Decision Support for Vehicle Dispatching Using Genetic Programming

Ilham Benyahia and Jean-Yves Potvin

Abstract—**Vehicle dispatching consists of allocating real-time service requests to a fleet of moving vehicles. In this paper, each vehicle is associated with a vector of attribute values that describes its current situation with respect to new incoming service requests. Using this attribute description, a utility function aimed at approximating the decision process of a professional dispatcher is constructed through genetic programming. Computational results are reported on requests collected from a courier service company and a comparison is provided with a neural network model and a simple dispatching policy.**

I. Introduction

In the transportation sector, vehicle dispatching is a vital activity which must be effectively and efficiently performed. Broadly speaking, this activity is concerned with the allocation of vehicles to transportation requests that (often) require immediate service. Many different types of vehicle dispatching problems are found in practice. Typical dispatching contexts include demand-responsive transportation systems (e.g., transportation for the handicapped or the elderly) and emergency services (e.g., police, ambulances). In this paper, the focus is on a particular application domain—a courier service operating in an urban area. The interested reader will find an exhaustive review of different types of time-constrained vehicle routing problems in [4]. A good analysis and overview of problems where each service request comprises both a pickup and delivery location may also be found in [17].

Vehicle dispatching is still mostly performed by human experts because the dispatching task is very complex and requires high-level cognitive abilities. Typically, the professional career of a dispatcher lasts only a few years due to the stress associated with reacting properly and quickly to a dynamically evolving environment. A computerized decision support system would thus be of great help. Our own experience has shown that it is very difficult to formalize the dispatching task using mathematical models or explicit decision rules. This difficulty stems from the multiplicity of quantitative and qualitative factors that must be weighted to achieve a good tradeoff between conflicting requirements. Hence, adaptive techniques may be a promising solution. Given that human dispatchers are still unmatched with regard to the decision quality, it is natural to try to "mimic" their behavior. This paper thus focuses on genetic programming techniques to develop a decision support system for vehicle dispatching—a population of utility functions that evaluate candidate vehicles for servicing new requests evolves from one generation to the next to best approximate the dispatcher's decision process.

This paper is organized as follows. Section II introduces the courier service application. The multiattribute choice model is presented in Section III. Section IV provides an overview of genetic algorithms and genetic programming. Section V details how the problem is solved within the genetic programming framework. Section VI presents computational results obtained on real data collected from a courier service company. This section also compares genetic programming with a neural network model and a simple dispatching policy.

II. A Courier Service

In the context of a courier service company, the dispatching problem can be described as follows. The company receives calls for the pickup and delivery of express mail within an urban area. Each customer specifies a specific pickup and delivery location, as well as a soft upper bound on the service time at each location (e.g., a letter should be collected in the next 30 min and delivered in the next 90 min). Since customers demand fast service, the routing and scheduling is done in real time. In particular, the dispatching situation must be evaluated each time a new customer calls into the dispatch office. At that time, some of the earlier requests have been delivered and therefore are no longer considered. The remaining requests have been assigned to the vehicles and are about to be picked-up, or are heading to their destination. Since the planned route and schedule of each vehicle for previously allocated requests is known, the problem is to determine the assignment of the new request to a particular vehicle, as well as the new planned route and schedule for this vehicle. In solving this problem, the dispatcher must find a good compromise between conflicting objectives like minimizing operations cost (e.g., fuel consumption), maximizing system's throughput (e.g., number of serviced customers), and maximizing service quality (e.g., meeting due dates).

Note that the dispatcher may wait before dispatching a new request to gather additional information that could lead to a better decision. For example, a batch of new requests can be collected over a certain time period. These requests are then dispatched at once. This approach alleviates the undesirable

Manuscript received September 22, 1995; revised July 18, 1996 and July 5, 1997. This work was supported by the Canadian Natural Sciences and Engineering Research Council (NSERC) and by the Quebec Fonds pour la Formation de Chercheurs et l'Aide à la Recherche (FCAR).

I. Benyahia was with the Centre de Recherche sur les Transports, Université de Montréal, Montréal, P.Q., Canada H3C 3J7. She is now with the Institut de recherche d'Hydro-Québec, Varennes, P.Q., Canada J3X 1S1.

J.-Y. Potvin is with the Centre de Recherche sur les Transports and the Département d'Informatique et de Recherche Opérationnelle, Université de Montréal, Montréal, P.Q., Canada H3C 3J7.

Publisher Item Identifier S 1083-4427(98)02747-7.

"myopic" behavior associated with dispatching requests one at a time. Although this strategy is sometimes exploited by vehicle dispatchers when enough time is available, this paper will only consider the sequential insertion of new requests.

A few algorithmic approaches are reported in the literature for different types of dispatching problems. A dynamic programming algorithm for the single vehicle case is described in [13]. The multivehicle case is addressed in [6] for pickup (only) or delivery (only) routes, using a tabu search heuristic. Specialized algorithms are also found for demand-responsive transportation systems [20], [21]; bulk delivery of industrial gases [1]; dispatching of petroleum tank trucks [2]; and shipping of troops and material in mobilization situations [14], [15]. Related work for the dynamic allocation of freight carriers and for the management of emergency vehicles, where each vehicle is dedicated to a single customer, may also be found in [12] and [19].

