RC25639 (WAT1611-045) November 10, 2016
Mathematics

IBM Research Report

An Algorithm for Bus Network Design

Francisco Barahona[1], João P. M. Gonçalves[1], Richard Santiago[2], Chai Wah Wu[1]

[1]IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598 USA

[2]McGill University
Montreal, Canada

Research Division
Almaden – Austin – Beijing – Brazil – Cambridge – Dublin – Haifa – India – Kenya – Melbourne – T.J. Watson – Tokyo – Zurich

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties). Many reports are available at http://domino.watson.ibm.com/library/CyberDig.nsf/home.

AN ALGORITHM FOR BUS NETWORK DESIGN

FRANCISCO BARAHONA, JOÃO P. M. GONÇALVES, RICHARD SANTIAGO,
AND CHAI WAH WU

ABSTRACT. We consider the problem of designing the public bus transit network of a medium size city. The goal is to produce bus routes so that the overall travelling time of the passengers is minimized. Our approach consists in breaking down the original problem into two smaller subproblems or phases. In the first phase we generate a large set of good candidate routes. We describe and implement five different route generation heuristics and compare their performance. In the second phase we take the candidate routes as input and use an integer programming model to choose the final set of routes. For the second phase we compare the commercial solver CPLEX with a Lagrangian relaxation approach. It turns out that this second method is much faster to produce solutions of similar quality.

1. INTRODUCTION

As stated by Ceder & Wilson [5] and many others, the bus planning process is usually divided into the following activities: bus network design, setting frequencies, timetable development, bus scheduling, and driver scheduling. The reason for this breakup is the complexity of each of these tasks. This makes very difficult to treat all of them in one single model. Here we give an algorithm for the first task, namely, we develop a procedure that produces the routes for the buses that cover the transport demand of a medium size city.

In our case, we have a network whose nodes correspond to street corners, and whose arcs correspond to the streets. Some preprocessing is required to identify all forbidden crossings, forbidden turns and unusable streets. Then for every arc we assume that we have the time that it takes for a bus to drive through, and for a person to walk through. We also have an *origin-destination (OD) matrix* that specifies the demand for transportation between a selected set of pairs of nodes. Also in our case there are three terminal nodes, so that each route corresponds to a cycle in this graph going through one of the terminals. A route should take at most thirty minutes, and when generating the routes we use a penalty to discourage left turns. Our objective is to minimize the total travel time. This includes the time walking and the time in a bus for the different users. Several other objective functions have been considered in recent and older literature. Some examples

of these include minimizing the passenger waiting time, number of transfers, operating costs, fleet size, and many others.

Our approach consists in breaking down the original problem into two smaller subproblems that we call *route generation phase* and *route optimization phase*. In the former we study and compare different heuristics for generating a large set of candidate routes. In the latter, we take the candidates routes as an input and we formulate an integer linear program to choose the final set of routes. For this model we first tried the commercial solver CPLEX [8], but due to the large computing time we had to switch to a Lagrangian relaxation approach that was much faster and produced solutions of similar quality.

There is a large number of articles written on Transit Network Design, we mention just a few of the different techniques used to attack this problem. Heuristics for route generation have been presented in Ceder & Wilson [5], Baaj & Mahmassani [1], and many others. Integer programming techniques have been used in Bussieck et al [4], Claessens et al [6], Goossens et al [13]. Column generation has been used by Schöbel & Scholl [16] and Borndörfer et al [3]. Meta-heuristics have also been widely used, see Fan et al [11], [12], and Zhao et al [17], for instance. For a more complete set of references see the surveys by Desaulniers & Hickman [10], Kepaptsoglou & Karlaftis [14] and Schöbel [15].

The rest of the paper is organized as follows. Section 2 describes our two-step approach. In Section 3 we show our Lagrangian relaxation method. Section 4 reports computational results with a dataset corresponding to a medium size city in the US. We end with conclusions in Section 5.

2. Our Approach

We break down the original problem into two smaller subproblems: the route generation phase and route optimization phase. In the former, we look for ways (or heuristics) to generate "good" candidate routes. In the latter, given a fixed (large) number of (previously generated) candidate routes, we use an integer programming model to choose the final routes out of the candidate ones. We next discuss each of these two phases in more detail.

2.1. **Route Generation.**

We implement and compare different heuristics for generating candidate routes. Our route generation process breaks down into two stages: route skeletons creation and route skeletons expansion. This two-phase generation procedure is very similar to the one discussed in [1] by Baaj and Mahmassani. However, some of our route expansion heuristics are different from the ones discussed there. Moreover, [1] does not deal with the additional constraint that every route has to pass through one of the terminals.

