HIGH SCHOOL EDITION
Journal of Student Research
Volume 11 Issue 4 (2022)

Learning to Branch-and-Bound to Route an Autonomous Mobility on Demand System

Claire Xu[1], Robin Brown[2#], Somrita Banerjee[2#] and Marco Pavone[2#]

[1]Foothill High School, USA
[2]Stanford University, USA
[#]Advisor

ABSTRACT

Autonomous Mobility on Demand (AMoD) is a system consisting of a fleet of centrally-controlled, autonomous vehicles that take customers from their requested origins to their requested destinations. In order to minimize the distance traveled by the fleet, a routing scheme must be developed to service all customer requests. This paper investigates the usage of the branch-and-bound algorithm (BB) to find such a scheme, as well as the usage of a neural network (NN) to speed up BB. Given a fixed road network, sets of randomly generated requests were passed into BB to obtain the ordering of each set that minimized the number of computations, and a NN was trained on this data. New randomly generated request sets were then passed into the trained NN, and runtimes given the NN-predicted orderings were compared to average runtimes over all permutations. For a NN with 1024 nodes in the first and 512 in the second hidden layer and a learning rate of 0.1, using the NN resulted in an average 40% decrease from average runtimes; furthermore, NN-orderings never resulted in increased runtime. Other combinations of NN parameters resulted in around 25% decrease in runtime. Also, BB performed the same number of computations for all permutations with the same first request, simplifying the problem to only finding the *next* request. These results show that training the NN results in a more efficient, faster routing algorithm that is therefore easier to scale up, enabling at-scale adoption of AMoD.

Introduction

Background

In recent years, traffic and pollution have become increasingly problematic with the rise of private vehicles and uncoordinated transportation. A potential solution, especially in large urban areas, is Autonomous Mobility on Demand (AMoD), a system consisting of a fleet of electric, autonomous vehicles that are centrally controlled to service customers, similar to a taxi or Uber-like system [1]. A visualization and brief description is provided in Figure 1.

AMoD has many social and environmental benefits, including a) reducing carbon emissions, b) providing cheap and reliable transportation, c) optimizing travel time/power consumption for everyone in the area, and d) eliminating unpredictable human driving. However, one critical issue is how the system decides what routes its vehicles should take to bring customers from their requested origins to their requested destinations—given that each road has a certain capacity constraint. Because cars cannot be split, the solution to this optimization problem must be in integers, as opposed to real numbers, i.e. mixed integer programming. One way to solve this is to use the branch-and-bound algorithm (BB) [2].

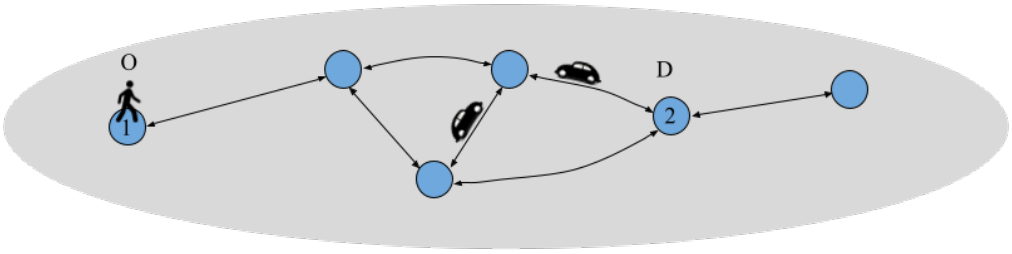


Figure 1. Autonomous Mobility on Demand. The blue nodes are places where vehicles can stop and customers can enter or leave the system. The nodes are connected by edges, representing the roads. Here, a customer enters at node 1, the *origin*, and requests a vehicle to get to node 2, the *destination*. The request is denoted (1, 2).

BB is a method used in discrete and combinatorial optimization. A tree structure is created, where each branch represents a subset of the entire solution set. The algorithm systematically enumerates the possible solutions and searches through them to find the one that is most optimal. Before computing on a certain branch, the upper and lower bounds are approximated and compared to the current optimum; if it is certain that this branch cannot produce a more optimal solution, the algorithm cuts it out of future consideration. If bounds cannot be computed or do not offer useful insight, BB will end up performing an exhaustive search. As such, finding a good initial ordering of searching can significantly reduce both the number of potential branches evaluated and overall computation time.

There are currently no universal methods to order the branches. Instead of finding the ordering specific to each context, a neural network can be trained to predict the ordering that would maximize BB's efficiency. This problem can be roughly split into two parts: "diving" and "searching" [3]. "Diving" refers to determining whether a given branch yields a feasible solution, while "searching" refers to determining the branch that will minimize the number of computations the algorithm has to perform. For the purposes of this paper, we will focus on the "search" side, and we will train the neural network to search specifically on BB data in an AMoD context.

In this paper, we show that BB alone can be used to route requests. We then show that a NN can successfully train on data generated from BB. Finally, we use the trained NN to significantly reduce BB computation time, i.e. we assume a fixed road network and show that the NN can output, from the randomly generated requests, which one should be searched through before the others. These results can be implemented in future AMoD systems as a lower-cost, power and time-efficient method that can easily be scaled up to service larger areas with larger populations.

Related Work

Several methods have previously been studied for modeling AMoD, such as a queuing-theoretical model [4]-[5], a simulation model [6]-[7], and a discrete-time model [8]-[9]. The queuing-theoretical model, as proposed in [4], as well as the discrete-time model, as proposed in [9], both account for time; that is, vehicles do not travel instantaneously, and customers continuously enter and leave the system. In addition, the models consider rebalancing—or how vehicles move from servicing one customer to the next—as well as routing. This is useful for controlling the system in the long-term, for example when coupled with real-time customer demand prediction [10]. However, since we are concerned with routing vehicles at one given moment in time, we use a network flow model, similar to [11]-[14]. We adopt the thresholded approximation of congestion used in [12], which assumes that no more vehicles can travel on any road once the limit is reached.