A different approach to this problem is described in [18]. This work uses a backpropagation neural network model [16] to learn the decision process of an expert dispatcher after a training phase with this expert. Through the learning mechanism, the system can adapt to different dispatching environments. In [11], a learning approach based on linear programming is also described. Since only linear or piecewise linear functions can be constructed to approximate the dispatcher's decision process, this approach did not prove to be competitive with the neural network model.

In this paper, an alternative adaptive approach is proposed: vehicle dispatching is seen as a multiattribute choice problem and genetic programming techniques are used to find a utility function that best approximates the dispatcher's decisions. Hence, the expertise is encoded in the final genetic program (or utility function) rather than the connection weights of a neural network. Furthermore, utility functions derived through genetic programming are only restricted by the set of mathematical functions available for constructing the programs.

III. Multiattribute Choice

Before introducing the multiattribute choice problem, we first illustrate a typical dispatching situation for a single vehicle. In Fig. 1, k and k' are the pickup and delivery locations, respectively, of customer k, $k = 1, 2, 3$. Currently, the vehicle is at location X and is planned to service customers 1 and 2 in the following order: pickup customer 1, pickup customer 2, deliver customer 2, and deliver customer 1. Then, customer 3 calls in. This customer is inserted at the location that minimizes the insertion cost $\Delta d + \Delta l$, where Δd and Δl are the modifications to the total travel time on the route and total lateness at customer locations, respectively [18]. In this example, Δd would be equal to

$$(d_{13} + d_{32} - d_{12}) + (d_{2'3'} + d_{3'1'} - d_{2'1'})$$

where d_{ij} is the travel time between i and j. The two components of the sum are the detour introduced by the pickup and delivery points, respectively. Then, Δl would correspond to the sum of additional lateness at service points 2, 2', and 1' plus any lateness at 3 and 3'. Note that the service time at

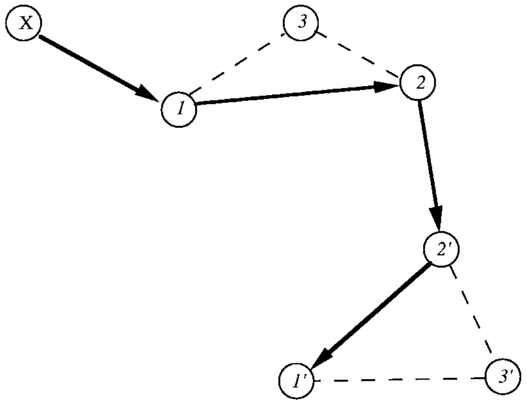


Fig. 1. Insertion of a new request in a planned route.

point 1 cannot be delayed, and thus cannot be associated with any additional lateness, since it is located before the insertion place of the new pickup point.

The simple insertion of each new request, without any reoptimization of the planned route, closely models the behavior of expert dispatchers in practice. This approach minimizes the amount of disruption to the planned route which still conforms to the "mental model" of the dispatcher (e.g., general shape and orientation). Furthermore, too much disruption would hardly be accepted by drivers.

The multiattribute choice problem can now be formalized as follows for a set $V = \{V_1, V_2, \cdots, V_n\}$ of vehicles. Whenever a new request is received, a vector of m attribute values $(a_{j1}, a_{j2}, \cdots, a_{jm})$ is associated with each vehicle $V_j, j = 1, \cdots, n$. These attribute values are obtained from the best insertion place of the new request in their planned route (as defined above). They are:

1) DE	Detour (in time) to service the new request.
2) PT	Pickup time of the new request.
3) DT	Delivery time of the new request.
4) *Avg. P. Lat.*	Average additional lateness at the pickup locations on the planned route, due to the insertion of the new request.
5) *Avg. D. Lat.*	Average additional lateness at the delivery locations on the planned route, due to the insertion of the new request.
6) *#P. Lat.*	Number of additional late services at the pickup locations on the planned route, due to the insertion of the new request.
7) *#D. Lat.*	Number of additional late services at the delivery locations on the planned route, due to the insertion of the new request.
8) *#Req.*	Number of requests on the planned route.
9) *Et. Tr.*	Ratio of additional empty travel time on total travel time due to the insertion of the new request. A vehicle is running empty when it is heading to the pickup point of the new request and there is no courier on board.

Attribute 1) emphasizes low operations costs and high throughput by favoring short routes; attributes 2)–7) emphasize

timely service to customers, which is a prominent issue for courier services; attributes 8) and 9) emphasize drivers' satisfaction through a fair distribution of the workload and a reduction of empty travel time (note that drivers are paid a commission for each serviced request). This set of attributes was determined in cooperation with an experienced dispatcher working in a courier service company. The same set was used for the neural network experiments described in [18].