In the first stage of the generation process we try to create three different skeletons for each OD pair. A skeleton is just a (directed) cycle that includes an OD pair, or one node from an OD pair and a terminal. We create

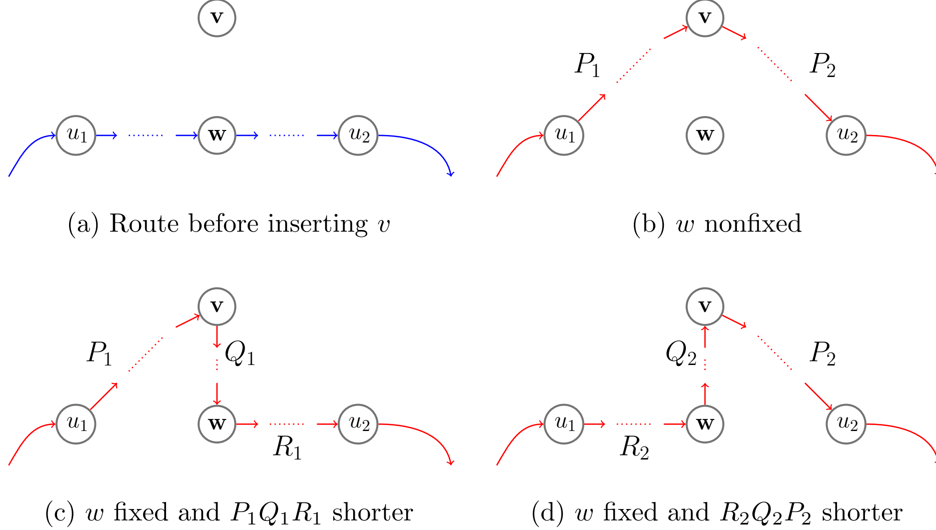
such three skeletons as follows. Given an OD pair, we look at the different terminals and pick the one that leads to the shortest cycle including the OD pair and the terminal. If this cycle does not exceed the maximum allowed duration for a route, we add it to the set of route skeletons. In a similar way, we take the origin of the OD pair, look at the different terminals, and pick the one that leads to the shortest cycle containing the origin and the terminal. If this cycle does not exceed the maximum allowed duration we add it to the set of skeletons. Finally, we repeat the same procedure for the destination node. One of our data-sets had 160 OD pairs, thus if we can successfully add all the generated skeletons, we end up with a total of (160 + 320 =) 480 route skeletons.

The second stage of the generation process is the expansion stage. Here, we expand the initial skeletons to actual routes. The expansion process consists of two steps: node(s) selection and node(s) insertion. In the former, we use different heuristics to decide what is (are) the node(s) to be added next to the current route in expansion. In the latter, we have a procedure for adding a node (or nodes) to a route in current expansion. We discuss the node insertion process first.

Given an initial route skeleton or a route under current expansion, we say that a given node is *fixed* if it is an OD node (i.e. a node from an OD pair) or a terminal, and *nonfixed* otherwise. We use this terminology since the idea is that once a fixed node is added to a route under current expansion, it will never be removed from that route in posterior insertion steps. On the other hand, nonfixed nodes can potentially be removed from the route at later insertion steps.

The insertion step works as follows. Assume we have a route under current expansion and we want to add a node v to it (see Figure 1). Then, we first find the node w in the current route that is closest to v (see Figure 1a). We have two possible cases: w is either fixed or nonfixed. If w is a nonfixed node (i.e. w is neither a terminal nor an OD node) we look at the fixed node in the route preceeding w (denote it by u_1) and proceeding w (denote it by u_2). That is, the segment $u_1a_1a_2...a_iwa_{i+1}...a_pu_2$ is part of the route, where all the nodes a_i are nonfixed and the nodes u_1, u_2 are fixed. Then, we perform the insertion by removing all the arcs in the route connecting u_1 with u_2 (i.e. the arcs $(u_1, a_1), (a_1, a_2), \dots, (a_p, u_2)$), and adding the two new segments $P_1 := u_1b_1b_2...b_lv$ and $P_2 := vc_1c_2...c_{l'}u_2$, where P_1 and P_2 denote the shortest paths from u_1 to v and from v to u_2 respectively (see Figure 1b). In the case that w is a fixed node, we repeat the same procedure as above and identify the fixed preceeding and proceeding nodes u_1 and u_2 respectively. Also, let P_1 be the shortest path from u_1 to v, P_2 the shortest path from v to u_2, Q_1 the shortest path from v to w, Q_2 the shortest path from w to v, R_1 the shortest path from w to u_2, and R_2 the shortest path from u_1 to w. Notice that R_1 is just the segment in the route joining w and u_2, and R_2 the segment in the route joining u_1 and w. Then we compare the paths $P_1Q_1R_1$ and $R_2Q_2P_2$ and use the shortest of the two alternatives to

expand the route. That is, if $P_1Q_1R_1$ is shorter than $R_2Q_2P_2$, we perform the insertion by removing all the arcs in the route connecting u_1 and w (i.e. R_2), and then adding P_1Q_1 (see Figure 1c). On the contrary (i.e. $R_2Q_2P_2$ shorter than $P_1Q_1R_1$), we perform the insertion by removing all the arcs in the route connecting w and u_2 (i.e. R_1), and then adding Q_2P_2 (see Figure 1d).



