The Dynamic Vehicle Routing Problem

Francis Fu Yao | francis_yao@pku.edu.cn | 2017.01.13

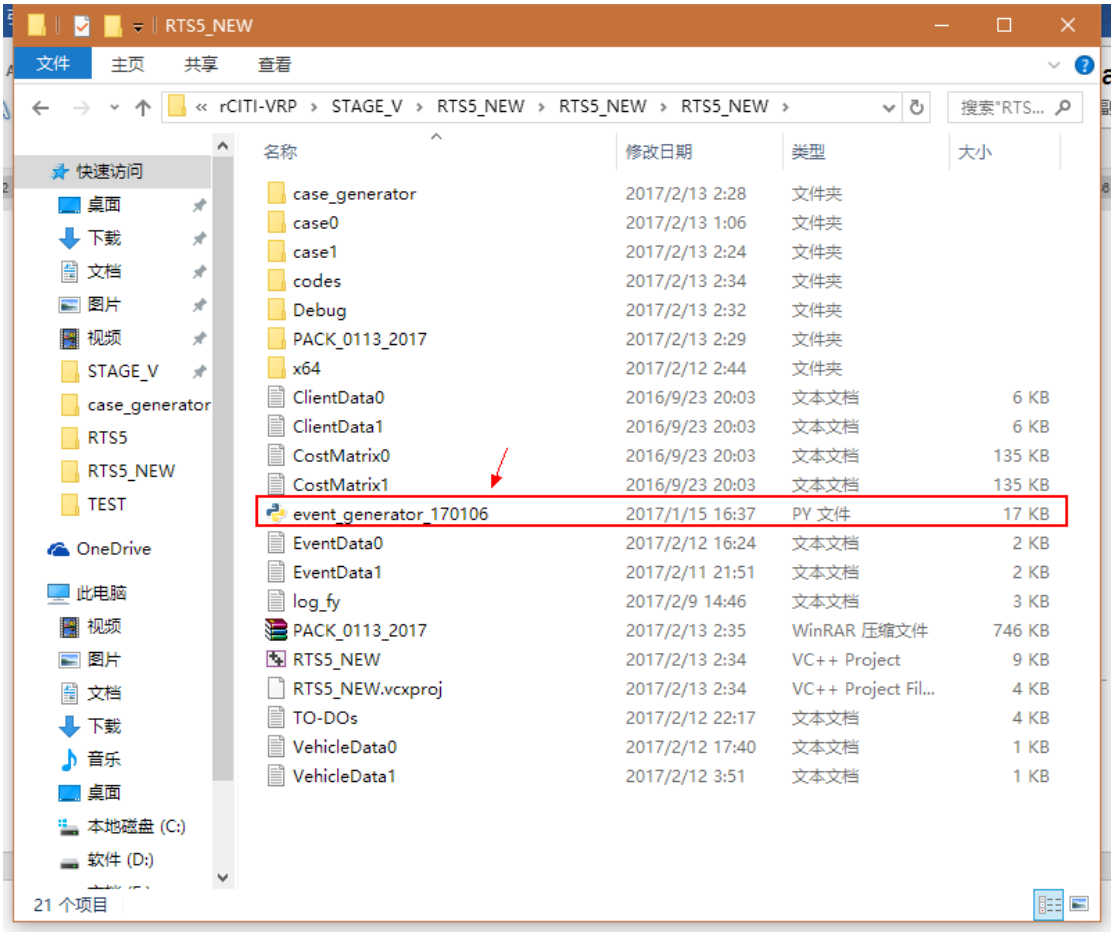
| Basic logic of the DVRP project

```
main()
|-routing.run()
  |-init()
  |-file_input()
  |-build_init_solution()
  |-perform_tabu_search()
  |-perform_opt()
  |-perform_waste_distribution()
  |-write_generator_input()
  |-system(generator_command.c_str())
  |-read_events()
  |-event_handling_init()
  |-route_events_main()
    |-loop:
    | -bestsolution.push_forward_time()
       |-get_next_upcoming_event()
       |-vehicles move to next nodes after next upcoming event
       |-collect events on the way
    | -bestsolution.batch_event_handling()
       |-loop:
       | -get_next_event()
       | -call different handlers
       | -NA_handler_dynamic()
          |-insert_event_na()
              |-try do find best insert
               |-if best insert is feasible, end event handling
               |-else, best insert is not feasible:
               | -operator_chain() // operators are re-writen to the DVRP version
                  |-perform_rearrange_dvrp()
                  |-perform_swap_dvrp()
                  |-perform_opt_dvrp()
       | -NC_handler_dynamic()
          |-change_supply()
            |-if feasible after current time
            | -end event handling
            |-else
            | -relax_demand_and_reload()
            | -operator_chain()
       | -NV_handler_dynamic()
          |- // the same as NC handler
       | -PV_handler_dynamic()
          |-change_supply()
            |-waste_distribution_after_current()
    | -good_solutions.solutions[i].follow_next_current()
    | -good_solutions.solutions[i].batch_event_handling()
       |- // the same as good solution
    | -check_good_solutions()
    |-end loop
    |-bestsolution.find_unhandled_events()
  |-end_case()  // then run next case
```