Since the attributes are expressed in different units, all values are normalized between 0 and 1. This is done as follows for each attribute i and vehicle j:

$$a''_{ji} = 1 - \frac{a'_{ji}}{\max_{k=1,\cdots,n} a'_{ki}}$$

where

$$a'_{ji} = a_{ji} - \min_{k=1,\cdots,n} a_{ki}, \qquad j = 1, \cdots, n; \quad i = 1, \cdots, m.$$

That is, the gap between the current attribute value and the minimum (best) attribute value over all vehicles is first computed. This gap is then divided by the largest gap over all vehicles. The final value is subtracted to one, so that better vehicles have higher attribute values. In particular, the normalized attribute values associated with the best and worst vehicles are 1 and 0, respectively.

The attribute description of each vehicle provides a context for decision making. The objective of the vehicle dispatching task is to select a suitable vehicle in a given context for servicing a new request. This decision process can thus be modeled as a function F from the set of contexts C to the set of candidate vehicles V:

F: $C \rightarrow V$ where

$$(a_1, a_2, \cdots, a_n) \in C$$

$$\begin{aligned} a_j &= (a_{j1}, a_{j2}, \cdots, a_{jp}, \cdots, a_{jm}), & j &= 1, 2, \cdots n, \\ a_{jp} &= \text{attribute value } p \text{ of vehicle } V_j, & p &= 1, 2, \cdots, m. \end{aligned}$$

Genetic programming is used to derive such a function through a population of evolving programs. In Section IV, we briefly review the main concepts of genetic programming.

IV. Genetic Programming

Genetic programming is a recent development in the field of evolutionary algorithms which extends the genetic algorithm paradigm to nonlinear structures [9]. A genetic algorithm is a randomized search procedure working on a population of individuals (solutions) encoded as linear bit strings. This population evolves over time through the application of operators that mimic those found in nature: selection of the fittest, crossover, and mutation [7], [8]. First, "parent" chromosomes are selected in the population for reproduction. The selection process is probabilistic and biased toward the best individuals (to propagate good solution features to the next generations). Then, the crossover operator combines the characteristics of pairs of parent chromosomes to create new offspring in the hope that these offspring will have higher fitness than their parents. An example of one-point crossover is shown in Fig. 2, where the cross point is randomly chosen between the second and third bit. In this case, the end parts of the parent chromosomes are exchanged to create two new offspring.

$$\begin{array}{ccccccl} 1 & 0 & | & 1 & 0 & 0 & \text{(parent 1)} \\ \underline{0} & \underline{0} & | & \underline{1} & \underline{1} & \underline{1} & \text{(parent 2)} \\ \hline 1 & 0 & & \underline{1} & \underline{1} & \underline{1} & \text{(offspring 1)} \\ \underline{0} & \underline{0} & & 1 & 0 & 0 & \text{(offspring 2)} \end{array}$$

Fig. 2. One-point crossover on two bit strings.

Finally, the mutation operator is applied to each offspring. This operator processes the offspring position by position and flips the bit value from 0 to 1, or from 1 to 0, with a small probability at each position. Mutation is viewed as a "background" operator that slightly perturbs a small proportion of solutions. It is used to maintain or restore diversity in the population and to guarantee that every state of the search space is accessible from the current state.

Through this selection/crossover/mutation process, it is expected that an initial population of randomly generated individuals will improve as parents are replaced by better offspring. Genetic algorithms have been used to solve difficult problems in many different domains. A sample of applications may be found, for example, in [3] and [10].

Genetic programming extends the above paradigm by allowing the evolution of programs encoded as tree structures. These programs are constructed from a predefined set of functions and terminals (which can be either variables, like the state variables of a particular system, or constants, like integer 3 or Boolean *False*). The evolution of programs within the genetic programming framework can be summarized as follows.

Step 1. *Initialization*: Create an initial random population of P programs and evaluate the fitness of each program by applying it on a set of *fitness cases* (examples). Set the current population to this initial population.

Step 2. *Selection*: Select P programs in the current population (with replacement). The selection process is probabilistically biased in favor of the best programs.

Step 3. *Modification*: Apply reproduction or crossover to the selected programs.

Step 4. *Evaluation*: Evaluate the fitness of each offspring program in the new population.

Step 5. Set the current population to the new population of offspring programs.

Step 6. Repeat steps 2–5 for a prespecified number of generations or until the system does not improve anymore.

The final result is the best program generated during the search. The main components of this algorithm are

1) encoding of programs as tree structures;
2) generation of the initial population;
3) fitness evaluation;
4) selection;
5) reproduction and crossover;
6) setting of control parameters.

In the following subsections, each component is briefly described.

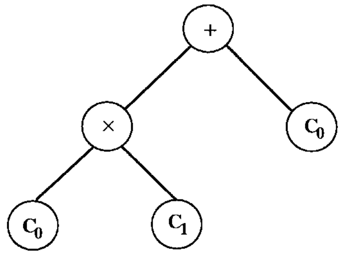


Fig. 3. A tree structure for a program.