(a) Route before inserting v

(b) w nonfixed

(c) w fixed and $P_1Q_1R_1$ shorter

(d) w fixed and $R_2Q_2P_2$ shorter

FIGURE 1. Node insertion

Finally, in the case we want to add two different nodes (e.g. an OD pair) v_1, v_2 to the route instead of only one, we repeat the above (adding a single node) procedure for both v_1 and v_2 separately. This completes the discussion about the node(s) insertion process.

We now discuss the node(s) selection stage. Recall that this step consists of deciding what is (are) the node(s) to be added next to the route in current expansion. We always start with a route skeleton and keep adding nodes until no more can be added. We consider several heuristics based on the idea of either adding the node or pair that is closest to the route (hence having longer routes that contain a larger number of OD nodes) or adding the node or pair with the highest demands (thus having routes that could be potentially shorter but that contain OD nodes and OD pairs with high demand). By a node's demand (or demand associated to a node) we mean the sum of the demands of all OD pairs that contain such node. Several of these ideas came motivated from the work in [1]. We consider the following five node(s) selection strategies:

(1) **Random Shuffle** (RS): Repeat the following twice: Shuffle the OD pairs in a random order. Go over the sorted list of OD pairs and try to add first the OD pair. If this fails (i.e. inserting the OD pair violates the maximum allowed duration), try to add the OD origin

node. If the latter insertion also fails, try to add the OD destination. Move to the next OD pair in the sorted list and repeat.

(2) **Greedy on the Distance** (GD): Try to add the OD node that is closest to the current route in expansion. If it cannot be added stop expanding the route.

(3) **Greedy on the Pairs Demand** (GPD): Presort the OD pairs by demand in decreasing order. Go over the whole sorted list trying to add the OD pairs. Then, go again over the whole list and try to add the OD origin nodes in the cases where the corresponding OD pair could not be added before. Finally, go one more time over the list and try to add the OD destinations in the cases where neither the OD pair nor the OD origin could be previously added.

(4) **Greedy on the Nodes Demand** (GND): Presort the OD nodes by demand in decreasing order. Go over the whole sorted list trying to add the nodes.

(5) **Greedy on the nodes demand/distance ratio** (GNDDR): Try to add the OD node that maximizes the ratio: (node demand / node distance from the route). If it cannot be added stop expanding the route.

Finally, after completing the generation process, we do some postprocessing where we delete routes that are duplicated and routes that do not satisfy the minimum duration constraint. Notice that we never generate routes that exceed the maximum time allowance given the way the insertion step works (i.e. before performing the node(s) insertion we always check whether this would lead to exceeding the maximum time allowance).

2.2. **Route Optimization.** Recall that in the route optimization phase, given a set of candidates routes (previously generated in the route generation phase) the goal is to choose a subset of N routes that minimizes the total travelling time of the passengers. We do this by formulating an integer linear programming model, and solve it using both CPLEX and Lagrangian relaxation. Several comparisons of the performance of these two methods are presented in Section 4.2.

In this phase we have a *walking graph* $G_W = (V, A_W)$ and a *bus routes graph* $G_R = (V, A_R)$. The former comes from the original input data of the problem, where each arc $(u, v) = a \in A_W$ corresponds to going from u to v by walk. The bus routes graph is built after running the route generation phase by putting together all the arcs of the generated routes. That is, if we generated routes (where we see routes as subsets of subsequent arcs) $R_1, R_2, \ldots, R_M$, we let A_R be the union of the R_i's, where if an arc a belongs to p different candidate routes we include p different copies of a in the set A_R, one for each candidate route. So we can think of A_R as containing M disjoint (or non-overlapping) routes. This fact will be important when

formulating the integer program. We now introduce the rest of the notation for the variables and parameters.

x_a^k : n° of people riding a bus on arc a for OD pair k
w_a^k : n° of people walking on arc a for OD pair k
y_r : takes the value 1 if route r is used
u_r : bus capacity in route r
c_a : time it takes to travel arc a by bus
e_a : time it takes to travel arc a by walk
$\delta_R^-(u)$: set of arcs in A_R entering u
$\delta_W^-(u)$: set of arcs in A_W entering u
f_k: demand for OD pair k
o_k: origin node of OD pair k
d_k: destination node of OD pair k
$G_W = (V, A_W)$: walking graph
$G_R = (V, A_R)$: bus routes graph
A_r: set of arcs in route r
$\delta_R^+(u)$: set of arcs in A_R leaving u
$\delta_W^+(u)$: set of arcs in A_W leaving u