Less work has been done regarding the fact that any solution to the AMoD routing problem must be in integers. Many of the aforementioned papers either do not incorporate road capacity constraints into their models, or do not address the integer constraint. This paper investigates a routing scheme that solves this issue by using the branch-and-bound algorithm. We also investigate using machine learning to improve the algorithm's efficiency; this is known as "learning to branch-and-bound" and is a powerful tool in mixed integer programming [15]. In particular, [3] and [16] show that machine learning can be used to pick variables to simplify the algorithm ("searching"). In this paper, we identify and adapt this tool "learning to branch-and-bound" as a means to accelerate AMoD algorithms, furthering the state-of-the-art and addressing a key limitation (computation speed) of the current literature.

Methods

AMoD Problem Formulation

Mathematical Representation

We use a network-flow model, similar to the mathematical representation given in [12]. Define a road network R, consisting of a set of nodes R_n, where each node is a potential origin or destination, and a set of edges, R_e, which represent the roads that a vehicle can travel over. Here, the physical length of the edges does not matter. We represent edges as

$$[i, j]$$

for some
$i, j \in R_n$.
Every edge also has its own capacity, or a maximum limit on the flow, which we call
$C_m[i, j]$.
The *flow* over an edge represents the number of vehicles traveling along that road. Denote the flow as
$f_m[i, j]$.

Finally, we define the cost function, J, which we seek to minimize. Since we want to minimize the total distance traveled by all the vehicles, we let J be the sum of flows over all edges. Vehicles enter the graph at the nodes corresponding to the request origins and exit the graph at nodes corresponding to the request destinations. Thus, the problem can be stated as follows:

$$\min\ J(f_m[i,j]) = \min\ \sum_{i,j \in R_n} f_m[i,j] \tag{1}$$

s.t.

$$\sum_{i:(i,j)\in R_e} f_m[i,j] + \mathbf{1}\text{:(j is origin)} = \sum_{k:(j,k)\in R_e} f_m[j,k] + \mathbf{1}\text{:(j is destination)} \tag{2}$$

s.t.

$$0 \le f_m[i,j] \le C_m[i,j],\ f_m[i,j] \in \mathbb{Z} \tag{3}$$

Equation (2) contains the boolean indicator **1:** and ensures that the mass of vehicles is conserved, as vehicles cannot instantaneously leave one node and appear at a separate one. Equation (3) ensures nonnegative, integer flows within capacity limits

AMoD Elements

The road network R is constant for all experimentation. A visual representation is given in Figure 2. Capacities are chosen arbitrarily, then experimentally decreased until around 1% of routing problems are infeasible. This is to ensure that routing is non-trivial, otherwise further measures would not be necessary (BB, machine learning, etc).

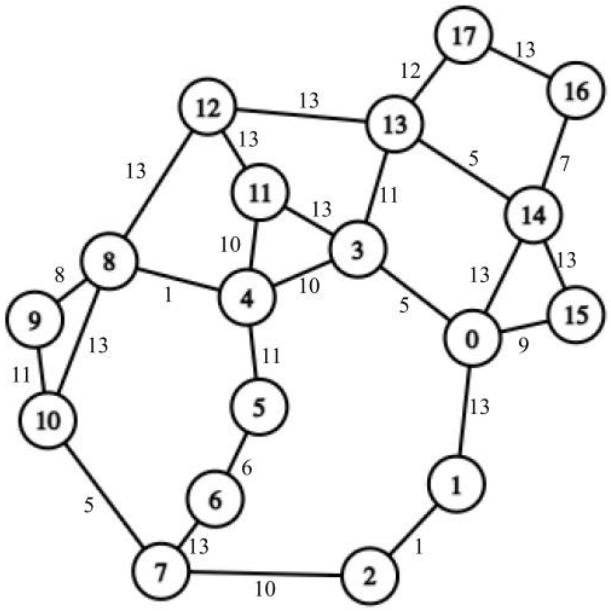


Figure 2. The road network, R. Each node is a possible origin or destination, with 18 in total (0-index). Capacities on each edge are labeled. Requests would be encoded as (origin node, destination node), for example (4, 17).

Requests are in the form

$$(o, d)$$

where $o, d \in R_n$, o is the origin node, and d is the destination node. For example, a customer who wants to enter R at node 4 and leave at node 17 would request (4, 17). An origin node cannot be the same as its corresponding destination node. Requests (a, b) and (b, a) are considered the same because in our static-time graph, the flows for (a, b) and (b, a) use the same edges, resulting in identical problems. For a graph with k nodes, there are a total of (k choose 2) total possible requests. In our case, there are (18 choose 2) total requests. Call the set of all requests Q.

One thing to note here is that the model assumes that all customers will be serviced at the same time, and that a vehicle travels along its designated route instantaneously such that it is occupying all edges in its route at the same moment in time. Thus, given a subset of Q, a routing scheme is said to be feasible if a path from the origin node to the destination node can be found for all (o, d) pairs in the subset, such that for each edge, the number of paths that traverse it does not exceed its capacity.

