THE UNIVERSITY OF QUEENSLAND

AUSTRALIA

The Vehicle Scheduling Problem for Fleets with Alternative-Fuel Vehicles

Yupeng Wu s4596060

Xinqian Wang s4565489

Yanting Zhang s4603111

The University of Queensland

Submitted for MATH7202 coursework project

30th October 2020

1. Introduction and Background

Scheduling fleets of vehicles is an important aspect of designing many logistics systems. For environmental sustainability or government regulation reasons, a fleet of vehicles is changed to a fleet of alternative-fuel vehicles, and research is being done into how current infrastructure can accommodate them.

The problem of vehicle scheduling consists of assigning a fleet of vehicles to service a given set of trips with start and end times. Vehicle scheduling changes when alternative-fuel vehicles are used since the vehicles can carry only a limited amount of fuel and can refuel only at fixed locations. This paper mainly models the alternative-fuel multiple depot vehicle scheduling problem (AF-VSP), a modification of the normal VSP problem where a limited number of refueling stations and a relative low fuel capacity are provided.

Basically, the paper sets trips (T), depots (D) and fuel stations (S) as the mainly data-set and utilizes these sets for making a series of data and variables. The object of the model is for computing the minimum cost for the system AF-VSP.

The paper firstly uses column generation algorithm. Then, in order to increase running speed, the paper uses a Heuristic Algorithm by solving the problem in an extremely fast time.

2. Formal Definition of the Problem

In the stage of single line bus operation planning, the alternative-fuel multiple depot vehicle scheduling problem can be described as: given the number set with fixed starting and ending points and starting and ending time, the objective is to find a feasible minimum cost assignment of vehicles from depots to service trips (and fuel stations between service trips), such that

(1) each service trip is visited by exactly one vehicle;

(2) each vehicle must start and end at the same depot and obey the depot capacity;

(3) for any path a vehicle takes between two fuel stations or a depot, the sum of the fuel requirement of the service trips and dead-heading trips in the path must be less than or equal to the fuel capacity of each vehicle in the fleet.

Generate a graph $G(V, E)$, where V is the set of vertices ($V = L \cup D \cup F$). We define the V as follows.

Table 1. Sets

Service Trips	$T = \{\tau_1, \tau_2, \ldots, \tau_n\}$
Fuel stations	$F = \{\sigma_1, \sigma_2, \ldots, \sigma_s\}$
Depots	$D = \{\theta_1, \theta_2, \ldots, \theta_d\}$
Locations	$L = \{l_1, l_2, \ldots, lm\}$
$V = L \cup D \cup F$	$V = \{v1, v2, \ldots, v_k\}$
Start location (For each service trip τ_i)	$b_i \in L$
End location (For each service trip τ_i)	$e_i \in L$

E is the set of edges, where $(v_i, v_j) = e \in E$, $v_i \neq v_j$. There are five types of e.

(1) Trip from depot to start station

(2) Trip from end station to depot

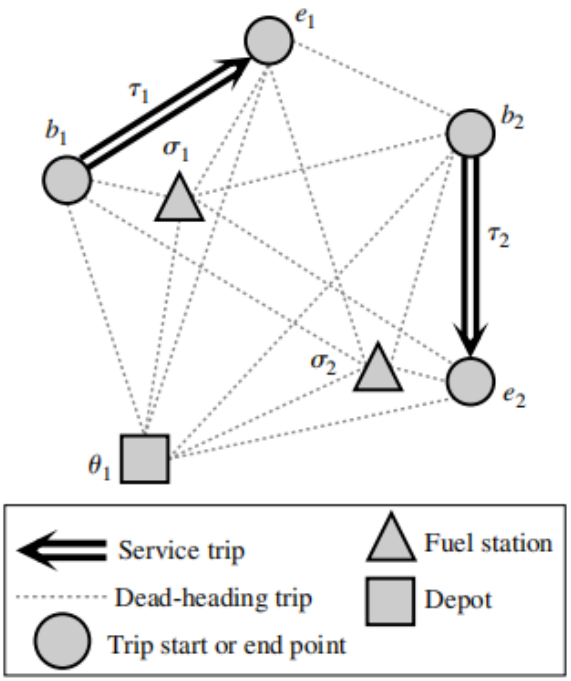
(3) Trip from start station to end station

(4) Trip from end station (fuel station) to fuel station (end station)

(5) Trip from start station (fuel station) to fuel station (start station)

We assume that the vehicles are in full fuel state when they leave the depot, and the vehicles can at least reach the next start station (or depot) after refueling. Figure 1 is an example about AF-VSP where $n = 2, s = 2$ and $d = 1$. (Dead-heading trips: given a set of n service trips, with costs and times assigned for vehicles to travel from the end location of one service trip to the start location of another.)