Then the integer program can be written as follows:

$$(1)\quad \min \sum_k \sum_{a \in A_R} c_a x_a^k + \sum_k \sum_{a \in A_W} e_a w_a^k$$

subject to

$$(2)\quad \sum_{a\in\delta_R^-(v)} x_a^k + \sum_{a\in\delta_W^-(v)} w_a^k - \sum_{a\in\delta_R^+(v)} x_a^k - \sum_{a\in\delta_W^+(v)} w_a^k = \begin{cases} f_k, & \text{if } v = d_k \\ -f_k, & \text{if } v = o_k \\ 0, & \text{otherwise} \end{cases},$$

for all k, for all $v \in V$

$$(3)\quad \sum_k x_a^k \le u_r y_r\ ,\ \forall r, \forall a \in A_r$$

$$(4)\quad x_a^k \le f_k y_r\ ,\ \forall k, \forall r, \forall a \in A_r$$

$$(5)\quad \sum_r y_r \le N$$

$$(6)\quad 0 \le x_a^k \le \min\{u_r, f_k\}, \quad x_a^k \in \mathbb{Z}, \quad \forall k, \forall r, \forall a \in A_r$$

$$(7)\quad 0 \le w_a^k \le f_k, \quad w_a^k \in \mathbb{Z}, \quad \forall k, \forall a \in A_W$$

$$(8)\quad y_r \in \{0,1\}, \forall r$$

Indeed, notice that the objective is just the total travelling time of the passengers, i.e., total walking time plus total bus riding time. Regarding the constraints, we have that (5) imposses that the number of chosen routes can be at most N. Constraint (3) makes sure that for each route r and each arc a belonging to route r, the bus capacity u_r for that route is not exceeded. Constraint (4) guarantees that if there is flow through arc a in route r then y_r should be greater than zero. Finally, (2) are the well-known flow conservation constraints. A simplification in this model is that it assumes that users can enter/leave a bus at any street corner. This is because the bus stops have not been defined. If the bus stops had been defined, then the nodes of the bus routes graph would be the bus stops.

3. Lagrangian Relaxation

As an alternative to use a commercial integer programming solver, we tried Lagrangian relaxation. We start by explaining the basics of this. Given a linear program

$$\min cx \tag{9}$$
subject to
$$Ax = b \tag{10}$$
$$Dx = r \tag{11}$$
$$x \geq 0, \tag{12}$$

we can multiply the constraints (10) by a set of *Lagrangian multipliers* π and solve

$$\begin{aligned}\theta(\pi) \quad = \quad & \min cx - \pi Ax + \pi b \\ & \text{subject to} \\ & Dx = r \\ & x \geq 0.\end{aligned}$$

Then for any vector π, the value $\theta(\pi)$ is a lower bound for the minimum in (9)-(12). Moreover the maximum of $\theta(\pi)$ over all π, is exactly the value of the minimum in (9)-(12). This procedure is effective if removing constraints (10) makes the resulting linear program much easier to solve. Here we say that constraints (10) have been *dualized*.

In our case, removing constraints (2) makes the problem considerably easier, we explain this below. The flow conservation constraints (2) are dualized with multipliers π_v^k, for $v \in V$ and all k. Let m be the number of OD pairs. Let $\pi = (\pi_v^k) \in \mathbb{R}^{m|V|}$ be the resulting vector of Lagrangian multipliers. We have to solve:

$$\theta(\pi) = \min \sum_k \sum_{a \in A_R} g_a^k x_a^k + \sum_k \sum_{a \in A_W} h_a^k w_a^k + \sum_k f_k(\pi_{d_k}^k - \pi_{o_k}^k) \tag{13}$$
subject to
$$\sum_k x_a^k \leq u_r y_r \ , \ \forall r, \, \forall a \in A_r \tag{14}$$
$$x_a^k \leq f_k y_r \ , \ \forall k, \, \forall r, \, \forall a \in A_r \tag{15}$$
$$\sum_r y_r \leq N \tag{16}$$
$$0 \leq x_a^k \leq \min\{u_r, f_k\}, \quad x_a^k \in \mathbb{Z}, \quad \forall k, \forall r, \, \forall a \in A_r \tag{17}$$
$$0 \leq w_a^k \leq f_k, \quad w_a^k \in \mathbb{Z}, \quad \forall k, \, \forall a \in A_W \tag{18}$$
$$y_r \in \{0, 1\}, \forall r. \tag{19}$$

Here for each arc $a = (u, v) \in A_R$, $g^k_a = c_a + \pi^k_u - \pi^k_v$. And for each $a = (u, v) \in A_W$, $h^k_a = e_a + \pi^k_u - \pi^k_v$. This is called the *Lagrangian subproblem*. In the next subsection we give a simple method to solve this.

3.1. **Solving the Lagrangian subproblem.** For any fixed vector of Lagrangian multipliers π, we have to solve (13)-(19). First notice that variables w^k_a only appear in (18), thus if $h^k_a < 0$ we set $w^k_a = f_k$, and $w^k_a = 0$ otherwise. Also if $g^k_a \geq 0$ we should set $x^k_a = 0$, so in what follows we have to concentrate on the remaining variables.

Because of constraint (16) we have to find N variables y_r that take the value one. Consider a route r and suppose that $y_r = 1$. We have to study constraints (14) to find the values for the variables x. Consider an arc a in route r. We treat the variables x^k_a with the procedure below.