Routing AMoD Using BB

Given a subset, S, of Q, we can solve the problem for S using the branch-and-bound method. In order to achieve this, we adapt the algorithm to the routing context. Specifically, the branches, or possible solution subsets, are generated by the possible routes for a given request in S. The objective value of each branch is the sum of the flows achieved by that possible combination of routes, which is the value we seek to minimize. Figure 3 lays out the general procedure on $S = \{A, B, C, D, \dots\}$:

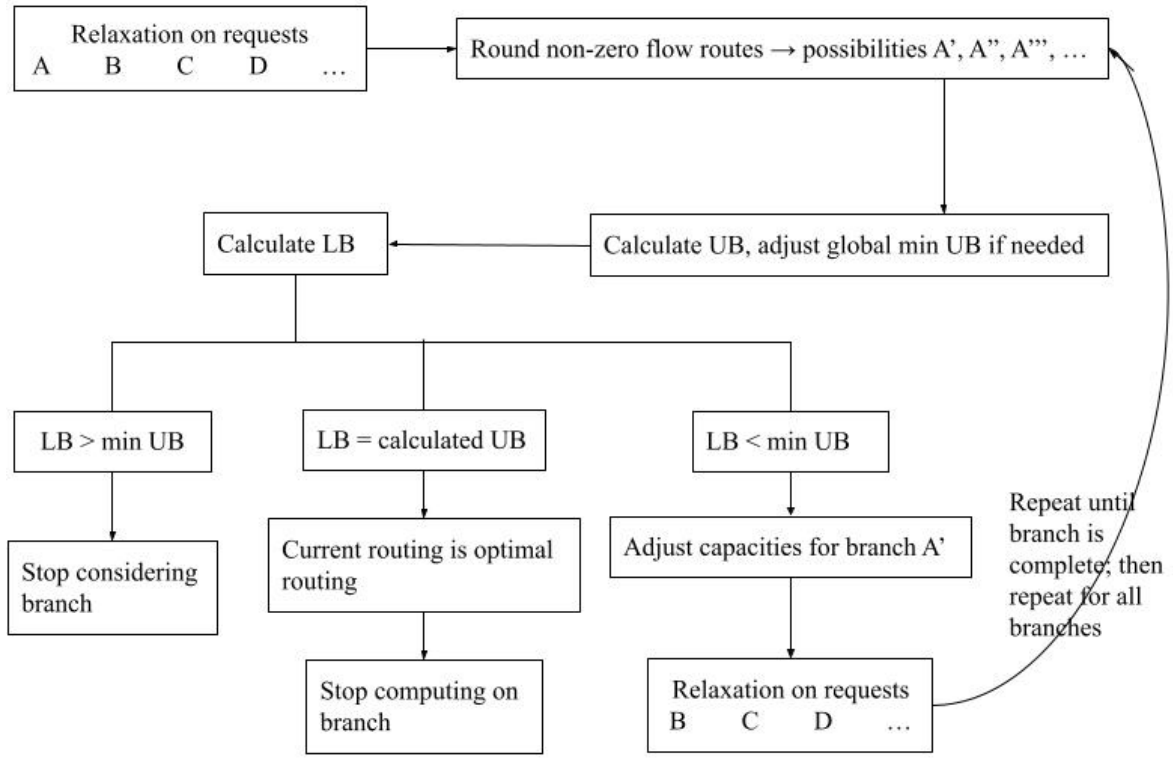


Figure 3. Using BB to route an AMoD problem. The flow chart shows how the concept of BB is adapted to an AMoD context and the steps of the algorithm.

Relaxation on requests

The problem described in the "Mathematical Representation" section can easily be solved in CVXPY, when the solution set isn't restricted to integers (the "relaxed" problem) [17]. As shown in Figure 3, in the first step, the optimal solution is computed on all the requests, which may yield non-integer flows and more than one path for each request.

Rounding non-zero flows

Take a request and call it $A = (o_a, d_a)$. Starting from o_a, call the current node p. The next node, q, in the path is chosen such that the flow $f_m[i = p, q]$ is maximized. Now set p to be the node just chosen, and the process is repeated until q is the destination node. Once the path has been determined, the flows over all edges in the path are

changed to 1. In the case of building the BB tree, for *any* path from o_a to d_a where every edge within the path has a nonzero flow, we change the flows of all edges in that path to 1. In the case of building the BB tree, each one of these "rounded" paths forms a "branch," or a subset of all possible solutions.

Calculating Upper Bound (UB)

The upper bound of a branch is the value of the cost function J for any known routing scheme. If none are known, set this value to be infinity. If one is found, the optimal solution on that branch either has the same or lower J value since the known routing scheme either happens to be the optimal solution or is less optimal. At the same time, the lowest UB over the entire tree will be continuously updated—if the most recent UB calculated is less than the current global minimum UB, then update the global value.

The method we used to find a known routing scheme is shown in Figure 4. For some set of requests, the relaxed problem is solved. The path of the first request is then determined by rounding the flows to 1. Adjust capacities based on the edges traversed by the path, then repeat with the rest of the requests: solve the relaxed problem on the current capacities and remaining requests, round the path of the next request in the list, and adjust capacities based on the rounded path.

