17519578, 2020, 13, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/iet-its.2020.0437 by UNIVERSITY OF MORATUWA, Wiley Online Library on [08/01/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

IET Intelligent Transport Systems

Research Article

IET Journals
The Institution of Engineering and Technology

Directionality-centric bus transit network segmentation for on-demand public transit

ISSN 1751-956X
Received on 16th July 2020
Revised 26th September 2020
Accepted on 13th October 2020
E-First on 6th January 2021
doi: 10.1049/iet-its.2020.0437
www.ietdl.org

Thilina Perera[1] ✉, Deshya Wijesundera[1], Lahiru Wijerathna[2], Thambipillai Srikanthan[1]

[1]*School of Computer Science and Engineering, Nanyang Technological University, 50, Nanyang Avenue, Singapore*

[2]*Department of Electronics and Telecommunication Engineering, University of Moratuwa, Bandaranayake Mawatha, Moratuwa, Sri Lanka*

✉ *E-mail: pere0004@e.ntu.edu.sg*

Abstract: The recent growth in real-time, high-capacity ride-sharing has made on-demand public transit (ODPT) a reality. ODPT systems serving passengers using a vehicle fleet that operates with flexible routes, strive to minimise fleet travel distance. Heuristic routing algorithms have been integrated in ODPT systems in order to improve responsiveness. However, route computation time in such algorithms depends on problem complexity and hence increases for large scale problems. Thus, network segmentation techniques that exploit parallel computing have been proposed in order to reduce route computation time. Even though computation time can be reduced using segmentation in existing techniques, it comes at the cost of degradation of route quality due to static demarcation of boundaries and disregarding real road network distances. Thus, this work proposes, a directionality-centric bus transit network segmentation technique that exploits parallel computation capable of computing routes in near real-time while providing high scalability. Additionally, a dynamic fleet allocation algorithm that exploits proximity and flexibility to minimise vehicle detours while maximising fleet utilisation is proposed. Experimental evaluations on a real road network confirm that the proposed method achieves notable speed-up in flexible route computation without compromising route quality compared to a widely used unsupervised learning technique.

Nomenclature

Variables

- Q queue of nodes to process (initially empty)
- s new elementary base cluster
- B list of junctions (initially empty)
- C list of elementary clusters (initially empty)
- C_i cluster at the ith index of C
- T list of leaf nodes (initially empty)
- R root node of the polyline tree
- W list of child nodes (initially empty)
- W_i child node at the ith index of W
- L all leaf nodes

1 Introduction

The global on-demand transit market is expected to be a 250 billion USD industry by 2026 [1] with its rapid growth in the recent past. Driven by the technological developments in fields such as GPS-based location tracking, mobile communication, smartphone technology and cashless payments, novel on-demand transit systems such as ride-hailing, ride-sharing, car-pooling etc. have emerged as attractive alternatives to public transit. Lately, the on-demand transit market has seen a trend in real-time, high-capacity, ride-sharing systems [2].

Commonly known as on-demand public transit (ODPT) systems, ODPT is characterised by a fleet of high-capacity vehicles such as minivans or minibuses, that respond to passenger demand in real-time by picking passengers from the origin and dropping-off at the destination. Furthermore, vehicles in an ODPT system do not follow fixed routes, unlike public transit, necessitating to compute routes prior to deployment. At the same time, in an ODPT system, the operator strives to increase profits by optimally utilising the fleet of vehicles while providing real-time service to passengers. Thus, the vehicle miles travelled (VMT) of the fleet need to be minimised in order to reduce operating costs. High responsiveness is also of paramount importance while serving the significantly high demand. This implies that the computed routes for the fleet of vehicles not only need to be optimised to reduce the VMT but also need to be generated rapidly to improve responsiveness. As a result, exact algorithms which produce optimal results but consume a significantly high computation time cannot be used to generate routes in an ODPT system as the responsiveness is crucial for system performance. Therefore, works propose efficient heuristic algorithms that generate near-optimal results significantly faster. However, a common drawback of the heuristic algorithms is the dependency of the computation time with the complexity of the network. This increases the computation time for scenarios with high demand such as ODPT. As a result, network segmentation techniques that can leverage on parallel computing to speed-up computations have been proposed.

Network segmentation breakdowns the large network into a set of independent sub-networks, which can be solved in parallel resulting in reduced computation time. Clustering is a well-known unsupervised learning method used to segment a network into a set of independent sub-networks that enables to exploit parallel computing architectures. Although clustering can reduce the computation time of route generation, it can lead to poor quality solutions if the clusters lead to a higher VMT of the fleet. Existing works rely on methods such as proximity-based clustering and static clustering [3, 4]. The former method is based on the proximity of the locations while the latter demarcates cluster boundaries based on major roads, common destinations etc. However, in an ODPT system, the locations of demand change dynamically. Therefore, existing clustering methods cannot be relied upon to generate near-optimal routes significantly faster. Hence, there is a need to develop rapid and robust, road network-aware clustering methods for ODPT to speed-up computations while maintaining accuracy.

ODPT systems have been specifically deployed to provide easy and convenient connections to rapid transit nodes [5]. In these systems, passengers request for rides indicating the pick-up location and the system matches the passenger with available vehicles, which operate in shared-ride mode. Thus, we use the setup proposed in our previous work [5] which studies an ODPT system consisting of a fleet of homogeneous, high-capacity, electric vehicles (EVs) dispersed in a neighbourhood, that responds to the passenger demand in real-time by picking passengers from

© The Institution of Engineering and Technology 2020

the origin and dropping off at the nearest metro station (common destination). The system generated near-optimal routes using a genetic algorithm (GA). However, the GA consumes several minutes to execute when the demand is high, which affects the responsiveness of the algorithm. Thus, in this work, we propose a bus transit network segmentation method that leverages the proposed GA to compute routes in each cluster parallelly. The contributions of this work are: (i) a directionality-centric technique for systematic segmentation of bus transit network; (ii) a technique for identifying clusters based on feasible shortest path routes from bus stops to the destination; (iii) a dynamic fleet allocation algorithm that exploits proximity and flexible clusters to minimise vehicle detours and maximise fleet utilisation, respectively; and (iv) analysis of the VMT by the fleet, computation time and failed requests for peak and off-peak period demands. The proposed method can generate routes significantly faster while maintaining accuracy.

The rest of the paper is organised as follows. In Section 2, we discuss the existing state-of-the-art work and highlight the limitations. Next, in Section 3, we present the proposed methodology. The results of the study are discussed in Section 4. Section 5 presents the conclusions and identifies future research directions.

2 Literature review

ODPT systems necessitate high responsiveness for real-time performance as well as near-optimal routes to reduce the VMT of the fleet. However, a drawback of heuristic methods commonly deployed to solve the underlying vehicle routing problem (VRP) is the increase in computation time with the problem size. Thus, works have explored methods to reduce the complexity of a problem. Segmentation is a common method used to decompose a large problem into independent small problems in order to leverage parallel computing architectures to speed-up computations. Existing segmentation techniques use proximity-based clustering methods (such as the well-known *k*-means and *k*-medoids algorithms), and methods that use static boundaries to segment road networks. Generated clusters can be either hard or fuzzy (soft). In hard clustering, a data point belongs to only a single cluster as opposed to soft clustering where a data point can belong to multiple clusters [6]. Although clustering can reduce the computation time, the solution quality can degrade drastically if performed unwisely. In this section, we present the existing bus transit network segmentation methods and highlight key shortcomings.

Ioachim *et al.* [7] presented a clustering algorithm for a door-to-door transportation service. The work presents a hard clustering method in combination with a heuristic routing algorithm to generate routes. Given a feasible set of itineraries, the method segments the itineraries into clusters, with each cluster containing a segment of an itinerary where the vehicle is never empty between the first pick-up point and the last drop-off point. Next, the authors created a network of clusters and solved a multiple travelling salesman problem by column generation. The authors also proposed a heuristic to reduce the size of the network without degrading the quality of the solution.

Jorgensen *et al.* [8] proposed a cluster-first, route-second method to a dial-a-ride problem. The authors used a GA to generate hard clusters, which consisted of a set of passengers and a vehicle. Next, independent routes are generated for each cluster using a heuristic routing algorithm.

Saez *et al.* [9] proposed a hybrid adaptive control method for a multi-vehicle dynamic pick-up and delivery problem. The work proposes a soft clustering method to predict the demand, which is used to dispatch vehicles. A GA is then used in each cluster to obtain routes.