Step 0. Set $\alpha_a = 0$, $U = u_r$, $L = \{k \,|\, g^k_a < 0\}$, $x^k_a = 0$ for all k.
Step 1. If $L = \emptyset$ stop. Otherwise let $l = \text{argmin}\{g^k_a \,|\, k \in L\}$. Remove l from L.
Step 2. If $f_l \leq U$ set $x^l_a = f_l$, otherwise set $x^l_a = U$. Set $U \leftarrow U - x^l_a$, $\alpha_a \leftarrow \alpha_a + g^l_a x^l_a$. Go to Step 1.

This gives tentative values for the variables x^k_a and the number α_a is their possible contribution to the objective function. Then for the route r we define $\beta_r = \sum_{a \in A_r} \alpha_a$. This is the contribution to the objective function obtained by setting $y_r = 1$. Thus we should order the values $\{\beta_r\}$ and pick the N smallest values. This gives the set of routes whose variables should take the value one. For those routes, we use the tentative values defined above for the variables x. All other variables y and x should take the value zero. This procedure is easy to implement and computationally fast.

3.2. **Solving the Lagrangian dual problem.** For a given vector π we have seen how to compute the lower bound $\theta(\pi)$. Now we have to see how to improve this lower bound, this is called the *Lagrangian dual problem*. For that we use the *Volume algorithm* which is an extension of the subgradient method, it produces primal solutions as well as dual solutions, see [2]. It can be seen as a fast way to approximate Dantzig-Wolfe decomposition [9]. The name "volume" is inspired by the fact that the primal values come from computing the volumes below the faces of the dual problem. A description of the algorithm is below.

Volume algorithm

STEP 0. [***Initialization***]
Let π^* be a vector of Lagrangian multipliers.
Solve (13)-(19) with $\pi := \pi^*$. Let $(\hat{x}_0, \hat{w}_0, \hat{y}_0)$ be an optimal solution.
Let $z^* = \theta(\pi^*)$. Set $t := 0$ and $(\bar{x}_0, \bar{w}_0, \bar{y}_0) := (\hat{x}_0, \hat{w}_0, \hat{y}_0)$.

STEP 1a. [***Subgradient Displacement***]

`Let` $\bar{v}_t = b - A^1\bar{x}_t - A^2\bar{w}_t$ `,`
`where` $A^1x + A^2w = b$ `is the system` (2)`.`
`Perform a` *`subgradient displacement`*`:`

$$\bar{\pi} := \pi^* + s_t\bar{v}_t\,,$$

`where the step size` s_t `is discussed below.`

STEP `1b.` **[*Solving the Subproblem*]**
`Solve` (13)-(19) `with` $\pi := \bar{\pi}$`.`
`Let` $(\hat{x}_{t+1}, \hat{w}_{t+1}, \hat{y}_{t+1})$ `be an optimal solution, and` $z_{t+1} = \theta(\bar{\pi})$`.`

STEP `2.` **[*Primal Update*]**
`Compute`

$$(\bar{x}_{t+1}, \bar{w}_{t+1}, \bar{y}_{t+1}) := \alpha\,(\hat{x}_{t+1}, \hat{w}_{t+1}, \hat{y}_{t+1}) \;+\; (1-\alpha)\,(\bar{x}_t, \bar{w}_t, \bar{y}_t). \tag{20}$$

`The value of` α `is discussed below.`

STEP 3 `.` **[*Dual-Test*]**
`Perform the` *`Dual-Test`*`:`

$$\texttt{if } z_{t+1} > z^*, \texttt{ then update } z^* := z_{t+1},\ \pi^* := \bar{\pi}. \tag{21}$$

STEP 4 `.` **[*Loop*]**
`Check stopping criteria. Do` $t := t+1$ `and go to` STEP `1a.`
□

The step size in Step 1a is given by

$$s_t = \lambda \frac{T - z^*}{\|\bar{v}_t\|^2},$$

where $0 < \lambda < 2$, and T is a *target* value. A more complete discussion on how to choose λ appears in [2].

To choose the value of α, an initial value 0.1 is used. Then every 50 iterations, the progress in the objective value is monitored. If the increase is less than 1% and $\alpha > 10^{-5}$, then α is divided by 2.

The stopping criteria was based on a maximum number of iterations, a time limit, and a bound for the gap between the lower bound $\theta(\pi)$ and the value of an integer solution produced by the procedure below (whichever was attained first).

We used the implementation of the Volume algorithm that is in COIN-OR [7].

3.3. **Producing a heuristic solution.** Every ten iterations of the Volume algorithm we use the (fractional) vector $\bar{y}_t$ to produce a $0-1$ vector that defines the set of routes to be open. This heuristic is as follows. For each route r let ν_r be the component of $\bar{y}_t$ associated with route r, we add a random number $\mu_r \in [0,1]$ to ν_r to produce the value γ_r. Then we pick the largest N values $\{\gamma_r\}$, this defines which routes should be open. Given a set of open routes, we treat the OD pairs sequentially. For each of them we find a shortest (time) path from origin to destination, using the walk arcs and the route arcs that are not full. We send flow through this path until an arc is saturated, or until the demand of the OD pair is satisfied. In the former case, we have to look for a new shortest path, and continue until the

demand is satisfied. Once the demand of the OD pair is satisfied, we treat the next OD pair.