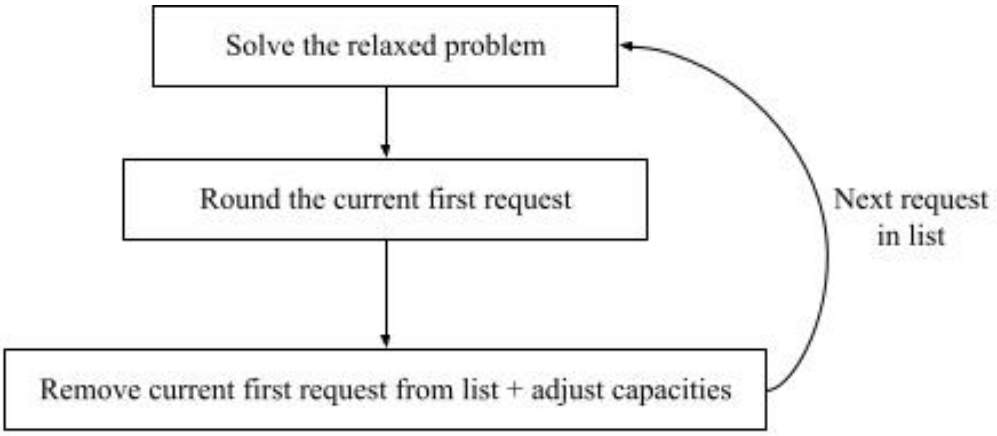


Figure 4. Method for finding one possible solution. For each request, in order, the relaxed routing problem is solved, the request path is rounded, and the capacities are adjusted to reflect the chosen path. This process is then repeated for the remaining requests.

If, at some point, it's impossible to solve the relaxed problem, then we set the UB to infinity to signify that no feasible routing scheme has been found yet.

Calculating and Comparing Lower Bound (LB)

The lower bound of a branch is the lowest possible value of J, which is to say the most optimal solution. In fact, we can say that the LB is simply the value of J for the relaxed problem. This is because the problem incorporating integer constraints satisfies all the criteria for the relaxed problem, so if it has some optimal solution, then that optimal solution must also be a solution to the relaxed problem.

If the LB is greater than the global minimum UB, that means that whatever solution this branch ultimately yields cannot be more optimal than that of the branch with the smallest current UB. Thus BB does not have to keep computing to find a less optimal solution. If the LB is equal to the UB calculated for this branch, then the optimal solution for this branch is found: the known routing scheme is the most optimal, so no further calculations need to be performed. If the LB is less than the global minimum UB, nothing can definitely be said, so calculations must continue on that branch.

Finding the overall solution

If calculations must continue, adjust the road network's capacities to reflect that the first request has already been routed by subtracting one from the capacity of each edge the first request's path passes through. Then remove the already routed request from the set of requests to be routed. Note that there is a separate copy of the road network's capacities and the original requests that is updated for each branch. With the modified capacities and requests, repeat the process—solve the relaxed problem for the remaining requests, round the paths of the current first request in the list to create sub-branches, calculate and compare UB and LB, then eliminate or continue computing, adjusting capacities and requests when necessary—until the solution is found or BB is done computing upon all branches.

If a problem is feasible and the right branch comes first, BB could immediately find the smallest possible global minimum UB. That UB would be the value of the solution, so for all subsequent branches, the LB will either equal this UB or will be greater than it. In either case, BB will stop computing on each of those branches, resulting in the minimum possible number of computations and maximum efficiency. We would like to have a maximally efficient algorithm, and we can achieve this by training a neural network to predict which request BB should branch on first. Figure 5 shows how the NN will be incorporated into the overall framework.

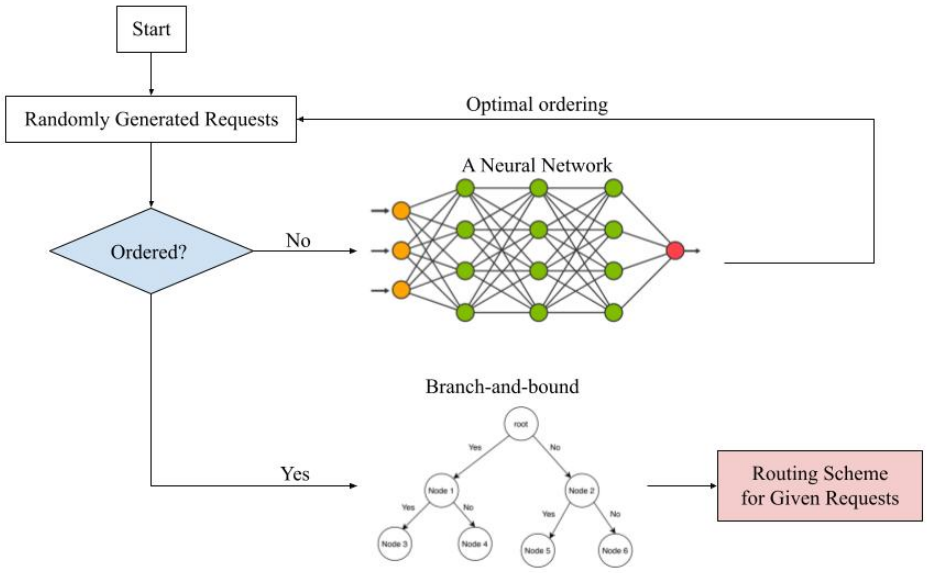


Figure 5. The overall routing scheme. Starting with a set of randomly generated requests, the trained NN outputs the optimal order of the set that minimizes the number of computations BB has to perform. BB then takes in the ordered requests and returns a solution to the given routing problem.

As we have demonstrated above, it's possible to use BB alone to solve the AMoD routing problem. However, we can maximize its efficiency by incorporating a NN.

Creating and Training a Neural Network

Simplifying the problem

An experiment shows that instead of predicting the entire order of requests to put into BB, a NN can equivalently predict which request should be first in the list. For each of 10 sets of randomly generated requests, a random request within the set was held in the first position while the rest of the requests were shuffled 7 times, and this was done for 5 different random requests from the original set. A BB tree was built and the number of computations BB performed was recorded each time the other requests were shuffled. Regardless of how the other requests were ordered, BB performed the same number of computations for any particular first request. In other words, the identity of the first request is enough to determine how many computations BB will perform for that set of requests. As such, the problem can be simplified so that the NN only has to predict which request should be first instead of predicting how all the requests need to be ordered.