Figure 1. An Example AF-VSP Where $n = 2, s = 2,$ and $d = 1$



For each service trip τ_i, each depot θ_j and any pair v_1, v_2 with $v_1 \neq v_2$. We give the following definitions:

Table 2. Data

Start time (For each service trip τ_i)	bt_i
End time (For each service trip τ_i)	et_i
Fuel requirement (For each service trip τ_i)	$f(\tau_i)$
Depot capacity (For each depot θ_j)	$D_{capacity}$
Travel times (For any pair (v_1, v_2))	$Trip_{ij}^{t}$
Cost (For any pair (v_1, v_2))	$Trip_{ij}^{c}$
Fuel requirements (For any pair (v_1, v_2))	$Trip_{ij}^{f}$

If two trips τ_i, τ_j can be completed by the same vehicle in sequence $(e_{ti} + t(l_i, l_j) \leq s_{tj})$, it is said that the two trips can be combined. For any depot θ_i, if we get the feasible paths from depot θ_i, through a number of trips and fuel stations and back to depot θ_i. We define this path as trip chain p. The set of trip chain is $P = \{p_1, p_2, \ldots, p_s\}$.

Intuitively, as long as we get all the trip chains. Calculate the cost of each trip chain. Taking every trip chain as the variable of the problem, we can solve the problem by solving a large-scale integer programming problem. Unfortunately, unless the number of trips is very small, it is a huge task to enumerate all available trip chain.

3. The Column Generation Algorithm

We will introduce the column generation algorithm, the mathematical model based on the column generation algorithm, the code running results using random numbers, and the code running results using real data

3.1 A brief introduction of column generation algorithm

Column generation algorithm is an optimization algorithm for solving large-scale integer programming problems. Its theoretical basis was proposed by Danzig in 1960, and in 1995, Desrochers applied the column generation algorithm to driver scheduling and VRP problems.

The problem is firstly formulated as a binary integer programming problem. The binary integer program will then be relaxed and solved using column generation. Column generation divide the problem into a restricted master-problem and sub-problem. Solving the master-problem is very similar to solving the simplex method, if a column with a negative reduced cost can be found, it would be added to the master-problem until no more column can be added. To formulate the binary integer programming problem, they first generate a graph corresponding to the problem and then make a formulation based on the graph, which represents all possible ways of visiting a fuel station following a service trip or a depot. Use constraint ensures that each service trip is traversed exactly once and ensures that no depot will have more schedules using it than the number of vehicles stored at that depot. The objective is to minimize the sum of all of the costs of the schedules used in the solution.

3.2 Math Model

Sets:

T	set of trips
L	set of stations
F	set of fuel stations
D	set of depots
P	set of trip chain

$V = L \cup D \cup F$

Data:

B	Bus cost
$F_{capacity}$	Fuel capacity
F_{charge}	Fuel recharge
$D_{capacity}$	Depot capacity
bt_i	start time for trip i
et_i	end time for trip i
$b_i, i \in T$	start station for trip i
$e_i, i \in T$	start station for trip i
$Trip_{ij}^{f}, i, j \in V$	fuel consumption for trip from i to j

$Trip_{ij}^{t}, i, j \in V$ time spent for trip from i to j

$Trip_{ij}^{c}, i, j \in V$ cost for trip from i to j

Variable:

$$x_p = \begin{cases} 1, & \text{The optimal solution contains trip chain } p \\ 0, & \text{The optimal solution does not contain trip chain } p \end{cases}$$

$$b_{ip} = \begin{cases} 1, & \text{trip } i \text{ is contained in the trip chain } p \\ 0, \text{else} & \end{cases}, i \in T, p \in P$$

Master problem:

In this paper, we define the master problem as a model which consider all feasible trip chain, and the objective function is to find the minimum cost trip chain strategy.

Objective:

Minimize cost:

$$min \sum_{p \in P} (B + \sum_{(i,j) \in p} Trip_{ij}^{c}) \times x_p$$

Constraints:

Each trip occurs once

$$\sum_{p} b_{ip} x_p = 1, \forall i \in T$$

Sub problem:

We apply the following constraints to optimize the columns generated by the master problem.

Constraints:

Each trip can go to next trip with f fuel

$$Trip_{e_i,b_j}^{f} \leq f, \ \forall i, j \in V$$

Each trip chain has enough fuel

$$\sum_{(i,j) \in p} Trip_{ij}^{f} \leq F_{capacity} + |\{(i,j) : i \in F, (i,j) \in p\}| \times F_{charge}, \ \forall p \in P$$

Each trip has enough time

$$et_i - bt_j \geq Trip_{ij}^{t}, \ \forall i, j \in V$$

The trip chain meets the capacity of the depot

$$|\{i : (d, i) \in p\}| \leq D_{capacity}, \ \forall p \in P, \forall d \in D$$

Each trip chain starts from depot and final back to depot

$$|\{i:(i,j) \in p, i \in D\}| = 1, \forall p \in P$$
$$|\{j:(i,j) \in p, j \in D\}| = 1, \forall p \in P$$

Conservation of flow

$$\sum_i Trip_{ij} = \sum_j Trip_{ij}, (i,j) \in p, p \in P, i,j \in V$$

3.3 Results using random numbers

We analyze the model according to different data scale. To find the best amount of data for this model, and find the optimal number of fuel stations and depots.