Quadrifoglio *et al.* [4] and Shen and Quadrifoglio [3] evaluated centralised and de-centralised clustering strategies for on-demand transit service. Here, the service area is divided into clusters based on natural boundaries such as highway corridors, administrative zones, depot locations etc. The authors strive to cluster based on natural boundaries to balance the number of inter-cluster trips, which results in reducing dead-head and empty vehicle miles. Furthermore, the authors proposed a soft clustering method to reduce the empty vehicle miles.

Pelzer *et al.* [10] proposed a clustering-based match-making algorithm for dynamic ride-sharing that strives to maximise the utilisation while limiting the vehicle detours. The authors introduced a segmentation method to divide the entire road network into clusters of custom size and shape. Similar to Shen and Quadrifoglio [3], the road network is used to identify the boundaries of each cluster. However, in this work, the authors proposed to scale certain clusters by a predefined factor, which results in overlapped clusters. The authors claimed that overlapped clusters increase the chances of matching passenger requests.

Zheng *et al.* [11] proposed a cluster-first, route-second method to generate routes for a shuttle service that caters large-scale peak demands. In the proposed method, the distance between two points is represented using the temporal distance instead of the actual routes. In addition, the authors also introduced a request rejection mechanism and a scheme for ensuring solution feasibility. Furthermore, the authors claimed that the soft clustering provides better results compared to conventional hard clustering.

Chen *et al.* [12] proposed a method to solve the first-mile ride-sharing problem using autonomous vehicles. The authors formulated a mixed Integer Linear Programming (ILP) model to determine the ride-sharing schemes to minimise operating costs. The authors also introduced a hard clustering method that facilitates parallel computing in order to reduce computational times in large problems.

Comert *et al.* [13] presented a case study of the cluster-first, route-second method using conventional hard clustering for the capacitated VRP. The authors explored three hard clustering algorithms, namely *k*-means, partitioning around medoids (PAMs) and random clustering based on vehicle capacity and concluded that PAM provides better solutions.

Lowalekar *et al.* [14] proposed to use the clustering method to reduce the computational complexity for handling large scale real-time ride-sharing. The authors explored three clustering algorithms, namely grid-based clustering, hierarchical agglomerative clustering with complete-linkage (HAC_MAX) and hierarchical agglomerative clustering with mean linkage. Based on the experiments, the authors concluded that HAC_MAX gives better results.

This literature review (summarised in Table 1) reveals that cluster-first, route-second methods reduce the computational complexity of large scale VRP problems.

Furthermore, it highlights that soft clustering outperforms standard hard clustering. It also shows that state-of-the-art works rely on proximity-based clustering methods and identifying static boundaries to demarcate clusters. However, a sub-optimal grouping of bus stops due to unawareness of real road network distance and disregarding the location of the destination respectively in the two techniques lead to poor solutions. This necessitates the development of a directionality-centric clustering method that can not only speed-up the computations but also be accurate.

3 Methodology

The proposed bus transit network segmentation method shown in Fig. 1 strives to reduce the computation time of flexible route generation while minimising the VMT of the fleet. The method considers the road network of the geographical area as the input. Then leverages the direction of travel and similarity of the shortest path route from the origin to the destination along the road network for clustering of bus stops followed by intelligent fleet allocation and parallel computation of flexible routes in each cluster. The method outputs the flexible routes for the fleet. The subsequent sections explain the proposed method in detail.

3.1 Directionality and shortest-path centric generation of clusters

In this section, we present the directionality and shortest-path centric clustering algorithm and describe each step in detail. The ODPT system assumes that the demand originates at existing bus

© The Institution of Engineering and Technology 2020

17519578, 2020, 13, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/iet-its.2020.0437 by UNIVERSITY OF MORATUWA, Wiley Online Library on [08/01/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

Table 1 Summary of literature

Property	Ioachim *et al.* [7]	Jorgensen *et al.* [8]	Saez *et al.* [9]	Quadrifoglio *et al.* [4]	Shen *et al.* [3]	Pelzer *et al.* [10]	Zheng *et al.* [11]	Chen *et al.* [12]	Comert *et al.* [13]	Lowalekar *et al.* [14]	Proposed work
on-demand transit	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
travel to a common destination	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✓
soft clusters	✗	✗	✓	✗	✗	✓	✓	✗	✗	✗	✓
directionality and shortest-path centric	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓

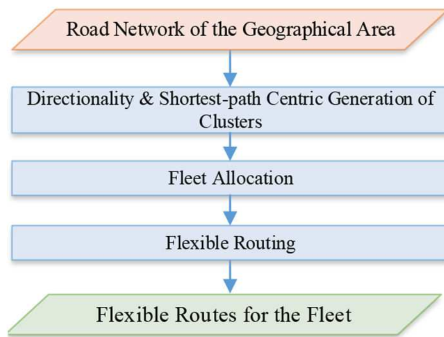

Fig. 1 *Proposed bus transit network segmentation method*

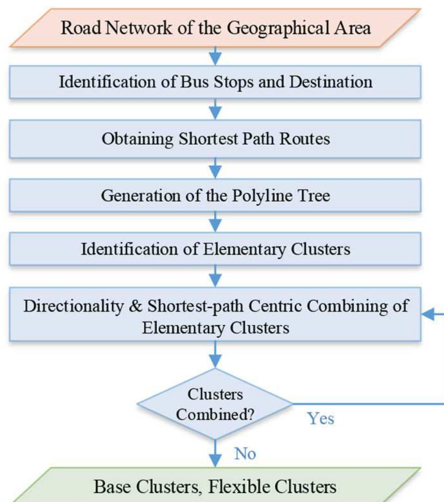

Fig. 2 *Technique for generation of clusters*

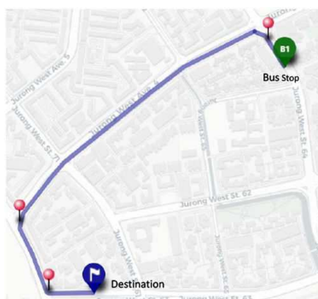

Fig. 3 *Shortest path route obtained from OSRM*

stop locations. Also, it assumes that vehicles are distributed at car parks near the bus stops and all passengers travel to a common destination. The method first identifies the bus stops and the destination of the geographical area. Next, the algorithm obtains the shortest path routes from each bus stop to the destination. Then, based on the shortest paths, the bus stops are arranged in a tree structure *(polyline tree)* to facilitate clustering. Next, leveraging the polyline tree, elementary clusters are identified. Thereafter, elementary clusters are combined to generate large base clusters. In addition, the method identifies flexible clusters that are utilised intelligently in the fleet allocation step in Section 3.2. The proposed technique for generation of clusters is shown in Fig. 2.

3.1.1 Identification of bus stops and destination: Initially, all the bus stops and the destination of the geographical area are identified. As mentioned, it is assumed that all passengers travel to the metro station, which is considered as the destination. The output of this step generates the GPS coordinates of bus stops and the metro station.

3.1.2 Obtaining shortest path routes: Shortest path data is obtained using open source routing machine (OSRM) [15] by providing the GPS coordinates identified in the above step. The output contains the shortest path route from each bus stop to the destination. In addition, the distance matrix, which gives the shortest path distances between all bus stops and the destination is also obtained. Information of the shortest path route from a bus stop to the destination is represented by a sequence of encoded polylines. In general, an encoded polyline represents a route segment which is constructed by connecting multiple way-points. For example, let the bus stop (origin) and destination GPS coordinates (latitude, longitude) be (1.344260°, 103.703293°), (1.337736°, 103.697289°), respectively. The points are shown using B1 green drop and the blue flag, respectively, in Fig. 3. The shortest path route obtained by OSRM for the given coordinates is shown using the blue line. The junctions (waypoints) along the path are marked in red markers.

The sequence of polylines for the respective route is given in Table 2. Each encoded polyline represents a segment of the route. For example, polyline 1 represents the route segment starting from the bus stop to the first junction marked in Fig. 3. This is shown in Fig. 4*a*. Similarly, polyline 2 represents the next segment of the route continuing from the first junction to the second junction as shown in Fig. 4*b*. The remaining route segments (polylines 3–5) can also be identified similarly by tracing the route along the shortest path. Furthermore, we observe that the length of the route segment is proportional to the number of encoded characters in the polyline (the length of the polyline). This property of polylines is exploited to generate the clusters.

