Asian Transport Studies 6 (2020) 100028

Contents lists available at ScienceDirect

Asian Transport Studies

journal homepage: www.journals.elsevier.com/asian-transport-studies

A Monte Carlo tree search for traveling salesman problem with drone

Minh Anh Nguyen [a,*], Kazushi Sano [a], Vu Tu Tran [b]

[a] Nagaoka University of Technology, Japan
[b] Ho Chi Minh University of Technology and Education, Viet Nam

ARTICLE INFO

Keywords:
Last-mile-delivery
Drone delivery
Traveling salesman problem
Synchronization
Heuristics

ABSTRACT

The use of drones and trucks working collaboratively has gained drastically attentions in recent years. We develop a new Monte Carlo Tree Search algorithm (MCTS) to solve the Traveling Salesman Problem with Drone (TSP-D) arising in the management of parcel last-mile-delivery systems. The approach seeks to find optimal decisions by taking random samples in decision space and building a search tree based on the results. We address several major issues in adapting the tree search concept to solve the TSP-D in this paper. The solution approach, aimed to minimize the total completion time, is tested on various problem instances derived from the well-known TSPLIB benchmark. Experimental results show a very promising performance of the proposed search method where it outperforms the compared heuristic method, providing new best solutions for 23 instances and an average 12% improvement in terms of solution quality.

1. Introduction

The world has witnessed a massive thrive of e-commerce in recent years. This highly competitive market has pushed the delivery standard to its finest. Indeed, next-day, same-day, even within 2-h delivery, to name a few, are no longer tomorrow dream in many cities. As e-commerce continues growing, the logistics of moving products along the last leg from warehouses to end-customers have become more important and more complex. To cope with this growth in term of expected volume and delivery standard, the capacity of last-mile delivery systems will have to expand significantly and seek ways to improve the efficiency.

The traditional modes (e.g., trucks, vans) have long been playing the main role in transport. Trucks are reliable in terms of capacity but are bulky and slow. Especially in urban context where traffic congestion and vehicle access restrictions usually occur, trucks are of less mobility and suffer high cost per-mile. These downsides demand more sustainable solutions.

One of the most talked-about developments in recent years is the potential use of unmanned aerial vehicles, commonly referred to as drones, for transportation, despite already being used in many other fields (e.g., military, agriculture, emergency response). Drone developments have undergone big leaps thanks to recent advancements in machine learning, remote sensing, image processing and so on. Fast and lightweight, one could consider drone as a highly versatile vehicle. What's more, drones fly in the sky, hence they can travel at high speeds over congested traffic without any delays.

Seeing numerous benefits drones can offer, Amazon has started using drones to deliver parcels from warehouses directly to customers in certain cities via Amazon's Prime Air. In the operation, a container containing packages is carried by a drone which departs from then returns to the warehouse after finishing its delivery at a customer location. DHL has joined the race with its prototype called Parcelcopter; Zookal has its textbooks delivered by drones; while Google has also launched its Project Wing on-fly. The use of drones for last-mile delivery promises to change the landscape of the logistics industry.

However, there are several reasons in terms of regulations and technological barriers that prevent drones to become mainstream vehicles for delivery tasks: low payload capacity and short battery endurance, both require frequent returns to central depots. Considering these facts, Stolaroff et al. (2018) investigate the energy efficiency of existing drone-based package delivery systems. They point out that there are plausible scenarios in which adopting drones even leads to higher overall operational cost as well as energy consumption. With that being said, trucks seem not to go anywhere far in years to come.

While waiting those constraints to be resolved, an idea to improve the mobility of the overall delivery system is to having drone-truck coordinated in an attempt to take advantage of complementary features (see Table 1). Recently, several prominent logistic companies in Europe and the US are seriously considering the idea of having drones launched from trucks and both working in parallel to deliver packages. In 2016,

* Corresponding author.
E-mail addresses: anh.nguyenor@gmail.com (M.A. Nguyen), sano@nagaokaut.ac.jp (K. Sano), tutv@hcmute.edu.vn (V.T. Tran).

https://doi.org/10.1016/j.eastsj.2020.100028
Received 30 September 2020; Accepted 8 October 2020
Available online 25 November 2020

Table 1
Complementary features of the drone and the truck (Source: Agatz et al., 2018).

	speed	mobility	capacity	range
Drone	high	high	small	short
Truck	low	low	large	long

Mercedes-Benz announced its new vehicle concept called the Vision Van which embraces this innovative idea (see Fig. 1). UPS was not out of the race as it tested the hybrid model of transport in early 2017. Both models have drones docked on the roof of the delivery trucks. The setup can carry up to two drones docked on a single truck, all together serve the given customers. While the delivery truck moves between different customer locations to make deliveries, the drones simultaneously serve another set of customer locations and return to the truck after each delivery to prepare for the next one.

While the vision of using drones alongside with trucks, providing massive relief to land transportation has become more obvious. However, contributions to this new phenomenon in Operations Research perspective are still scarce. The problem is to find both assignment decisions and routing decisions. Assignment decisions determine which vehicle, the drone or the truck, will serve which customers, while routing decisions determine the sequence in which the assigned customers are visited (Agatz et al., 2018). The problem can be considered as an extension of the Traveling Salesman Problem (TSP), the TSP with drone. There are still needs for problem-solving methods, which we aim to close the gap in this paper. Inspired by the success of AI in Go (Silver et al., 2017), the main contribution of this study is to propose a tree search approach based on Monte Carlo Tree Search concept to efficiently solve the TSP with drone variant.

2. Related works

Murray and Chu (2015) is one of the first attempts to address operational aspects of the combined drone-and-truck prototype for last-mile delivery sector. They define two problem settings depending on potential problem scenarios, one of which is the Flying Sidekick Traveling Salesman Problem (FSTSP). The FSTSP seeks optimal coordination of a truck with a drone for scenarios where direct flights from the distribution center to customers are impractical. They propose a Mix Integer Linear Programming (MILP) formulation for the problem. However, due to its complexity, the model can barely find optimal solutions for instances larger than 6 nodes. In an attempt to solve practical large-scale instances, they provide heuristic approaches based on the Route-first Cluster--second procedure. The framework starts with a TSP solver to obtained the truck's route. As the procedure progresses, the truck's route is repeatedly partitioned into numerous subroutes by using relocation procedure to reduce the total cost.

We are aware of another work from Agatz et al. (2018), which also investigates the same proctotype, considering a problem called Traveling Salesman Problem with Drone (TSP-D). The TSP-D shares most of the common characteristics of the FSTSP. The only major difference is that the TSP-D allows the drone to be retrieved at the same location from which it was launched by the truck. Those circumstances should have been infeasible to launch the drone in the FSTSP, making the truck the only option to serve remote customers. Fig. 2 illustrates an example of the mentioned situation. Serving node 2 by the drone, which is launched from node 1 and retrieved at node 3, is impossible due to battery capacity shortage ($distance^{12}+distance^{23}>$maximum flight range) in case of the FSTSP. Launching drone to node 2, however, is feasible if the truck waits at node 1 in case of TSP-D (given that $distance^{12}+distance^{21}\leq$maximum flight range). This new feature stimulates the use of drone, which is expected to reduce delivery effort.

