DTALite Implementation Details

* **Flow Chart Overview:**



Figure 0.1: Flow chart overview

We will illustrate the whole process of STALite by a specific example *two\_corridor,* and this programming guide will be expanded in the order of figure 0.1. You can find some basic parameter definitions in the user’s guide. Here, we pay more attention to the specific execution process of the program.

* **Classes Overview**

Here, we do an overall overview of the whole class.

Table 0.1: Classes Overview

|  |  |
| --- | --- |
| **Class** | **Description** |
| class CCSVParser | Class for reading and writing .csv. |
| class CDemand\_Period | Class for demand period /  Calculate starting(ending) time |
| class CAgent\_type | Class for different demand agent type  (including agent’s VOT/PCE/CRU) |
| class CLinkType | Class for linktype |
| class CColumnPath | Class for node sequence/link sequence of a column. |
| class CAgentPath | Path sequence and statistics for each agent |
| class CColumnVector | Path sequence and statistics for each column |
| class CAgent\_Column |  |
| class CAgent\_Simu | Class for the agent in the simulation |
| class Assignment |  |
| class CVDF\_Period | Calculate Volume Delay Function Period  (using BPR function) |
| class CLink | Class for roadlink |
| class CServiceArc | Class for service arc |
| class CNode | Class for node |
| class COZone | Class for origin zone |
| class CAGBMAgent |  |
| class NetworkForSP | Class for shortest path calculation |
| class VehicleScheduleNetworks |  |
| class CVSState | Class for vehicle scheduling states |
| class C\_time\_indexed\_state\_vector |  |

Here, we explain some important classes briefly:

* **Class Assignment**

*Class Assignment* includes some basic globe parameters or variables for computing assignment, such as number of nodes, number of links, etc.

Initialize the demand array for each zone, agent type, demand period, and total demand array for each zone, period.

* **Class NetworkForSP**

*Class NetworkForSP* provides some basic memory space to generate the shortest path tree for each origin zone.

**void AllocateMemory(int number\_of\_nodes, int number\_of\_links)**

Allocate memory and initialize the data for shortest path calculation.

**void UpdateGeneralizedLinkCost()**

The generalized link cost for each link can be calculated by the following mathematical formula.

genalized\_cost= travel\_time\_per\_period + route\_choice\_cost + toll / m\_value\_of\_time

**void BuildNetwork(Assignment\* p\_assignment)**

Build subnetwork in each workbench based on agent type and link-type at iteration 0.

The *m\_outgoing\_link\_seq\_no\_vecto*r and *m\_to\_node\_seq\_no\_vecto*r are defined to store outgoing link for each node and link.

**1. Read Input Data**

Section 1.1~1.5 introduces how to read basic traffic network(node/link) data, and you can find the detailed source code in function *void g\_ReadInputData(Assignment& assignment).*

Section 1.6 introduces how to read the demand file, and you can find the detailed source code in function *void g\_ReadDemandFileBasedOnDemandFileList(Assignment& assignment).*

**1.1 Read *demand\_period.csv***

Table 1.1: *demand\_period.csv*

|  |  |  |
| --- | --- | --- |
| **demand\_period\_id** | **demand\_period** | **time\_period** |
| 1 | AM | 0700\_0800 |

We use the *class CDemand\_Period* to store demand\_period and time\_period, and each demand period message will be stored in vector *assignment.g\_DemandPeriodVector.* For easy searching, we use the map data structure *assignment.demand\_period\_to\_seqno\_mapping.* In this example *two\_corridor*, there is only one piece of demand period information in vector *g\_DemandPeriodVector.*

**1.2 Read** ***link\_type.csv***

Table 1.2: *link\_type.csv*

|  |  |  |
| --- | --- | --- |
| **link\_type** | **link\_type\_name** | **type\_code** |
| 1 | Highway/Expressway | f |
| 2 | Major arterial | a |

We use the *class CLinkType* to store link type name and the abbreviation of link type(“f” means Highway/Expressway, “a” means Major arterial), and each link type message will be stored in array *assignment.g\_LinkTypeMap.*

**1.3 Read *agent\_type.csv***

Table 1.3: *agent\_type.csv*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **agent\_type** | **name** | **VOT** | **flow\_type** | **PCE** |
| p | passenger | 10 | 0 | 1 |

We use the *class CAgent\_type* to store some attributes of agent type, such as VOT(value of time), flow type(0:flow, 1:fixed path, 2:integer decision variables), and PCE(the vehicle conversion factor). Each agent type message will be stored in vector *assignment.g\_AgentTypeVector.*

**1.4 Read** ***node.csv***

Table 1.4: *node.csv*

|  |  |  |  |
| --- | --- | --- | --- |
| **node\_id** | **zone\_id** | **x\_coord** | **y\_coord** |
| 1 | 1 | 0.017882 | -0.12518 |
| 2 | 2 | 40.25393 | 0.053648 |
| 3 |  | 19.77825 | 14.80687 |
| 4 |  | 19.68884 | -9.69242 |

We use the class *Cnode* to store the node attributes including node\_id, zone\_id, x\_corrd, and y\_coord. The difference between node\_id and zone\_id is that zone\_id is prepared for the node with traffic demand or traffic attraction volume.