3.1.3 Generation of the polyline tree: After obtaining the shortest path routes from all the bus stops to the destination as a collection of polyline sequences, a tree is constructed using the similarities of polyline sequences. The tree facilitates clustering by arranging the bus stops in an organised structure based on the similarity of the shortest path routes. In the polyline tree, the bus stops, junctions and destination are represented using nodes and each polyline sequence is represented by an edge. The root, branch and leaf nodes of the tree represent the destination, junctions and bus stops, respectively. The edges starting from leaf nodes represent polyline sequences (route segments) starting at bus stops, while edges at branch nodes represent the polyline sequences starting from junctions. Therefore, traversing the tree from a leaf

© The Institution of Engineering and Technology 2020

17519578, 2020, 13, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/iet-its.2020.0437 by UNIVERSITY OF MORATUWA, Wiley Online Library on [08/01/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

<table>
<caption>**Table 2** Encoded polyline sequence</caption>
<tr><th>Polyline number</th><th>Encoded polyline generated using [16]</th></tr>
<tr><td>1</td><td>upeGsp}wRg@T_B'AaAd@EDMH</td></tr>
<tr><td>2</td><td>sweGel }wRMVKVEL? ^Xp@~AxDd@hAd@dABH
FJx@rArlhLpA~Aj@v@l@x@HJf@ x@v@|CvB’@\</td></tr>
<tr><td>3</td><td>{vdGm~ {wRt@HHBRBVAb@Q'Ae@hAc@rBi@|A]</td></tr>
<tr><td>4</td><td>mgdGac|wREUAiF</td></tr>
<tr><td>5</td><td>ugdGak|wR</td></tr>
</table>

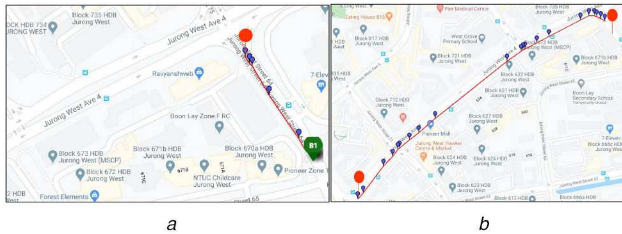

Fig. 4 *Route segments representing polylines in Table 2*
(a) Polyline 1 (bus stop to first junction), ***(b)*** Polyline 2 (first junction to second junction)

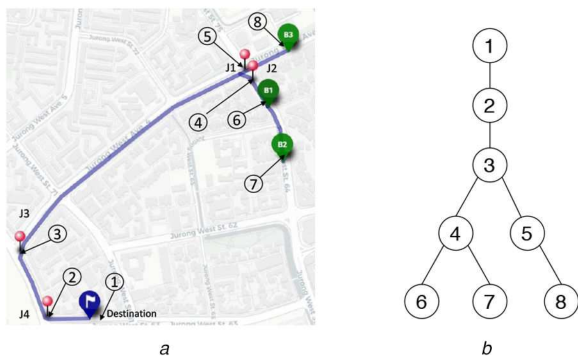

Fig. 5 *Shortest path routes from bus stops B1, B2 and B3 to the destination*
(a) Identified nodes, ***(b)*** Generated tree

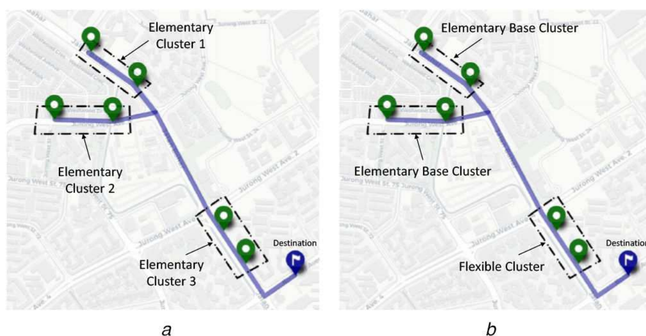

Fig. 6 *Example of elementary clusters*
(a) Identified elementary clusters, ***(b)*** Classified clusters

node to the root provides the polyline sequence of the shortest path route from the bus stop to the destination. Thus, it can be seen that a group of nodes which have the same parent will have an overlapped sequence of nodes. This is explained using the example shown in Fig. 5. Here, three bus stops (green drop) $B1$, $B2$ and $B3$, four junctions (red ball) $J1$, $J2$, $J3$ and $J4$, destination (blue flag) and the shortest path routes from each bus stop are shown. The nodes along the shortest path routes are marked in Fig. 5*a*. The generated polyline tree for this scenario is shown Fig. 5*b*. In this polyline tree, the destination is node 1, while the bus stops $B1$, $B2$ and $B3$ are represented by nodes 6, 7 and 8, respectively. The junctions $J1$, $J2$, $J3$ and $J4$ are represented by nodes 5, 4, 3 and 2, respectively. The polyline sequence of the shortest path routes from each bus stop can be shown as a set of nodes; Bus stop $B1$: 6, 4, 3, 2, 1; Bus stop $B2$: 7, 4, 3, 2, 1; and Bus stop $B3$: 8, 5, 3, 2, 1.

3.1.4 Identification of elementary clusters: The next step is the identification of elementary clusters by leveraging the generated polyline tree. Elementary clusters contain a group of bus stops with similar shortest path routes to the destination except for the first polyline, which is the polyline starting from the bus stop. Assume that the bus transit network consists of only six bus stops as shown in Fig. 6. Based on the given definition, the six bus stops can be classified into three elementary clusters as shown in Fig. 6*a*. However, an elementary cluster where the shortest path route does not overlap with any other shortest path route of another elementary cluster is termed as an elementary base cluster. For example, elementary clusters 1 and 2 in Fig. 6*a* are elementary base clusters. This is shown in Fig. 6*b*. The shortest path route of bus stops in the elementary cluster 3 falls along the path of the bus stops in the elementary base clusters (elementary clusters 1 and 2). Therefore, the vehicles travelling from an elementary base cluster can pick-up the passengers in the elementary clusters along the shortest path route. Thus, elementary cluster 3 is termed as a flexible cluster. The algorithm for identifying elementary base clusters and flexible clusters is given in Algorithm 1 (see Fig. 7). Variables used in Algorithm 1 are summarised in the Nomenclature section. Objects and functions used in Algorithm 1 are introduced in Tables 3–6.

Algorithm 1 is explained using the example scenario shown in Fig. 8*a*. This shows five bus stops ($B1 - B5$) and two junctions ($J1, J2$), which connects the remaining bus stops in the vicinity in the road network. The corresponding polyline tree is given in Fig. 8*b*. The polyline tree is traversed using a breadth-first search starting from the root node (denoted by R in Fig. 8*b*). Initially, an empty queue (Q) is defined and the root node of the polyline tree is enqueued to Q (line 2). Then, while Q is not empty, a node (n) is dequeued from Q (line 5) and the child nodes of n are put into the list W (line 6). Next, the child nodes in W are sorted in the descending order of polyline length (line 7). After sorting, the first node (W_0) in W is the node with the polyline that ends furthest away from the polyline represented by the parent node (n). While W is not empty, in each iteration, the first element (W_0) of W is removed (lines 9–10) and either an elementary base cluster or flexible cluster is initialised. For the example given in Fig. 8*a*, W_0 will be a leaf node (bus stop $B1$). Therefore, the case when W_0 is a leaf node is explained first followed by the case when W_0 is a non-leaf node.