Agatz et al. (2018) develop a new MILP formulation which allows

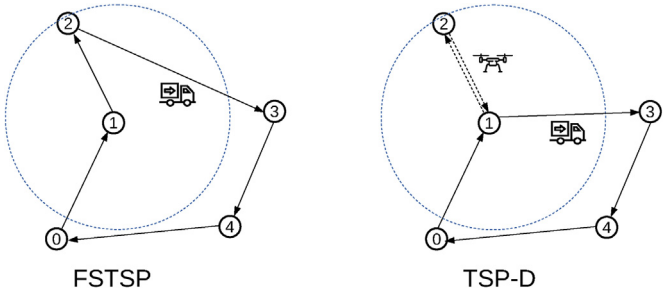

Fig. 2. The difference between the FSTSP and the TSP-D.

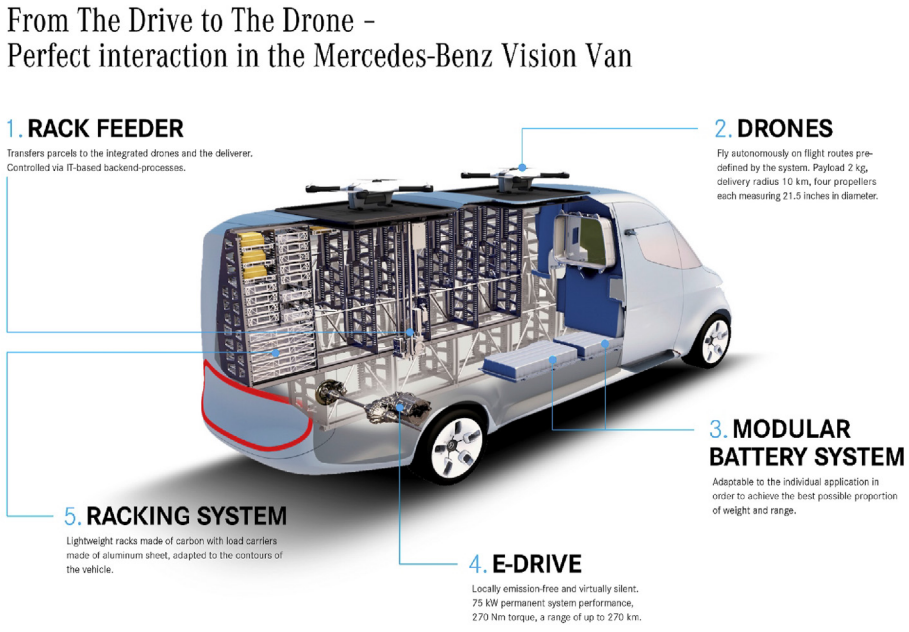

Fig. 1. A "Vans and Drones" prototype from Mercedez-Benz.
(Source: Aerotime Team. "Vans and Drones to be the future of delivery?", Aerotime News Hub, Sep. 10, 2016, https://www.aerotime.aero/aerotime.team/13956-vans-and-drones-to-be-the-future-of-delivery)

Table 2
Related works and characteristics.

	Problem	Objective function	Solution methods
Murray and Chu (2015)	FSTSP	minimize makespan	MILP formulation & heuristic approach
Ponza (2016)	FSTSP	minimize makespan	Simulated annealing
Freitas and Penna, 2020	FSTSP	minimize makespan	Hybrid General Variable Neighborhood Search
Agatz et al. (2018)	TSP-D	minimize makespan	Kruskal's MST heuristic and DP, MILP formulation
Bouman et al. (2018)	TSP-D	minimize makespan	Dynamic programming and A*
Ha et al. (2018)	TSP-D	minimize cost	GRASP, TSP-LS, MILP formulation
This paper	TSP-D	minimize makespan	Monte Carlo Tree Search

solving larger instances with up to 12 customers to optimality. They address large instances by proposing two fast heuristics using route-first cluster-second principle. Two algorithms are introduced for the cluster-phase: a greedy partitioning heuristic, and an exact partitioning algorithm based on dynamic programming. An iterative improvement procedure adopting several local search operators is employed to reach the best possible solution. Bouman et al. (2018) extend this work by proposing a 3-pass dynamic programming approach. They also provide several techniques to shorten computation times which enable the exact method to solve larger instances than the previous work.

Ponza (2016) puts his focus on developing efficient algorithms for the solution improvement stage. Using the same idea of heuristics described in Murray and Chu (2015), he provides a Simulated Annealing along with several local search operators.

Freitas and Penna, 2020 provide a Hybrid General Variable Neighborhood Search for the FSTSP. The algorithm is a simple hybrid heuristic integrating MILP and Variable Neighborhood Search.

In contrast with those aforementioned so far that aim to minimize the makespan of services, Ha et al. (2018) consider a new min-cost variant of TSP-D. The variant objective is to minimize operational costs including total transportation cost and another incurred by wasted time. A MILP model and two heuristics: a Greedy Randomized Adaptive Search Procedure (GRASP) based on split algorithms with local search, and a TSP-LS heuristic - a route-first cluster-second procedure in conjunction with local search - adapted from Murray and Chu (2015) for min-cost variant are introduced. Numerous experiments show that GRASP outperforms TSP-LS in terms of solution quality.

A brief overview of related works mentioned in Section 2 is provided in Table 2.

3. Problem description

A problem instance of the TSP-D can be defined on a directed graph $G = (V, A)$ comprised of a node set $V = \{0, \ldots, n\}$ and arc set A, where n is the number of customers who need to be visited exactly once. We denote node 0 as the depot where vehicles start their tour at the beginning and return to after finishing all given tasks. The problem might capture vehicle-access restrictions which are commonly seen in urban areas. Therefore, n customers might be further distinguished into two subsets: (1) those can be served by the truck only, and (2) those are eligible for both drone and truck services. Heavy or bulky parcels may exceed drone capacity, parcels may require signatures, those are possible reasons behind this differentiation.

It is observed that the drone is able to fly between points without having to follow road network. In other words, the drone can nearly travel on Euclidean distance, while the truck cannot. Hence, given the two nodes $i, j \in V$, the two vehicles are likely to traverse from i to j with different distances. Define two cost function $d^T, d^D : V^2 \to \mathbb{R}$ model distances between locations. Given two arbitrary nodes i, j in the network, we denote $d^T(i,j)$ as the distance the truck has to drive from i to j, and $d^D(i,j)$ as the distance the drone has to fly from i to j. In addition, we define v^T, v^D as the average speed of the truck and the drone, respectively. Hence, in the presence of vehicle speed, one can easily calculate amount of time $t^T(i,j), t^D(i,j)$ required for the truck and the drone, respectively, to travel from i to j.

The objective of the TSP-D is to minimize the makespan, defined as the total time needed for the two vehicles to visit all customers and return to depot. In common sense, a TSP-D solution consists of two routes: one representing the truck route, and the other representing the drone route. However, one cannot simply sum up the truck and the drone driving times to compute the makespan since the two vehicles need to be synchronized. By synchronized, it means that the two vehicles have to wait for each other at designate locations in order to continue their tours. Hence, as a way to consider synchronization and to facilitate the computation, a TSP-D solution is decomposed into a sequence of *m operations* $(o_1, o_2, \ldots, o_m)$ (Agatz et al., 2018).