Code for stage i-iv are implemented in functions before perform_waste_distribution()

After that, the function write_generator_input() will write the route constructed by static VRP into file, and then read by the event case generator(the .py file) to generate events

Then the system() function will call the event generator written in python, please put the event generator in the visual project path, i.e. the same path with input data:



Here is the control flow of the project:

1. C++ project finish static VRP
2. C++ write best solution into files, as input file for event generator
3. C++ project call event generator written in python using the system() function
4. Event generator generate event based on best solution from static VRP generated by C++
5. Control switches back to C++, C++ read event file
6. C++ perform dynamic VRP in `route_events_main()`

Note that step 2-4 can be skipped if we already have the event file, i.e. we only need to call the event generator once.

So if you already have the event file, and you want to skip step 2-4 to reproduce the same output, please modify the variable `EVENT_DATA_EXIST` in `Routing::init()` function. If you keep it as false, then the second time you run the code, the event file will be generated again, which will make you lose your previous output.

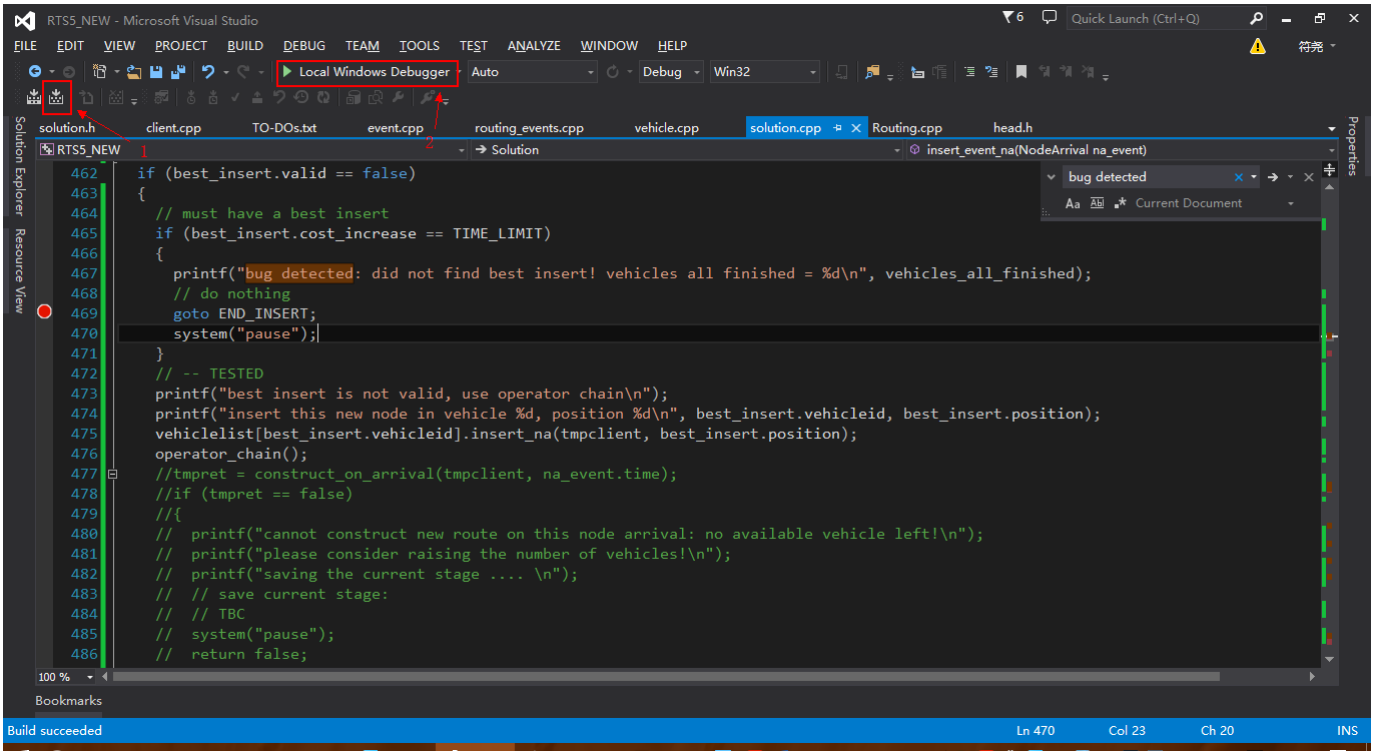
You may also want to modify some parameters in the C++ code while keep the event data the same to do some comparison. In this case you also need to keep `EVENT_DATA_EXIST = true`.