Case 1: W_0 is a leaf node (Bus stop) (lines 11–25). If W_0 is a leaf node, a new cluster (c_{base}) is initialised and W_0 is added to c_{base} (lines 12–13). Then, starting from the first index, all remaining nodes are read iteratively from W (let a removed node be denoted by W_i). Here, node W_i is determined to be combined with W_0 in order to form an elementary base cluster if the character match between the polyline sequences representing the shortest path route to the destination from the two nodes are greater than or equal to the P_n% (*similartyPercentageForNodes*) (line 16). The value for P_n (70%) was determined through experimentation. The relatively high value ensures that the bus stops in elementary base clusters are physically nearby in the road network. However, it should be noted that only leaf nodes are added to the newly created cluster, c_{base} (lines 17–19). Else, the algorithm moves on to the next node and repeats the process until a non-leaf node with a similarity percentage greater than or equal to P_n% is encountered (non-leaf node $J1$). In Fig. 8*b*, leaf nodes $B1$ and $B2$ are added to c_{base}. In this case, c_{base} is determined to be an elementary base cluster since there are no existing clusters which can be formed such that c_{base} is flexible. Also, c_{base} is added as a new elementary base cluster for the flexible clusters that will be discovered during the search process of the remaining nodes in W (line 24). New base elementary cluster functions similarly to elementary base clusters. However, new base elementary cluster is denoted separately since it is not possible to identify these clusters using the contents of the list of junctions (B), which is used to identify the elementary base

© The Institution of Engineering and Technology 2020

17519578, 2020, 13, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/iet-its.2020.0437 by UNIVERSITY OF MORATUWA, Wiley Online Library on [08/01/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

```
Input: Polyline Tree (T_p)
Output: List of Clusters (C)
 1: Execution:
 2: Q.enqueue(R)
 3: s = null
 4: while Q ≠ ∅ do
 5:     n = Q.dequeue()
 6:     W = n.getChildren()
 7:     Sort the nodes in W in the descending order of polyline
        length
 8:     while W ≠ ∅ do
 9:         W_0 = W.get(0)
10:         W.remove(0)
11:         if W_0 ∈ L then
12:             make a new cluster: c_base
13:             c_base.addNode(W_0)
14:             for W_i ∈ W|i ≠ 0 do
15:                 m = W_0.getMatchScore(W_i)
16:                 if m ≥ P_n then
17:                     if W_i ∈ L then
18:                         c_base.addNode(W_i)
19:                         W.remove(i)
20:                     else
21:                         break
22:                 else
23:                     continue
24:             s = c_base
25:             C.add(c_base)
26:         else
27:             Q.enqueue(W_0)
28:             B.add(W_0)
29:             for W_i ∈ W|i ≠ 0 do
30:                 m = W_0.getMatchScore(W_i)
31:                 if m ≥ P_n then
32:                     W.remove(i)
33:                     if W_i ∈ L then
34:                         T.add(W_i)
35:                     else
36:                         Q.enqueue(W_i)
37:                         if T ≠ ∅ then
38:                             make a new cluster → c_flex
39:                             add all the nodes in T to c_flex
40:                             c_flex.setListofJunctions(B)
41:                             c_flex.setFlexibility(true)
42:                             if s ≠ null then
43:                                 c_flex.setNew
                                        BaseCluster(s)
44:                             C.add(c_flex)
45:                             T.clear()
46:                             B.add(W_i)
47:                         else
48:                             continue
49:                 else
50:                     continue
51:             if T ≠ ∅ then
52:                 make a new cluster → c_flex
53:                 add all the nodes in T to c_flex
54:                 c_flex.setListofJunctions(B)
55:                 c_flex.setFlexibility(true)
56:                 if s ≠ null then
57:                     c_flex.setNewBaseCluster(s)
58:                 C.add(c_flex)
59:                 T.clear()
60:     s = null
61:     B.clear()
```

Fig. 7 *Algorithm 1 Identifying elementary clusters*

clusters of a flexible cluster. Then, the new instantiated cluster (c_{base}) is added to the list of elementary clusters, C (line 25), and the algorithm moves to the next iteration of the nested while loop, starting at line 8. In the next iteration, the removed node from W at lines 9 and 10, W_0, will be a non-leaf node, representing the junction $J1$. Therefore, the algorithm moves to case 2.

Table 3 Objects and functions: node (n)

Function	Description
n.getChildren()	returns the list of child nodes of the node *n*
n.getMatchScore(n_2)	returns the matching score between the two nodes *n* and n_2
n.getClusters()	returns the list of base clusters where the root node is *n*

Table 4 Objects and functions: cluster (c)

Function	Description
c.getMatchScore(c_2)	returns the matching score between the two clusters *c* and c_2
c.addNode(*n*)	adds node *n* to the cluster *c*
c.isFlexible()	returns true, if the cluster *c* is a flexible cluster, otherwise false
c.setFlexibility(*T/F*)	set the value for the flexible attribute of *c* (as true or false)
c.setListofJunctions(*b*)	sets the list of nodes, *b*, as the tailing list of junctions of the cluster *c*
c.getListofJunctions()	returns the list of tailing junctions of the cluster *c*
c.setNewBaseCluster(c_s)	sets new base cluster c_s of flexible cluster *c*
c.getNewBaseCluster()	returns the new base cluster of cluster *c*

Table 5 Objects and functions: list (h)

Function	Description
h.add(*m*)	add the element *m* to the end of the list *h*
h.remove(*i*)	removes the element at index *i* from the list *h*
h.get(*i*)	returns the element at index *i* in the list *h*
h.clear()	removes all the elements from the list *h*
h.size()	returns the number of elements in the list *h*

Table 6 Objects and functions: Queue (q)

Function	Description
q.enqueue(*m*)	adds the element, *m*, to the end of the queue *q*
q.dequeue()	returns the element that is in front of the queue and removes that element from the queue

Case 2: W_0 is a non-leaf node (Junction) (lines 26–59). If W_0 is a non-leaf node, then W_0 is enqueued to Q (line 27) and added to the list of junctions (B) (line 28). B will be used to identify the corresponding elementary base clusters of a flexible cluster. Then, starting from the first index, all remaining nodes are read from W iteratively (let a removed node be denoted by W_i). If the similarity percentage of W_i is higher than P_n%, W_i is validated if it is a leaf node. If it is valid, W_i is added to T (temporary buffer of leaf nodes) in each iteration (in this example, leaf nodes $B3$ and $B4$ will be added to T in two iterations) (lines 33 and 34). If W_i is a non-leaf node (in this example, the non-leaf node representing $J2$), it is enqueued to Q (line 36) and a new flexible cluster, c_{flex} is instantiated with all the leaf nodes stored in T (lines 37–39). At the same time, all the contents of B (in this example, $J1$) are added to the list of junctions of c_{flex} (line 40). The motivation of identifying c_{flex} as a flexible cluster is that it will be placed along the shortest path route of elementary base clusters formed from the child nodes of the nodes in B. For example, the bus stops $B3$ and $B4$ in c_{flex} will be flexible to all the elementary base clusters formed farther away from the destination, where the routes overlap from junction $J1$. Similarly, if a new elementary base cluster (s) exists, it is also updated (lines 42–43). Then, the new instantiated flexible cluster c_{flex} is added to list of elementary clusters, C (line 44). Afterwards, T is cleared (line 45) and W_i is added to the list of junctions, B (line 46, now B will contain $J1$ and $J2$). This process is repeated until

© The Institution of Engineering and Technology 2020

17519578, 2020, 13, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/iet-its.2020.0437 by UNIVERSITY OF MORATUWA, Wiley Online Library on [08/01/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

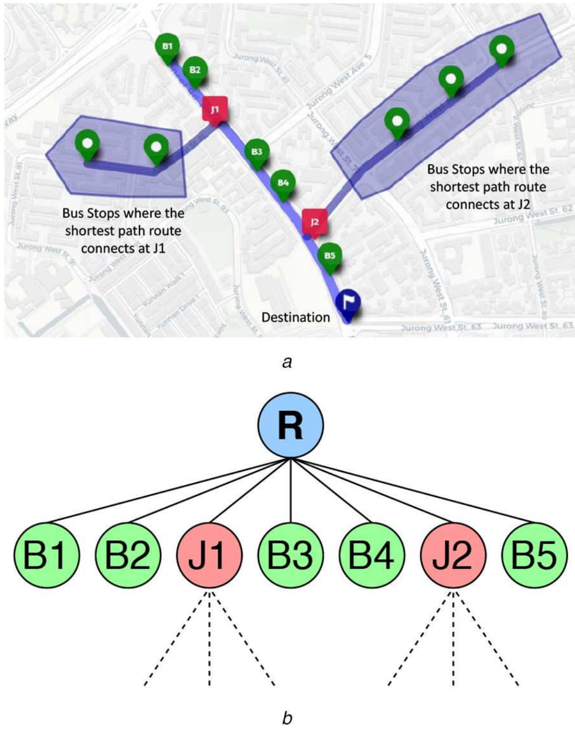

Fig. 8 *Example network for Algorithm 1*
(a) Network, ***(b)*** Corresponding polyline tree

Table 7 Clusters of the example in Fig. 8

Cluster No.	Flexible (Y/N)	Bus Stops	Junctions	New base elementary cluster
1	N	B1,B2	—	—
2	Y	B3,B4	J1	Cluster 1
3	Y	B5	J1, J2	Cluster 1

