2023

Solver for vehicle routing problems

VrpSolver

DOCUMENTATION V. 1.4.2

SHOBB@YANDEX.COM

Contents

Description of the task for which this program is intended

This program is designed to solve various types of vehicle routing problems (VRPs) with pickups and deliveries. The program can handle the following features and constraints:

- Each vehicle has its own attributes, such as cost, availability, capacity, route, and skills.
- Each order has its own specifications, such as pickup and delivery locations, time windows, weight, and required skills.
- The program aims to assign orders to vehicles in a way that minimizes the total cost and meets the time requirements.

Some examples of VRPs that this program can solve are:

- Pickup and Delivery Problem with Time Windows (PDPTW)
- Dial-a-Ride Problem (DARP)
- Capacitated Vehicle Routing Problem (CVRP)
- Capacitated Vehicle Routing Problem with Time Windows (CVRPTW)

The algorithm used

- The optimizer uses a highly advanced meta heuristic algorithms to solve the problem – simulated annealing SA (or late acceptance high climbing LAHC) with local neighborhood search, the effectiveness of which is confirmed by several world records and some achievements in solving reference tests. These algorithms have proven to be very effective and efficient in solving complex problems.
- Its speed, with very good quality of the solution is quite high. All operations that are possible are performed in parallel. Thus, you can increase the speed of the solution by increasing the number of processor cores.

The data you need to solve the problem:

- A set of starting and end points for vehicles and points of departure and destination. Some of these points may be the same
- The time/distance matrix between all these points. Each element of the matrix may depend on time. In addition, up to four different matrixes can be set (for each vehicle, you can specify which matrix to use if the characteristics of the vehicles are different)
- The details of each order, such as:
    - Pickup and delivery points for order
    - Weight and volume of cargo
    - Time required to load and unload cargo
    - Multiply time windows to load and unload cargo
    - You can also set unloading time windows relative to load time. This gives the possibility to restrict maximal time to transfer cargo – required at example at DARP
    - The skills required from vehicle to handle the order.
- The characteristics of each vehicle, such as:
    - The starting and ending points of its route. These may be the same or different.
    - Attribute, if vehicle needs to return to end point or route ends at destination of last order
    - The maximum capacity in terms of weight and volume
    - The maximum length of its route and number of orders
    - The delay at each point, which can be used instead of separate loading and unloading times (optional)
    - Own vehicle weight – to calculate total transport work (gross ton-mile - I refer this value further as job)
    - Separate costs for:
        - Unit of distance
        - Unit of job (transport work) – you can plan more greener and logic routes, because fuel consumption and passenger convenience depend on this
        - Unit of time – time is also costs something; you do not use even small value for this – not to plan route which is 1% shorter but requires 100% time more.
        - Own cost for using this vehicle
    - The skills that the vehicle has

- The program requires a zip archive with several json files as input data. You can find the description of these files below.
- You need to load the zip archive into the program and start the optimization process. This may take some time depending on the size and complexity of the problem.
- After the optimization is done, you can save the results as another zip archive with a different set of files. You can find the description of these files below as well.

Program limits

- This version of solver is suitable for tasks with approximately 500 to 800 orders and hundreds of vehicles.
- For such tasks, 8GB of memory should be sufficient.

Input file description

- The program requires a zip archive (with the .zip extension) as input data. You can see an example of this archive in *example.zip*. Autogenerated Json schemas of files in this archive - vrpsolver_schemas.zip.
- The archive should contain these files:
    1. **Points.json**: A list of all the points in the problem, such as where vehicles start and end, and where orders are picked up and delivered.
    2. **Vehicles.json:** A list of all the vehicles in the problem, with their features and limits.
    3. **Orders.json:** A list of all the orders in the problem, with their details and needs.
    4. **Matrix.json**: A time/distance matrix that shows how long and how far it takes to travel between any two points. You can have up to four different matrices for different kinds of vehicles. If you need more than one matrix, you can name them as matrix1.json, matrix2.json, and matrix3.json.
    5. **options.json** (optional): A file that has some options for the solver, like:

