Dynamic Ridesharing (D-RIDE) Tool

Dynamic Mobility Application

Booz Allen Hamilton

Arizona State University

TABLE OF CONTENTS

Introduction

This repository contains the D-RIDE source-code, executable file, input files, and NEXTA graphical user interface (GUI). The D-RIDE application is a sequence of algorithms designed to calculate the pick-up and drop-off sequence for a given set of vehicles to transport given passengers. Each passenger has a specific time window requirement for picking up and dropping off. This algorithm can minimize the number of required vehicles and travel the shortest distance to meet all the passengers' requirements. The application is coded in C++ language and is developed by Arizona State University.

D-RIDE

The D-RIDE application is an executable file recognized as a .exe extension. The application provides a car-pooling system in which drivers and riders arrange trips within a relatively short time in advance of departure. A person could arrange daily transportation to reach a variety of destinations, including those that are not serviced by transit. Used on a one-time, trip-by-trip basis, and would provide drivers and riders with the flexibility of making real-time transportation decisions. The application increases the use of non-transit ride-sharing options including carpooling and vanpooling, and improve the accuracy of vehicle capacity detection for occupancy enforcement and revenue collection on managed lanes.

D-RIDE Input Files and Execution

The D-RIDE application requires four basic input files recognized as .csv native format and are listed as follows: 1. input_agent.csv; 2. input_configuration.csv; 3. input_link.csv; and 4. input_node.csv. The input agent file defines the agent trip demand with specifications on origin, destination, departure time, arrival time, and capacity. The input configuration file defines several attributes for the algorithm which includes the number of iterations for optimization, shortest path debugging details, vehicle cost per hour, and number of computer processing threads used. Finally the input link and node files define the roadway network used as the environment where D-RIDE will perform path finding. Users are encourages to reference the following documentation for more information on the input data structure https://sites.google.com/site/dtalite/.

The D-RIDE executable file is labeled 'AgentPlus.exe' and must be located within the same directory as the four input files. Once the D-RIDE file is executed and the agent routes have been produced, an 'agent_path.csv' file is generated listing the list of paths, or links the agent will traverse.

Graphical User Interface (GUI)

The following Figure 1 shows an example of the NEXTA GUI application. The program is loaded with the output file from the agent_path.csv file and the network file from the link and node files.