Input: List of clusters (C)
Output: List of clusters after combining (C')
1: **Execution:**
2: $L = getAllBaseClusters(C)$
3: **while** $L \neq \emptyset$ **do**
4: $L_0 = L.remove(0)$
5: **for** $L_i \in L | i \neq 0$ **do**
6: **if** $L_0.getMatchScore(L_i \geq P_c)$ **then**
7: Combine L_i with L_0
8: $L.remove(i)$
9: $G.add(L_i)$
10: $L'.add(L_0)$
11: $F.getAllFlexibleClusters(C)$
12: $F' = updateFlexibleClusters(F, G)$
13: $C' = L' \cup F'$

Fig. 9 *Algorithm 2 Generating base clusters*

Table 8 Supplementary functions for Algorithm 2

Function	Description
getAllBaseClsuters(C)	returns a list containing all base clusters in the list of clusters C
getAllFlexibleClusters(C)	returns a list containing all flexible clusters in the list of clusters C
updateFlexibleClusters(F, G)	given the list of elementary flexible clusters (F) and the list of base clusters that were combined with other base cluster (G), returns the updated list of flexible clusters, F'

the algorithm iterates over all the elements in W. Then, the algorithm checks if T is empty (line 51). T can be non-empty, if the last node removed from W is a leaf node (in this example, last node removed from W is the leaf node $B5$). Therefore, another flexible cluster is added in the same manner explained above (lines 52–59). Then, s is set to null (line 60) and B is cleared (line 61), which indicates the end of clustering the nodes, whose parent is n. Thereafter, the next node in Q is dequeued (line 5) and the whole process is repeated until Q is empty. Table 7 summarises the clusters and the corresponding data for the example in Fig. 8.

3.1.5 Combining of elementary clusters: Output of Algorithm 1 provides all the elementary base and flexible clusters of the polyline tree. Generally, an elementary base cluster contains 3–4 bus stops, which implies that the possibility of improving routes during optimisation is limited. Moreover, the high-capacity vehicles used in the ODPT system will be underutilised if the number of passengers in the elementary base clusters are low. Thus, elementary base clusters are combined to form base clusters considering the geographical placement. For example, combining two elementary base clusters that are placed apart in the road-network will significantly increase the VMT. Therefore, the proximity of the elementary clusters is crucial for enabling the combination of clusters. While combining clusters, the following two cases are considered.

Case 1: In this case, we consider combining two elementary base clusters into a single base cluster. The criteria for combining elementary base clusters considers the similarity of the route to the destination. However, since a cluster is a collection of leaf nodes, which represent different bus stops, a single leaf node must be selected to represent the whole cluster. For this, the node which has the median distance to the destination among all the nodes of the cluster (henceforth referred to as median node) is selected. Here, distance is identified by the length of the polyline sequence representing the route to the destination. During the combining process, the median nodes of the two clusters are compared for a P_c% (*similartyPercentageForClusters*) character match in the encoded polylines. If the comparison is valid, the two elementary base clusters are combined to a single base cluster.

Case 2: In this case, we consider combining a base cluster and a flexible cluster into a single base cluster. This case occurs either when two or more elementary base clusters which originally had a common flexible cluster, is combined into a single base cluster using case 1 or if a flexible cluster was only flexible to one elementary base cluster. Therefore, the flexible cluster is now only flexible to a single base cluster thus, necessitating to combine and form a base cluster.

The algorithm for generating base clusters is given in Algorithm 2 (see Fig. 9). The supplementary functions used in Algorithm 2 are given in Table 8. As shown in Fig. 2, combining clusters is an iterative process. Therefore, Algorithm 2 is iteratively executed until there is no change in the set of clusters. Therefore, in the first iteration, all the base clusters extracted are elementary base clusters. From the second iteration onwards, combining can occur between combined base clusters and elementary base clusters. However, for clarity of explanation these are referred as base clusters irrespective of the iteration number.

Initially, all base clusters are extracted from the list of clusters, C, and assigned to a list, L (line 2). Then, one base cluster (L_0) is removed from L (line 4). Thereafter, all remaining base clusters which can be combined with L_0 is combined, while removing each combined base cluster from the remaining cluster list (lines 6–8). Also, each combined cluster is added to a list G (line 9). This is used to update the flexible clusters. Then, L_0 is added to the updated list of base clusters, L' (line 10). This process is iterated until the remaining base cluster list, L, is empty. Next, all flexible clusters are extracted from C into a list F (line 11). Then, base clusters are combined with flexible clusters (line 12). Algorithm 3 (see Fig. 10) shows the pseudo code of the function that updates flexible clusters. In Algorithm 3, for all flexible clusters (F_i), a list (Z) is compiled with the corresponding new elementary base cluster (s) and the base clusters that F_i is flexible. Then, if Z

© The Institution of Engineering and Technology 2020

17519578, 2020, 13, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/iet-its.2020.0437 by UNIVERSITY OF MORATUWA, Wiley Online Library on [08/01/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

Input: List of flexible clusters (F), List of base clusters that were combined with other base clusters (G)
Output: List of updated flexible clusters (F')

1: **Execution:**
2: $F'.clear()$
3: $Z \leftarrow$ empty list of clusters
4: **for** $F_i \in F$ **do**
5: $s = F_i.getSiblingBaseCluster()$
6: **if** $s \neq null$ **then**
7: **if** $s \in G$ **then**
8: $F_i.setSiblingBaseCluster(null)$
9: **else**
10: $Z.add(s)$
11: $Y = F_i.getFlexibleNodesList()$
12: **for** $Y_i \in Y$ **do**
13: $N = Y_i.getClusters()$
14: **for** $N_i \in N$ **do**
15: $Z.add(N_i)$
16: **if** $Z.size() == 1$ **then**
17: combine F_i with the only base cluster
18: **else**
19: $F_i.setFlexibleToList(Z)$
20: $F'.add(F_i)$
21: **return** F'

Fig. 10 *Algorithm 3 Pseudo code:* updateFlexibleClusters(F, G)

Input: List of clusters, vehicle locations, passenger locations
Output: List of vehicles and passengers for each cluster

1: **Execution:**
2: $vehiclePool \leftarrow$ empty list of vehicles
3: Extract excess vehicles from each base cluster and put into $vehiclePool$
4: Extract all the vehicles from each flexible cluster and put into the $vehiclePool$
5: Allocate vehicles in the $vehiclePool$ to clusters that require additional vehicles
6: Allocate passengers in flexible clusters into possible base clusters

Fig. 11 *Algorithm 4 Fleet allocation*

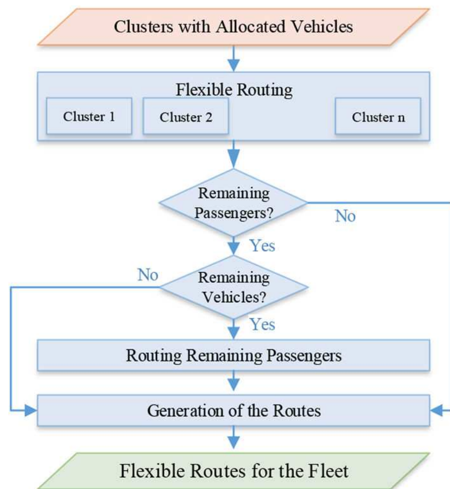

Fig. 12 *Flexible routing technique*

contains only one element, F_i is combined with this base cluster and the new list (F') is updated. Algorithm 3 returns the updated list of flexible clusters F'. Finally, the algorithm outputs the list of updated clusters, C', by adding elements in L' and F' (line 13).

3.2 Fleet allocation

After generating the list of base and flexible clusters, the next step in the method is to allocate vehicles to each cluster based on the demand of base clusters while leveraging the demand of flexible clusters. The objective of the fleet allocation algorithm is to minimise vehicle usage by optimally allocating vehicles to clusters. Since vehicles are assumed to be distributed in car parks near the bus stops, each cluster will contain a specific number of passengers and vehicles. Therefore, some clusters contain enough vehicles to satisfy the passenger demand while some clusters require additional vehicles. Thus, the fleet allocation algorithm has to optimally manage the supply and demand parameters. For example, clusters with excess vehicles can assign the excess to other clusters which require vehicles. However, the main problem that needs to be addressed in the allocation is to determine the vehicle/s that must be retained to satisfy the demand in the cluster. Also, the fleet allocation algorithm can leverage the demand in flexible clusters which can be picked-up by vehicles in multiple base clusters. This will significantly affect the quality of the results. The fleet allocation algorithm is given in Algorithm 4 (see Fig. 11).