        - ***MinInuse***: A flag that tells the solver to use fewer vehicles in the solution, even if it costs more. The default value is 1 (true).
        - ***MinReturns***: A flag that tells the solver to visit points that are pickup locations for many orders less often. The default value is 1 (true).
        - ***NumSolvers***: The number of solvers that run concurrently. The default value is 1, which works well for most problems. Increasing it allows multiple solvers to run in parallel and choose the best solution. The value must be positive.
        - ***NumIterations***: The number of times that the solver tries to find the best solution. The default value is 100000, which works well for most problems. For small tasks, you can reduce this value; for large tasks, you can increase it.
        - ***InitRebuilds***: The number of times that the solver tries to find a starting solution. A good value is between 500 and 1000.
        - ***Timeout***: The most time (in minutes) that the solver can run. If zero, there is no time limit.
        - ***KInitTemp***: A parameter that changes the starting temperature in Simulated Annealing (SA) mode. If positive, it is used as a factor for auto-finding the starting temperature. If negative, it is used as a fixed value. The default value is 0.5.

- ***LahcLength***: A parameter of Late Acceptance Hill Climbing (LAHC) algorithm. If positive, the solver uses LAHC instead of SA. The default value is zero. If you want to use LAHC, you can try a value around 100. SA and LAHC modes have their own advantages and disadvantages.

6. sdata.txt (optional): A file that has a starting solution from a previous run.
7. Optionally: *sdata.txt* – The initial solution from previous run.

Detailed descriptions of input structures in C# notation:

Point data structure:

```
struct SPtInfo {
        int Id              // Unique point ID
        string Name         // Name
        string Info         // optional: Additional information
        string Address      // optional: Address
        string LatLng       // optional: GPS coordinates
}
```

Distance matrix element data structure:

```
struct SVDistInfo {
   int Point_from_id     // Identifier of the point of the beginning of the section
   int Point_to_id       // Identifier of the end point of the section
   STimeInfo [] ATime    // Array of time/distance description structures (at least 1 element)
       // Time / distance array element:
       struct STimeInfo {
         double StartTime        // Start time of this element
         double TravelTime       // Travel time
         double Distance         // Travel distance
         }
         // ATime elements must be sorted by StartTime and StartTime of all elements must be different
   }
```

Vehicle data structure:

```
struct SVehData
  {
     int Id                    // Unique identifier of vehicle.
     string Name               // Name
     int Start_Point_Id        // Identifier of the route start point
     int End_Point_Id          // Identifier of the route end point (can be the same as Start_Point_Id)
     double Capacity           // load capacity
     double Volume             // optional: maximum cargo volume
     int Matrix_Index          // Index of the time/distance matrix (0 - 3)
     double Max_run            // maximum allowed path length
     int Max_Order_Count       // maximum allowed number of orders to be executed
     int Attr                  // 0 - if the vehicle ends the path at the destination of the last order,
                               otherwise 1
    int LoadConstr             // Restrictions on the order of loading / unloading: 0 - no, 1 - LIFO, 2 -
                               FIFO
     double TimeStart          // Start time
     double TimeEnd            // End time of work
     double delay_at_pt        // Delay time when for loading / unloading
     double own_weight         // Own weight of vehicle
     double cost_vehicle       // Fixed cost for using vehicle
     double cost_distance      // Cost of distance unit
     double cost_time          // Cost of a unit of time
     double cost_job           // Cost of a unit of transport work
     double kcost_order        // Coefficient of the cost of an order for this vehicle
     double coefficient_of_delay // optional:  Coefficient of the cost of the delay in the start of the order
     double ktime_speed        // optional:  Speed coefficient - the time of distance matrix is multiplied
                               by this coefficient
     string Skills             // - optional: list of additional properties of this vehicle
}
```

Skills string format:

<NAME OF SKILL1:string>:<COUNT:int>:<TYPE:int>;<NAME OF SKILL2:string>:<COUNT:int>:<TYPE:int>

Where
NAME OF SKILL – some abbreviation for skill
Count – number of such skills (max value - 10000)
Type of skill: 0, 1 or 2.
If type of skill is 1 – then it is released after using by some order, is 0 – vehicle simply “has” such property, type 2 – if skill has limit for whole route – it is not released by order.
At example skills for taxi: front seat, two rear seats and lux car – “FRS:1:1;REAR:2:1;LUX:1:0”
For Order it does not contain type, only name of skill and count.
At example:
“REAR:1;LUX:1” – Order needs 1 rear seat in LUX car.
If Skills for order – empty string – order does not need any special resources.