We further distinguish nodes in a given tour into the following types:

- *Drone node*: a node that is visited by the drone separated from the truck
- *Truck node*: a node that is visited by the truck separated from the drone
- *Combined node*: a node that is visited by both truck and drone

An operation always consists of two *combined nodes* including a *start node* and an *end node*. The start node is the node at which the truck launches the drone. The launching procedure must be carried out at a customer location or the depot. The end node is a customer node or the depot at which the drone rejoins the truck. The end node is also called *rendezvous node* in the literature (Agatz et al., 2018; Freitas and Penna, 2020; Ha et al., 2018).

Those sorties served by the drone are indicated by *drone nodes*. From here, there are two cases that could happen. If an operation does not contain any drone node, it indicates the drone stays idly docking on the truck and let the truck serve customers along the operation between the two combined nodes. Otherwise, if an operation contains a drone node, the drone departs from the truck at the start node, then serves the drone node, and flies back to meet up with the truck again at the end node. In the mean time, the truck makes its deliveries to all customers in between the two combined nodes. As mentioned before, there is a special case of the TSP-D where an operation is defined by a drone node, and both the start and end nodes are the same. Such operation describes the scenario illustrated in Fig. 2 where the drone is launched and returns back to the exact same location where the truck has been waiting. In normal cases, any vehicle, either the truck or the drone, which arrives the end point first has to wait for the other to be able to continue its designate tour. The drone after gathering with the truck is immediately available for the next launch. Besides, once being launched, the drone must visit a customer and return to either the truck or the depot within the drone's endurance t^{max}.

An operation k is characterized by a pair of subtours $(\mathscr{R}_k, \mathscr{D}_k)$ where $\mathscr{R}_k, \mathscr{D}_k$ are the truck tour and drone tour, respectively, starting at the k^{th} combined node and ending at the $(k+1)^{\text{th}}$. The cost function therefore is determined by:

$$t(o_k) = \begin{cases} \max\{t^T(\mathscr{R}_k), t^D(\mathscr{D}_k)\} & \text{if } o_k \text{ contains drone node} \\ t^T(\mathscr{R}_k) & \text{if } o_k \text{ does not contains drone node} \end{cases} \quad (1)$$

Then we can calculate the maximum time needed to serve all customers in a full TSP-D tour by summing up the time duration of all operations: $T = \sum_{k \in K} t(o_k)$

To have the problem logical in terms of practical sense, we take the

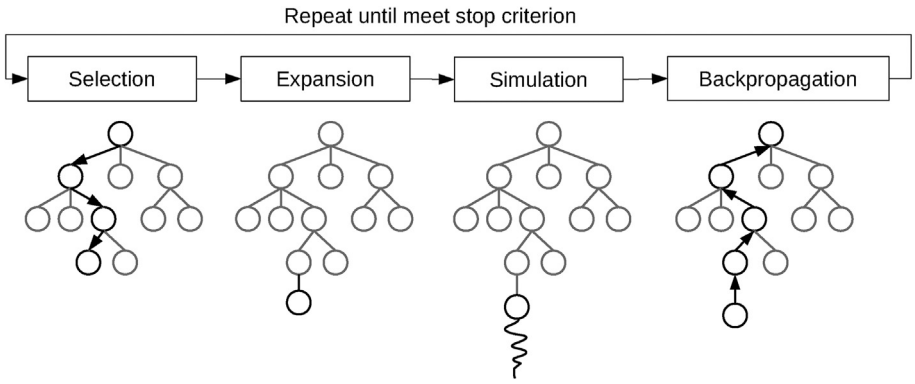

Fig. 3. The four steps of MCTS (Source: Browne et al., 2012).

following assumptions:

1. The truck's capacity is ignore. That means it is capable of handling all customers' packages.
2. The truck has total coverage of considered customers, each of which is visited only once. This assumption guarantee the continuity between operations to satisfy Eq. (1).
3. The drone has unit capacity and has to return to the truck after each delivery.
4. We take the common assumption in the literature (Agatz et al., 2018; Bouman et al., 2018; Freitas and Penna, 2020; Ha et al., 2018; Ponza, 2016; Murray and Chu, 2015): launching and landing operations occur only at customer points. It can be justified by the fact that (1) launching and receiving drones on a moving truck is technologically still very challenging, and (2) parking at intermediate points on the route may not always be possible given that usually one staff/driver operates one truck.
5. Loading and drop-off time can be neglected, assuming that there are enough spared batteries to swap.

Interested readers are referred to Agatz et al. (2018); Murray and Chu (2015) for detailed MILP formulations.

4. Problem-Solving methodology

Planning problems are among the most important and well-studied problems in Artificial Intelligence (AI) field. They are most typically solved by tree search algorithms that simulate ahead into the future, evaluate future states, and back-up those evaluations to the root of the search tree. *Monte-Carlo tree search* (MCTS) is one of the most general, powerful and widely used. While MCTS has already earned its reputation in the AI community, there is relatively little experience of its use in Operations Research (OR) community (Bertsimas et al., 2017; Browne et al., 2012). There are only few studies applied MCTS to solve OR problems (Bertsimas et al., 2017; Edelkamp et al., 2016; Mańdziuk and Nejman, 2015). Other than AI games, researchers recently start to see its potential applications in a number of difficult problems, especially those with combinatorial move plannings.

In this section, we present a MCTS approach to solve the TSP-D. We first describe the MCTS in general, and then discuss particular modifications we have tailored to the problem at hand.

4.1. The Monte Carlo Tree Search

In general, MCTS is a best-first search method that does not require a positional evaluation function. To do so, MCTS combines a traditional tree search with Monte Carlo simulations (also known as *rollouts*), and uses the outcome of these simulations to evaluate states in a look-ahead tree. The algorithm progressively builds a partial tree, guided by the results of previous explorations. The tree is used to estimate the values of moves, with these estimates becoming more accurate as the tree is built.

The basic MCTS process is conceptually simple. MCTS usually starts with a tree containing only the root node. From there, the tree gradually grows by iteratively going through four main steps (see Fig. 3) before making a decision:

1. Selection: starting at the root node representing the current state, recursively select the most "interesting" child nodes (actions) until a leaf node L is reached.
2. . Expansion: if L is not a terminal node then create one or more child nodes and select one child node C.
3. Simulation: from C, perform random simulation until reaching the final state.
4. Back-propagation (update): with the obtained simulation results, update the current move sequence.

The idea is to develop an algorithm that can reduce the estimation error of the nodes' values as quickly as possible to minimize the error probability when being stopped prematurely and converged to optimum given sufficient time. To do so, the algorithm must balance exploitation of the currently most promising action with exploration of alternatives which may later turn out to be superior. How the nodes in the tree are selected affects how the tree is built. Hence, the breakthrough rests primarily on the selectivity mechanism, the way in which it ranks the available actions for selection.

Having that in mind, Kocsis and Szepesvári (2006) treat the *selection step* as a multi-armed bandit problem. Having the MCTS as the backbone, they propose *Upper Confidence Bounds (UCB1)* and combine with MCTS to serve as a *tree/selection policy* to guide the node selection. This MCTS implementation using UCB1, known as the *Upper Confidence Bounds for Trees*, is going to be applied for the considered TSP-D and is referred as MCTS for short in this paper. Considering the UCB1 embodied in MCTS, one selling point we see in the MCTS is the ability to address the dilemma of "*exploitation vs exploration*". The issue of balancing the search orientation: discovering potential search space and focusing "good" areas is proven to be elegantly and effectively solved in the MCTS (Auer et al., 2002). We explain this issue later in subsection 4.2.2.