Initially, the number of vehicles required for a base cluster is calculated based on the demand. Then, excess vehicles in each base cluster are added to a list of vehicles, *vehiclePool*. When selecting the excess vehicles, the algorithm uses the polyline sequence of each location to determine the proximity of the vehicles to the destination. Then, the vehicles farthest away from the destination in each cluster are allocated to pick-up the passengers in the respective cluster. The motivation of selecting the vehicles farthest away is that it will be able to pick-up a higher number of passengers from other flexible clusters along the way to the destination. Next, all vehicles in flexible clusters are also added to the *vehiclePool*. Here, the demand in flexible clusters is not considered since it is preferable to satisfy the demand from vehicles of the corresponding base clusters. Thereafter, the algorithm identifies the base clusters which require additional vehicles and allocates from the *vehiclePool*. Here, the actual road network distance to the median node of the cluster from each vehicle is used for allocation. Therefore, vehicles closest to the median node are selected. Next, the algorithm computes the excess number of seats in a base cluster by subtracting the *no of passengers* in a base cluster from the total number of seats available in vehicles (*total number of seats* = *no of vehicles* × *vehicle capacity*). Finally, passengers in flexible clusters are assigned to the relevant base clusters with excess capacity as computed above. Here, if the algorithm must select between multiple base clusters, priority is given to the base cluster with the highest excess capacity. This facilitates to reduce the probability of filling up the remaining base clusters as there could be other flexible clusters that needs more seats. On the other hand, if a few passengers have to be allocated to the remaining seats, a separate selection mechanism is not considered since selecting any passenger in a flexible cluster has the same impact.

However, at the end of this step some passengers may not be allocated to a cluster. Consider the following two scenarios: (i) lack of vehicles to satisfy demand; and (ii) base clusters that can accommodate flexible clusters are already filled up. In case 1, since the demand is higher than the supply, excess passengers are indicated as skipped passengers. In case 2, there can be vehicles in the *vehiclePool*, which can be allocated to pick-up the passengers. However, a separate cluster is created with all the unallocated passengers and remaining vehicles (henceforth referred as unallocated cluster). The routing algorithm for this cluster is executed as given in Section 3.3.

3.3 Flexible routing

The technique for flexible route generation is given in Fig. 12. The input contains clusters with a set of vehicles and passengers that need to be routed such that all constraints are satisfied while minimising the VMT. To this end, we use the GA proposed in our previous work [5]. After executing the algorithm in parallel for each cluster, passengers that are not picked-up due to the violation of a constraint is added to the unallocated cluster created in Section 3.2. Next, the unallocated cluster is executed using the GA. However, it should be noted that this step can only be executed if excess vehicles are available in the *vehiclePool*. If this is not the

© The Institution of Engineering and Technology 2020

17519578, 2020, 13, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/iet-its.2020.0437 by UNIVERSITY OF MORATUWA, Wiley Online Library on [08/01/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

case, passengers in the unallocated cluster are assigned as skipped passengers. Finally, the flexible routes that consists of the routes of each vehicle is generated.

4 Results

This section evaluates the performance of the proposed method. Section 4.1 explains the selected geographical area and Section 4.2 gives the experimental parameters. Section 4.3 presents the comparison strategy. Finally, Section 4.4 evaluates the suitability of the proposed methods for route generation in ODPT systems. The method in Section 3 uses JAVA, while the rest of the code is implemented in C++. The computation time is measured on a PC with 32 GB RAM, running Windows 10 Pro on an Intel Xeon E5-1630V4 CPU at 3.70 GHz.

4.1 Geographical area

We have selected a locality in western Singapore to conduct the experiments. The selected area encompasses the largest university in Singapore and the adjoining residential neighbourhood. The nearest metro station is selected as the common destination. The university has a large population and currently operates a fixed-route shuttle to the nearest metro station. Therefore, the students and staff of the university can further benefit from the ODPT system. Selecting the adjoining residential neighbourhood not only increases the complexity of the road network but also provides an opportunity to validate the proposed methods over a large area. Fig. 13 shows the selected locality, bus stops in the area (blue dot) and the metro station (blue flag).

4.2 Experimental parameters

Here, the parameters used in test cases (TCs) and the GA are presented. TC parameters in Table 9 give the number of passengers, number of available EVs in the fleet, EV capacity and driving range. The passenger count in the TCs are selected based on the observed demand on fixed-route transit using historical

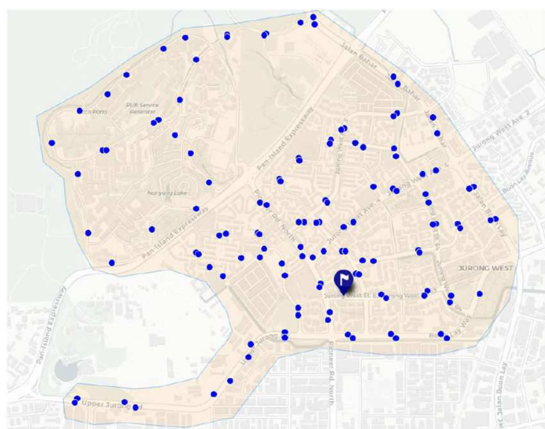

Fig. 13 *Selected locality, bus stops and metro station*

demand data. In each TC, passengers and EV are randomly distributed in the bus stops shown in Fig. 13. A medium size EV capacity and driving range is selected such that the operator can use super charging facilities which would eliminate the additional complexities of EV downtime. However, it should be noted that the GA generates routes such that the driving range is always sufficient to traverse the entire path to pick-up passengers and reach the destination. Alternatively, this implies that EV charging can only occur while the vehicle is empty. In TCs 1–3, the supply exceeds the demand as opposed to TCs 4–6 where the supply is marginally sufficient to satisfy the demand. These TCs are henceforth referred as 'relaxed test cases' and 'tight test cases' respectively. Tight TCs have a higher possibility of skipping passengers. They have been selected to show the robustness of the proposed methods for different scenarios. Table 10 gives the parameters of the GA. GA parameters specify the maximum number of iterations and the population size for both the proposed method and [5] used for comparison.

4.3 Comparison strategy

4.3.1 Case 1: The performance of the proposed method in terms of VMT of the fleet and computation time of routes is assessed by comparing against [5]. In the experiments, each relaxed TC in Table 9 is repeated three times while randomly distributing passengers and EVs in the selected area. The parameters given in Table 10 are used in the respective methods. The lower number of iterations and population size of the proposed method compared to [5] enables to validate the rapidity.

4.3.2 Case 2: The benefit of the proposed directionality and shortest-path centric clustering algorithm is compared with widely-used clustering algorithms, namely *k*-means and *k*-medoids clustering, and a static clustering technique based on dividing the locality using main road segments similar to Quadrifoglio *et al.* [4] and Shen and Quadrifoglio [3]. Here, we compare the VMT of the fleet for the four methods. We use the same vehicle allocation algorithm and the GA in each method in order to ensure a fair comparison. In this case, one instance of each relaxed TC is used for evaluation. However, a common drawback of *k*-means and *k*-medoids is the requirement to specify *k* (the number of clusters). Thus, we compare the VMT for values of 5, 10 and 20 of *k*. For the static clustering technique the selected locality is divided into three clusters based on the main road network. The three static clusters are shown in Fig. 14.

4.3.3 Case 3: In this case, the robustness of the proposed method in handling tight demands is evaluated. For the tight TCs in Table 9, the supply is marginally sufficient to satisfy the demand. Therefore, passengers maybe skipped during the routing phase as described in Section 3.3. Therefore, in this case, the number of skipped passengers, VMT of the fleet and computation time of routes are compared with [5]. One instance of each tight TC is used for evaluation.