Transfer order data structure:

```
public struct SOrderData {
    public int Id                       // Unique order identifier
    public string Name                  // Name
    public int Point_from_id            // loading point identifier
    public int Point_to_id              // destination point identifier
    public double Cargo_weight          // Cargo weight
    public double Cargo_volume          // optional: cargo volume
    public double TimeToLoad            // Time to Load
    public double [] TimeStartLoad      // Array of start time of time windows to load
    public double [] TimeEndLoad        // Array of the end time of time windows for loading
    // Arrays TimeStartLoad and TimeEndLoad must have equal length

    public double TimeToUnload          // Array of start times of time windows to unload
    public double [] TimeStartUnload    // Array of start times of time windows for unloading
    public double [] TimeEndUnload      // Array of the end time of time windows for unloading
    // Arrays TimeStartUnload and TimeEndUnload must have equal length

    public bool adjust_UnloadTw         // true - if the time window for unloading is set relative to the
                                        load time, and in fact it represents the time limit for order
                                        fulfillment.
    public double CostOfDelay           // optional: the cost of delaying the start of the order
    public string Skills                // optional: Requires additional vehicle properties to complete
                                        the order.
}
```

Solution file description:

- The solution file is zip archive with the following elements:
    1. **result. json** – information about the solution in json format. *See below*.
    2. **result.txt** – a solution in human-readable text form.
    3. **result**.xlsx - a solution in Microsoft Excel format.
    4. **sdata.txt** – a file that, if necessary, can be used as an initial solution to the task.

Result.json structure description:

(Json schema - schema_result_solution.json)

```
solution
{
      solution_summary summary        // Total values across all routes
      List<route> routes               // list of all routes
}
```

Summary structure:

```
solution_summary
  {
      double duration                  // Total duration of all routes
      double distance                  // Total length of all routes
      double job                       // Total transport work
      double cost                      // Total cost
      double orders_cost               // Total orders cost
      double cargo_weight              // Total cargo weight
      int orders_count                 // Total number of orders
      double wait_time                 // Total waiting time
      int number_of_routes             // Total routes
      List<error> errors               // List of errors, if any
  }
```

Error structure:

```
error
 {
   string description                  // Error description
   double value                        // Value
}
```

Route structure:

```
route
{
   int route_nr                        // Route number
   vehicle vehicle                     // Vehicle description
   summary summary                     // Summary for this route
   List<segment> segments              // List of route segments
}
```

Vehicle structure:

```
vehicle
 {
      int id                          // Unique identifier of vehicle.
      string name                     // Name
      string loading_constraints      // Restrictions on the order of loading / unloading
      double delay_at_point           // Delay time when for loading / unloading
      double ktime_speed              // Speed coefficient
      List<skill> skills              // List of additional properties of this vehicle
      point depot                     // Depot point
      point destination               // Destination point
      int Matrix_Index                // Index of the time/distance matrix
  }
```

Skill structure:

```
skill
{
    string name                       // Skill name
    short as_resources                // Skill type
    int count                         // Skill count
}
```

Point structure:

```
point
{
    int id                            // Unique point ID
    string name                       // Name
    string info                       // Additional information
    string address                    // Address
    string latlng                     // GPS coordinates
    List<operation> operations        // List of operations performed at this point, if any
}
```

Operation structure:

```
operation
 {
   string type                    // Type of operation: Load / Unload
   double start_time              // Start time of this operation
   double end_time                // End time of this operation
   double late                    // Delay, if any
   double wait                    // Wait time before operation, if any

  // Information about related order:
   int order_id                   // Unique order identifier
   string order_name              // Name
   double order_weight            // Cargo weight
   double order_volume            // Cargo volume
   List<skill> order_skills       // Additional vehicle properties, required to complete the order,
                                  if any
}
```

Segment structure:

```
segment
 {
   point from_point               // Segment start point
   point at_point                 // Segment end point
   double distance                // Travel distance
   double duration                // Travel duration
   double start_time              // Start time
   List<onboard> onboard;         // Items on board
}
```

Onboard structure:

```
onboard
{
// Information about order:
   int order_id                   // Unique order identifier
   string order_name              // Name
   double order_weight            // Cargo weight
   double order_volume            // Cargo volume
   List<skill> order_skills       // Additional vehicle properties, required to complete the order,
                                  if any
 }
```