Table 3. Quantity

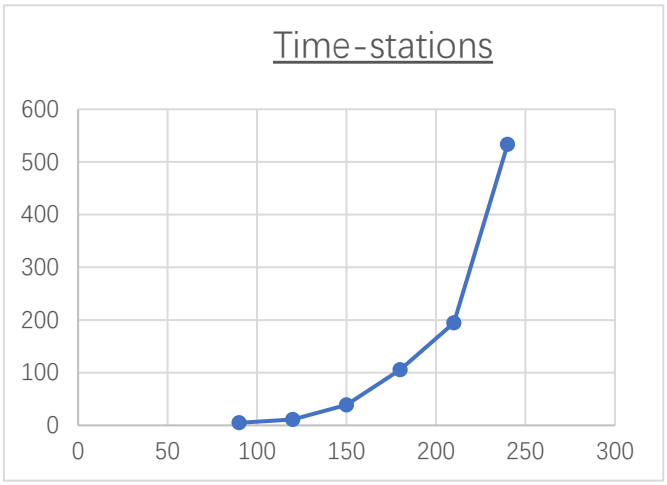
Fuel stations	5
Depots number	5
Depot capacity (The maximum number of bus a depot can hold)	10
Bus cost	15000
Fuel cost	2×Distance between two points
Time cost	Distance between two points

We use different numbers of trips and stations to analyze the running results of the model. Since in the real world, there may be more than one vehicle passing through the same station, so the fixed station / trip = 1.5

Table 4. Time consumed by different data sizes

Stations	90	120	150	180	210	240
Trips	60	80	100	120	140	160
Time(sec)	4.84	11.09	38.82	105.63	194.49	533.26
Columns	64352	184157	365514	928923	2003820	4399704
Cost (10^4)	1.97	2.28	3.05	3.08	3.88	4.70

Figure 2. Time Stations Line chart (x: stations, y: time)



3.4 Branch and Price

For a given twenty trips, restricting the max tour, the max numbers that each schedule can do, from 1 to 7. Calculating the π_t^{mt} $\forall t \in T, mt \in max\_tour$, the dual value of trip t with max tour equals to mt and made the table below.

Table 5.

Trip	dual value with max tour 1	dual value with max tour 2	dual value with max tour 3	dual value with max tour 4	dual value with max tour 5	dual value with max tour 6	dual value with max tour 7
0	15256	7695	5222.666667	276	276	276	276
1	15198	7671	5086.666667	184	186	186	186
2	15340	7811	5274.666667	374	382	382	382
3	15388	7937	5440.666667	15388	15368	15368	15368
4	15406	7873	5442.666667	15338	15320	15320	15320
5	15286	7703	5184.666667	14996	15006	15006	15006
6	15506	7803	5358.666667	15218	15216	15216	15216
7	15592	8093	5680.666667	15576	15558	15558	15558
8	15298	7691	5282.666667	15096	15088	15088	15088
9	15358	7807	5334.666667	450	438	438	438
10	15296	7715	5156.666667	224	224	224	224
11	15314	7733	5124.666667	214	214	214	214
12	15312	7659	5202.666667	258	276	276	276
13	15530	7867	5452.666667	384	402	402	402
14	15572	7893	5404.666667	478	480	480	480
15	15250	7687	5246.666667	248	248	248	248
16	15222	7653	5186.666667	188	206	206	206
17	15320	7709	5158.666667	200	200	200	200
18	15274	7723	5090.666667	194	194	194	194
19	15208	7753	5142.666667	194	196	196	196

At the 3rd column of Table 5, when the max tour equals to 2, the algorithm generates a schedule, (4, (0, 17)) (Basically means a schedule starts from depot-4 with have trip-0, trip-17) with the price 15404. We can see in 3rd column, the dual value of trip 0 is 7695 and the dual value of trip 17 is 7709 which means the price 15404 = 7695(π_0^2) +7709(π_{17}^2).

Interestingly, $\pi_0^2=c(\theta_4,\tau_0)+c(\tau_0)+0.5*B-1$ and $\pi_{17}^2=c(\tau_0,\tau_{17})+c(\tau_{17})+0.5*B+1$.

So, we can deduce the dual values in the algorithm explaining the cost that's a schedule needs to pay if it wants to contain a certainly trip. The cost can be made by the cost of the trip itself and the cost of driving to the start of that trip, and, it also shares a nearly equivalent percent of the BusCost with other trips in the schedule when all the length of the schedule is equivalent.

However, in the case max tour is greater or equal than 4, we can see that from each column, trip 4,5,6,7,8 has the biggest dual value and in the calculated schedule each of these trips are not contained in a same schedule (Otherwise it would be Impossible).

Table 6.