```
void Routing::init()
{
  // what is the reaction when event happens?
  REACTION_TYPE = DYNAMIC_HANDLER;

  // if first time to test this case, set this variable to be false
  // else, set it to be true to reproduce the same output
  EVENT_DATA_EXIST = false;

  // tabu parameter
  lumbda = 0.02;
  alpha = 1;
  beta = 1;
  gamma = 1;
  mdfactor = 2;

  // new id to be assigned to arrival nodes
  new_id = 500;

  // current time when doing routing
  current_time = EPSILON;
  EPSILON = 1e-4;
  TIME_LIMIT = 1000000;
  event_time_ptr = 0;
```

You can do basic configurations of the codes in head.h and in Routing::init() (see below) in Routing.cpp (see above).

```
// ---- General definition ----
#define VEHICLE_LIMIT 10
#define CLIENT_LIMIT 250
#define SOLUTION_LIMIT 10
#define SHIFT_LIMIT 1000000
#define EVENT_LIMIT 200

// route construction
#define RELAX_RATIO 0.8

// tabu search
#define TABUTENURE 8
#define STOPCRITERIA 2000

// this type define will consist with previous files
#define PICK 1
#define DELV 0

// event handling
#define TIME_RELAXATION 1.25
#define GOOD_SOLUTION_LIMIT 5
```

You do not need to do special configuration in your visual studio because I have already handled them. But if you want to modify the code and run it again, please click the [build] button first, then the [local windows debugger] button. Not doing this may lead your modified code not effective.



| Dynamic Handlers

All pickup and supply is modified to be the saturated version, i.e. load/deliver whatever we can.