Data Generation

For each set of n randomly generated requests, two random permutations of the set were created for each of the n requests being first. Although we can be reasonably certain that the ordering of the 2nd through nth request doesn't impact the number of BB's computations, two random permutations were generated instead of one as an extra precaution. Both permutations were inputted into BB, and a tree was built for each. The larger number of computations performed by BB from the two permutations was assigned to the current first request. When BB had run through all 2n permutations and a number of BB computations was determined for each of the n requests, the request that, being first, resulted in the least number of BB computations was said to be the corresponding output for the whole set of requests, the input.

The outputs were determined for 3000 sets of randomly generated requests, and these 3000 input-output pairs formed the data that would be used to train the NN. The data was then randomly split such that the NN would only train on 75% of the data, while the other 25% would be used to test the NN's performance.

Hyperparameter Sweep

Several NNs with different combinations of parameters were trained on the data. All NNs had the same structure with two hidden layers, so the variables were the size of each hidden layer, the learning rate, and the number of epochs. However, as the number of epochs is essentially determined by the hidden layer sizes and learning rate, we will not discuss it further. Let the size of a NN be denoted as (s_1, s_2), where s_1 is the size of the first hidden layer and s_2 is the size of the second hidden layer. The sizes we tested were (512, 512), (512, 1024), (1024, 512), and (1024, 1024). The learning rates we tested were 0.01, 0.03, 0.05, 0.07, 0.1, 0.3, and 0.5. Thus, with 4 different sizes and 7 different learning rates, we tested a total of 28 different NNs.

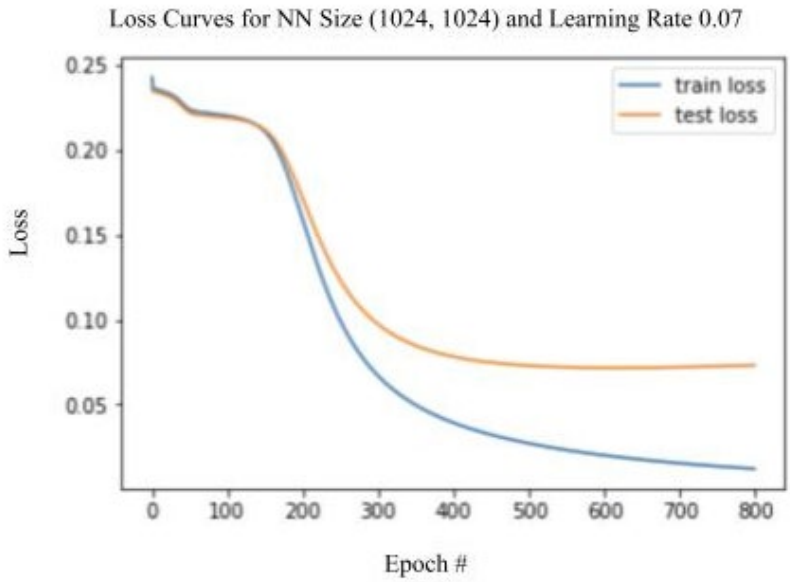
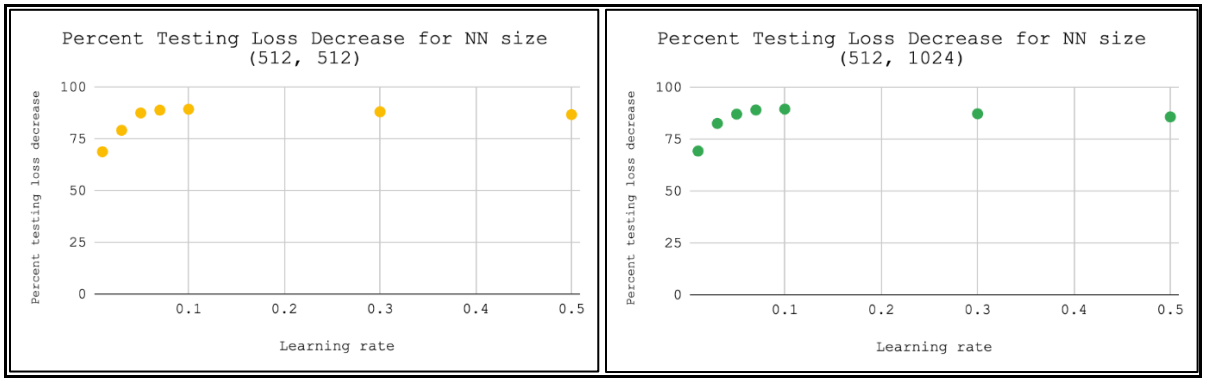


Figure 6. Training and testing loss curves per epoch. The NN has size (1024, 1024) and a learning rate of 0.07. Both the train and test loss decrease, indicating that the neural network is fitting to the data. After 800 epochs of training, the losses begin to plateau, signifying that the model optimization has reached a local minimum.