Trip	$\Delta\pi^{1,2}$	$\Delta\pi^{2,3}$	$\Delta\pi^{3,4}$	$\Delta\pi^{4,5}$	$\Delta\pi^{5,6}$	$\Delta\pi^{6,7}$
0	-7561	-2472.333333	-4946.666667	0	0	0
1	-7527	-2584.333333	-4902.666667	2	0	0
2	-7529	-2536.333333	-4900.666667	8	0	0
3	-7451	-2496.333333	9947.333333	-20	0	0
4	-7533	-2430.333333	9895.333333	-18	0	0
5	-7583	-2518.333333	9811.333333	10	0	0
6	-7703	-2444.333333	9859.333333	-2	0	0
7	-7499	-2412.333333	9895.333333	-18	0	0
8	-7607	-2408.333333	9813.333333	-8	0	0
9	-7551	-2472.333333	-4884.666667	-12	0	0
10	-7581	-2558.333333	-4932.666667	0	0	0
11	-7581	-2608.333333	-4910.666667	0	0	0
12	-7653	-2456.333333	-4944.666667	18	0	0
13	-7663	-2414.333333	-5068.666667	18	0	0
14	-7679	-2488.333333	-4926.666667	2	0	0
15	-7563	-2440.333333	-4998.666667	0	0	0
16	-7569	-2466.333333	-4998.666667	18	0	0
17	-7611	-2550.333333	-4958.666667	0	0	0
18	-7551	-2632.333333	-4896.666667	0	0	0
19	-7455	-2610.333333	-4948.666667	2	0	0

From the table above we can see that the gap between each step's dual value is fluctuating and decreasing to zero when nothing is changed. Which give us an idea about if we pick the right trip and package it into a schedule, the total dual price would go down. More detailed to say, if we calculate the dual price correctly with the candidate trip, we can decide whether it can be contained in some king of schedule. Which would be different from the ordinary column generation and save much time. Because the normal generation process would only contain the feasible column

independently instead of taking the whole processing into consideration. We eliminated the trip cost τ_t from the cost calculating algorithm because no matter what the schedule is, these costs would always be costed.

We make a general function for calculating the current fuel f and expense with a particularly schedule.

Figure 3. CostSuccessor Algorithm

$function$ ***CostSuccessor***
$On\ input\ a\ schedule(\theta_d, \{\tau_i, \tau_{i+1} \dots, \tau_{i+n}\}, f = 0, expense = 0)$
$if\ f == 0\ and\ expense == 0:$
    $Initialize\ the\ f\ and\ expense\ with\ the\ value\ after\ doing\ the\ first\ trip$
    $if\ len(\{\tau_i, \tau_{i+1} \dots, \tau_{i+n}\}) == 1:$
        Return $f, expense$
$if\ len(\{\tau_i, \tau_{i+1} \dots, \tau_{i+n}\}) \geq 2:$
    $Update\ the\ f, expense\ when\ the\ condition\ meets\ refueling\ or\ keepgoing\ cases$
    Return $CostSuccessor(\theta_d,\ \{\tau_i, \tau_{i+1} \dots, \tau_{i+n}\}[1:],\ f,\ expense)$
$Else:$
    $NoComplible \leftarrow \emptyset$
    Return $NoComplible, NoComplible$
$Update\ the\ f\ and\ expense\ in\ the\ end\ of\ schedule$
Return $f, expense$

To start, we follow general approach of Danztig-Wolfe Decomposition:

1. Generate initial set of Variables.
2. Solve the resulting Restricted Master Problem.
3. Use the dual variables to find new Master Problem variables which will improve the objective.

$$min \sum_{p \in P} Cost_p Z_p$$

$$\sum_{p \in P_t \forall t \in T} Z_p = 1 \quad Z_p \in [0,1]$$

Denoted $\pi_t\ \forall\ t \in T$ as the dual value of trip t. The dual value of Z_p is denoted as below:

$$Dual(Z_p) = Cost\_current - \sum_{t \in p} \pi_t$$

We keep updating the dual value of Z_p, and do the column generation when we find a price that is less than the current value for $p \in P$.

Figure 4. The Algorithm for finding a cheap route according to the dual value

For θ_d , τ *in* $CanReach$: $(\forall\ \tau \in T)$

$Initialize\ short_{t\ \forall t \in T} \leftarrow (\infty, \emptyset)$

$short_\tau \leftarrow (\ B + c(\theta_d, \tau) - \pi_\tau\ , (\tau,)\ )$

For j *in* $CanReach[\theta_d\ , \tau]$: $(\forall\ j \in T)$

if $short_j[0] + c(j, \theta_d) < 0$

$f, expense\ = CostSuccessor(\theta_d, short_j[1])$

if $f \geq c(j, \theta_d)$ *and* $expense < NoComplible$:

Add the column with the cost

For tt *in* $Successors[\theta_d\ , \tau\ , j]$: $(\forall\ tt \in T)$

$f, expense\ = CostSuccessor(\theta_d, short_j[1])$

if $f \geq c(j, \theta_d)$ *and* $expense < NoComplible$:

$ff, expense\_next\ =$ ***CostSuccessor***$(\theta_d, short_j[1] + (, tt))$

if $ff \geq c(tt, \theta_d)$ *and* $expense < NoComplible$:

if *no need reful befor doing Trip* tt :

$short_{tt}[0] = \min(short_j[0] + c(j, tt) - \pi_{tt}\ , short_{tt}[0])$

if the cost is updated, then do update to the $\boldsymbol{short_{tt}[1]}$

Elif *reful befor doing Trip* tt :

