Documentation Algorithm

Brecht Wuyts - r0668122 & Felix Cornelis - r0720203

January 6, 2022

Contents

1 Introduction

Over the past weeks we have been working on the implementation of the Monte Carlo Tree Search algorithm for the various dial a ride problems. At the moment we have a working implementation for the Multi-Depot Heterogeneous Dial a Ride Problem (MDHDARP). However, given the way the problem is set up this implementation should be suited for the other, less complex, problem settings as well.
In the following text we will detail the workings of our algorithm. Specifically, we will first introduce several definitions and conventions we made while modelling the problem. Next, the general logic or workflow of the algorithm will be explained through the main/core functions. Afterwards, all the other functions which are called by the core functions will be detailed. Finally, some important remarks and to do's are stated in the final section.

2 Definitions & Conventions

We have made the following conventions for modelling the problem:

2.1 State

During the execution of the algorithm users are allocated to the different vehicles. Whenever not all users are allocated to a vehicle we speak of a partial solution. We define a state as the current allocation of the users over the vehicles, thus they represent partial or complete solutions.
A state is compromised of 6 arrays. The first three (succ, pre and RI) represent the actual (partial) solution, the last three (earliest, latest and cap) contain additional information to facilitate and speed up feasibility checks in the move-function (how a move is defined will be explained below).
A state is thus a list containing these arrays with the following structure: Sate = [succ, pre, RI, earliest, latest, cap].

1. Succ:
   A successor array of size equal to the number of vertices related to the users in the

problem. In other words, a length equal to two times the number of users in the problem as each user has a pickup and delivery vertex.
At each index the array gives the successor vertex of the vertex with that index. succ[i] = j, if vertex number j is visited right after vertex i.

2. Pre:
A predecessor array of the same size as the succ-array. Here the value of the array at index i does not indicate vertex i's successor but the vertex that was visited right before vertex i.
pre[i] = j, if vertex j is visited right before vertex i.
Together the arrays Succ and Pre form a double linked list.

3. RI:
A route information array, which contains the first and last user vertex of each tour/vehicle in the current state.

4. Earliest:
This is an array of length equal to the total number of vertices in the problem. Hence, two times the number of users plus two (for start and end-depot). At index i the array contains the earliest time vertex i can be visited, taking into account travel times and time windows (not ride times).

5. Latest:
This array has the same length as array 'Earliest'. However, it does not contain the earliest arrival time of vertex i at index i but the latest. Again, only travel times and time windows are considered, not user ride times.

6. Cap:
This is a multidimensional array with a size equal to the number of resources in the problem times the total number of vertices in the problem.
At each position the array contains the usage of a resource type after visiting vertex i. This array will be initialized with zero and updated gradually as a solution is built.

2.2 Node

A node in the search tree differs from a state as it stores additional information. Next to the actual state itself also the number of times the node was visited, the back-propagated simulation results, its child nodes, parent node, the parent_action (action taken from parent node to reach the node) and the untried actions/moves given the current state are stored. It is not necessary to store all this additional information for every generated state, for example during the simulation, hence the distinction.
We will model a Node in a class, such that all generated nodes are objects of this class.

2.3 Move

We define a move as the insertion of a user's pickup and delivery vertex in the current partial solution such that increase in length of the solution is minimized while respecting all constraints i.e. precedence constraints (pickup before delivery vertex), time windows, capacity constraints.
(IMPORTANT: we currently haven't yet implemented the maximum user ride times and maximum route duration constraints.)
Given the current definition of a move all possible insertion combinations across all tours need to be evaluated. Which takes quite a long time and hence results in slow rollouts.
An alternative definition for a move could be the best possible insertion in a random route while respecting all constraints (again user ride times and route duration is not yet implemented). If no possible insertion could be found another random route would be selected.
In the 'MCTS_all_routes.py' the first definition of a move is used to expand the the search tree as well as in the rollouts. While, in 'MCTS_hybrid.py' the first definition of a move, the one considering all routes, is used in the expand function while the second definition, with random route allocations, is used in the rollouts. This method returned only slightly worse solutions but was a lot faster. We also made a version in which the random routes were used both in expand and rollout but the results of this implementation were a lot worse.