4.2. Tailoring the MCTS to TSP-D

In this subsection, we describe how our proposed constructive heuristic for the TSP-D applies the MCTS concept in action. Based on Mańdziuk and Nejman (2015), we go through several key issues we find necessary to address to adapt the MCTS to such a combinatorial optimization problem:

- Representation of the problem in a tree-like structure;
- A method for evaluating simulation results;

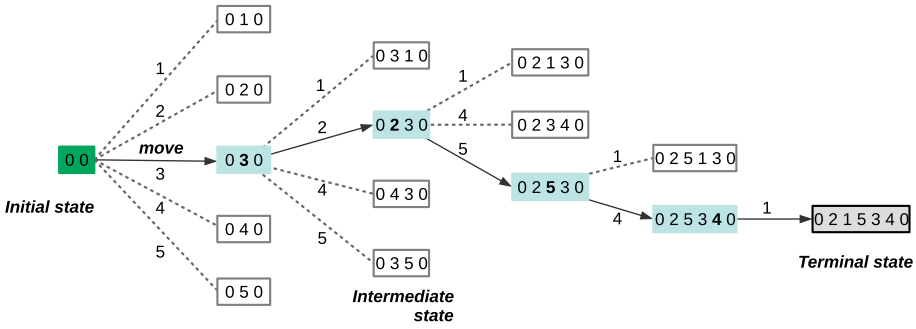

Fig. 4. An example of tree search for TSP.

- Node selection strategy based on simulation data.

As a constructive heuristic, our method heavily relies on insertion heuristics which build up solutions step-by-step. These heuristics have a common characteristic: the earlier insertions might affect the later. According to Ropke and Pisinger (2006), an obvious problem with these myopic heuristics is that they often postpone the placements of "hard" requests (requests that are expensive to insert) to the last iterations where we do not have many opportunities for inserting the requests because many of the routes are "full" (due to, e.g., capacity constraints, time windows constraints). That said, the problem turns into another problem which seeks to *determine the best sequence of insertions in order to achieve the best solutions possible*. Having discussed the core idea of our route search, theoretical features of MCTS address this issue. MCTS offers greedy insertion the ability to "look ahead" several moves by simulation insertions, and thus can see which sequence of insertions are likely to lead to "good" tours. In the same order of MCTS 4-step procedure in which a tree is grown, we explain our method in what follows.

4.2.1. Move definition

First, we define a move in the context of a combinatorial optimization problem like TSP-D. Normally, tree data structure is a standard way to formulate state space in many games, e.g., chess, go or checker. A state space essentially consists of a set of nodes representing *states* of the problem, arcs between nodes representing the legal *moves* from one state to another, a root node represents *initial state*, and a *terminal state* where there is no children and game result is concluded.

However, for routing problems, it is not straightforward and natural to directly translate the route configuration into tree structure itself. To optimize performance, Fredman et al. (1995) suggest that the tour data structure should facilitate queries supporting search operators. By this, we do not mean to use tree structure but rather the traditional array-based for solution representation. It is worth noting that our method incrementally builds a complete route from scratch by progressively assigning customers into the considered tour. Thus a naive option one could think of is the *in-order insertion*: to append "proper" customer node at the end of the current partial route in each iteration. Yet, this assignment policy ended up with poor results in preliminary experiments during our implementation. It might due to the fact that strict insertions easily make sub-optimal decisions at the early stage, which would likely lead to unpromising search space. A more flexible alternative, *cheapest insertion*, is expected to alleviate this drawback. This policy inserts customer i where it would cause the least increase in the tour cost.

With that being said, we define a move i is the action that inserts the unvisited customer p_i into the current tour $\mathscr{T} = \{p_0, p_1, \ldots, p_m\}$ (where $p_0 = p_m = 0$, $p_i \notin \mathscr{T}$, $m \leq n$) right after node p_k such that $k = \arg\min_{1 \leq k \leq m-1} c(p_k, p_i, p_{k+1})$. Hereby, going one level down in the tree corresponds to assigning one customer to the current TSP tour. Thus, the *branching factor* and the *full height* of the tree are equal to the number of unvisited customers at a particular state. The *root node,* representing the initial state of solution, has branching factor equals n which implies n available moves. The *terminal node*, representing the terminal state of solution where all customers are visited, has branching factor equals 0, indicating no more moves to execute and the rollout ends here. Fig. 4 demonstrates an example of translate the TSP in the tree-form.

4.2.2. Node selection (tree policy) and tree reuse

In selection step, from the current root node, the tree picks a child to expand according to the *tree policy*: if there are any child nodes that have never been expanded before, then expand one of the unexplored child at random. Otherwise, compute the "exploration/exploitation" value of each child node and expand the one with highest value computed by Eq. (2) (Browne et al., 2012; Kocsis and Szepesvári, 2006):

$$i = \text{argmax}_i \left(v_i + C \times \sqrt{\frac{\ln N}{n_i}} \right) \tag{2}$$

where v_i is the estimated value of node i, n_i is the number of times node i has been visited, N is the total number of times its parent node has been visited, and $C > 0$ is the explore bias parameter which controls how much exploration to perform in the tree. C can be tuned experimentally depending on the balance between "exploration/exploitation". For instance, $C = \sqrt{2}$ can yield good results for reward in range [0,1], whereas rewards outside this range, other values of C may yield better results (Browne et al., 2012).

Unlike minimax and $\alpha\beta$ algorithm, MCTS can evaluate large trees without sophisticated evaluation functions. As a specific class of rollout algorithms, MCTS repeatedly performs multiple rollouts and then uses the average outcome to approximate the minimax value of the tree. After performing a number of rollouts R, a decision for the best move (i.e., which customer to be inserted into current tour $\mathscr{T}$) must be made based on simulation results. A node i therefore records important information such as: children nodes $\mathscr{C}$, the total number of simulations n_i, and the total rewards v_i.

The idea of MCTS with UBC1 variant is to balance between exploitation and exploration (Browne et al., 2012), which is equivalent to "intensification" and "diversification" in OR literature. The first component of the UCB1 formula (2) above corresponds to exploitation, as it is high for moves with high average win ratio. The second component corresponds to exploration, since it is high for moves that are rarely visited and have fewer simulations. As each node is visited, the denominator of the exploration term increases, which decreases its contribution. On the other hand, if another child of the parent node is visited, the numerator increases and hence the exploration values of unvisited siblings increase. The exploration term ensures that each child has a non-zero probability of selection, which is essential given the random nature of the rollouts. This also imparts an inherent restart property to the algorithm, as even low-reward children are guaranteed to be chosen eventually (given sufficient time).

MCTS runs a large amount of simulations, thus it constructs variety of solutions. We occasionally encountered circumstances where the final

solutions from MCTS were not as good as solutions found of simulations. Therefore, it is important to memorize and keep updating the best solutions *best* found by random walking. To this end, the MCTS can avoid missing useful information and return its very best results.

One of advantages of MCTS is that the search tree can be preserved and reused for the next iterations. This allows us to make use of results of earlier simulations. Particularly, the child node corresponding to the played move becomes the new root node; the sub-tree below this child node is retained along with all its statistics, while the remainders of the tree are discarded (Silver et al., 2017). Fig. 5 displays this idea.