Table 9 Parameters of TCs

Parameter	TC number					
	1	2	3	4	5	6
number of passengers	200	250	300	120	200	320
number of EVs	40	50	60	15	25	40
EV capacity	8					
EV driving range	30 km					

Table 10 GA parameters

Parameter	Ref. [5]	Proposed
iterations	100	20
population size	50	10

© The Institution of Engineering and Technology 2020

17519578, 2020, 13, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/iet-its.2020.0437 by UNIVERSITY OF MORATUWA, Wiley Online Library on [08/01/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

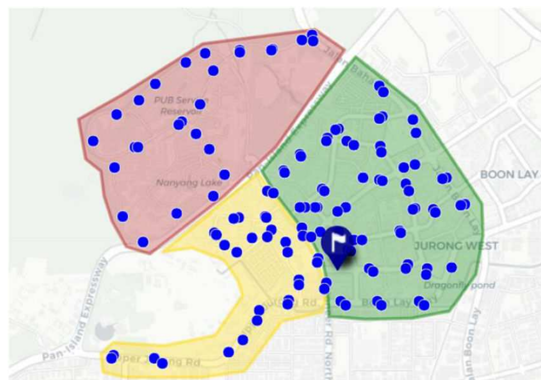

Fig. 14 *Generated static clusters based on main road segments*

Table 11 VMT and computation time comparison

Test case	Ins.	VMT, km			Computation time, s		
		Ref. [5]	Proposed	PI, %	Ref. [5]	Proposed	PI, %
1	1	125.8	117.8	6	864.75	42.51	95
	2	134.6	119.9	11	817.23	44.56	95
	3	136.1	119.9	12	813.87	41.31	95
2	1	139.5	131.7	6	1723.79	77.11	96
	2	146.7	135.9	7	1736.91	82.73	95
	3	142.5	136.0	5	1608.72	74.13	95
3	1	146.7	135.0	8	2810.05	120.95	96
	2	168.7	157.3	7	2744.12	122.68	96
	3	154.4	152.7	1	2733.24	114.89	96
average				7			95.4

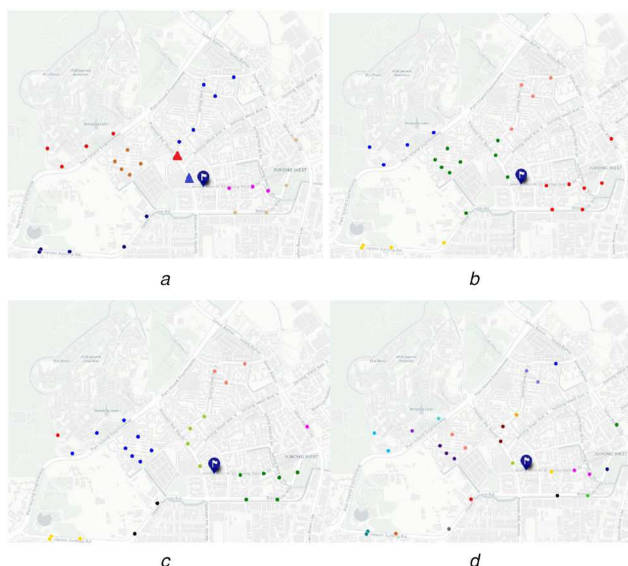

Fig. 15 *Generated clusters for proposed and K-medoids algorithms*
(a) Proposed, ***(b)*** $k = 5$, ***(c)*** $k = 10$, ***(d)*** $k = 20$

4.4 Evaluation

4.4.1 Case 1: Table 11 shows the VMT of the fleet and computation time of routes of [5] and the proposed method. It also shows the percentage improvement (PI) of the results. Here, three instances (Ins.) of the relaxed TCs are executed to show the consistency of the proposed method. The results confirm that the proposed method not only rapidly generates routes with a 95.4% average improvement in computation time but also produces superior routes with a 7% average improvement in VMT. The significant improvement of computation time can be accounted for the parallel implementation of the routing algorithm and the reduced number of population and iterations in the GA. The improvement of the VMT of the fleet can be accounted to the directionality-centric segmentation of bus transit network. For example, the clusters formed by the proposed algorithm ensures that the detours in the vehicle routes are significantly low. Therefore, the GA can generate superior routes.

4.4.2 Case 2: This case compares the VMT of the fleet of *k*-means, *k*-medoids, static and the proposed clustering algorithm. The VMT of the fleet will depend on the clusters created by the algorithm. Fig. 15 shows a sample set of clusters generated using the proposed clustering algorithm and the corresponding clusters of *k*-medoids for $k = 5$, 10 and 20. In each sub-figure of Fig. 15, the coloured circles show the clustered bus stops while the triangle in Fig. 15*a* shows bus stops in flexible clusters. The diagrams show that the proposed algorithm considers the directionality to the destination when generating clusters, while the *k*-medoids algorithm generates clusters based on proximity of locations. This would enable the GA to generate superior routes. This is evident in the results shown in Fig. 16. Figs. 16*a–c* show the VMT of the fleet for the relaxed TCs. The vertical axis in each graph gives the VMT while the horizontal axis shows the value of *k*. The two columns represent *k*-means and *k*-medoids algorithms, respectively. The results of the proposed and static clustering algorithms are shown in an orange solid and yellow dashed line, respectively. It is observed that in all instances, the proposed algorithm generates superior routes with low VMT compared to *k*-medoids and static clustering. However, in the case of *k*-means, in two instances the VMT of the proposed algorithm is marginally higher (the VMT obtained using *k*-means with $k = 5$ for TCs 1 and 2 is 0.25% and 0.95% higher, respectively). However, finding the best value for *k* is practically infeasible. Therefore, *k*-means clustering algorithm cannot be relied upon to produce consistently superior routes. On average, the proposed algorithm generates clusters that result in a 6% reduction in VMT compared to the *k*-means and *k*-medoids algorithms and a 5% reduction compared to static clustering.

4.4.3 Case 3: For the tight TCs, we compare the number of skipped passengers, VMT of the fleet and computation time of routes of the proposed method and [5]. The results are shown in Fig. 17*a–c*, respectively. The horizontal axis in each graph gives the TC number and the vertical axis gives the number of skipped

© The Institution of Engineering and Technology 2020

17519578, 2020, 13, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/iet-its.2020.0437 by UNIVERSITY OF MORATUWA, Wiley Online Library on [08/01/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

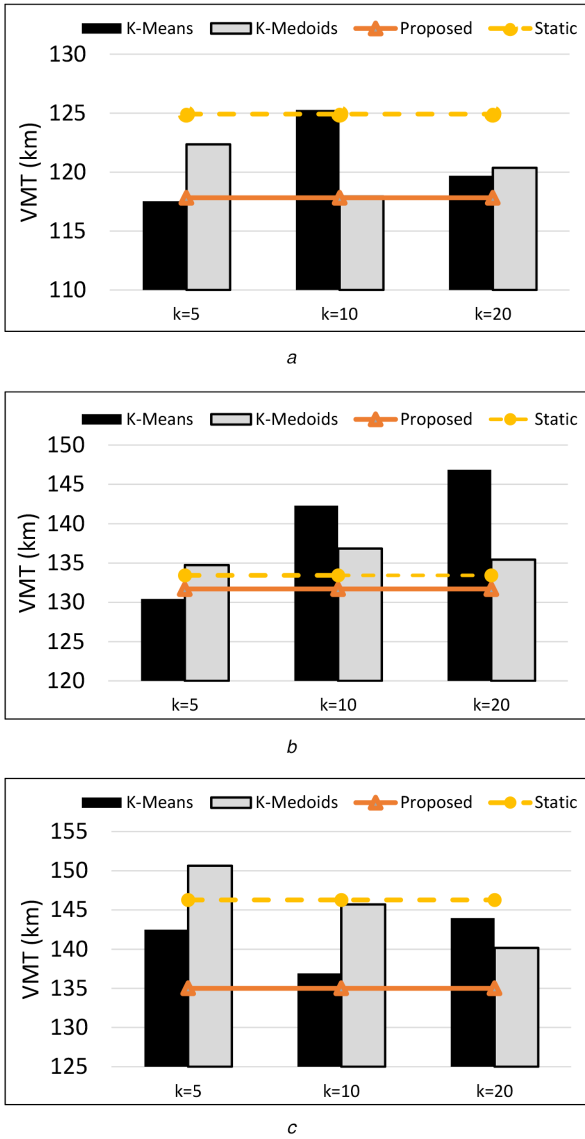

Fig. 16 *Comparing clustering algorithms*
(a) Test case 1, ***(b)*** Test case 2, ***(c)*** Test case 3