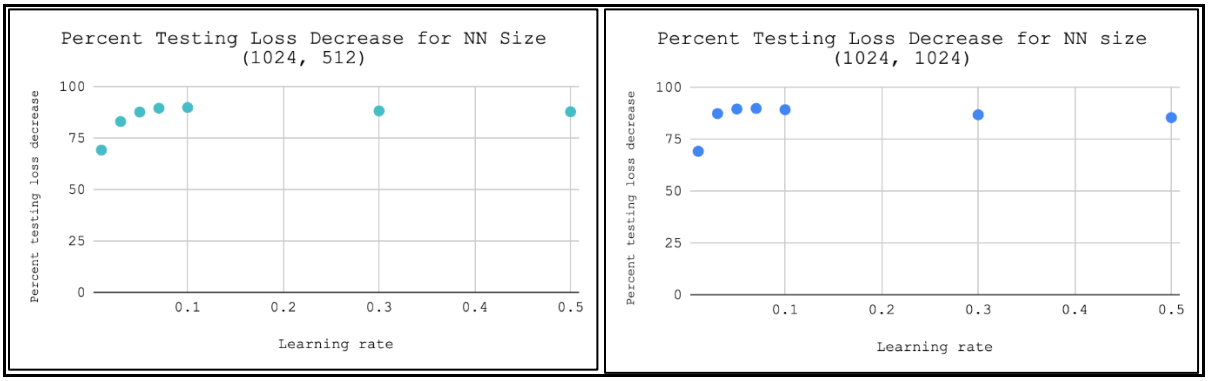
Two NNs were created and trained for each combination of size and learning rate. The loss, or the error between the NN's predictions and the actual output, was collected for the training and testing data, before and after training each NN. A sample training and testing loss curve is shown in Figure 6. However, as our goal is for the NN to learn broadly applicable patterns, the testing loss is a better metric than the training loss to measure how well the NN trained. For each combination of parameters, the average of the testing losses before and the average of the testing losses after training were recorded, and the percent loss between the two averages was calculated. The results for different NNs are summarized in Figures (7a) - (7d).

Because all percentage decreases between the testing loss before and after were positive, we concluded that all NNs trained to some extent. We also observed the same pattern in all of the NN sizes, where the NN trained poorly when the learning rate was too low. Moreover, apart from learning rates of 0.01 and 0.03, there was no significant difference in percent testing loss decrease when using any other NN size and learning rate combination. As such, the NN with size (1024, 512) and learning rate 0.1 had the greatest percent testing loss decrease—89.77%—so this was the combination of parameters that we used to conduct the rest of the experiments.

(7a) (7b)



(7c) (7d)



Figures (7a) - (7d). Percent testing loss decreases for NN of varying sizes. The percent loss was calculated between the testing losses before and after training an NN, for each of 28 NNs with different combinations of parameters (i.e. NN size and learning rate). All NNs trained to some extent—since all the plotted values are positive—although for all NN sizes, lower learning rates performed worse.

Results

Given the trained network, we ran trials using randomly generated requests to see how well BB performed on the NN predicted ordering versus all the possible orderings. In all trials, an ordering was generated for each possible request being first. BB built a tree for each ordering, and both the number of computations performed, and the amount of time needed to build the tree were recorded. Figure 8 summarizes the data for the number of computations performed, and Figure 9 summarizes the data for the run times.

	Number of BB Computations				
Trial #	NN predicted ordering	Average of all orderings	Minimum of all orderings	Maximum of all orderings	Percent improvement (from average)
1	19	29	19	57	34.48275862
2	22	30	22	66	26.66666667
3	28	48.21428571	28	140	41.92592593
4	23	30	23	69	23.33333333
5	20	29	20	60	31.03448276
6	28	47.85714286	28	134	41.49253731
7	25	28	25	50	10.71428571
8	15	17	15	45	11.76470588
9	19	21	19	38	9.523809524
10	29	45	29	145	35.55555556

Figure 8. The number of computations BB performed when building a tree for different orderings of sets of randomly generated requests. Over any set of requests, BB performed less computations than average given the NN predicted ordering. Furthermore, the NN predicted ordering always resulted in the minimum number of computations, which is exactly what we trained it to do.

When given the NN predicted ordering, BB always performed the minimum number of computations possible for that set of requests. This is exactly what we trained it to do, so we can say that the NN training was successful (see Discussion). Since it always achieved the minimum, it could never have performed worse than average; in fact, the NN predicted ordering resulted in, on average, 26.6% less computations than the average of all possible orderings.

Similarly, the amount of time BB took to build a tree when given the NN predicted ordering was always almost the minimum possible for that set of requests. One thing to note here is that BB didn't take the absolute minimum amount of time but was nevertheless very close to the minimum and relatively far from the average and maximum. Again, the NN predicted orderings never performed worse than average and were, on average, 24.1% faster than the averages of all possible orderings.

	Time Taken to Build Tree (s)				
Trial #	NN predicted ordering	Average of all orderings	Minimum of all orderings	Maximum of all orderings	Percent improvement (from average)
1	1.741202831	2.380709033	1.640149117	4.186253071	26.86200594
2	2.100915194	2.624884248	2.002646923	5.220301867	19.9616061
3	2.677391052	5.265179472	2.56871295	24.02265906	49.14910182
4	2.031543016	2.489379271	2.022150993	5.090121269	18.391583
5	1.881418943	2.473899388	1.843220949	4.379544973	23.94925387
6	2.974802971	6.847914892	2.902956963	31.39216995	56.55899616
7	2.214895964	2.422634716	2.175100803	3.890120029	8.574910241
8	1.361700058	1.437596242	1.2663908	3.200922251	5.279381056
9	1.641494989	1.782203298	1.636470079	2.94604516	7.895188424
10	3.028402328	4.022616049	2.900734186	10.5824542	24.71560071

Figure 9. Time taken for BB to build the tree for different orderings of sets of randomly generated requests. Here, BB took close to the minimum amount of time possible to run given the NN predicted ordering compared to all other possible orderings, still with a significant decrease from the average.