A. Encoding of Programs as Tree Structures

The individual structures that undergo adaptation are computer programs expressed as tree structures (or S-expressions when the programs are constructed using Lisp code). In the courier service application, for example, these structures encode utility functions that evaluate the suitability of a vehicle to service a new request based on its attribute description.

The size, shape, and contents of the trees can dynamically change during the evolution process. They are made up of all possible composition of functions that can be constructed recursively from a set G of functions and a set T of terminals. Each particular function g_i in the function set takes a specified number of arguments and should be able to accept as argument the value returned by any other function in the set (which is typically the case for numerical applications). Fig. 3 illustrates a program for $G = \{+, \times\}$ and $T = \{C_0, C_1\}$.

Such a program computes $C_0 \times C_1 + C_0$. For many problems, C_0 and C_1 would correspond to numerical constants. Since there is no way to know in advance what constants are needed to solve a given problem, an ability to arbitrarily create such constants must be provided. In [9], this problem is addressed by adding a special terminal, called the ephemeral random constant R. This terminal is transformed into a random number in a prespecified interval whenever it is found within a program. Typically, ephemeral constants are generated in the initial population of programs only. New constants are created in the following generations through the action of crossover or other secondary operators (as described below).

B. Generation of the Initial Population

Typically, the initial population is made of randomly generated programs or trees. One of the three following methods is used to create the initial population.

1) *Full*: This method creates trees such that the length of every path between a terminal and the root is *equal* to a prespecified maximum depth.
2) *Grow*: This method creates trees of variable shapes. That is, the length of a path between a terminal and the root is *at most* a prespecified maximum depth.
3) *Ramped*: This is a mix of *Full* and *Grow*. That is, an equal number of both types of trees are created in the initial population.

C. Fitness Evaluation

The fitness of a program is evaluated by applying this program to a set of fitness cases and by measuring the correspondence between the desired responses and the responses produced by the program. For example, the correspondence value could be set to 1 if the two responses are the same and to 0 otherwise. Hence, the fitness would be related in this case to the number of fitness cases that are correctly handled. Obviously, other measures could be devised as well (e.g., based on the magnitude of the difference between the two responses).

D. Selection

Different methods have been reported in the literature for selecting programs that either reproduce or are transformed by crossover. These methods are described below.

1) Fitness-Proportionate Selection: Under this scheme, the selection probability of program i is

$$p_i = \frac{f_i}{\sum_{p=1}^{P} f_p}$$

where f_i is the fitness of program i and the denominator is the sum of fitness values over all programs in the current population. Hence, the selection probability of a given program is proportional to its fitness and the best programs are more likely to be selected for reproduction and crossover.

Some problems are reported with fitness-proportionate selection. For example, a “super-program,” the fitness of which is much higher than the fitness of other programs in the current population, will typically be overselected (leading to premature convergence to a possibly suboptimal solution). Conversely, fitness-proportionate selection behaves as a random selection process when the fitness values of all programs in the current population are similar (preventing exploitation of the marginal differences). Hence, new selection methods like rank selection and tournament selection have been devised in order to reduce the difference between fitness values when a super-program is found in the population, or to increase this difference when fitness values are similar.

2) Tournament Selection: Under this scheme, two individuals are selected at random from the current population and the one with better fitness is selected. Consequently, the selection probability of a super-program is the same as the one of any other program in the population (although it will surely win the competition if it is selected). Furthermore, a marginally better fitness is sufficient to win a competition. Since tournament selection does not emphasize the magnitude of the difference between fitness values, but rather focuses on their relative ordering (i.e., which one is better), tournament selection alleviates the problems observed with fitness-proportionate selection.

3) Rank Selection: Here, selection is based on the rank of the fitness value of each program in the population. First, all programs in the current population are sorted from best (rank 1) to worst (rank P). A new fitness value is then associated with the program of rank i as follows:

$$f_i = \text{Max} - \left[(\text{Max} - \text{Min})\frac{i-1}{P-1}\right].$$

Hence, the best program gets fitness Max, the worst program gets fitness Min, and the fitness values of the remaining programs are equally spaced between Min and Max. In the computational results of Section VI, Max and Min were set to 1.5 and 0.5, respectively. For example, when the population is composed of $P = 5$ programs, these programs are evaluated from best to worst as follows: $f_1 = 1.5, f_2 = 1.25, f_3 = 1.0, f_4 = 0.75$, and $f_5 = 0.5$. Fitness-proportionate selection is then invoked with these new values.

Like tournament selection, rank selection separates the selection process from the original distribution of fitness values. It exaggerates the difference between almost identical fitness values and minimizes the effect of a super-program by bounding its fitness to the Max value.