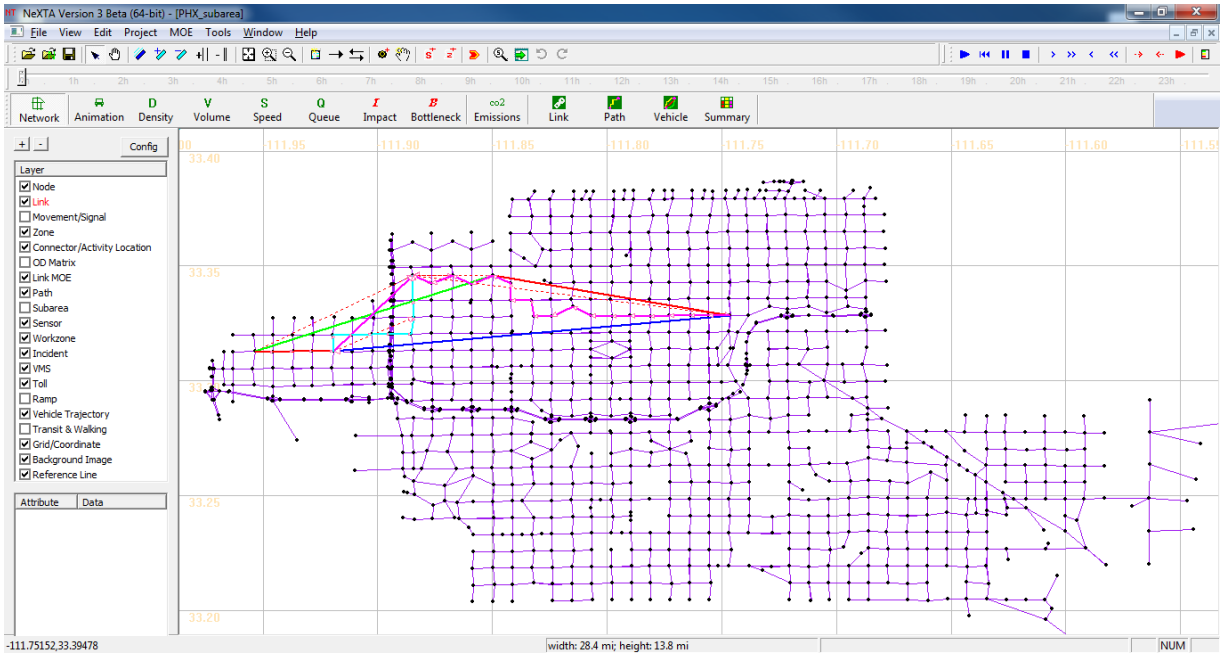

Figure 1: Sample NEXTA Graphical User Interface

To display the paths generated from D-RIDE along the network node and link input files, the file must be loaded into the NEXTA environment through the "Open Traffic Network Project" option which allows the network and data be imported. The D-RIDE paths can be displayed by selecting the Config button located on the left side of the GUI. The following Figure 2 shows the pane for the Display Configuration.

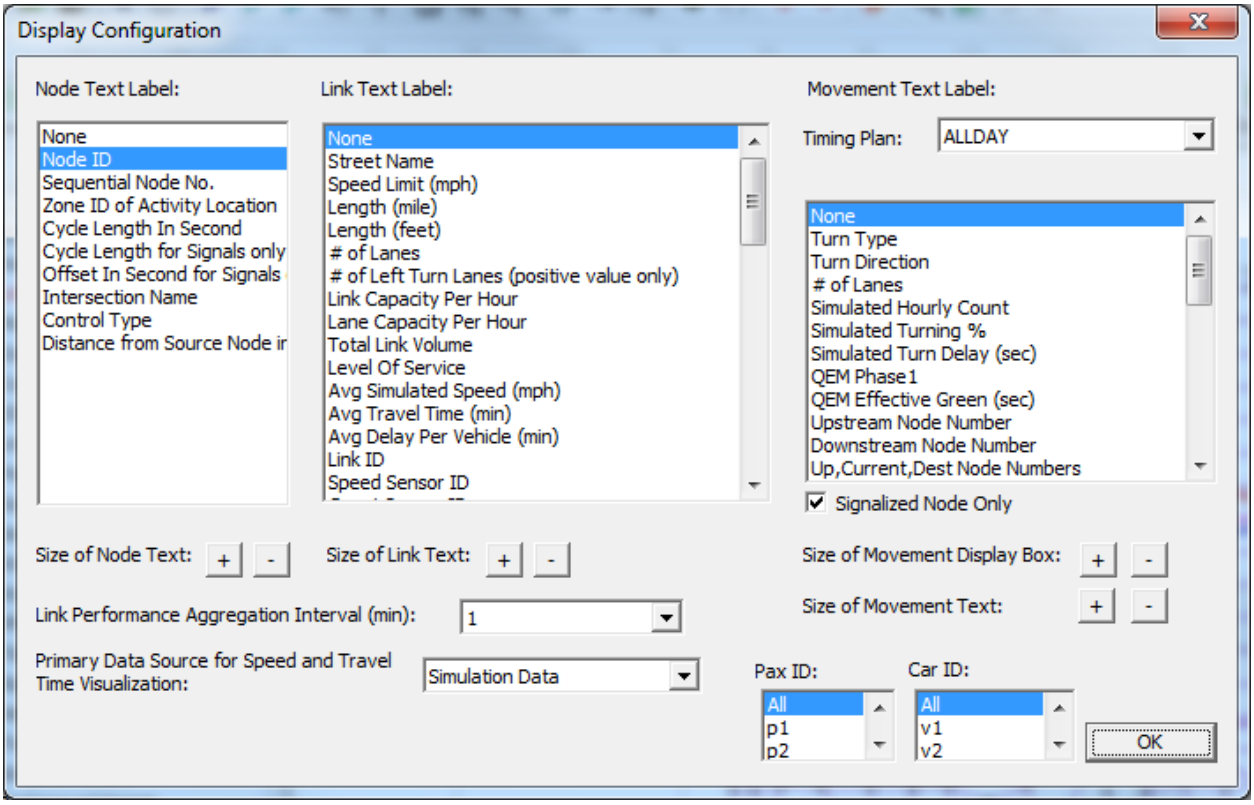

Figure 2: Display Configuration Pane

Located at the bottom right next to the "OK" button is the passenger and car paired paths found by D-RIDE. Selecting on individual agents displays their corresponding individual paths. Selecting the "All" option displays all paired paths as shown in Figure 3.