Discussion

We trained our NN to predict which request(s) being placed first in any set would minimize the amount of computations BB would perform when building a tree. The results show that the NN trained successfully, since in all 10 trials, the ordering predicted by the NN indeed led to the minimum possible number of computations for the given set of requests. Similarly, this ordering resulted in fewer computations, and therefore lower runtimes, subject to natural variability in runtime. The NN also worked with a high accuracy—for this sample, it worked 100% of the time—so that it could take any arbitrary set of requests and would never predict an ordering that performs worse than picking one at random.

Regarding time reduction, 24.1% of the times listed in the table would usually be around 1 second. But in a real-world AMoD system with thousands of requests, BB would take much longer, at which point decreasing run time by 24.1% would be significant, i.e., on the order of hours saved. The efficient algorithm would be more capable of handling large numbers of requests—this makes implementing a real-world AMoD system more feasible, since we would have a mechanism that could route requests in a timely manner. With greater efficiency, the algorithm would also require less power, making it more environmentally friendly as well as lowering the cost. AMoD would become an affordable, reliable form of transportation, widely accessible to all.

Future Improvements

There are several directions for improving our model. First, in order to be employed in a real-world AMoD system, the algorithm must be able to service the population of a large city. This would involve hundreds of thousands of requests compared to the maximum of 30 that we dealt with in this paper. Our model wasn't tailored to any specific number of requests, so it should still work on a larger scale with relatively little modification. However, our model was tailored to one specific road network, and we would like for it to be able to work on any arbitrary road network.

Second, as mentioned in Mathematical Representation, we only considered requests at one instant in time, and we assumed vehicles in the system travel along all edges in their routes instantaneously. More work would need to be done to incorporate vehicle travel times and shift to continuously received requests: that is, change the AMoD problem formulation to describe a dynamic, rather than static, system. One possible solution is to make the network-flow model resemble the one in [11], where each new time step is represented by a new copy of the road network.

We can also combine the results of this paper with other research into AMoD, especially once a continuous timeline has been added. Since vehicles are reused, a natural extension is to consider *rebalancing* as well as routing [13], [14]. Additionally, since vehicles are electric, the interaction with the power grid [18]-[19] and coordinating charging [20] should also be considered. Finally, the AMoD problem can be converted to an Intermodal-AMoD problem when other forms of transportation within a system are allowed [21]-[23].

Conclusion

In this paper, we adapted the branch-and-bound algorithm to an AMoD context and showed that it can find integer solutions to the AMoD routing problem. Using the algorithm, we generated data and trained a neural network to predict the ordering of a set of randomly generated requests such that BB performed as few computations as possible. We then demonstrated that the NN predicted orderings indeed resulted in a minimum number of computations and significantly reduced computation time. These results, when scaled up to a real world AMoD system, can make AMoD time-efficient and can save computing power, thereby reducing the cost. This allows reliable, eco-friendly transportation to become more feasible and accessible to all.

Acknowledgements

I would like to thank my mentors, Robin Brown and Somrita Banerjee, for teaching me everything I needed to complete this project. I would like to thank Professor Marco Pavone and Stanford University's Autonomous Systems Laboratory for facilitating this research. Finally, I would like to thank my computer for dealing with all my coding errors and not crashing.

References

[1] Pavone, M. (2015). Autonomous mobility-on-demand systems for future urban mobility. In Autonomes Fahren (pp. 399-416). Springer Vieweg, Berlin, Heidelberg. Available at https://link.springer.com/chapter/10.1007/978-3-662-45854-9_19.

[2] Land, A. H., & Doig, A. G. (2010). An automatic method for solving discrete programming problems. In 50 Years of Integer Programming 1958-2008 (pp. 105-132). Springer, Berlin, Heidelberg. Available at https://link.springer.com/chapter/10.1007/978-3-540-68279-0_5.

[3] He, H., Daume III, H., & Eisner, J. M. (2014). Learning to search in branch and bound algorithms. Advances in neural information processing systems, 27. Available at https://proceedings.neurips.cc/paper/2014/hash/757f843a169cc678064d9530d12a1881-Abstract.html.

[4] R. Zhang and M. Pavone. Control of robotic Mobility-on-Demand systems: A queueing-theoretical perspective. Int. Journal of Robotics Research, 35(1–3):186–203, 2016. Available at https://journals.sagepub.com/doi/abs/10.1177/0278364915581863.

[5] Iglesias, R., Rossi, F., Zhang, R., & Pavone, M. (2019). A BCMP network approach to modeling and controlling autonomous mobility-on-demand systems. The International Journal of Robotics Research, 38(2-3), 357-374. Available at https://journals.sagepub.com/doi/full/10.1177/0278364918780335?casa_token=celUGTO7nC4AAAAA%3AU9TeABMhkCHjwoQn3X4XruP8ACMaDDFtAZqQeuB4387rgGaK9t9zX9GqS3-hK_4y9FS_73xipoTW.

[6] Hörl, S., Ruch, C., Becker, F., Frazzoli, E., & Axhausen, K. W. (2018). Fleet control algorithms for automated mobility: A simulation assessment for Zurich. In 2018 TRB Annual Meeting Online (pp. 18-02171). Transportation Research Board. Available at https://trid.trb.org/view/1495210.

[7] Levin, M. W., Kockelman, K. M., Boyles, S. D., & Li, T. (2017). A general framework for modeling shared autonomous vehicles with dynamic network-loading and dynamic ride-sharing application. Computers, Environment and Urban Systems, 64, 373-383. Available at https://www.sciencedirect.com/science/article/pii/S0198971516302 37X?casa_token=dt530Laj49sAAAAA:c2Jr6ayr7lbPrhlG5BJ__sDAEik1sIBukQRoVHr0eHt96IlM4kthOl_xJHH5t1G8V5z39E-0JQ.