In order to discuss the general flow of the algorithm and for the remainder of this documentation we will refer to the 'MCTS_hybrid.py' file as this seems to be the most promising implementation.

3 Prepossessing of Data

In the first lines of code the data of the instance is loaded and cleaned. Resulting in an array with the x coordinates (x_co), an array with the y coordinates (y_co), an array with the service time (s), an array with the maximum user ride time (L), a matrix with the demand for each resource type (q), an array with the earliest arrival times (e) and an array with the latest arrival time (l) for each vertex.
In this section, also the number of rollouts that will be performed in each execution is set.

Next the coordinates of the depot for each of the vehicles are set. This is done as described in the literature. In case of a single depot all possible locations should simply all be set to the same values, probably coordinates (0,0) but other choices work too.

4 Initialization Algorithm

In order to see the main flow of the algorithm you have to scroll down towards the code below the line 'if __name__ == "__main__":'.
Firstly the root state is initialized via the **initialize_state() function**. The succ, pre and RI arrays are initialized with 'None', earliest and latest respectively with e and l arrays and cap with 0.
Afterwards, based on this state the root node is generated, this is done by creating an object of the class 'MonteCarloTreesearchNode' which contains this initial state and the other attributes described in section 2 as well as several class methods. Next to these class methods some other (helper) functions can be found in the script. All of these, class methods, regular functions and their relations will be explained as we progress.

Note on the distinction between class methods and regular functions:
When building the search tree nodes are created via the 'MonteCarloTreesearchNode' class, when performing simulations/rollouts we only need states to evaluate the final outcome. Hence no nodes are created as the additional information stored in them is not required and it would slow down the rollout. Thus, in order to be able to use certain functions during these rollouts on non class objects we excluded them from the class.

5 Main Logic Algorithm

For each user an execution of the Monte Carlo Tree Search algorithm is done. At the end of this loop the current node contains the final state where each user is allocated to one of the vehicles and the cost/length of this solution can be calculated.

The entire execution is packaged in the **'best_action()'-function**. In this function the following functions are executed. First, the **'_tree_policy()'-function** is called on the current node, then a simulation is performed with the **'rollout()'-function** and finally the result of this rollout is backpropagated to all the nodes which were on the path used to reach the node from which the simulation was started with the **'backpropagate()'-fuction**.
This entire procedure, inside best_action, is called an iteration and is done as many times within one execution as is set with the 'sim_num' parameter.
In short, one execution contains several iterations in which tree_policy, rollout and backpropagate are called and in doing so a search tree is built.
Ultimately the best child node of the current node is returned, according to the 'best_child()'-function. This child becomes the new current node from where a new execution is started.
We will now have a look at the three functions inside 'best_action()' and all other functions

which they call in turn.

6 Core Functions

6.1 _tree_policy():

This function takes a node as input, the root node or the node returned in the previous execution, and yields the next node from where to start the rollout/simulation.
It does so by traversing the current tree according to the **'best_child()'-function** until it reaches a terminal node or a node which is not fully expanded.
Note the value for the parameter c_param of the best_child function in the tree_policy() function is different than the value used in the main loop described in section 5, see the explanation of the best_child function why.
In case of a terminal node the function returns this one, in case of a not fully expanded node it calls the **'expand()'-function** which generates one of its possible child nodes and returns this child.
Not fully expanded means not all possible moves for this node have been explored, every unserved customer in the current solution is a possible move. Thus, for every unserved customer a child node can be created in which this customer is served, a node has thus as many child nodes as there are unserved users in its state.
A node is terminal if all customers are served in the state of the node, hence no more child nodes can be created.

6.2 rollout():

This function takes the node returned by _tree_policy() as input and performs a rollout starting from it.
It takes the state stored in this node and returns all possible moves which can be done from it with the **get_legal_action()-function**, one move is selected according to the **rollout_policy() function** and finally the next state given the selected move is generated with the **move() function**. This procedure is repeated for the newly generated state until a final state is reached in which all users are served. For this terminal state the **game_result() function** is called, which calculates the reward which will be back-propagated based on the length of the found solution.
Note that in the rollout no nodes are generated, we only move from one state to a next until a final state is reached in which all users are served. This is done because we will discard these states after the execution of the rollout. It is thus not necessary to create a node for every state storing a lot of additional information and hence slowing down the rollout.