E. Reproduction and Crossover

A new generation of programs is obtained through the action of reproduction and crossover. Other secondary operators can also be applied to the programs (like mutation). However, to the extent that these operators are aimed at restoring lost diversity in a population, they are not really needed in genetic programming since 1) the functions and terminals are not associated with fixed positions in a fixed structure like the chromosomes in genetic algorithms and 2) there are relatively few functions and terminals to construct the programs, so it is unlikely for a particular element to disappear from the population [9]. Accordingly, only the reproduction and crossover operators are considered in the following.

1) Reproduction: Reproduction simply copies a selected program to the new population without any modification (where the selection process is based on the methods presented in the previous subsection).

2) Crossover: Crossover applies to two selected programs. A crossover point, i.e., a function or terminal within the tree, is randomly selected within each parent. This crossover point defines a fragment which is the subtree consisting of the crossover point plus the entire subtree lying below the crossover point. Two new offspring programs are then produced by exchanging the two fragments, as depicted in Fig. 4. In this example, the functions $\div$ and $\times$ are selected in the first and second tree, respectively. Then, the subtrees are exchanged to create two new programs. Obviously, when both crossover points are terminals (i.e., leaves of their respective tree), the effect of crossover is simply to exchange the two terminals. Typically, the probability distribution for the selection of a crossover point favors the exchange of subtrees rather than terminals.

F. Setting of Control Parameters

Genetic programming is controlled by many parameters. Two important parameters are population size and number of generations. However, many other parameters must also be set such as the following:

1) initialization method;
2) selection method;
3) probability of reproduction and crossover;
4) probability of choosing an internal point or a terminal during crossover;

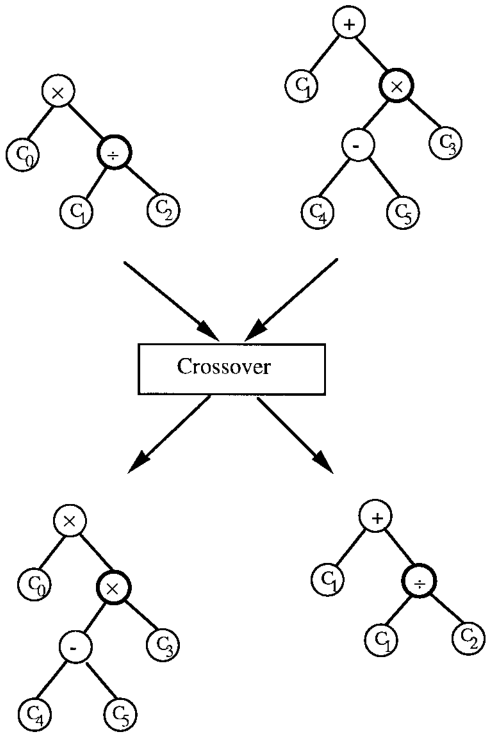


Fig. 4. Crossover in genetic programming.

5) maximum depth for the programs in the initial population;
6) maximum depth for the programs in the following generations.

The settings used in our experiments are described in Section VI.

V. Solving the Courier Service Application Using Genetic Programming

In this section, we explain how genetic programming is applied to our problem. To this end, we follow the general presentation of Section IV, but we restrict ourselves to the components that really need further explanation.

A. Encoding Programs as Tree Structures

Here, a program computes a utility function. This function combines the attribute values associated with a given vehicle to assess the suitability of this vehicle to service a new request. Hence, the set of terminals are the nine attributes introduced in Section III plus the ephemeral constant R (the value of which is generated in the interval $[-1, +1]$). The functions are the mathematical operators $\{+, -, \times, \div\}$. A typical program is illustrated in Fig. 5. In this case, the (normalized) detour value is multiplied by 0.9, the (normalized) pickup time is divided by 0.7, and both results are summed up to obtain the utility of a vehicle for servicing the current request.

B. Fitness Evaluation

Fitness evaluation is based on a set of fitness cases. Here, each fitness case is a dispatching situation involving a new request and n vehicles (with $n = 12$ in our application).

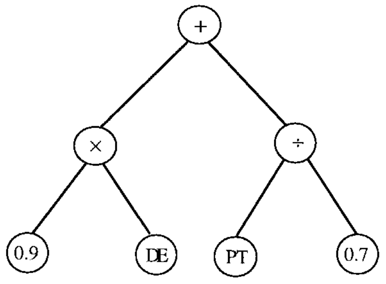


Fig. 5. A program.

TABLE I
FITNESS OF A PROGRAM BASED ON THE RANK GIVEN BY THIS PROGRAM TO THE VEHICLE SELECTED BY THE DISPATCHER

Rank	1	2	3	4	5	6	7	8	9	10	11	12
Fitness	150	130	110	70	50	30	20	15	10	0	0	0

A program must be applied to each fitness case and to each vehicle within a fitness case to evaluate its overall fitness. The vehicles are first ranked within each fitness case according to their utility (as computed by the program). Then, the rank of the vehicle selected by the dispatcher in each fitness case is used to compute the overall fitness of the program (as explained below). Two different types of fitness functions were tested:

1) $F_1(i,j)$ sets the fitness of program i to 1 on fitness case j if the vehicle selected by the dispatcher is ranked first by the program. The fitness is set to 0 otherwise.
2) $F_2(i,j)$ sets the fitness of program i on fitness case j according to Table I where the values monotonically decrease from rank 1 to rank 12. Preliminary experiments have shown that genetic programming is rather insensitive to the exact value associated with each rank, as long as a sufficiently large gap is observed between the highest and lowest ranks.