4.2.3. Default (rollout) policy

After a child node is chosen to become the starting point for the expansion, a *rollout* (i.e., simulation) is performed. Our rollout generates a TSP-D tour based on the route-first-cluster-second principle (Beasley, 1983), which is commonly known for its success in solving VRP variants (Prins, 2004). The same principle can also be found in proposed heuristic algorithms of Agatz et al. (2018); Ha et al. (2018). The procedure simply has two steps as follows:

1. Construct a giant TSP tour $\mathscr{T}$ containing all customers. We assume the initial plan only let the truck visit all customers;
2. Given the TSP tour generated above subjected to the fixed relative order of the customers, we split the tour $\mathscr{T}$ into multiple *operations* including flying and driving compounds.

Algorithm 1. Pseudocode of rollout function for TSP-D.

Algorithm 1 Pseudocode of rollout function for TSP-D

```
1  function ROLLOUT(moves)
2    Shuffle(moves)
3    for i = 0 to |moves| do
4      Assign_to_T(moves[i])
5    BELLMAN-BASED SPLIT(T)
6  return T
```

The key-point is to search the space of giant tours TSP rather than that of TSP-D, which is much larger, without loss of information. This is due to the fact that any giant tour TSP can be converted into a TSP-D solution using split procedure. Especially, any giant tour TSP can be split optimally, subject to its sequence.

As show in Algorithm 1, given the current state corresponding to a partial TSP tour, a rollout takes actions by inserting unvisited-customers in random order. The rollout ends as soon as it reaches the terminal state at which the giant TSP tour is completely built and all customers are served. Recall that we use the cheapest insertion heuristic as our pilot heuristic, which inserts one by one customers to positions where it would incur the least increase in tour cost. It is then followed by the split procedure to transform the TSP as indirect solution representation into an actual TSP-D solution. One can see the split procedure as a stage in which remote customers are discarded from its truck tour. Those are then assigned to drone deliveries, each of which require two combined nodes at two ends. They later form multiple consecutive partitions in the form of operations. To this end, it is expected to shorten the total travel time.

Consider a giant TSP tour encoded as a permutation $T = (T_0, T_1, \ldots, T_n, T_{n+1})$ of n customers with the depot at the beginning and at the end denoted as T_0 and T_{n+1}. As shown by Beasley (1983), an optimal splitting (subject to the sequence) can be obtained by computing a shortest path in an auxiliary graph $H = (Y, U)$. Y contains $n+2$ nodes numbered from 0 to $n+1$.

Each subsequence of customers $(T_i, T_{i+1}, \ldots, T_j)$ as an operation within the giant tour is modeled in U by one arc (i,j), weighted by time $t(i,j) = \max\{t_{\text{truck}}, t_{\text{drone}}\}$. Each operation is evaluated to see if it is feasible, i.e., if its drone delivery is possible given the battery and customer-type constraints. The optimal splitting corresponds to a shortest path from node 0 to $n+1$ in H. The shortest path can be computed using Bellman's algorithm for directed acyclic graphs. The split algorithm (Algorithm 2) is modified based on Bellman-based algorithm, many of which can be found in Prins (2004).

Optimal TSP tours do not necessarily lead to optimal TSP-D solutions after splitting. However, given one TSP tour $\mathscr{T}$, split algorithm always guarantees optimal partitioning. The Bellman split algorithm has $O(n^2)$ complexity. To avoid exhaustively enumerating the auxiliary graph, we limit the upper bound value B, which means we only consider arcs (i,j) for $j < i + B$. With the constant upper bound B, our split algorithm reduces complexity to $O(nB)$ in practice.

Algorithm 2. Pseudocode of Bellman-based Split Algorithm for TSP-D.

Algorithm 2 Pseudocode of Bellman-based Split Algorithm for TSP-D

```
1  function BELLMAN-BASED SPLIT(T)
2    set V_0 to 0 and other labels V_i to ∞ (cost of path to node i)
3    set Drone_i to 0
4    for i ← 0 to n − 1 do
5      for j ← i + 1 to n while route(T_i, T_{i+1}, ..., T_j) feasible
6        for any node k betwwen (i, j) do
7          determine the best node k* to assign to drone delivery
8        compute arc cost z_{i,j} of arc (i, j)=max(fly time, drive time)
9        if V_i + z_{i,j} < V_j then
10         V_j ← V_i + z_{i,j}
11         Drone_j = k*
```

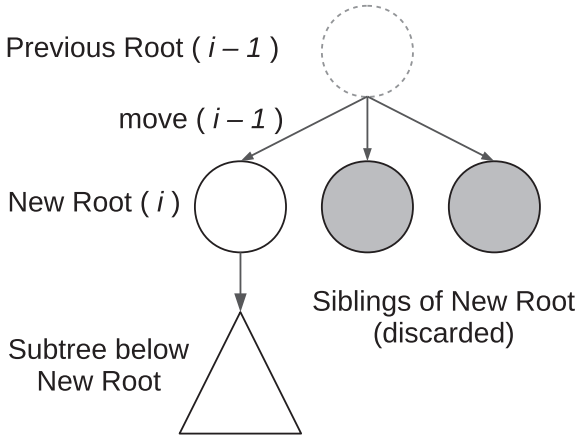

Fig. 5. Tree reuse in the MCTS (Source: Soemers et al., 2016).

The role of Algorithm 2 is to determine and store the predecessor of node j on the shortest path leading to j. After running split procedure, another extra step is needed to extract the actual TSP-D solution. Algorithm 3 shows how to extract the TSP-D solution S using these predecessors.

Algorithm 3. Extraction of TSP-D solution after Split.

Algorithm 3 Extraction of TSP-D solution after *Split*

```
1   function CONVERT TSP-D
2     S = ∅
3     j ← n
4     repeat
5       operation ← ∅
6       add node V_j at the end of operation
7       drone node index k ← Drone_j
8       if T_k > 0 then
9         add T_k at the end of operation
10      add customer node T_j at the end of operation
11      add operation at the beginning of S
12      j ← V_j
13    until j = 0
```

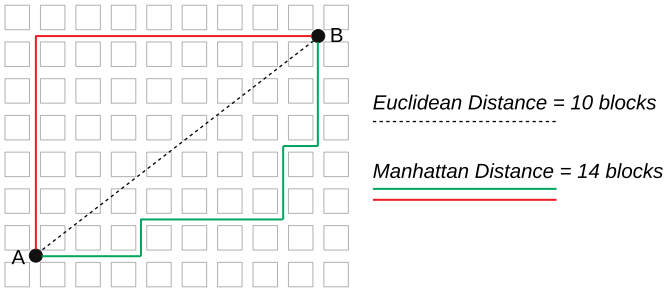

Fig. 6. Illustration of two types of geometry distance.

$$f(x) = \begin{cases} 1 & \text{for} \quad x < \sigma_{best} \\ g(x) & \text{for} \quad \sigma_{best} \le x \le \alpha\sigma_{best} \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

where x is the cost of considered solution, $g(x)$ is the inner function, σ_{best} is the cost of the best solution so far. The inner function $f(x)$ defines the result gradation pattern, i.e., the policy decides whether it should consider promoting or degrading particular solutions. It is worth to mention that the value of α should be greater than or equal to 1, indicating the threshold at which a solution having objective function equal to x is considered *good* if $x \le \alpha\sigma_{best} (\alpha \ge 1)$. The closer to 1 α is, the more it demands on quality of a solution to be considered "good" enough as a reward determined by the function $g(x)$. As a result, intensification of the algorithm is promoted as the reward would only distribute to minority of nodes those are elite. In the contrary, the higher α is, the more even among nodes in terms of reward if x is sufficiently good, which stimulates diversification to generate diverse solutions so as to explore the search space on a global scale.