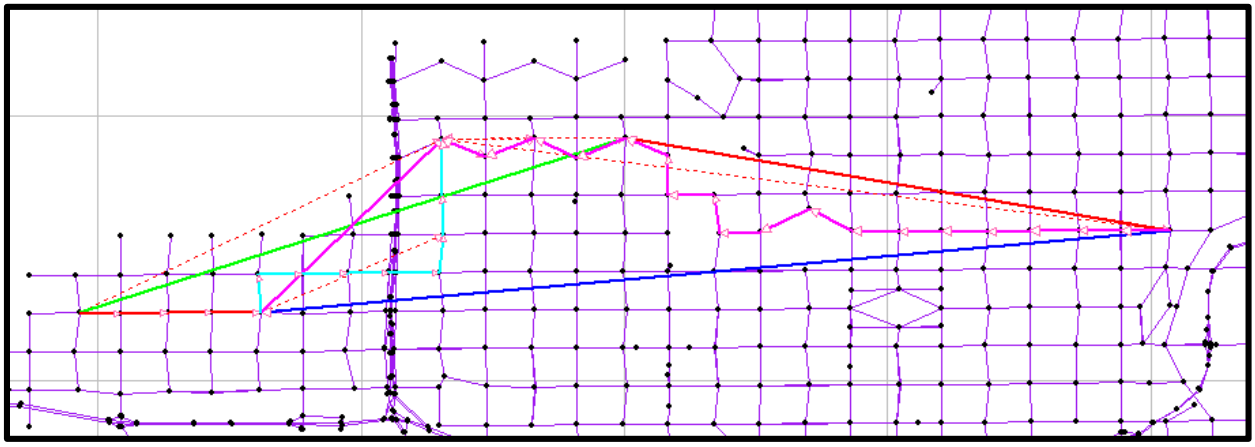

Figure 3: D-RIDE Paths in NEXTA

D-RIDE Framework

Shown in Figure 4, after transportation network geometry, available vehicles and passenger demands are loaded, they first go into the AgentPlus, a program containing D-RIDE and vehicle routing algorithms. The outputs of AgentPlus include suggested each vehicle's pickup and delivery sequence and corresponding paths to satisfy all passengers' needs (if possible at all) while minimizing the overall cost. The outputs of AgentPlus will then go into DTALite. In the meantime, the proposed algorithms also have a mechanism to determine the best dynamic pricing strategy for vehicles which is important to have a sustainable development of D-RIDE applications.