**1.5 Read *road\_link.csv***

We use the class *Clink* to store the link attributes. Here, we use an example link to illustrate the details. All link messages will be saved into a globe vector *g\_link\_vector.* Some attributes are based on BPR function and VDF(Volume delay function) to link a.



Table 1.5: *road\_link.csv*

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Example value** | **Storage location** | **Description** |
| road\_link\_id | 1003 | link.link\_id |  |
| from\_node\_id | 1 | link.from\_node\_seq\_no  (after mapping) |  |
| to\_node\_id | 3 | link.to\_node\_seq\_no  (after mapping) |  |
| facility\_type | Freeway |  |  |
| dir\_flag | 1 |  |  |
| length | 20 | link.lenth |  |
| lanes | 1 | link.number\_of\_lanes | Roadlink lane numbers |
| capacity | 4000 | link.link\_spatial\_capacity |  |
| free\_speed | 60 |  |  |
| link\_type | 1 |  | 1: Highway/Expressway  2: Major arterial |
| cost | 0 |  |  |
| VDF\_fftt1 | 20 | link.free\_flow\_travel\_time | Free flow travel time in VDF |
| VDF\_cap1 | 4000 | The calculation result  according to BPR function  will be saved in  link.PCE\_at and Link.CRU\_at. | The capacity parameter in VDF |
| VDF\_alpha1 | 0.15 | Parameter α in VDF |
| VDF\_beta1 | 4 | Parameter β in VDF |
| VDF\_theta1 | 1 | Parameter θ in VDF |
| VDF\_gamma1 | 1 | Parameter γ in VDF(Optional) |
| VDF\_mu1 | 100 | Parameter μ in VDF(Optional) |

\*The rest of the parameters in *Clink* are not read from the file and will be introduced later.

**1.6 Read *demand\_file\_list.csv* and *demand\_p.csv***

Table 1.6: *demand\_file\_list.csv*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **scenario\_no** | **file\_sequence\_no** | **file\_name** | **format\_type** | **demand\_period** | **agent\_type** |
| 0 | 1 | demand\_p.csv | column | AM | p |

Table 1.7: *demand\_p.csv*

|  |  |  |
| --- | --- | --- |
| **from\_zone\_id** | **to\_zone\_id** | **number\_of\_passengers** |
| 1 | 2 | 7000 |

The *demand\_file\_list.csv* provides document indexes for different types of demand(*demand\_p* means the passenger demand, *demand\_v* means the demand for the vehicle, etc.).

We use 4 different statistics and arrays to save demands,

* total\_demand[agent\_type\_no][demand\_period\_no]: total demand for each agent type and period
* g\_column\_pool[from\_zone\_seq\_no][to\_zone\_seq\_no][agent\_type\_no][demand\_period\_no].

od\_volume: demand for each column(for each OD, agent type and peroid)

* total\_demand\_volume: total demand
* g\_origin\_demand\_array[from\_zone\_seq\_no][agent\_type\_no][demand\_period\_no]: trip generation volume for each origin zone, agent type and period.

For this example:

total\_demand[0][0] = 7000;

g\_column\_pool[0][1][0][0].od\_volume = 7000;

total\_demand\_volume = 7000;

g\_origin\_demand\_array[0][0][0] = 7000;

**2.Allocate memory for optimization and assign computing tasks**

To achieve parallel computing, we should prepare memory space for calculation tasks and assign each OD into corresponding workbench or thread. You can find the detailed source code in function *void g\_assign\_computing\_tasks\_to\_memory\_blocks(Assignment& assignment).*

Take 4 threads and 8 od pairs as an example, we divide all OD pairs into 4 parts and realize parallel computing in different workbenches(as shown in figure 2.1). But the *two corridor* example only has 1 OD, so we can only one thread.



Figure 2.1: Schematic diagram for parallel computing

Here, we use the array *NetworkForSP\* PointerMatrx* to record the corresponding position in different threads for each OD pair. The main purpose of establishing the workbench is to calculate the shortest path for each OD. You can refer *class NetworkForSP* for details.

For this example:



Figure 2.2: Parallel computing for example

**Notes:**

Outer Loop: Loop for column pool optimization iterations, which are set as the default value(Outer iteration number=1).