Regarding $g(x)$, we choose hyperbolic function as show in Eq. (4) such that the search would concentrate on promoting the very best results while ignoring those are poor and average ones:

$$g(x) = \left(\frac{\alpha\sigma_{best}}{x} - 1\right)^2 \quad \text{for } \sigma_{best} \le x \le \alpha\sigma_{best} \tag{4}$$

As a result, value of $f(x)$ varies in range of $[0, 1]$.

4.2.4. *Terminal state evaluation*

This part is important to answer how we evaluate a TSP-D tour obtained at the terminal state of a rollout. In most games, which are the original applications of MCTS, player agent can easily evaluate results of the terminal states by checking whether a game win, lose, or draw. In TSP-D, however, the concept of winning and losing are a bit more complex given the entire tour and its total cost, from which MCTS classifies as good/bad results to guide the search.

The simplest idea is to define that a rollout wins $(f(x) = 1)$ if it produces a solution better than the best one so far $(x < \sigma_{best})$, otherwise a rollout loses and is ignored $(f(x) = 0)$. In order for the MCTS to expand the tree in the right directions without missing useful information from simulation, Mańdziuk and Nejman (2015) introduce the inner function $g(x)$ to take full advantage of good solutions found. We generalize the reward function proposed in Mańdziuk and Nejman (2015) with the *sufficiency threshold* α as follows:

4.2.5. *The general outline*

After providing details of several fundamental features in the MCTS modified for the TSP-D, we outline the pseudocode in Algorithm 4.

Algorithm 4. General MCTS-based search framework for TSP-D.

Algorithm 4 General MCTS-based search framework for TSP-D

```
function SEARCH()
    Initialize root, s ← [], R_best ← ∞
    while there exists possible move do
        move ← BEST MOVE(root)
        Add move to s
        root ← root.best_child
    T ← ROLLOUT(s)
return T
function BEST MOVE(node)
    while there are enough time left do
        leaf ← TRAVERSE(node) // 1. Select the best MCTS children
        leaf ← ADDCHILD(leaf) // 2. Expand from the leaf
        R ← ROLLOUT(leaf) // 3. Simulation
        BACKPROPAGATE(leaf, R) // 4. Update stats
        if R < R_best then
            R_best ← R
return node.best_move
```

5. Numerical experiments

5.1. Benchmark instance, and experiment setting

Seeking large instances to evaluate our heuristic, Freitas and Penna's instances were chosen for numerical experiments. Based on the existing well-known TSPLIB benchmark, created by Reinelt (1991), Freitas and Penna, 2020 modified 24 instances derived from TSPLIB for the FSTSP with the size ranging from 52 to 200 nodes with various properties in terms of problem size and geographical distribution of cities.

Unfortunately, we were unable to access the proposed instances in Freitas and Penna, 2020. For that reason, we replicated the considered instances for our own testbed, which based on the exactly same idea. We proceeded as follows. Firstly, customer locations are remained where they are as in the original TSPLIB instances. Then the customers, considered eligible to drone delivery, are randomly generated such that for every instance there are between 85% and 90% serviceable customers and the other 10%–15% are truck-only customers. For all instances, both vehicle speeds are set at 40 and the drone's endurance t^{max} equal 40 of flight time.

In order to simulate the city block network through which the truck has to traverse, *Manhattan distance* is used to represent the truck travel distance in the road network, while the *Euclidean distance* metric is used to describe the drone flying distance. Fig. 6 demonstrates the two variants of geometry distance metrics.

In the effort to have the comparison valid, each original instance was randomly generated 10 variants each of which has different setting of customer drone eligibility. The experiment was performed by running 10 times for each variant. As such, given 24 instances, 10 setting variants each, we have in total: $24 \times 10 \times 10 = 2400$ tests.

Table 3 shows the parameter setting we use for the numerical experiments. The fixed number of rollouts R is set 200 based on the results obtained in the first experiment (sub-subsection 5.2.1) in which we evaluate the performance of MCTS given different R. The value 200 is found to have the balance between runtime and solution quality trade-off. Finally, the split upper bound B is limited at value of 10 in order to keep the simulation runtime reasonable.

Table 3
Default parameter configuration for the MCTS.

Parameter		Value
Fixed number of rollouts	R	200
Bias exploration parameter	C	2
Sufficiency threshold	α	1.5
Split upper bound	B	10

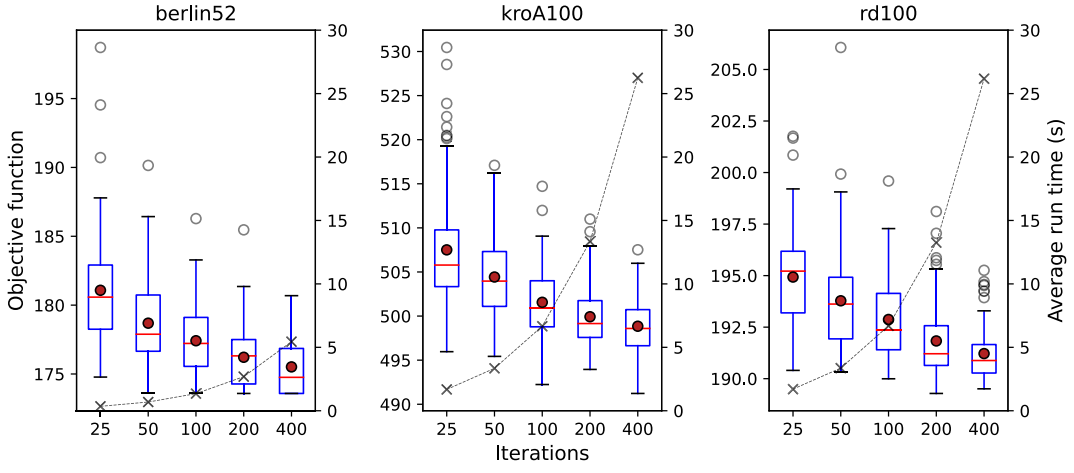

Fig. 7. Performance of MCTS on different amount of rollouts (the lower the better).

For the results, we denote σ, *T*, and ρ as the value of objective function, run time in seconds, and performance ratio, respectively. We define ρ as: $\rho = \frac{\sigma' - \sigma}{\sigma'} \times 100$, where σ is the objective function value obtained by our algorithm and σ' is that of reference algorithm, taken from results in Freitas and Penna, 2020.

The proposed method is implemented in C++ and executed on a desktop computer with AMD Ryzen 5–1600 @ 3.2 GHz and 32 GB of RAM, running 64-bit version of Ubuntu 19.1 operating system.

5.2. Results

5.2.1. Performance of the MCTS on different number of rollouts

We first study the impact of the number of rollouts per decision *R* on MCTS behavior. Kocsis and Szepesvári (2006) proved that the failure probability at the root (i.e., the probability of selecting suboptimal actions) converges to zero at polynomial rate as the number of simulation grows to infinity. This implies that given enough time and memory (if possible), MCTS is allowed to converge to a minimax tree, and thus optimal solutions can surely be found. That said, we expect that the more Monte Carlo simulations (i.e., rollouts) we perform, the more likely we are able to find better solutions.