$\sigma_s = Trip\_Trip\_Cost[j, tt]\ (\forall\ \sigma_s \in F)$

$short_{tt}[0] = \min(short_j[0] + c(j, \sigma_s, tt)$

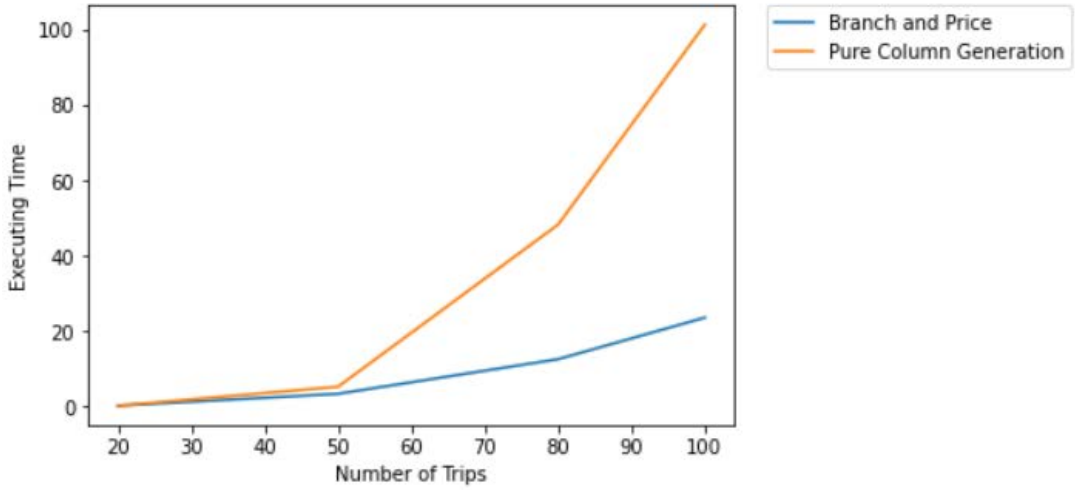
$- \pi_{tt}\ , short_{tt}[0])$

if the cost is updated, then do update to the $\boldsymbol{short_{tt}[1]}$

Else :

conitue next step

Figure 5. Execute Time for Two Algorithms



We can see a huge improvement by implementing the Branch and Price algorithm. What's for sure is that the algorithm can certainly be revised and saving a huge amount time because there is a lot of code checking if the current station is feasible to continue and many of them would be redundant. Moreover, the algorithm **CostSuccess** takes a complete trip chain and compute the current condition from the beginning is repetitive. Whereas, it seems there is no better way saving the fuel condition or calculating it fleetly.

Sadly, the algorithm itself is not complete correct. We deduced the outcome could give the number of correct buses so there is a little bug. It could be the reduced cost in refueling condition or some code just restrict the generation of feasible trip chain.

Figure 6. The Gap between pure Col-Generation and Branch and Price

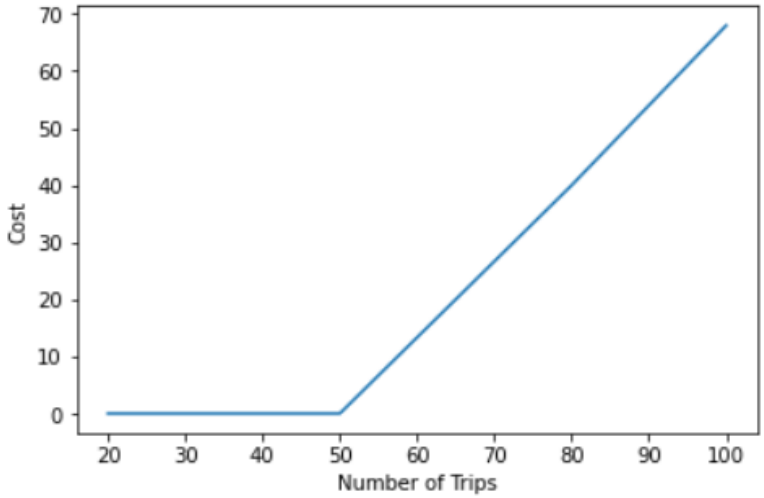


Table 7. Comparison Between Pure Column Generation and Branch and Price

Pure Column Generation	Branch and Price
(0, (15, 44, 63, 75))	(0, (20, 36, 61, 76))
(0, (24, 55, 79))	(1, (0, 16, 25, 47, 59))
(1, (0, 16, 25, 47, 59))	(2, (24, 55, 79))
(1, (5, 26, 32, 64))	(3, (3, 18, 29, 43, 65))
(1, (19, 30, 41, 53, 69))	(3, (9, 22, 40, 66))
(2, (1, 17, 42, 51, 68, 78))	(3, (11, 35, 57, 77))
(2, (8, 39, 58, 72))	(2, (1, 17, 42, 51, 68, 78))
(2, (20, 36, 61, 76))	(2, (4, 21, 62, 70))
(3, (2, 14, 27, 37, 54))	(2, (15, 41, 53, 63, 75))
(3, (3, 18, 28, 34, 48, 67, 73))	(3, (12, 38, 46, 50, 74))
(3, (4, 29, 52))	(1, (5, 26, 32, 69))
(3, (6, 23, 43, 65))	(3, (2, 14, 27, 37, 54))
(3, (9, 22, 40, 66)) 1.0 16000	(3, (6, 23, 67, 73))
(3, (10, 21, 62, 77))	(3, (13, 28, 34, 52))
(3, (11, 35, 57, 70))	(1, (19, 30, 48, 64))
(3, (12, 38, 46, 50, 74))	(0, (8, 39, 58, 72))
(3, (13, 33, 45, 49, 60))	(3, (10, 33, 45, 49, 60))
(5, (7, 31, 56, 71))	(0, (7, 31, 44, 56, 71))

The Table 7 lists the two algorithms' outcome for 80 trips and without any constrain but only going back to the starter depot, meeting the time and fuel constrain.