Node arrival:
Note that I changed this handler different with your previous document because I found problem with it.
Algorithm:

- Try to find the best insertion, i.e. dvrp_feasible with lowest cost increase.
  - Dvrp_feasible = feasible after current time.
  - Not that after insertion, the pickup and delivery is modified to be the saturated version.
- If best insertion exists
  - Insert at best insertion.
  - End event handling.
- Else if no dvrp_feasible insertion, accept infeasible insertion with the lowest cost
  - Perform operator chain.
  - The operator chain may save the infeasibility of a route.
  - Note that this is different with your documentation. We will not construct new route anymore -- why we should make this modification? Later in this documentation we will discuss this.
  - End event handling.
- Else, no place to insert (because all vehicles have finished routing).
  - Mark this event as not_successfully_handled.
  - End event handling

Negative cancellation:

- Change the supply of the node, update the route with saturated load& deliver.
- If dvrp_feasible
  - End event handling
- Else, not dvrp_feasible
  - Relax demand and reload
  - Use operator chain
  - End event handling

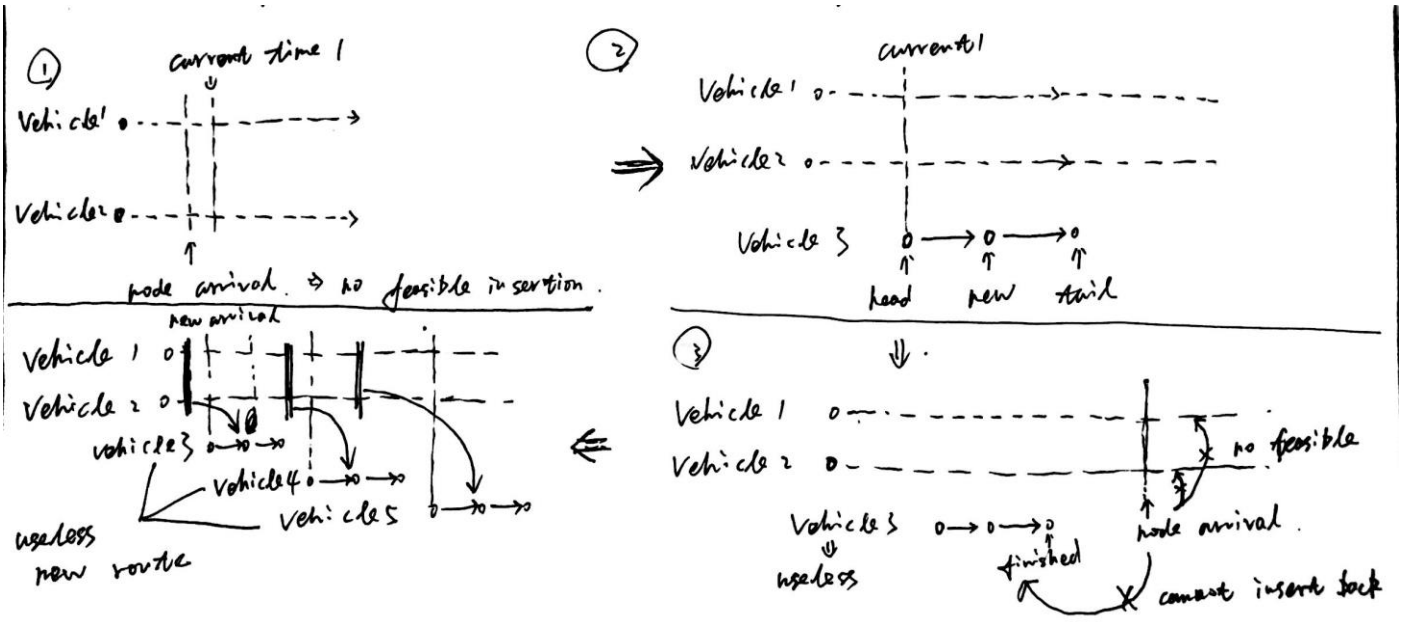
Node cancellation: the same as negative variation.