The overall fitness of a given program i over r fitness cases (service requests) is computed as follows:

$$f_i = \sum_{j=1}^{r} F_1(i,j) \quad \text{or} \quad f_i = \sum_{j=1}^{r} F_2(i,j).$$

Accordingly, the maximal fitness for any given program i is r when F_1 is used and $150 \times r$ when F_2 is used. Clearly, function F_1 emphasizes perfect matches between the dispatcher and the program. That is, the function does not care about the rank given by the program to the vehicle selected by the dispatcher, except if it is rank 1 (i.e., "all or nothing" approach). Function F_2 allows a better average performance over the set of fitness cases. By summing up over all fitness cases, programs that assign high ranks to vehicles selected by the dispatcher over the entire set of fitness cases are favored (even if fewer vehicles are ranked first).

C. Stopping Criterion

The genetic programming algorithm stops when the maximum fitness value or the maximum number of generations is reached.

D. An Example

A small example with $n = 3$ vehicles and $r = 2$ fitness cases is now provided to fix the ideas. We suppose that the fitness function is F_2 (using only the first three ranks) and the program or utility function to be evaluated is

$$DE. + PT. + DT. + Avg.\ P.\ Lat. + Avg.\ D.\ Lat. \\ + \#P.\ Lat. + \#D.\ Lat. + \#Req. + Et.\ Tr.$$

Let assume that the attribute vector a_j associated with each vehicle $j = 1, 2, 3$ on the two fitness cases are

	Evaluation
fitness case 1	
$j = 1$: (1.0, 0.6, 0.0, 0.7, 0.9, 1.0, 0.4, 0.0, 1.0)	5.6
$j = 2$: (0.6, 1.0, 1.0, 1.0, 0.0, 0.0, 1.0, 0.3, 0.0)	4.9
$j = 3$: (0.0, 0.0, 0.5, 0.0, 1.0, 1.0, 0.0, 1.0, 0.2)*	3.7
fitness case 2	
$j = 1$: (1.0, 1.0, 0.0, 0.3, 1.0, 1.0, 0.1, 0.0, 1.0)*	5.4
$j = 2$: (0.2, 0.6, 0.4, 1.0, 0.8, 0.0, 0.0, 0.3, 0.0)	3.3
$j = 3$: (0.0, 0.0, 1.0, 0.0, 0.0, 0.3, 1.0, 1.0, 0.5)	3.8.

The vector marked with a $*$ symbol corresponds to the vehicle selected by the dispatcher, while the evaluation of each vehicle by the utility function is found at the end of each line (namely, the sum of all values in the vector). On the first fitness case, the vehicle chosen by the expert is ranked third by the program. A fitness of 110 is obtained by looking at rank 3 in Table I. On the second fitness case, the vehicle chosen by the dispatcher is ranked first by the program. Thus, a fitness of 150 is obtained and the overall fitness of this program is $110 + 150 = 260$.

VI. COMPUTATIONAL RESULTS

We tested the genetic programming paradigm on $r = 140$ service requests collected from a courier service company located in Montréal, P.Q., Canada, using a fleet of $n = 12$ vehicles. The data were collected from 9H00 in the morning to 15H00 in the afternoon. As illustrated in Fig. 6, the distribution of requests is not homogeneous along the time line. In particular, there is a substantially lower number of requests between 12H00 and 13H00. Each request had to be collected 30 min and delivered 90 min after the customer's call, respectively.

The travel times between service points were estimated using a previously computed "distance table." To obtain a table of reasonable size, the network was divided into 502 zones of one square kilometer. Within each zone, a major street intersection was identified as its "centroid." The exact distance between each pair of centroids was then computed using an adaptation of Dijkstra's shortest path algorithm [5].

These distances were exploited to estimate the travel times between two service points in the network. The distance found in the table was first slightly modified because the

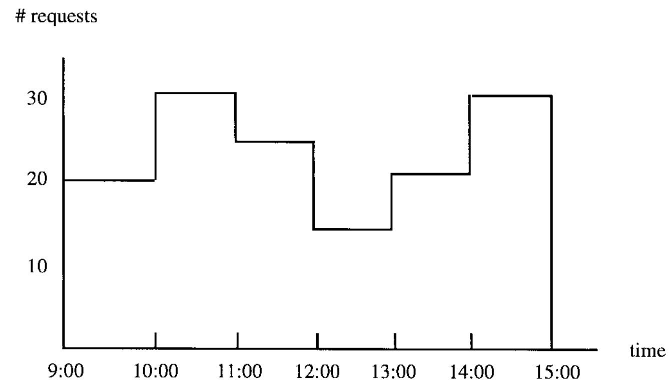


Fig. 6. Distribution of requests along the time line.