The yellow line is just entirely two same trips and the green line is the same trip with different starting depot. We can just focus on the first green column on the right and see why the schedule have not worked from deport 2. Interestingly, you can also see the sequence, for pure column generation, the algorithm first fixed the depot and do the circulation for trips. For Branch and Price, the sequence may tell us what's the first trip chain that is generated in the algorithm. Anyway, we would keep fixing what's going on and emailed you as soon as we solved.

When talking about the Branch and Price, the paper said that when a non-integer solution coming, as in my algorithm I usually got 0.5 for two trip chains contained by a same trip. In paper, the author let it branches into two cases, one trip chain with pair $[\tau_i, \tau_j]$ must stay together, the other trip must not the pair $[\tau_i, \tau_j]$ can stay together. Then, at the end of the Branch and Price part, the author also talks that he used a list to save these branch point so that for the following iteration it could come to a fast react and just exclude the pair for being excluded by the former branches.

3.5 Conclusion

We can see that the running time increases exponentially with the number of stations. And before 150 stations, the ratio of site to running time is good.

Although the column generation algorithm optimizes the original linear programming problem to some extent, it still needs a lot of time to calculate in the scale of big data. On this basis, we establish the model of the latter two algorithms.

Basically, The Column Generation is just like another kind of Lazy Constrain. The later one uses the constrains and the former one uses the objective coefficient generated by all the possible solution. Then, to accelerate the executing time for fitting the huge data. The Danzig-Wolf Decomposition and Benders Decomposition is combined respectively.

Moreover, there is an important technique which we thought it really speed up the whole process when we explored the random data. This technique is generating

as much data as you can at the beginning. We also tried to implement the Python NumPy Array to make the calculation faster. But this idea still stays on the imagination. The file CODCG really helps us a lot, the dictionary CanReach and Successors would just like two filters which make things move smooth at the beginning. We also generate the optimal fuel station between two different trips before run the algorithm, the reason we can do that is because we think it is just the Pareto Optimality case. If we have to fuel, we need to pick the most optimal location without doing any changes to other conditions.

Another interesting thing would be giving the restrictions which could fit the normal life situation and at the same time speed up the algorithm. We have tried many concepts such as put a restriction on max working time, max tour, depot capacity and using depot as a refuel station only if a bus finished the trip.

4. A Heuristic Algorithm for AF-VSP

As you can see from the above line chart, the AF-VSP problem can be really time consuming when there are many trips and stations in this NP-hard problem. Thus, it is necessary to have a heuristic algorithm to serve thousands of trips in a day. In order to bring the problem closer to the reality, we assume that (1) each depot has limited available vehicles ($nBus$: number of buses in each depot), (2) vehicles can refuel both in fuel stations and depots and (3) always make sure a vehicle can return to a depot during its schedule. The first assumption is set in order to avoid using too many buses in the problem. The second assumption can help us reduce the optimal cost for the AF-VSP problem. The third assumption prevent having an infeasible solution during the algorithm.

4.1 Algorithm

This algorithm is constructed based on the concurrent scheduler algorithm (Bodin, Rosenfield, & Kydes, 1978). Which assigns the first service trip to vehicle 1 in a depot that has a minimum cost, then iterates through the remaining service trips. If this vehicle cannot serve the trip, consider the refuel process for this vehicle and compare the fuel cost with bus cost. When it is cheaper to assign this trip to another vehicle in a depot, assign the trip to a new vehicle that with have the smallest cost. Adler and Mirchandani (2017) referred this method as the fuel scheduling algorithm (FSA). The process of using FSA to allocate trips to bus schedule is as follows:

1. Assign $t_1 \in Trip$ to vehicle b from depot d which has the minimum cost according to FSA. Set $i = 2$.

2. Determine all feasible schedules for trip t_i (i.e.: which vehicles could serve in terms of time and fuel). Use FSA to find which vehicle has the minimum cost and then assign trip t_i to this vehicle. We developed a **CanSchedule** function to determine whether the vehicle needs refuel before it takes trip t_i. After having a feasible schedule, detect whether this is a first trip for a vehicle. If this is the first trip of a vehicle, compare the bus cost and the trip cost from its depot to trip t_i with the cost of having an activated vehicle go to refuel. Only keep the schedule with the fewest cost.

3. Set $\mathrm{i} = \mathrm{i} + 1$. If $\mathrm{i} > \mathrm{n}$, which means all the trips are assigned to a vehicle and we can return the schedules for vehicles; otherwise, return to 2.