To test this deduction, it is necessary to test on a large sample of instances. However, due to the significant computational effort required, we only investigated several instances with no more than 100 customers. As such, the berlin52, kroA100, and rd100 were randomly selected. In each instance, we varied the $R \in \{25, 50, 100, 200, 400\}$, each of which was run 100 times and aggregated the results. The experiment recorded solution's cost, and run time (in seconds) of each run. A summary of the results is illustrated in Fig. 7. In each subplot presenting the result of each instance, we have two datasets plotted together: solution's cost and average run time associated with the left y-axis and the right y-axis, respectively. The x-axis labels the set of *R* values.

Table 4
Results of MCTS compared with HGVNS on TSPLIB instances.

Instance	σ_{TSP}	HGVNS				MCTS					ρ_{avg}	ρ_{best}
		σ_{best}	σ_{avg}	T_{avg}	δ_{avg}^{TSP}	σ_{best}	σ_{avg}	T_{avg}	δ_{avg}^{TSP}	*RSD*		
				(s)	(%)			(s)	(%)	(%)	(%)	(%)
Berlin52	239.75	172.25	196.25	6.50	18.14	173.59	178.95	2.73	25.36	1.34	8.81	−0.78
bier 127	3712.00	3456.80	3587.88	23.69	3.34	3439.74	3538.49	20.36	4.67	1.12	1.38	0.49
ch130	190.96	178.16	180.40	44.13	5.53	145.61	150.81	35.56	21.02	1.25	16.40	18.27
d198	472.63	461.83	461.83	27.69	2.28	417.56	429.23	74.89	9.18	0.78	7.06	9.59
eil51	13.55	13.45	13.68	11.57	−0.96	9.76	10.02	3.12	26.05	0.76	26.76	27.47
eil76	17.20	16.35	16.68	27.14	3.02	11.78	12.19	8.88	29.11	1.16	26.90	27.94
kroA100	661.30	587.80	609.71	10.95	7.80	495.85	510.90	13.56	22.74	0.62	16.21	15.64
kroA150	832.60	729.95	758.43	10.95	8.91	622.49	642.61	37.46	22.82	1.20	15.27	14.72
kroA200	932.85	870.65	873.99	6.53	6.31	715.54	738.50	79.26	20.83	0.98	15.50	17.82
kroB150	836.40	763.15	773.72	20.20	7.49	607.61	624.93	39.71	25.28	1.03	19.23	20.38
kroB200	932.85	804.47	835.89	60.00	10.39	701.07	724.39	89.90	22.35	1.32	13.34	12.85
kroC100	666.15	575.30	611.31	20.07	8.23	503.83	515.35	14.14	22.64	0.55	15.70	12.42
kroD100	663.35	606.45	652.34	9.15	1.66	507.31	521.23	14.36	21.42	0.65	20.10	16.35
kroE100	695.00	556.30	570.30	8.57	17.94	528.59	540.08	15.58	22.29	0.76	5.30	4.98
lin 105	424.45	378.25	380.43	10.64	10.37	336.57	350.28	17.75	17.48	1.31	7.93	11.02
pr107	1222.50	1017.50	1059.00	15.48	13.37	1004.25	1018.63	16.86	16.68	0.53	3.81	1.30
pr124	1687.25	1553.80	1566.62	25.45	7.15	1552.23	1582.42	22.23	6.21	0.59	−1.01	0.10
pr136	2800.00	2542.00	2559.00	44.50	8.61	2406.01	2491.21	25.71	11.03	1.35	2.65	5.35
pr144	1688.75	1666.25	1675.75	62.63	0.77	1659.52	1679.41	32.01	0.55	0.59	−0.22	0.40
pr152	2126.70	1919.35	1938.47	70.33	8.85	1912.03	1942.13	38.96	8.68	0.63	−0.19	0.38
rat 99	38.25	37.15	37.33	9.76	2.41	29.47	30.09	15.35	21.33	0.95	19.40	20.67
rat 195	75.40	71.40	71.64	24.13	4.99	57.79	59.04	88.58	21.69	0.94	17.58	19.06
rd100	248.44	240.46	243.84	10.83	1.85	188.36	192.21	16.22	22.63	0.70	21.17	21.67
st70	21.00	20.50	21.00	3.85	0.00	15.45	15.83	5.68	24.62	0.69	24.62	24.61
Average					6.60				18.61		12.65	12.61

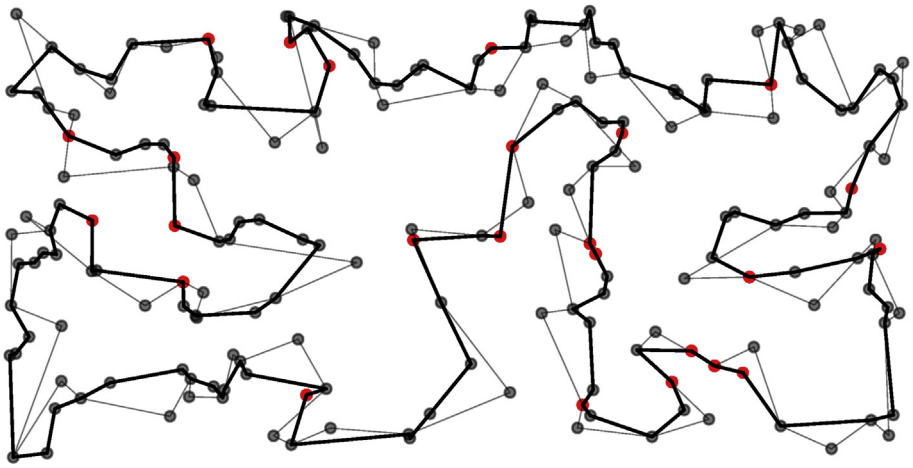

Fig. 8. An illustration of a TSP-D solution for kroA200 instance.

Unsurprisingly, we can observe the similar patterns in the three boxplots. First, the figure shows MCTS tends to produce better solutions as R increases. This is indicated in berlin52 instance for example, we find that σ_{median} steadily decreases from 180.58 to 174.76 as R increases from 25 to 400. Besides, the minimum value, as the best σ of 100 runs, appears to be lower in higher R. Second, the plot indicates less variation with tighter interquartile range (IQR) and shorter whisker as R increases. This shows the progress in performance consistency from left to right. Third, one may empirically observe that run time is proportional to the amount of rollouts R. In fact, run time grows linearly from left to right. Regardless of high variation, we notice that low R settings are able to produce comparable results to those with higher R. Apart of that, increasing R does not always guarantee for a boost in outcome quality. We still detect outliers beyond the high end in settings with high R. The phenomenon could be explained due to the noise from random simulation which did not scale well for large instances, hence, the algorithm tends to struggle to have enough useful information to make moves at the beginning. Making wrong decision at the early stage would cost the loss of the optimal result. Intuition suggests that the performance of MCTS algorithm is highly dependent on the magnitude of the action branching factor, characterized by Rin this paper. In this respect, the concern is all about the trade-off between run time and solution quality.

5.2.2. Performance of heuristics on TSPLIB instances

In this subsection, we aim to evaluate the performance of proposed algorithm under modified TSPLIB instances. Our method is compared with the Hybrid General Variable Neighborhood Search (HGVNS) presented in Freitas and Penna, 2020. Two methods are tested on the same benchmark setting given in section 5.1.