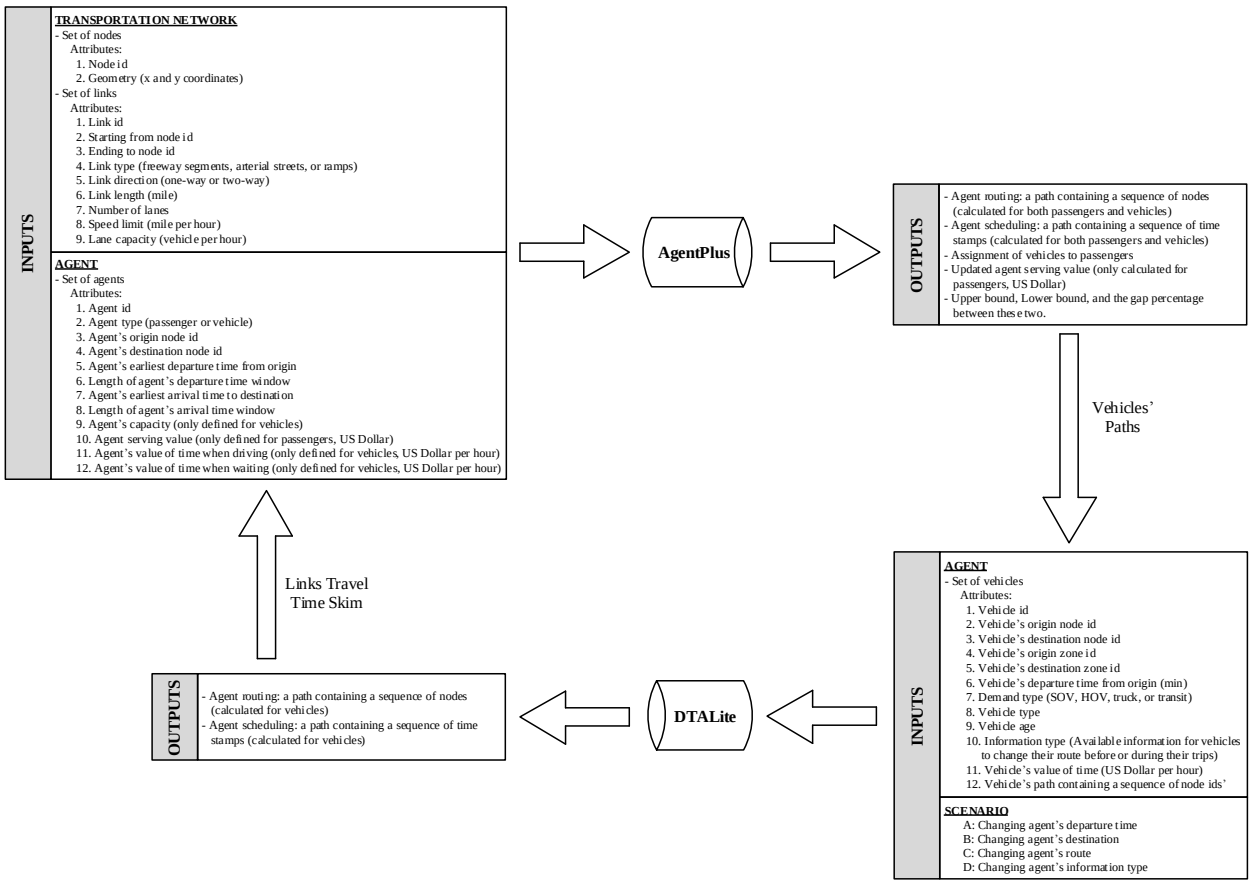

Figure 4: The Framework of D-RIDE simulation platform

Data flow process in the D-RIDE simulation platform

Given an agent file in a transportation network, DTALite simulates the route decision and produces the network-wide and link-level performances, as well as output agent results. An activity based model (ABM) can read the traffic performance data to further generate an updated agent file for another iteration of traffic simulation. Here are main files involved within the D-RIDE simulation. Table 1 provides a more detailed description of the files' functions and explanation of data content.

Table 1: Types of Data Block in Integration

Data Block	File Name		Source of Data	Major Attributes
A: Basic Input Files	A1: input_node.csv		Prepared from static transportation planning packages, GIS data sets	Node id, geometry
	A2: input_link.csv			Link capacity, length, speed limit
	A5: input_scenario_settings.csv			Traffic assignment methods, traffic flow models
Day-by-day Integration	B: Input file	B1: input_agent.csv	AgentPlus provide input to DTA	Departure time, path node sequence, information types, demand types
	C: Output files	C1: output_agent.csv	DTA's simulation output	Path node sequence, OD, trip time, distance
		C3: output_summary.csv		DTA version, simulation setting process, simulation results

Working Steps

Step 0: Use NeXTA to prepare the basic network and demand data definitional data files from A1 to A5

Step 1: AgentPlus prepares input agent file B1 as a starting point: Input agent file B1 stores agents' essential travel information (origin, destination and departure time, demand type and vehicle type). Go to Step 1:

Step 2: DTA assignment/simulation: DTA here finds the least cost routes for individual agents and further simulates the interactions of agents to generate network-wide and link-level traffic performance. DTALite generates output agent file C1 with detailed trip trajectories and experienced travel time along selected travel path sequence. For running DTA for multiple days, users could access file A5 scenario setting to specific the day-to-day iterative runs, DTA will run toward user equilibrium for K iterations or days using Methods of Successive Average (MSA). Go to Step 3.

Step 3: Iteratively updating agent input from ABM to DTA .The output files in Data Block C (including zone-to-zone travel cost skim data C2) are feedback files for ABM to generate updated trip rates, destination and departure times for the agents for the next integration iteration. Go back to Step 2 to run another round of DTA simulation to update network wide traffic conditions.

Algorithm Development for Ride Sharing Applications

Time-dependent Dynamic Programming

In this section, the time-dependent dynamic programming algorithm we use for solving the least cost path problem is presented. As we mentioned before, there are associated a node id i, a time stamp t, and a passengers' carrying state w with each vertex of our state-space-time network. Let $L(i,t,w)$ denote the label of vertex (i,t,w). The algorithm can be described as follows:

```
// time-dependent dynamic programming algorithm
    for each vehicle v ∈ (V ∪ V*) do
    begin
        // Initialization
        L(i,t,w) := ∞;
        node predecessor of vertex (i,t,w) := −1;
        time predecessor of vertex (i,t,w) := −1;
        state predecessor of vertex (i,t,w) := −1;
        // vehicle v starts its route from the empty state at its origin at the earliest departure time
        L(o'_v, e_v, w_0) := 0;
        for each time t ∈ [e_v, l_v] do
        // each unit of time is assumed to be one minute.
        begin
            for each link (i,j) do
            // node i and j can be a physical transportation or a dummy node
            // node i and j should be accessible for vehicle v
            begin
                for each current passengers' carrying state w do
                // infeasible states and those states exceeding vehicle v's capacity should be skipped
                begin
                    for each next passengers' carrying state w' do
                    // only feasible states and skip all states exceeding vehicle v's capacity
                    begin
                        s := t + TT(i,j,t) ;
                        // TT(i,j,t) is the travel time from node i to node j starting at time t.
                        if (L(i,t,w) + ξ(v,i,j,t,s,w,w') < L(j,s,w')) then
                        begin
                            L(j,s,w') := L(i,t,w) + ξ(v,i,j,t,s,w,w') ;
                            node predecessor of vertex (j,s,w') := i;
                            time predecessor of vertex (j,s,w') := t;
                            state predecessor of vertex (j,s,w') := w;
                        end;
                    end;
                end;
            end;
        end;
    end;
```