Positive variation

- Change the supply and update the route with saturated load& deliver.
- Call waste_distribute_after_current()
  - The same as previous waste distribution function but only effect node after current time.

| Issue to discuss

Why we do not construct new node at new arrival? Consider the following situation:



In figure 1, the upper left figure, there is a node arrival which cannot be inserted into vehicle 1 or vehicle 2 feasibly. In figure 2, we construct a new route (vehicle 3) start at current time.

In figure 3, the lower right figure, there is a node arrival which cannot be inserted into vehicle 1 or vehicle 2 feasibly. However, it also cannot be inserted into vehicle 3 because by that time vehicle 3 has already finished routing, so we have to construct a vehicle 4. In this case, vehicle 3 is useless to accept new nodes.

The situation discussed in figure 1-3 can happen repeatedly, leading to more and more useless routes.

How to tackled this? Remember that now we allow infeasible solutions. So we can insert the new arrival node into vehicle 1 or 2 infeasibly, and use operator chain to make them feasible again.

The first case in the file case0 demonstrates the above solution. If you print out the intermediate state you will find that the solution comes into infeasible state after new arrival, but come back to feasible state after further operators.

| Static Handlers

Static handlers are modified from static VRP, but modified to the “after current time” version, which includes multiple changes of previous functions.

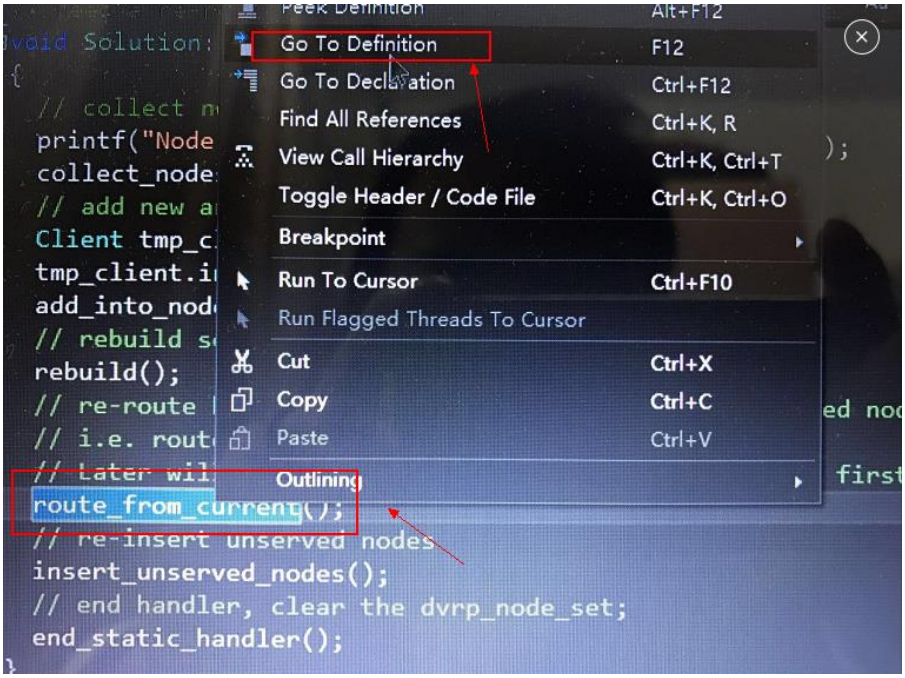
Below is the call hierarchy of static handlers. When an event happens, we first collect the nodes that have not been visited after current time, if new arrival, put it into the collected node set, or if variation, make the variation happen, then reconstruct nodes after current time using rebuild() function. The route_from_current() function performs tabu search, opt operator and waste optimization.

Note that the rebuild() function does not use clustering, rather when we find unserved nodes, we will temporally leave it and perfrom tabu search and so on. After waste optimization, we will re-insert them infeasibly like what we do in dynamic handler.

Basically all the auxiliary functions in this stage are modified to be the “after current version.” Operators do not influence node before current time, and update of vehicles and solution will also not touch the route structure and the parameters before current time. You can find them in the code.