4. Computational Results

We now present and discuss the results of our computational experiments. We had to deal with a city of about 55000 inhabitants in the US. Our network has a total of 2487 nodes and 5192 arcs, where three of the nodes are terminals. Our OD-matrix contained travel demand among centroids of different zones. Statistics show that the demand is low, and buses run with only a small occupancy. Thus the OD-matrix had very low values, and we ignored all the ones below a certain threshold. We repeated this procedure for two data sets, one for peak hours and the other for off peak. The first dataset has a total of 160 pairs while the second one has 157. We want to design a total of 16 routes (i.e. $N = 16$). Our bus routes have a minimum duration of 20 mins and a maximum duration of 30 mins. All buses had a capacity of forty passengers, and in our routings this capacity was never reached. All travel times are in minutes. All experiments were performed on a 2.33 GHz Intel Xeon E5410 machine with 32 GB of RAM, running Linux.

We are mainly interested in the following two questions:

(1) What are good heuristics for generating candidate routes?
(2) Is the Volume algorithm a better alternative than CPLEX when solving the integer program?

4.1. **Heuristics Performance.**

We generate candidate routes following each of the five heuristics described in Section 2.1. We do this for the two different datasets of customer demands (i.e. peak hours and off peak). We present the total number of generated routes for each case in Table 1. Recall that we delete duplicated routes and routes that do no meet the minimum duration criterion of 20 minutes.

Table 1. Number of generated routes

		RS	GD	GPD	GND	GNDDR
# Routes	Dataset 1	302	138	173	169	123
	Dataset 2	248	114	135	120	119

In order to study which sets of candidate routes lead to a better solution, we use the Volume algorithm to solve the corresponding integer program (see Section 2.2) associated to each set of candidate routes. We justify this choice (i.e. using Volume instead of CPLEX) later on Section 4.2 where we show that the Volume algorithm runs much faster than CPLEX for instances of this size, and moreover, it also tends to give a better solution than CPLEX.

We run the Volume algorithm with the following stopping criteria (whichever it is reached first): a maximum of either 3 hours, 40 000 iterations, or a 3% gap between the integer solution value and the lower bound. We describe

in Table 2 the results obtained for the first dataset of customer demands, where for each set of candidate routes we show the total travelling time, the best lower bound found, the relative gap between these two, the share (in percentage) of total available memory used by the algorithm, and the total elapsed time (in hh:mm format) before the algorithm stopped.

Table 2. Heuristics comparison (first dataset)

Volume Algorithm (parameters: 3 hours, 40 000 iter, 3% gap)					
	RS	GD	GPD	GND	GNDDR
# Routes	302	138	173	169	123
Travel time	599 729	618 712	595 934	598 629	608 308
Lower bound	574 763	593 712	578 841	578 105	590 746
Relative gap	4.5%	4%	3%	3.5%	3%
Memory	0.5%	0.3%	0.3%	0.3%	0.3%
Time	3:00	2:30	2:50	2:55	2:00

We can observe from Table 2 that the best results are given by the heuristics GPD (Greedy on the Pairs Demand), GND (Greedy on the Nodes Demand) and RS (Random Shuffle), in this order.

Now, we create a new set of candidate routes by combining the N chosen routes (recall that in our case $N = 16$) from each of the five different heuristics discussed above. Certainly, we know that the objective value for this new set of candidate routes will not be worst than the best value attained for each of the heuristics separately. However, there is no obvious reason to think that the N optimal routes from each case will "complement" each other and give an improvement over each of the heuristics considered separately. Surprisingly, this seems to be the case with our data.

Table 3. Combination of optimal routes (first dataset)

# Routes	80	80	80
Travel time	581 043	574 675	568 787
Lower bound	562 735	562 794	562 989
Relative gap	3%	2%	1%
Time	3:00	4:00	5:00

We can observe on Table 3 how by running the Volume algorithm for 3 hours in this new candidate set, we obtain an objective value that is 2.5% better than the best solution we previously had. In addition, we also see how the lower bound in this case is better (i.e. lower) than the ones obtained for each of the heuristics separately, leading to think that the objective value could be further improved. In this spirit, we run the Volume algorithm for an additional 1 and 2 hours respectively, and observe how in fact this is the case. That is, we see how the total travelling time decreases further, and

how the relative gap with the lower bound is closed to 1%. Moreover, notice that there is an almost 5% improvement between the best solution found by combining the N optimal routes from each of the heuristics (with value 568 787) and the one found by optimizing for each heuristic separately (with value 595 934).

We perform the same computational experiments for our second dataset of customer demands. The network data (i.e. nodes, arcs, terminals, etc) stays the same, but we now have a different set of OD pairs (and different demands) containing a total of 157 pairs. Table 4 shows the results obtained for this case.