Besides, it is worth to notice that the HGVNS in Freitas and Penna, 2020 was designed for the FSTSP, while the MCTS algorithm proposed in this paper is designed for the TSP-D. Recall that the TSP-D generalizes the FSTSP by allowing the drone to launch and return at the same node. Hence, we turned off that feature in an attempt to make the comparison as fair as possible. As a result, the obtained results by our algorithm can be considered as FSTSP solutions.

The results obtained in Freitas and Penna, 2020 by performing HGVNS ten times for each instance. In overall, the comparison results in Table 4 shows that MCTS outperforms the HGVNS in terms of solution quality. In the table, column σ_{TSP} represents the cost of TSP solutions. The improvement of TSP-D tour over TSP tour is showed in columns δ_{avg}^{TSP}. We report the average and best performance ratio of our method to the HGVNS in ρ_{avg} and ρ_{best} columns, respectively.

Given 24 instances, except berlin52 instance, the proposed method improves all best known solutions found by HGVNS in Freitas and Penna, 2020. The obtained improvement is roughly 12.6% on average. TSP-D solutions found by MCTS can reduce the cost of TSP tours up to nearly 30% (18.6% on average) while HGVNS can only reduce maximum 18.1% (6.6% on average). Oddly, we observed several solutions produced by HGVNS were even not as good as TSP tours we found in the experiment (namely eil76, st70).

Moreover, the table shows the performance consistency of the MCTS algorithm indicating by the relative standard deviation percentage (*RSD* column), with less than 1.35% through all instances. As to run time, MCTS is slightly faster in those tests with instances less than 150 customers. Run time then escalates quickly and is significantly higher when it comes to larger instances. This can be improved by reducing B which might result in degrading solution quality. Nevertheless, it is still acceptable given that only less than one and a half minutes are needed for a single run on a 200-customers instance.

Fig. 8 demonstrates an example of a TSP-D solution for kroA200 instance. The solution shows the routing plan for 200 customers scattered all over the map. The red dots are customers who are only eligible to truck delivery, while the gray dots are customers who can be served by both the truck and the drone. The truck tour is indicated by black bold arcs. Drone deliveries, including the launch and the return flights, are indicated by pairs of gray thin arcs with a customer served by the drone in between.

6. Discussion

In this study, we consider the Traveling Salesman Problem with Drone (TSP-D) applied for last-mile-delivery systems, in which a combined truck-drone model is employed to handle package distributions. We develop a tree search method based on the Monte Carlo Tree Search (MCTS). Numerous experiments are conducted on variety of instances derived from the TSPLIB benchmark. The results show promising performance of the proposed MCTS as it outperforms the compared heuristic method.

Our study shows how generic and flexible of MCTS in the lack of domain-knowledge. While the current literature on TSP-D basically rely on multiple local search operators, the MCTS approach basically needs a simple rollout algorithm. We see the potential for further extensions, like time windows, e.g., those can be easily adapted into our algorithm.

Many reported that significant improvements in performance can often be achieved using domain-specific knowledge although it can be applied in its absence. With that in mind for the further works, we expect to see better performance with existing heuristics, e.g., nearest neighbor insertion, farthest insertion in consideration rather than uniform random simulations in our current work.

One thing we found in the results, run time significantly grew as the problem size increased. This is due to the complexity of the Bellman-based split algorithm, which is called inside every rollout. The future work should take into account the linear split algorithm introduced by Vidal (2016) to accelerate the rollout phase. It also allows more intensive search in the space of giant tours without pruning auxiliary graph, hence, giving potential improvements in terms of solution quality.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

Agatz, N., Bouman, P., Schmidt, M., 2018. Optimization approaches for the traveling salesman problem with drone. Transport. Sci. 52 (4), 965–981.

Auer, P., Cesa-Bianchi, N., Fischer, P., 2002. Finite-time analysis of the multiarmed bandit problem. Mach. Learn. 47 (2–3), 235–256.

Beasley, J.E., 1983. Route first-cluster second methods for vehicle routing. Omega 11 (4), 403–408.

Bertsimas, D., Griffith, J.D., Gupta, V., Kochenderfer, M.J., Mišić, V.V., 2017. A comparison of Monte Carlo tree search and rolling horizon optimization for large-scale dynamic resource allocation problems. Eur. J. Oper. Res. 263 (2), 664–678.

Bouman, P., Agatz, N., Schmidt, M., 2018. Dynamic programming approaches for the traveling salesman problem with drone. Networks 72 (4), 528–542.

Browne, C.B., Powley, E., Whitehouse, D., Lucas, S.M., Cowling, P.I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., Colton, S., 2012. A survey of Monte Carlo tree search methods. IEEE Transactions on Computational Intelligence and AI in games 4 (1), 1–43.

Edelkamp, S., Gath, M., Greulich, C., Humann, M., Herzog, O., Lawo, M., 2016. Monte-carlo tree search for logistics. In: Commercial Transport. Springer, pp. 427–440.

Fredman, M.L., Johnson, D.S., McGeoch, L.A., Ostheimer, G., 1995. Data structures for traveling salesmen. J. Algorithm 18 (3), 432–479.

Freitas, J.C.d., Penna, P.H.V., 2020. A variable neighborhood search for flying sidekick traveling salesman problem. International Transactions in Operational Research 27, 267–290. https://doi.org/10.1111/itor.12671.

Ha, Q.M., Deville, Y., Pham, Q.D., Hà, M.H., 2018. On the min-cost traveling salesman problem with drone. Transport. Res. C Emerg. Technol. 86, 597–621.

Kocsis, L., Szepesvári, C., 2006. Bandit based monte-carlo planning. In: European Conference on Machine Learning. Springer, pp. 282–293.

Mańdziuk, J., Nejman, C., 2015. Uct-based approach to capacitated vehicle routing problem. In: International Conference on Artificial Intelligence and Soft Computing. Springer, pp. 679–690.

Murray, C.C., Chu, A.G., 2015. The flying sidekick traveling salesman problem: optimization of drone-assisted parcel delivery. Transport. Res. C Emerg. Technol. 54, 86–109.

Ponza, A., 2016. Optimization of Drone-Assisted Parcel Delivery (PhD thesis).
Prins, C., 2004. A simple and effective evolutionary algorithm for the vehicle routing problem. Comput. Oper. Res. 31 (12), 1985–2002.
Reinelt, G., 1991. Tsplib - a traveling salesman problem library. ORSA J. Comput. 3 (4), 376–384.
Ropke, S., Pisinger, D., 2006. An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. Transport. Sci. 40 (4), 455–472.
Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., et al., 2017. Mastering the game of go without human knowledge. Nature 550 (7676), 354.
Soemers, D.J., Sironi, C.F., Schuster, T., Winands, M.H., 2016. Enhancements for real-time monte-carlo tree search in general video game playing. In: 2016 IEEE Conference on Computational Intelligence and Games (CIG), Pages 1–8. IEEE.
Stolaroff, J.K., Samaras, C., O Neill, E.R., Lubers, A., Mitchell, A.S., Ceperley, D., 2018. Author correction: energy use and life cycle greenhouse gas emissions of drones for commercial package delivery. Nat. Commun. 9 (1), 1054.
Vidal, T., 2016. Split algorithm in o (n) for the capacitated vehicle routing problem. Comput. Oper. Res. 69, 40–47.