One small useful tip: write click a function and click “go to definition” will help you to track function call chain, thus helping you find out functions I have modified in the more underlying level:

If you need specifications of what functions I have touched and need a more comprehensive documentation, please ask me.




```
main()
|-routing.run()
  |-init()
  |- … /multiple previous functions as before
  |-route_events_main()
    |-loop:
     | -bestsolution.push_forward_time()
        |-get_next_upcoming_event()
        |-vehicles move to next nodes after next upcoming event
        |-collect events on the way
     | -bestsolution.batch_event_handling()
        |-loop:
        | -get_next_event()
        | -call different handlers
        | -NA_handler_static()
           |-collect_nodes() // collect node that haven't been visited from current time into a node set
           |-add_into_node_set(tmp_client) // add the new arrival node into node set
           |-rebuild(); // rebuild route from current time using nodes from node set collected above
           |-route_from_current();
             |-tabu_search_after_current() // the most complicated function in this stage
             |-opt_after_current()
             |-waste_distribution_after_current()
           |-insert_unserved_nodes()
        | -NC_handler_static()
           |-change_supply()
           |-collect_nodes()
           |-rebuild()
           |-route_from_current()
        | -NV_handler_static()
           |- // the same as NC handler
        | -PV_handler_static()
           |- // the same as NC handler
     | -good_solutions.solutions[i].follow_next_current()
     | -good_solutions.solutions[i].batch_event_handling()
        |- // the same as good solution
     | -check_good_solutions()
     |-end loop
     |-bestsolution.find_unhandled_events()
  |-end_case()  // then run next case
```

| About the case generator

Case generator is the same I wrote last September … The parameter in this generator is tuned to generate 100-node case with 6 vehicles. You can tune the parameters in this generator to make the case smaller or larger:

The nodes are chosen from a 100*100 grid with supply and demand following certain distribution and ratio, you can also modify it in the case_generator.py:

```
class Generator(object):
    # ---- Constructor ----
    def __init__(self):
        # how many node do you want in total?
        self.totalnode = 100
        # ratio of numbers of pick-ups and deliveries
        self.pdratio = 1.25
        # ratio of pick-up demand and delivery demand
        self.upper_pd_dmd_ratio = 1.0
        self.lower_pd_dmd_ratio = 0.5
        # fixed demand ratio, if we cannot get a hit in large number
        self.pd_dmd_ratio = 0.8
        # upper limit and lower limit of the demand of a single node
        self.upperdmd = 400
        self.lowerdmd = 0
        # how many grid do you want in total?
        self.totalgrid = 9
        # how many dense grids do you want?
        self.ndensegrid = 2
        # how many node do you want at least in these dense grid?
        # lstnddsgrid = least node in dense grid
        self.lstnddsgrid = 20
        # how many node do you want in single grid at least?
        # lstndingrid = least node in grid
        self.lstndingrid = 5
        # note: the code will ensure the following
        #       number of node in dense grid > least node in dense grid
        #       least node in grid < number of node in ordinary grid < leas
```

Also do not forget to modify the limit of vehicles and nodes in head.h

| About the event generator

The event generator read the best solution file generated by C++ code, and decide cancellations and variations based on the nodes of best solution of static VRP.

The event generator should also generate node arrival events based on the time span of the best solution. Please tune the parameters to make the time of node arrival coherent with time span of best solution. If a node arrival is too late that all vehicles are finished their routing, this node arrival will not be handled.

The tuning of parameters is based on multiple factors including number of nodes, number of vehicles, time span, demand and requestion of nodes. Please tune them carefully to ensure coherency.

Now it is tuned for the 100 nodes case, coherent with the parameters of the case generator.

Note that the event generation part based on real world data has not been finished yet. And the static handlers are not finished yet. Will try to finish the static handler tomorrow.