Lagrangian Relaxation Algorithm

In this section, we present Lagrangian relaxation algorithm in order to show how $\xi(v,i,j,t,s,w,w')$ is updated. According to equation (2-14),

$$\xi(v,i,j,t,s,w,w') = \begin{cases} c(v,i,j,t,s,w,w') + \lambda(p) & \forall (i,j,t,s,w,w') \in \Psi_{p,v} \\ c(v,i,j,t,s,w,w') & Otherwise \end{cases},$$

so $\xi(v,i,j,t,s,w,w')$ is only updated for $\forall (i,j,t,s,w,w') \in \Psi_{p,v}$. The Lagrangian relaxation (LR) algorithm can be described as follows:

```
// Lagrangian relaxation algorithm
// Initialization for the first LR iteration
```

Initialize arc multiplier $\lambda(p)^1 = 0$ for $\forall p \in P$.

Initialize step size θ_1. // θ_1 can be interpreted as the amount of money ($) passenger p offers to be served.

for each LR iteration k **do**

begin

reset the visit count for each arc $(v, i, j, t, s, w, w') \in \Psi_{p,v}$ to zero.

global lower bound $= 0$;

global upper bound $= 0$;

for each vehicle $v \in V$ **do**

begin

solve time-dependent dynamic programming algorithm for vehicle v.

update the visit count for each arc $(v, i, j, t, s, w, w') \in \Psi_{p,v}$.

end;

calculate the total number of visits by $\sum_v \sum_{(i,j,t,s,w,w') \in \Psi_{p,v}} y(v, i, j, t, s, w, w')$

update arc multiplier $\lambda(p)^{k+1} = \lambda(p)^k + \theta_k(total\ number\ of\ visits - 1)$ for $\forall p \in P$

end;

Step 1. **Initialization.**

Set iteration number $k = 0$, choose a positive value for the Lagrangian multiplier $\lambda(p)^k$ for $\forall p \in P$.

Step 2. **Passenger routing.**
For $\forall p \in P$
Initialize least cost path= passenger p's travel budget
For $\forall v \in V$

- Given $\pi^k_{i,j,t,s}(v)$, solve the decomposed dual sub-problem P_X using a standard TDLCP algorithm and find the least-cost-path solution X for passenger p.
- If the least-cost-path < existing least-cost-path, then update the vehicle assignment to passenger p.

Step 3. **Vehicle routing.**
For $\forall v \in V$

- Given $\pi^k_{i,j,t,s}(v)$, solve the decomposed dual sub-problem P_Y using a standard TDLCP algorithm and find the least-cost-path solution Y for vehicle v.
- If the least-cost-path < existing least-cost-path, then update the vehicle route.

Step 4. **Duality gap.** Calculate the objective function value of the primal and dual problems, then find the value of duality gap.

Step 5. **Checking capacity constraints.** If all vehicles' capacity constraints are satisfied, the optimal solution has been reached, otherwise:

- Compute sub-gradients $\nabla L_{\pi_{o_p,j,t,s}(v)}$
  $\nabla L_{\pi_{o_p,j,t,s}(v)} = \sum_{p' \in P} x_{o_p,j,t,s}(p', v) - Q(v) y_{o_p,j,t,s}(v)$
- Update Lagrangian multiplier $\pi^k_{P_{o_p},j,t,s}(v)$ by
  $\pi^{k+1}_{o_p,j,t,s}(v) = \pi^k_{o_p,j,t,s}(v) + \theta_k \nabla L_{\pi_{o_p,j,t,s}(v)}$
  Where θ_k is the step size at iteration k which can be defined by the method of successive average.

Step 6. **Termination condition test.** If *k* is less than a predetermined maximum iteration value, or the duality gap is less than a predefined toleration gap, terminate the algorithm, otherwise *k=k+1* and return to Step 2.