It might be possible that given a current state an action cannot be performed, i.e. the user selected to be inserted has no feasible insertion in any of the current routes. If so the infeasible action will be deleted from the possible actions and the procedure will be restarted, if no other actions exist the rollout is terminated and a score of 0 is back-propagated.

6.3 backpropagate():

This function accredits every node on the path from where the rollout started to the root with the result obtained in the rollout.

7 Other Functions

In this section the functionality of all other functions and methods will be explained. We suggest going through the core functions described above and looking in this section for the explanation of other functions they call in turn.

7.1 dist(k,i,j)

This function calculates the distance between vertex i and vertex j. It does so by gathering the x and y coordinates of both vertices and calculates the euclidean distance between them.
If either of these vertices correspond to a depot the function accounts for this by taking the appropriate depot coordinates for the route in which the vertices are inserted, which is route k.

7.2 compute_costincrease(k,a,tour,i,j)

This function calculates the total increase in the length of a a tour when inserting the pickup vertex of user a at index i and the delivery vertex at index j in the tour.
The parameter k indicates which vehicle corresponds to the tour such that the appropriate depot coordinates can be used if necessary.
The tour parameter is a list containing the vertices in order they are visited in the tour.
Two cases are identified. The first one in which i is equal to j and the delivery vertex is inserted straight after the pickup vertex in the tour, the second where this is not the case. This is necessary as a different number of arcs needs to be deleted and created in each scenario. When calculating the cost increases the function calls the dist() function to get the length of the new and the deleted arcs.

7.3 length_tour(tour, k)

This function calculates the total length of a tour.
The tour parameter again is a list with the order in which the vertices in the tour are visited.
The parameter k indicates which vehicle serves the tour in order to get the appropriate depot coordinates.
The function calculates the total distance by calling the dist() function for all of the consecutive vertices in the tour list.

7.4 length_total(state)

Given the inputted state this function calculates the total length of all tours in the state. It does so by creating all the tours in the state in list format and calling the length_tour() function for each of them.

7.5 get_legal_actions(state)

This function returns a list of all possible moves/action given the inputted state. This corresponds to all users who have not been served in the current state.
It does so by checking the first n values (n = total number of users) of the succ array and adding all indices with value 'None' to the unserved_requests list as a vertex with no successor can not be visited. In other words, checking which of the user pickup vertices have no successor.
Note in python indices start at 0 the users start from user 1, user 2, and so on. Thus if index 0 has a 'None' value this corresponds to user 1 not being served.

7.6 is_game_over(state)

This returns a boolean indicating whether the current state is final or not i.e. whether all users have been served.
It does so by checking if the succ array of the state has no values left which are equal to 'None'. (Also, here only pickup vertices are checked, as with every move both the pickup and delivery vertex of a user are inserted.)

7.7 dipslay_routes(state)

This function is used nowhere in the script but at the end to display the routes in a list format a described above rather than with the succ, pre and RI arrays. (Which is just a bit more intuitive to look at the results)

7.8 game_result()

This function accredits a result to the outcome of a rollout. It does so by comparing the found solution to the global best. The global best is the best solution found over all of the rollouts in all the executions and iterations.
If the found solution is shorter a score of one is back-propagated, if it is longer than the best but still shorter than two times the best score a partial score is accredited through a linear inner function, else a score of zero is accredited.