TABLE II
PARAMETER SETTINGS

Population size	100 250 500
Number of generations	100
Initialization method	Full Grown Ramped
Selection method	Fitness-proportionate Rank Tournament
Probability of reproduction	10%
Probability of crossover	90%
Probability of choosing an internal point during crossover	90%
Probability of choosing a terminal during crossover	10%
Maximum depth of the programs in the initial population	6
Maximum depth of the programs in the following generations	17

TABLE III
RANKS ASSIGNED BY THE BEST PROGRAM TO THE VEHICLES

	P=100			P=250			P=500		
Rank	140/0	90/50	50/0	140/0	90/50	50/0	140/0	90/50	50/0
1	63	23	34	68	25	36	73	25	39
2	20	10	7	18	10	5	12	9	2
3	15	4	2	20	3	2	21	5	0
4	7	3	3	10	4	0	12	6	2
5	15	1	2	2	5	2	6	1	4
6	4	4	1	8	0	3	2	2	0
7	1	2	0	5	0	0	4	1	1
8	1	1	0	1	2	1	5	0	0
9	4	0	0	1	1	1	2	0	1
10	3	0	0	4	0	0	3	1	1
11	6	1	1	3	0	0	0	0	0
12	1	1	0	0	0	0	0	0	0

TABLE IV
RANKS ASSIGNED BY THE BEST PROGRAM TO THE VEHICLES

	P=100			P=250			P=500		
Rank	140/0	90/50	50/0	140/0	90/50	50/0	140/0	90/50	50/0
1	56	21	32	61	22	33	65	25	33
2	34	13	11	31	14	13	34	15	13
3	20	6	5	20	5	4	18	4	4
4	10	2	0	10	3	0	11	3	0
5	6	2	0	5	3	0	3	0	0
6	5	1	0	2	2	0	2	2	0
7	5	2	0	3	0	0	2	0	0
8	2	1	0	2	0	0	3	0	0
9	1	1	0	3	1	0	1	0	0
10	0	0	0	2	0	0	1	1	0
11	1	1	0	1	0	0	0	0	0
12	0	0	0	0	0	0	0	0	0

service points are not exactly located at the centroid within their respective zone. Then, an "average speed function" was applied to estimate the travel time. This function is such that the average speed increases with the distance between the service points (since the driver is more likely to use large avenues or highways when two service points are far apart). See [18] for details.

Table II shows the parameter settings of the genetic programming algorithm. Note that most values are suggested in [9].

As indicated in the table, three different initialization and selection methods were tested, as well as three different population sizes. Preliminary experiments have shown that no significant improvement is obtained after 100 generations with the largest population size $P = 500$. The same number of generations was kept for smaller population sizes (although, it would have been possible to reduce this number, since convergence occurs earlier on smaller populations).

In our experiments, the algorithm was first applied on the entire set of requests and the best program was reported at the end (140/0). The algorithm was also applied on a training set of 90 requests and the best program was tested on the 50 remaining requests (90/50). Note that the training and testing sets were the same as those used in [18] (where the training set was randomly selected among the entire set of requests). Finally, the genetic programming algorithm was applied on the 50 requests found in the testing set of the second experiment and the best program was reported at the end (50/0).

The results are summarized in Tables III and IV for fitness functions F_1 and F_2, respectively. For each population size and type of experiment, the tables show the best solution over nine different runs (i.e., 3 initialization methods $\times$ 3 selection methods).

In most cases, *tournament* selection provided the best results. Only four exceptions (rank selection three times, *fitness-proportionate* selection once) are found over the 18 solutions shown in Tables III and IV. Furthermore, *full* and *ramped* initialization are associated with the best results in all cases but one.

Function F_1 generally produces a larger number of perfect matches between the dispatcher and the program than function F_2. However, a more "graceful" degradation over the ranks is observed with F_2 (as illustrated in Fig. 7 for the 140/0 experiments). Note that the vehicle selected by the dispatcher is ranked quite low in a few instances. This situation occurs when most vehicles look equally good (or equally bad) to

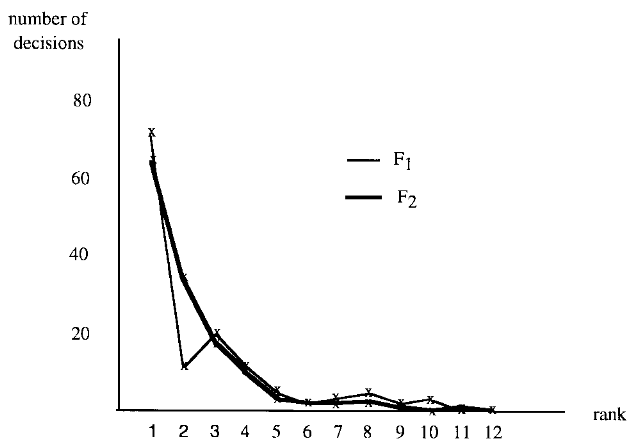


Fig. 7. Number of decisions associated with each rank for F_1 and F_2 $(P = 500)$.

service the new request and cannot be usefully discriminated, even by the dispatcher.