TABLE 4. Heuristics comparison (second dataset)

Volume Algorithm (parameters: 3 hours, 40 000 iter, 3% gap)					
	RS	GD	GPD	GND	GNDDR
# Routes	248	114	135	120	119
Travel time	472 146	480 593	463 629	482 336	472 907
Lower bound	450 000	466 596	451 051	472 200	460 332
Relative gap	5%	3%	3%	2%	3%
Memory	0.4%	0.3%	0.3%	0.3%	0.3%
Time	3:00	1:10	2:15	1:55	1:45

Notice that for this new dataset of demands the number of generated routes for each heuristic is lower than for the previous dataset (see Table 1). Hence, the Volume algorithm tends to run faster than before, since its running time is proportional to the number of candidate routes that we are optimizing over. In addition, we can observe how the heuristics GPD, RS, and GNDDR give the best results in this order. We create again a new set of candidate routes by combining the N optimal routes chosen for each heuristic. Table 5 shows the results obtained after running Volume over this new set of candidate routes. We can observe how by combining the optimal routes we get again something strictly better, although in this case the improvement is not as significant as for the previous dataset.

TABLE 5. Combination of optimal routes (second dataset)

# Routes	80	80	80
Travel time	462 476	461 946	457 654
Lower bound	445 231	445 868	445 916
Relative gap	4%	3.5%	2.5%
Time	3:00	4:00	5:00

We conclude this section by pointing out that our computational results seem to imply the following:

(1) GPD gives the best results among the different heuristics for generating candidate routes.
(2) Combining the optimal routes from the five different heuristics and optimizing again gives something strictly better than the best value found by optimizing over each candidate set separately.

4.2. **CPLEX vs Volume algorithm.** We now focus on the route optimization part of the problem and compare the performance of two different software approaches: CPLEX and the Volume algorithm. Recall that in the optimization part we are interested in finding the N optimal routes out of a larger number M (usually $N << M$) of candidate routes.

We present in Table 6 the results obtained after running CPLEX on the five different sets of candidate routes (see Table 1) generated from the first dataset of customer demands. We run CPLEX with a time limit of 10 hours for the integer programming optimizer (notice that this does not include the time that it takes to solve the LP relaxation) and a 5% optimality gap.

TABLE 6. CPLEX performance (first dataset)

CPLEX stopping parameters: 10 hr optimizer, 5% gap					
	RS	GD	GPD	GND	GNDDR
# Routes	302	138	173	169	123
Travel time	829 851	888 908	930 000	656 578	598 194
Optimality gap	45%	50%	60%	14%	$<1\%$
Memory	75%	39%	45%	45%	35%
Total Time (LP time)	15:00 (5:00)	18:00 (8:00)	28:00 (18:00)	12:00 (2:00)	14:30 (7:00)
# variables	4 900 000	2 700 000	3 200 000	3 100 000	2 300 000

By comparing these results with the ones obtained by running the Volume algorithm on the same instances (see Table 2), we observe two interesting facts. First, it seems clear that Volume runs much faster than CPLEX for problems of this size. In particular, just solving the LP relaxation in CPLEX seems to take a long time, and in most cases just solving the LP in CPLEX takes even longer than solving the whole problem using Volume. Second, it seems to be the case that Volume gives a much better solution than CPLEX, even though as we mentioned before, it runs for a much shorter amount of time. We should also notice that while Volume uses less than 1% of the total amount of available memory, CPLEX requires 35% or more of the available memory.

We present in Table 7 a case by case comparison between CPLEX and Volume for each of the five sets of candidate routes generated from the first dataset. We show the difference (in %) between the best solution found and between the running times. For instance, in the case of RS heuristic (second column of the table), we write in the first row "Vol 40%" to denote that the travelling time found by Volume is 40% better than the one found by CPLEX, and we write in the second row "Vol 500%" to denote that the running time of Volume is five times faster than the one of CPLEX.

Table 7. Volume vs CPLEX comparison (first dataset)

	RS	GD	GPD	GND	GNDDR
Travel time	Vol 40%	Vol 43%	Vol 53%	Vol 10%	CPLEX 3%
Running time	Vol 500%	Vol 600%	Vol 1400%	Vol 400%	Vol 1000%

We repeat the same computations for the second dataset of demands. That is, we run CPLEX on the five different sets of candidate routes generated from the second dataset and compare these results with the ones obtained by running Volume over the same instances (see Table 4). We present in Table 8 the results obtained with CPLEX and in Table 9 the comparison between CPLEX and Volume.