Here is the comparison between the column generation method and FSA method. Here we set $number\ of\ Depots = 5, number\ of\ Fuel\ Stations = 5, nBus = 5.$

Although this method is likely to provide a suboptimal solution, it has the advantage of being extremely fast. Each service trip has to be analyzed only once, and then to be assigned into a bus schedule. Since this algorithm runs quickly, it can be used to generate a feasible starting point for more effective algorithms such as large neighborhood search.

4.2 Pseudocode

Function CanSchedule:

On input is the current schedule $tour$ for a bus b from depot d and a new trip $t_i \in T$ that needed to be assigned. $tour$ contains the previous trips ($[pretrips]$), where last trip $t_l = tour[-1]$, and initial trip $t_0 = tour[0]$, current total cost ($cost$) and the remaining fuel of the bus (f).

Returns bus schedule contains tour, tour cost, and remaining fuel after assigned t_i.

If $cost == 0$*:* \\ Which means this is the first trip for this bus, then activate this bus

Return $([t_i], B + Trip^t_{dt_i}, F_{capacity} - Trip^f_{dt_i})$

\\ Whether the bus can finish t_i. Using **CanFuel** function to detect whether the bus can finish t_i and whether it needs to refuel or can refuel. Return the optimal fuel plan.

FuelPlan = ***CanFuel****(*d, t_0, t_l, t_i, f*)*

\\ *FuelPlan[0]* contains the additional cost after assigned t_i

\\ *FuelPlan[1]* contains the fuel after assigned t_i

\\ *FuelPlan[-1]* contains a flag for refueling status

If not FuelPlan[-1]:

\\ When **CanFuel** returns a negative flag

Return False \\ Which means this is a bad assignment

Else if FuelPlan[-1] == 'Trip':

Return ($[pretrips] + [t_i]$, $cost$ +*FuelPlan[0]*, *FuelPlan[1]*)

Else if FuelPlan[-1] == 'Fuel'':

s *= FuelPlan[2]* \\ Fuel location

Return ($[pretrips] + [s, t_i]$, $cost$ +*FuelPlan[0], FuelPlan[1]*)

Function CanFuel:

Inputs are the depot d, last trip t_1, next trip t_2, and remaining fuel f.

Returns next step's trip cost, remain fuel, and fuel status (*'Trip'* or *'Fuel'*).

If t_1 end too late or t_2 start too early:

Return False

If there is not enough fuel to finish t_2: \\ Find the refuel plans.

FuelPlans = []

For s *in* FuelStations and Depots:

If the bus can finish refuel process before the start time of t_2

AND

the bus has enough fuel to the station:

FuelPlans.append($Trip^t_{t_l s} + Trip^t_{st_i}$, $F_{capacity} - Trip^f_{st_i}$, t_2, s, *'Fuel'*)

If $len($*FuelPlans*$) == 0$: \\ Cannot refuel

Return False

Else:

Return $min($*FuelPlans*$)$ \\ Return the plan with minimum cost

Return ($Trip^t_{t_l t_i}, f - Trip^f_{t_l t_i}$, *'Trip'*) \\ When don't need refuel

4.3 Results and Comparisons

Table 8. The Results of Testing the Column Generation and Concurrent Scheduler Algorithms on random data. $(nDepots = 5, nFuels = 5, nBus = 5)$

Run number	Number of trips	Column Generation			Concurrent scheduler		Increase in cost for heuristic (%)
		Runtime (sec)	Cost	Number of columns	Runtime (sec)	Cost	
1	50	2.2668	14480	45152	0.0477	27589	90.53
2	60	5.6293	19479	84350	0.0407	28045	43.98
3	70	9.6180	21302	142112	0.0469	29135	36.77
4	80	12.4053	24003	180829	0.0660	29714	23.79
5	90	22.5922	24594	270551	0.0520	29379	19.46
6	100	61.2862	26437	779557	0.0469	29972	13.37

Since the concurrent scheduler restricted the number of available bus in each depot and enable buses to refuel in depots, we add these constraints to the column generation method and then compare the performance of them. From the row of "Increase in cost for heuristic (%)" in table 1, we can find out that the concurrent scheduler has unacceptable high cost when there are small number of trips. However, when the number of trips grows, the running time for column generation grows rapidly, while the increase cost for heuristic decreases. Besides, the concurrent scheduler keeps a fast speed for solving the bus scheduling problem.

4.4 Conclusions

In the heuristic algorithm we additionally have two more constraints: (1) each depot has limited buses, and (2) vehicles can refuel both in the depots and fuel stations in order to make the simulation more realistic