Inner Loop: Loop for column generation iterations, which can be set in *setting.csv.*

**3.Column Pool Generation**

Find Shortest Path and Update Path Cost in each iteration. Unless otherwise specified, we will take the first iteration as an example to talk about the program execution process in detail.

**3.1 Reset and update link volume and cost**

There are 4 modes to reset or update link volume based on BPR function, and you can find the detailed source code in the corresponding function.

* **void update\_link\_travel\_time\_and\_cost()**

We use the BPR function to update travel time(TT):



Where,

variables ta (variable avg\_travel\_time in code) represents generalized travel cost (or time) on link a;

variable *fftta* (variable FFTT in code) is the free-flow travel time of link a;

variable *Va* or *Xa*(variable flow\_volume\_per\_period or volume in code) represents flow volume on link a. Typically, *Va* is used to cover composite flow volume in different applications, while Xa is used to denote nonnegative pure flow or fluid from the network flow modeling perspective.

Ca (variable capacity in code) represents the (ultimate) capacity of link a;

θa, αa, βa are the coefficients of VDF function for link a.

* **void g\_reset\_link\_volume\_in\_master\_program\_without\_columns()**

Only for User Equilibrium(UE) mode.

* **void g\_reset\_link\_volume\_for\_all\_processors()**

Only for User Equilibrium(UE) mode.

* **void g\_reset\_and\_update\_link\_volume\_based\_on\_columns()**

Only for System Optimization(SO) and User Equilibrium(UE) with resource constraints.

For this example:

**Iteration 1:**

Column Pool = { }

Because the pool is empty, this step is skipped.

**Iteration 2:**

Column Pool = {Column 1}

Column 1 = {Path: 1-3-2; Volume = 7000}

After this step, we can get the travel time for column 1 and global link volume and travel time. As for how to generate a column, you can refer to section 3.2

**3.2 Label Correcting Algorithm**

According to the road network with updated link cost, we can use the label method to generate the columns. There are 3 steps in generating a path column: build the sub-network for each origin zone, find the shortest path tree in sub-network, and do back-trace by the label.

* **void BuildNetwork(Assignment\* p\_assignment)**

Build the network for each O-D pair using the adjacency list.

For this example(sub-network for origin\_zone\_no=1):



Figure 3.1: Physical network for example

* **float optimal\_label\_correcting(int processor\_id, Assignment\* p\_assignment, int iteration\_k, int o\_node\_index, int d\_node\_no, bool pure\_travel\_time\_cost)**

Label correcting algorithm with double queue implementation.

Generate the shortest path tree.

The pseudo-codes details of Label Correcting are as follows:

|  |
| --- |
| **Algorithm：Deque Label Correcting Algorithm** |
| **begin**  //Initialization  m\_label\_distance\_array[from\_node] = 0;  m\_node\_predecessor[from\_node] = 0;  m\_label\_distance\_array[j] = ∞ for each node j∈N-{from\_node}；  SEList.pushback(from\_node);  **while** SEList ≠ Φ **do**  **begin**  **remove** the first element on the left from SEList;  **for each** arc (i,j)∈A(i) **do**: //A(i) means the outgoing link set of node i  **if** m\_label\_distance\_array[j] > m\_label\_distance\_array[i] + link\_cost(i,j) then:  **begin**  m\_label\_distance\_array[j] = m\_label\_distance\_array[i] + link\_cost(i,j);  m\_node\_predecessor[j] = i;  **if** node j has not been in SEList earlier then add node j to the right end of SEList;  **else** add node j to the left end of SEList;  **end;**  **end;**  **end;** |

For this example:

**Iteration 0:**

The current road network is shown in figure 3.1. According to the label correcting method, we can easily find a one to shortest-path tree for each origin zone.

Here are the shortest path tree and node predecessor table for origin\_zone\_no = 1:



Figure 3.2: Shortest path tree for example

Table 3.1: Node predecessor for SPP tree

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 |
| Shortest path cost | 0 | 20 | 10 | 15 |
| node\_predecessor | -1 | 3 | 1 | 1 |

**3.3 Backtrace and generate column pool**

* **void NetworkForSP::backtrace\_shortest\_path\_tree(Assignment& assignment, int iteration\_number\_outterloop, int o\_node\_index)**

For each node, find it's the shortest-path tree and backtrace the SPP tree from destination to origin and load flow count at the same time. Using the node-sum method to append a new path(like the hash-table method, as shown in figure 3.3).



Figure 3.3: Node-sum method

For this example:

**Iteration 0:**

For origin zone = 1 and destination zone = 2, we can get a feasible column(1-3-2) by the node predecessor table.



Figure 3.4: Backtrace for shortest path tree

At this point, we get the column pool in iteration 0:

column pool = {column 1};

column 1 = {1,3,2}.

You can use the following standard format to achieve the label correcting algorithm with parallel computing in different workbenches.