Table 8. CPLEX performance (second dataset)

CPLEX (parameters: 10 hr optimizer, 5% gap)					
	RS	GD	GPD	GND	GNDDR
# Routes	248	114	135	120	119
Travel time	830 000	478 832	695 365	495 862	468 939
Optimality gap	85%	2%	53%	5%	1%
Memory	70%	38%	38%	38%	38%
Total Time (LP time)	19:00 (9:00)	13:00 (7:00)	18:30 (8:30)	15:00 (7:30)	17:00 (9:00)
# variables	4 200 000	2 300 000	2 600 000	2 400 000	2 300 000

Table 9. Volume vs CPLEX comparison (second dataset)

	RS	GD	GPD	GND	GNDDR
Travel time	Vol 76%	CPLEX <1%	Vol 50%	Vol 5%	CPLEX <1%
Running time	Vol 600%	Vol 1100%	Vol 800%	Vol 800%	Vol 1000%

We can observe again how the Volume algorithm is much faster than CPLEX for this problem. Also, how Volume either gives a (usually much) better solution than CPLEX or in the cases where CPLEX outperforms Volume how this difference is very small (less than 1%).

5. Conclusions

We considered the problem of designing the public bus transit network of a medium size city. The goal is to design a total of N routes so that the overall travelling time of the passengers is minimized. Our approach consisted in breaking down the original problem into two subproblems: route generation and route optimization. In the former we are interested in finding heuristics to generate good candidate routes. In the latter we formulate an integer program that chooses N optimal routes out of a larger number of candidates routes, and we study efficient ways to solve this integer program for our given data.

Our computational results seem to imply that the GPD (Greedy on the Pairs Demand) heuristic is a good way to generate candidate routes. This heuristic is very simple to implement and consists in initially presorting the OD pairs by demand in decreasing order, then try to add first as many pairs as possible, and finally add as many single nodes as possible. In addition, the best results were achieved by first optimizing separately over the candidate set generated by each heuristic, and then combining these five sets of optimal routes into a new set of candidate routes.

On the route optimization part we solve the integer program with both CPLEX and the Volume algorithm and compare their performances. Our computational results in this case suggest that Volume is a better choice than CPLEX for approaching this problem. In particular, Volume was much faster than CPLEX in all the instances that we considered. It also gave much better solutions than CPLEX in many of the instances, while in the few cases where CPLEX outperformed Volume this difference was very small (less than 1% in two cases and 3% in one case). Also the amount of memory that Volume requires is much less than the memory used by CPLEX.

References

[1] M Hadi Baaj and Hani S Mahmassani. Hybrid route generation heuristic algorithm for the design of transit networks. *Transportation Research Part C: Emerging Technologies*, 3(1):31–50, 1995.

[2] Francisco Barahona and Ranga Anbil. The volume algorithm: producing primal solutions with a subgradient method. *Mathematical Programming*, 87(3):385–399, 2000.

[3] Ralf Borndörfer, Martin Grötschel, and Marc E Pfetsch. A column-generation approach to line planning in public transport. *Transportation Science*, 41(1):123–132, 2007.

[4] Michael R Bussieck, Peter Kreuzer, and Uwe T Zimmermann. Optimal lines for railway systems. *European Journal of Operational Research*, 96(1):54–63, 1997.

[5] Avishai Ceder and Nigel HM Wilson. Bus network design. *Transportation Research Part B: Methodological*, 20(4):331–344, 1986.

[6] MT Claessens, Nico M van Dijk, and Peter J Zwaneveld. Cost optimal allocation of rail passenger lines. *European Journal of Operational Research*, 110(3):474–489, 1998.

[7] COIN-OR. `http:www.coin-or.org`.

[8] IBM ILOG CPLEX. V12. 6: Users manual for cplex. *International Business Machines Corporation*, 2015.

[9] George B Dantzig and Philip Wolfe. Decomposition principle for linear programs. *Operations research*, 8(1):101–111, 1960.

[10] Guy Desaulniers and Mark D Hickman. Public transit. *Handbooks in operations research and management science*, 14:69–127, 2007.

[11] Wei Fan and Randy B Machemehl. Optimal transit route network design problem with variable transit demand: genetic algorithm approach. *Journal of transportation engineering*, 132(1):40–51, 2006.

[12] Wei Fan and Randy B Machemehl. Using a simulated annealing algorithm to solve the transit route network design problem. *Journal of transportation engineering*, 132(2):122–132, 2006.

[13] Jan-Willem Goossens, Stan Van Hoesel, and Leo Kroon. A branch-and-cut approach for solving railway line-planning problems. *Transportation Science*, 38(3):379–393, 2004.

[14] Konstantinos Kepaptsoglou and Matthew Karlaftis. Transit route network design problem: review. *Journal of transportation engineering*, 135(8):491–505, 2009.
[15] Anita Schöbel. Line planning in public transportation: models and methods. *OR spectrum*, 34(3):491–510, 2012.
[16] Anita Schöbel and Susanne Scholl. Line planning with minimal traveling time. In *ATMOS 2005-5th Workshop on Algorithmic Methods and Models for Optimization of Railways*. Internationales Begegnungs-und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, 2006.
[17] F Zhao and X Zeng. Optimization of transit network layout and headway with a combined genetic algorithm and simulated annealing method. *Engineering Optimization*, 38(6):701–722, 2006.

(F. Barahona) IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598

(J. Gonçalves) IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598

(R. Santiago) McGill University, Montreal, Canada

(C. W. Wu) IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598