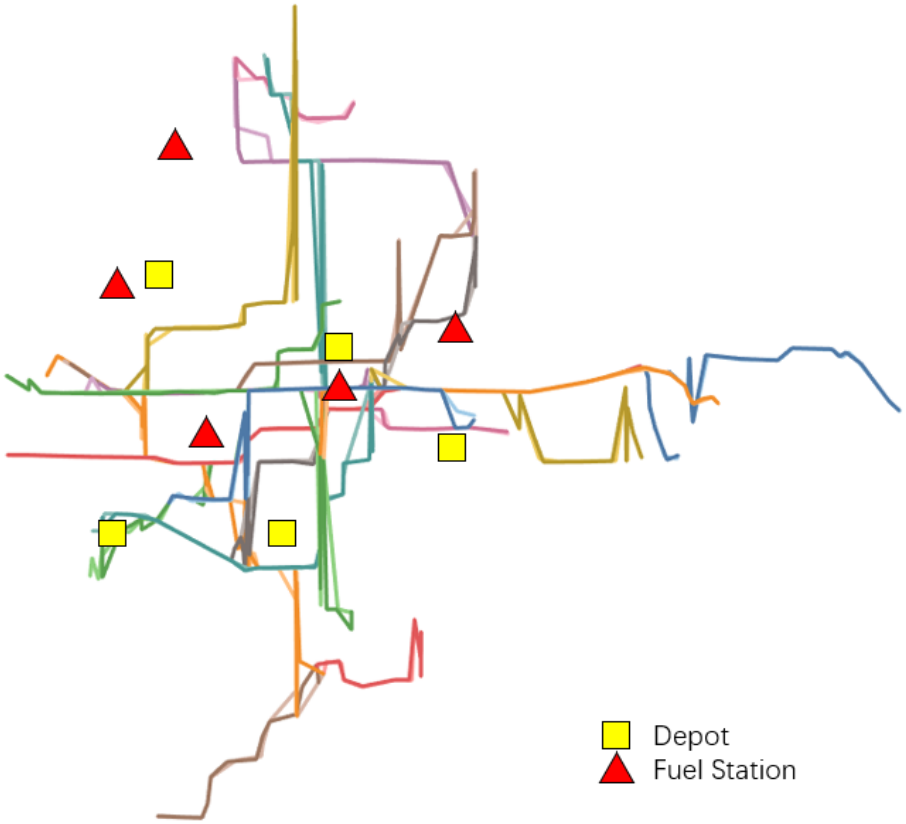
We also compared the suboptimal results from the concurrent scheduler with the optimal results from column generation. In general, when we have a small data, we should use column generation method for a better optimal cost. For large dataset, we should apply the concurrent scheduler for a suboptimal but quick solution, also this solution will not increase too much cost in this case.

5. Data from the Baidu Map

Baidu Map collected bus service information from 300 cities in China in 2015 and part of the data from XiAn is publicly available on the data shop website (http://www.data-shop.net/). This includes the bus ID, start and end locations, and starting time for each trip. However, we still need the trip time, depot locations, and fuel station locations in our problem. Five depot locations were found based on the location of BanPo Bus Centre, ChangAn Bus Centre, South XiAn Passenger Terminal, West XiAn Passenger Centre and XiAn Bus Station. Five fuel stations were found based on the location of five Sinopec stations (the China Petroleum and Chemical Corporation).

The trips from Baidu Map are organized into routes, where a route is a set of service trips that continuously start from one bus station to another. Buses are organized to travel in both direction of the route every hour. For example, route 102 travels between the South Park Road and the Labor Road in both directions. For our analysis we used 42 bus routes and 315 bus stations. Unfortunately, this dataset does not provide the information about time cost and fuel cost for each route, so in order to obtain these data we used Baidu Map API. A map of the 42 routes and the depots and the fuel stations can be seen in Figure 2.

Figure 8. The XiAn Bus Network and Fuel Station and Depot Locations



The starting time and the end time are set from 6 a.m. to 6 p.m. and trip will start every hour according to the data. That is to say for each route we have 12 trips per day

and we totally have 504 trips.

Based on the fuel volume and money cost conversion formula:

$$Fuel\ price\ for\ driving\ 100\ km = \ ¥120\ (In\ China) \approx AUD\$24$$

The buses are assumed to have the full fuel mileage of 200 kilometers. The bus cost is $24, which represents the daily salary and cost for driver. In order to speed up the computation, we restricted the maximum tour for each bus from its depot to its first trip to be no more than 400 kilometers.

5.1 Results and Discussions

Because of the massive trips and tremendous maximum tour setting, the column generation method will cost too much time that we are not able to run. On the other hand, the concurrent scheduler provided a quick solution for this problem. However, this solution may not seem to be reasonable, which shows that no bus needs to refuel during. We have tried two ways to detect what is wrong.

First, we increased the bus cost and decrease the fuel capacity so that it becomes more expensive to activate a new bus and buses are forced to refuel early. However, the model becomes infeasible. It will be either do not have enough fuel for bus to return to a depot or cannot assign a trip which is far from the depots.

Second, we add an error flag in the **CanSchedule** function in order to detect whether the buses refueled. The flag shows that some bus tours include the trip to a fuel station, so our fueling function is also correct and some buses also triggered the refuel process. So, we are still struggling about the reason why this do not perfectly work as it was on the random generated data.

Table 9. Results from concurrent scheduler based on real dataset.

Bus cost	100000
Maximum Tour (km)	400000
Fuel capacity (km)	200000
Bus limit (per depot)	10
Number of depots	5
Number of stations	5
Cost	5289656
Number of buses used	50
Runtime (sec)	0.0607

Reference

Jonathan D. Adler, Pitu B. Mirchandani. (2017). The Vehicle Scheduling Problem for Fleets with Alternative-Fuel Vehicles. Transportation Science 51 (2):441-456. http://doi.org/10.1289/trsc.2015.0615