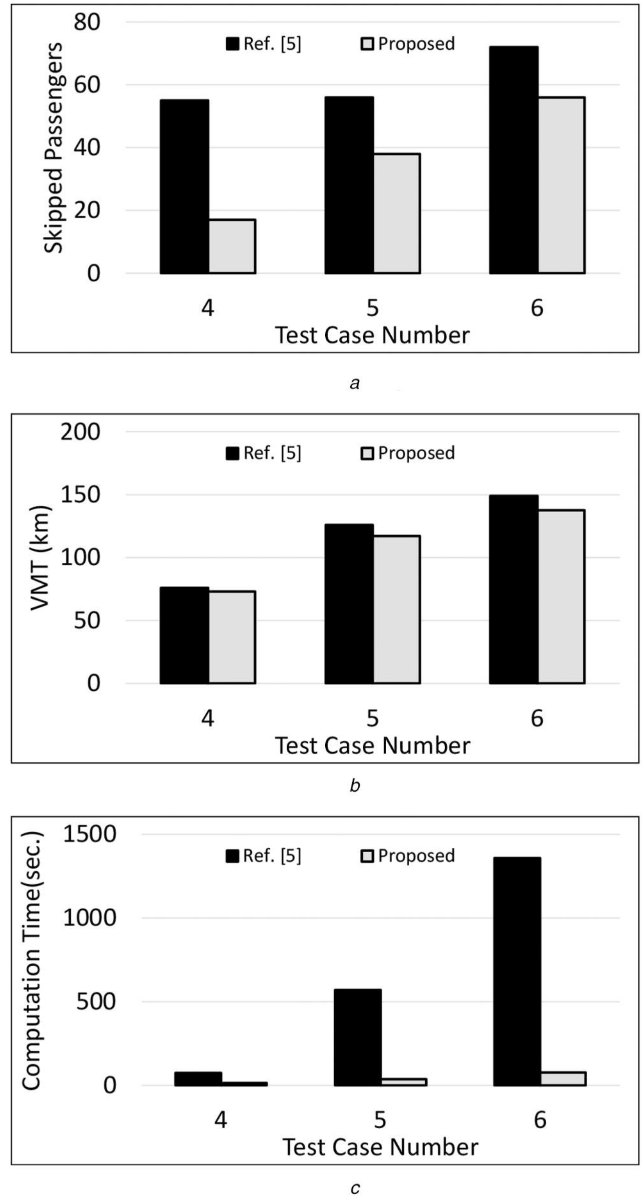

Fig. 17 *Comparison of results for tight TCs*
(a) Skipped passengers, ***(b)*** VMT, ***(c)*** Computation time

passengers, VMT and computation time, respectively. Results show that the proposed method on average has reduced the number of skipped passengers by 16.25%. In addition, the method generates routes with 8.25% less VMT with an 89.5% improvement in computation time. It should be noted that this improvement is achieved with additional passengers transported to the destination. The number of skipped passengers is reduced due to the superior clusters, which makes the solution space for exploration small compared to an un-clustered approach. For example, the population size of the GA will limit the number of solutions that can be present in each generation. However, in tight TCs, the constraints will create a significant number of low-quality solutions. Since [5] is unable to search the full design space, it is forced to find a few low-quality solutions. In contrast, due to the superior clusters that are generated in the proposed method, the number of low-quality solutions in the design space is significantly reduced. Therefore, the proposed method can generate superior routes. The results confirm the robustness of the proposed method to be deployed for route generation in ODPT systems.

5 Conclusion & future directions

This work presented a directionality-centric, bus transit network segmentation method to speed-up route computation for an ODPT system. The proposed methodology groups all the bus stops in a geographical area into clusters by leveraging the similarity of the shortest-path routes to the destination. The proposed method also takes into account the demand and flexible clusters for intelligent fleet allocation. The flexible routes are generated by leveraging a GA in parallel for each cluster. The proposed method not only generates results rapidly but also with minimum degradation of solution quality.

The method has been evaluated using TCs generated on a real road network encompassing the largest university in Singapore. Openly available public transit demand data has been used to quantify demand and supply. Experimental evaluations confirm that a notable improvement in computation time compared to an un-clustered method is achieved while also improving the route quality. Experiments also show that the proposed directionality and shortest-path centric clustering method outperforms two widely-used unsupervised learning algorithms and a method that uses static boundaries to segment the road network. The robustness of the methodology has also been demonstrated for scenarios where the supply is marginally sufficient to meet the demand, which can be useful for peak periods. This affirms the suitability of the proposed method to generate rapid and robust routes for real-time deployment of ODPT system.

Furthermore, the methods proposed in this work can be used to generate superior routes for a fleet of vehicles significantly faster. However, the method relies on a homogeneous fleet of vehicles with similar capacities and driving ranges. Therefore, in the future, we plan modify the algorithms to incorporate heterogeneous fleets.

© The Institution of Engineering and Technology 2020

17519578, 2020, 13, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/iet-its.2020.0437 by UNIVERSITY OF MORATUWA, Wiley Online Library on [08/01/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

Also, the proposed methods are based on all passengers traveling to a common destination. However, passengers may travel to multiple destinations. Although, multiple destinations can be dealt as separate instances of the proposed method, it can be possible to derive more robust algorithms by identifying sub-trees in the generated polyline tree. Therefore, we plan to adapt the methods proposed in this work for scenarios involving multiple destinations.

6 Acknowledgments

This research project is partially funded by the National Research Foundation, Singapore under its Campus for Research Excellence and Technological Enterprise (CREATE) programme with the Technical University of Munich at TUMCREATE.

7 References

[1] Bhutani, A., Saha, P.: 'Global mobility on-demand market worth $250bn by 2026'. Accessed June 2020. Available at https://bit.ly/2CIHVmZ

[2] Alonso.Mora, J., Samaranayake, S., Wallar, A., *et al.*: 'On-demand high-capacity ride-sharing via dynamic trip-vehicle assignment', *Proc. Natl. Acad. Sci.*, 2017, **114**, (3), pp. 462–467

[3] Shen, C.W., Quadrifoglio, L.: 'Evaluating centralized versus decentralized zoning strategies for metropolitan ADA paratransit services', *J. Transp. Eng.*, 2013, **139**, pp. 524–532

[4] Quadrifoglio, L., Dessouky, M., Ordóñez, F.: 'A simulation study of demand responsive transit system design', *Transp. Res. A, Policy Pract.*, 2008, **42**, pp. 718–737

[5] Perera, T., Prakash, A., Gamage, C.N., *et al.*: 'Hybrid genetic algorithm for an on-demand first mile transit system using electric vehicles', In Shi, Y., Fu, H., Tian, Y., *et al.*, (Eds.): '*Computational science – ICCS 2018*' (Springer International Publishing, Cham), 2018, pp. 98–113

[6] Kaufman, L., Rousseeuw, P.: '*Finding groups in data: an Introduction to cluster analysis*' (John Wiley & Sons, Inc., USA 1990)

[7] Ioachim, I., Desrosiers, J., Dumas, Y., *et al.*: 'A request clustering algorithm in door-to-door transportation', *Transp. Sci.*, 1995, **29**, pp. 63–78

[8] Jorgensen, R., Larsen, J., Bergvinsdottir, K.: 'Solving the dial-a-ride problem using genetic algorithms', *J. Oper. Res. Soc.*, 2007, **58**, pp. 1321–1331

[9] Saez, D., Cortés, C., Núñez, A.: 'Hybrid adaptive predictive control for the multivehicle dynamic pick-up and delivery problem based on genetic algorithms and fuzzy clustering', *Comput. Oper. Res.*, 2008, **35**, pp. 3412–3438

[10] Pelzer, D., Xiao, J., Zehe, D., *et al.*: 'A partition-based match making algorithm for dynamic ridesharing', *IEEE Trans. Intell. Transp. Syst.*, 2015, **16**, pp. 1–12

[11] Zheng, H., Chen, J., Zhang, X., *et al.*: 'Designing a new shuttle service to meet large-scale instantaneous peak demands for passenger transportation in a metropolitan context: a green, low-cost mass transport option', *Sustainability*, 2019, **11**, (18), https://www.mdpi.com/about/announcements/784

[12] Chen, S., Wang, H., Meng, Q.: 'Solving the first-mile ridesharing problem using autonomous vehicles', *Comput.-Aided Civ. Infrastruct. Eng.*, 2020, **35**, (1), pp. 45–60

[13] Comert, S.E., Yazgan, H.R., Kır, S., *et al.*: 'A cluster first-route second approach for a capacitated vehicle routing problem: a case study', *Int. J. Procurement Manage. (IJPM)*, 2018, **11**, pp. 399–419

[14] Lowalekar, M., Varakantham, P., Jaillet, P.: 'ZAC: a zone path construction approach for effective real-time ridesharing'. Proc. of the Twenty-Ninth Int. Conf. on Automated Planning and Scheduling, Menlo Park, California, 2019, pp. 98–113

[15] Open Source Routing Machine. 'OSRM API Documentation',accessed April 12 2020. Available at https://bit.ly/2XCChvM

[16] Google Maps Platform. 'Encoded polyline algorithm format',accessed September 17 2020. Available at https://bit.ly/2FAqMxC

© The Institution of Engineering and Technology 2020

17519578, 2020, 13, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/iet-its.2020.0437 by UNIVERSITY OF MORATUWA, Wiley Online Library on [08/01/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License