Empty state).

**Sample Code – Building a Solution**

Step 1: Initialize the Locations

// STEP 1 - Define the locations to be used by the Optimizer

// This includes all starting points, end points,

// route stop locations, driver starting locations

// driver starting location

var startLocation = new Location

{

DisplayName = "Home",

Latitude = 35.006,

Longitude = -89.8913

};

// route stop locations

var location1 = new Location

{

DisplayName = "Location 1",

Latitude = 35.0803,

Longitude = -89.963

};

Step 2: Initialize the Drivers

var d1 = new Driver

{

AvailableDrivingHours = 11.0,

AvailableDutyHours = 14.0,

DisplayName = "Driver 1",

EarliestStartTime = Helper.GetTimeSpan(6, 0),

StartingLocation = startLocation

};

Step 3: Create Job(s)

var job = new Job

{

DisplayName = "Job 1",

Id = 1

};

Step 4: Add Route Stops to the Job with corresponding StopActions.

job.RouteStops = new List<RouteStop>

{

Helper.CreateRouteStop(job, StopActions.PickupEmptyWithChassis, location1,

60, Helper.GetTimeSpan(6, 0), Helper.GetTimeSpan(22, 0)),

Helper.CreateRouteStop(job, StopActions.DropOffEmpty, location2,

60, Helper.GetTimeSpan(6, 0), Helper.GetTimeSpan(22, 0)),

Helper.CreateRouteStop(job, StopActions.PickupLoadedWithChassis, location2,

90, Helper.GetTimeSpan(6, 0), Helper.GetTimeSpan(22, 0)),

Helper.CreateRouteStop(job, StopActions.DropOffLoadedWithChassis, location3,

60, Helper.GetTimeSpan(6, 0), Helper.GetTimeSpan(22, 0))

};

Step 5: Place all created Jobs and Drivers within a collection.

var jobs = new List<Job>() {job, job2};

var drivers = new List<Driver>() {d1, d2};

Step 6: Initialize the Optimizer and generate the solution.

var optimizer = GetService<IDrayageOptimizer>();

optimizer.Initialize();

// build the solution

var solution = optimizer.BuildSolution(drivers, drivers[0], jobs);

Step 7: Output solution results:

Console.WriteLine("Solution Created");

Console.WriteLine(solution.RouteSolutions.Count + " route solutions.");

Console.WriteLine(solution.UnassignedJobNodes.Count + " unassigned jobs.");

Console.WriteLine(solution.RouteStatistics.TotalTime + " : total time.");

int count = 0;

foreach (var routeSolution in solution.RouteSolutions)

{

count++;

Console.WriteLine("\n\tROUTE #" + count);

Console.WriteLine("\n\tAssigned to Driver: " + routeSolution.DriverNode.Driver.DisplayName);

Console.WriteLine("\t\tTravel Distance: " + routeSolution.RouteStatistics.TotalTravelDistance + " miles");

Console.WriteLine("\t\tEstimated Total Time: " + routeSolution.RouteStatistics.TotalTime);

Console.WriteLine("\t\tEstimated Travel Time: " + routeSolution.RouteStatistics.TotalTravelTime);

Console.WriteLine("\t\tNodes:");

foreach (var node in routeSolution.AllNodes)

{

foreach (var routeStop in node.RouteStops)

{

Console.Write("\t\t\t" + routeStop.Location.DisplayName);

if (routeStop.StopDelay.HasValue)

Console.Write(" - " + routeStop.StopDelay.Value.TotalMinutes + " minute stop" );

Console.Write("\n");

}

}

}