\*Please refer to Guid**e** into OpenMP:Easy multithreading programming for C++**.**

\*The code template:

#pragma omp parallel for

for (int ProcessID = 0; ProcessID < g\_NetworkForSP\_vector.size(); ProcessID++)

{

……

//Perform one to all shortest-path tree calculation in each processor

}

**4.Column Pool Optimization**

Given the generated column pool, we can optimize the column pool by adjusting flow counts between columns.

* **void g\_column\_pool\_optimization(Assignment& assignment, int column\_updating\_iterations)**

Call *g\_update\_gradient\_cost\_and\_assigned\_flow\_in\_column\_pool* for *column\_updating\_iterations* times

* **void g\_update\_gradient\_cost\_and\_assigned\_flow\_in\_column\_pool(Assignment& assignment, int inner\_iteration\_number)**

Based on the newly calculated path volume and column pool, we update volume-based travel time and update volume-based resource balance using the reduced gradient method. Here, we use this example to illustrate the implementation details.

|  |
| --- |
| **Algorithm: Column Pool Optimization** |
| **Input:** Generated Column Pool(after 2 column generation iterations) |
| **//Step 1: Calculate the gradient cost for each column cost.**  Column 1 total cost(path\_gradient\_cost[1]) = 38.34  Column 2 total cost(path\_gradient\_cost[2]) = 21.76  least\_gradient\_cost = min{path\_gradient\_cost}= 21.76  **//Step 2: Calculate gradient cost difference for each column cost.**  **for each non-shortest path column(only column 1 here):**  path\_gradient\_cost\_difference[1] = path\_gradient\_cost[1] - least\_gradient\_cost= 16.58  path\_gradient\_cost\_relative\_difference[1] = path\_gradient\_cost\_difference / max(0.0001, least\_gradient\_cost) = 0.7619  **//Step 3: Update path flows**  step\_size = 1.0 / (iteration\_number + 2) \*od\_volume =1/(0 + 2) \* 7000 = 3500  **//Step 3.1: Shift flow form non-shortest path to shortest path**  path\_volume[1] = max(0, path\_volume[1] - step\_size \* path\_gradient\_cost\_relative\_difference) = max(0, 3500 – 3500 \* 0.7619) = 833.35  total\_switched\_out\_path\_volume = previous\_path\_volume[1]- path\_volume[1] =  3500 – 833.35 = 2666.65  **end for**  **//Step 3.2: Consider the least-cost path(column 2 here), receive all volume shifted from the non-shortest path.**  path\_volume[2] = path\_volume[2] - total\_switched\_out\_path\_volume = 3500 + 2666.65 = 6166.65  **//Step 4: Based on newly calculated path flows, update volume, and travel time.** |

**5.Simulation**

* **void Assignment::STTrafficSimulation()**

Given an optimization result, make the simulation, output the specific flow for each link and time, and output the departure time and arrival time for each agent.

For each simulation interval, active the agent, update the entrance queue, and exit queue.

* **void Assignment::AllocateLinkMemory4Simulation()**

Prepare memory space for the simulation process.

|  |
| --- |
| **STALite Simulation** |
| **//Step 1: Scan the different continuous paths in the column pool and generate a simulation agent for each column(In this example, an agent represents a vehicle).**  We use the class *CAgent\_Simu* to store simulation time and simulation route, etc., and each simulation agent will be stored in vector *g\_agent\_simu\_vector.*  **for each** simulation\_interval:  **//Step 2: Simulate traffic conditions at time t(as shown in figure 5.1).**  Inherit the CumulativeDeparture and CumulativeArrival value from the previous simulation-interval period for each link.  //**Step 2.1: Active simulation agents that are going to depart from the origin node.**  **//Step 2.2: Push back the simulation agents into the corresponding entrance queue.**  **//Step 2.3: Pop the simulation agents from the entrance queue and push it back into the exit queue, and update the total travel time at the same time**.  **//Step 2.4: Check whether the agent has reached the destination node.**  **//Step 3: Count the departure numbers and arrival numbers for each link at time t.**  **//Step 4: Check link capacity and termination condition for each activated simulation agent.**  t = t->next simulation\_interval;  **end for** |



Figure 5.1: Simulation principle

**6.Output Results**

The output results mainly include two files, *link\_performance.csv and agent.csv*(based on simulation), and you can find the detailed source code in function *void g\_output\_simulation\_result (Assignment& assignment).***7.** **Calculation Process Overview**

Table 7.1: Calculation Process Overview

|  |  |  |  |
| --- | --- | --- | --- |
| Iteration | Network | Column Generation | Column Pool |
| 0 |  |  |  |
| 1 |  |  |  |
| 2 |  |  |  |
| Column Pool Optimization | | | |
|  | | | |