[8] Iglesias, R., Rossi, F., Wang, K., Hallac, D., Leskovec, J., & Pavone, M. (2018, May). Data-driven model predictive control of autonomous mobility-on-demand systems. In 2018 IEEE international conference on robotics and automation (ICRA) (pp. 6019-6025). IEEE. Available at https://ieeexplore.ieee.org/abstract/document/8460966.

[9] Zhang, R., Rossi, F., & Pavone, M. (2016, May). Model predictive control of autonomous mobility-on-demand systems. In 2016 IEEE International Conference on Robotics and Automation (ICRA) (pp. 1382-1389). IEEE. Available at https://ieeexplore.ieee.org/abstract/document/7487272.

[10] Xu, J., Rahmatizadeh, R., Bölöni, L., & Turgut, D. (2017). Real-time prediction of taxi demand using recurrent neural networks. IEEE Transactions on Intelligent Transportation Systems, 19(8), 2572-2581. Available at https://ieeexplore.ieee.org/abstract/document/8082792.

[11] Tsao, M., Milojevic, D., Ruch, C., Salazar, M., Frazzoli, E., & Pavone, M. (2019, May). Model predictive control of ride-sharing autonomous mobility-on-demand systems. In 2019 International Conference on Robotics and Automation (ICRA) (pp. 6665-6671). IEEE. Available at https://ieeexplore.ieee.org/abstract/document/8794194.

[12] Rossi, F., Zhang, R., Hindy, Y., & Pavone, M. (2018). Routing autonomous vehicles in congested transportation networks: Structural properties and coordination algorithms. Autonomous Robots, 42(7), 1427-1442. Available at https://link.springer.com/article/10.1007/s10514-018-9750-5.

[13] Salazar, M., Tsao, M., Aguiar, I., Schiffer, M., & Pavone, M. (2019, June). A congestion-aware routing scheme for autonomous mobility-on-demand systems. In 2019 18th European Control Conference (ECC) (pp. 3040-3046). IEEE. Available at https://ieeexplore.ieee.org/abstract/document/8795897.

[14] Wollenstein-Betech, S., Houshmand, A., Salazar, M., Pavone, M., Cassandras, C. G., & Paschalidis, I. C. (2020, September). Congestion-aware routing and rebalancing of autonomous mobility-on-demand systems in mixed traffic. In 2020 IEEE 23rd International Conference on Intelligent Transportation Systems (ITSC) (pp. 1-7). IEEE. Available at https://ieeexplore.ieee.org/abstract/document/9294258.

[15] Khalil, E., Le Bodic, P., Song, L., Nemhauser, G., & Dilkina, B. (2016, February). Learning to branch in mixed integer programming. In Proceedings of the AAAI Conference on Artificial Intelligence (Vol. 30, No. 1). Available at https://ojs.aaai.org/index.php/AAAI/article/view/10080.

[16] Etheve, M., Alès, Z., Bissuel, C., Juan, O., & Kedad-Sidhoum, S. (2020, September). Reinforcement learning for variable selection in a branch and bound algorithm. In International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research (pp. 176-185). Springer, Cham. Available at https://link.springer.com/chapter/10.1007/978-3-030-58942-4_12.

[17] Diamond, S., & Boyd, S. (2016). CVXPY: A Python-embedded modeling language for convex optimization. The Journal of Machine Learning Research, 17(1), 2909-2913. Available at https://www.jmlr.org/papers/volume17/15-408/15-408.pdf.

[18] Rossi, F., Iglesias, R., Alizadeh, M., & Pavone, M. (2019). On the interaction between Autonomous Mobility-on-Demand systems and the power network: Models and coordination algorithms. IEEE Transactions on Control of Network Systems, 7(1), 384-397. Available at https://ieeexplore.ieee.org/abstract/document/8737720.

[19] Estandia, A., Schiffer, M., Rossi, F., Luke, J., Kara, E. C., Rajagopal, R., & Pavone, M. (2021). On the interaction between autonomous mobility on demand systems and power distribution networks—an optimal power flow approach. IEEE Transactions on Control of Network Systems, 8(3), 1163-1176. Available at https://ieeexplore.ieee.org/abstract/document/9354039.

[20] Boewing, F., Schiffer, M., Salazar, M., & Pavone, M. (2020, July). A vehicle coordination and charge scheduling algorithm for electric autonomous mobility-on-demand systems. In 2020 American Control Conference (ACC) (pp. 248-255). IEEE. Available at https://ieeexplore.ieee.org/abstract/document/9147734.

[21] Salazar, M., Lanzetti, N., Rossi, F., Schiffer, M., & Pavone, M. (2019). Intermodal autonomous mobility-on-demand. IEEE Transactions on Intelligent Transportation Systems, 21(9), 3946-3960. Available at https://ieeexplore.ieee.org/abstract/document/8894439.

[22] Wollenstein-Betech, S., Salazar, M., Houshmand, A., Pavone, M., Paschalidis, I. C., & Cassandras, C. G. (2021). Routing and rebalancing intermodal autonomous mobility-on-demand systems in mixed traffic. IEEE Transactions on Intelligent Transportation Systems. Available at https://ieeexplore.ieee.org/document/9541261.

[23] Zgraggen, J., Tsao, M., Salazar, M., Schiffer, M., & Pavone, M. (2019, October). A model predictive control scheme for intermodal autonomous mobility-on-demand. In 2019 IEEE Intelligent Transportation Systems Conference (ITSC) (pp. 1953-1960). IEEE. Available at https://ieeexplore.ieee.org/abstract/document/8917521.