7.9 move_allroutes(state, a)

This function performs action/move a on the inputted state and returns the new state as a consequence of this state.
As a reminder a move is defined as the best possible insertion of vertex a and (a+n), thus both the pickup and delivery vertex of user a, over all of the routes of the current state. When doing so precedence of pickup to delivery, time windows, and capacity constraints are respected. We will later add the final two constraints, maximum user ride time and route duration, in this code to ensure that solutions are feasible with respect to all constraints of the DARP-problems.
We will briefly go through the code and explain what each code snippet does.
The following procedure is repeated for every route/vehicle (k-loop), whenever a feasible solution is found it is compared to the current best and replaces it if better.
Firstly, it is checked whether an empty vehicle has enough capacity of each resource type to serve the user at all. If this is not the case we can immediately move on to the next vehicle.
Next, the tour is generated in list format, as described previously, to facilitate computations.
In a following step, all possible insertion indices for the pickup vertex are considered. This is done by a loop for i in range(1, length(tour)), in python this means looping from the index 1 until but excluding the last index of the tour, this way the depot is always the first element in the tour (index 0) and the last index in the tour. By inserting a vertex at a certain index everything before this index remains the same, the values from this index until the end of the list are shifted one spot to the right and the new vertex is inserted in the open spot.
For every possible insertion index it is checked whether the inserted vertex can be reached in time (before l[a]), if this is not the case the next possible index is considered else the code moves down.
In the next step, it is checked whether a vehicle has sufficient available capacity at the insertion index for the pickup vertex to service the user. Again, if this is not the case we move on to the next index else we continue.

Firstly, we consider the insertion of the delivery vertex at the same index as the pickup vertex in the tour, thus immediately after it. For this insertion index for the delivery vertex we check if it can be reached in time. If so we move on, else we can stop the entire loop (i-loop going through insertions for pickup), as postponing the insertion of vertex (a+n) will always break the time window constraint. (We don't have to check capacity here as we only drop off the previously picked up capacity)
If at this point the insertion of vertex a and (a+n) at index i is feasible we compute the cost increase and compare it to the current best (lowest) cost increase, if it is lower we save the current action [k, tour, i, j], meaning inserting user vertices a and (a+n) in the tour of vehicle k respectively at index i and j (in this case i is equal to j).
The tour value is not strictly necessary as we have have the state variable and the vehicle k, but we generated it in the code above already so we keep using it.
Remember we are still considering the same index i for the pickup vertex and have already considered to use this same index for the delivery vertex. We will now consider all other indices from (i+1) until the end of the tour (excluding the last one as this is always the depot) for the insertion of the delivery vertex, this is done with the j-loop in the code.
In essence, the same feasibility checks regarding the time window and the available capacity in the vehicle are done each time. Whenever a possible combination if insertion indices is found the cost increase is calculated and compared to the current best one, if it is lower combination of insertion indices is stored.
At the end, the state values are updated and the new state is returned.

7.10 move(state,a)

This function does basically the same thing as the move_allroutes(), but it chooses a route randomly and searches for the best insertion combination of the vertices in it and returns the new state based on this.
If no possible insertion combination can be found the function randomly takes another route

7.11 untried_actions(self)

Returns a list of the untried actions/moves given the state (self). It does so by calling the function get_legal_actions().

7.12 is_terminal_node(self)

The function checks whether the current node is terminal/final by calling the is_game_over() function described above.

7.13 is_fully_expanded(self)

Checks whether all possible child nodes have been created for the current node (self) by checking whether the length of _untried_actions is equal to zero, returns a boolean.

7.14 expand(self)

Generates one of the child nodes of the current node by taking one of the untried actions/moves and generating the new state from it with the move function and ultimately creating a node object.

7.15 best_child()

Function encoding the upper confidence bound for trees as described in the literature of the Monte Carlo Tree Search. It balances exploration and exploitation of nodes with c_param, i.e. exploring not yet visited nodes vs exploring nodes with high potential.
The function is used to return the next node from where to start an executing, in this case the exploration component is not important and c_param is set to equal here. The function is also used for traversing the search tree in the tree_policy() function here balancing exploration and exploitation is important and the c_param is set to sqrt(2).

7.16 rollout_policy(self, possible_moves)

This function determines the policy by which a move is selected out of all the possible moves in the rollout. We currently implemented it as a random, uninformed selection.

8 To Do's & Remarks

1. implement maximum user ride time & maximum route duration constraints
2. Truncated rollouts
   i.e. stopping the rollout after moving a set number of states forward in order to be able to execute many more rollouts as they are currently very slow.