The tables show that better solution quality is obtained through an increase in population size, particularly in the 140/0 experiments. The 90/50 and 50/0 experiments also indicate that a program obtained through genetic programming performs better on its own set of fitness cases than on a new, previously unseen, set of fitness cases (as expected).

Genetic programming has clearly produced useful programs. However, the distribution of ranks obtained with function F_2 is nicer and would probably be more appropriate if the goal of the program is to focus the attention of the dispatcher on a certain number of top ranked alternatives (leaving the final choice to the dispatcher).

The best programs produced by genetic programming are rather intricate and are thus difficult to interpret. However, the program produced by *ramped* initialization and *fitness-proportionate* selection for the 90/50 experiment with function F_1 and $P = 500$ is simpler and is as follows:

$$PT. \times (Avg.\ D.\ Lat. + DT. + DT.^2 \times Avg.\ P.\ Lat.^2 \\ \times (\#D.\ Lat. - DE.) - \#D.\ Lat).$$

This program uses seven attributes (out of nine) and does not contain any numerical constant. However, this is not typical of the best programs found in other experiments. Many numerical constants are contained in our programs and the nine attributes of Section III are used. The difficulty of interpreting programs obtained through genetic programming is certainly a weakness of this approach, although it is shared by other adaptive algorithms (e.g., the connection weights in neural networks).

Genetic programming is computationally expensive in this application, because each program in the population must be evaluated over a substantial number of fitness cases and each fitness case contains a substantial number of candidate vehicles. The computation times vary mostly according to the population size and initialization methods. Table V shows the average computation time in seconds (over the three selection methods and two fitness functions) for the 140/0 experiments for each combination of values, using a Sun Sparc 10 workstation.

TABLE V
AVERAGE COMPUTATION TIMES IN SECONDS FOR THE 140/0 EXPERIMENTS

Initialization	P=100	P=250	P=500
Full	3044	8551	18166
Grow	1408	2751	6470
Ramped	1839	4274	9941

TABLE VI
RANKS ASSIGNED BY DIFFERENT METHODS TO THE VEHICLES SELECTED BY THE DISPATCHER

Rank	Dispatching rule	Neural network	Genetic programming	
			F_1	F_2
1	12	20	25	25
2	10	14	9	15
3	5	6	5	4
4	8	7	6	3
5	5	1	1	0
6	5	0	2	2
7	0	1	1	0
8	2	1	0	0
9	2	0	0	0
10	0	0	1	1
11	1	0	0	0
12	0	0	0	0

The computation times shown in Table V are substantial. However, the genetic programming code is not run “on-line” but rather “off-line” using files of historical data. The best utility function found through genetic programming can then be used to respond to new service requests in real-time. That is, the nine attribute values are computed for each vehicle whenever a new service request is received. Then, the utility function is applied to each vehicle to rank them from best to worst.

In this application, a ranking was obtained a few seconds after the occurrence of each request, which is acceptable for a courier service application. The attribute vector associated with each vehicle was readily computed because each planned route only contains a few customer requests (which is usually the case in courier service applications). On the other hand, the fleet of vehicles operated by the courier service company was relatively small. Adding new vehicles to the fleet would obviously impact the response times. However, a “reasonable” increase of the fleet size would still produce acceptable computation times given that the time pressure is not as stringent in this application as in other domains (e.g., military applications). Otherwise, alternative measures could be considered, like implementing the software on a parallel platform.

Table VI compares the results obtained through genetic programming $(P = 500)$ with those reported in [18] with a backpropagation neural network model, using the same experimental setting (the 90/50 experiment was the only one performed and reported in the above paper). A dispatching rule that simply assigns the next service request to the vehicle with minimum detour (in time) is also presented in the table.

By comparing these numbers, we observe that utility functions produced through genetic programming are competitive with the trained neural network. Furthermore, both genetic programming and the neural network produce substantially better results than the simple dispatching rule.

VII. Conclusions

This paper has described an application of genetic programming for vehicle dispatching. Here, this paradigm was exploited to construct a utility function aimed at approximating the decision process of an expert vehicle dispatcher. Our experiments have shown that genetic programming found useful programs. However, these programs are rather intricate and do not help much to better understand the dispatcher's decision process.

Better results could possibly be obtained through additional refinements. For example, the travel times between service points are based on average conditions and do not incorporate "dynamic" parameter values, like the current congestion level of the transportation network or the current weather conditions. Clearly, an improved assessment of travel times would help to better approximate the dispatcher's decision procedure.

References

[1] W. Bell, L. M. Dalberto, M. L. Fisher, A. J. Greenfield, R. Jaikumar, P. Kedia, R. G. Mack, and P. J. Prutzman, "Improving the distribution of industrial gases with an on-line computerized routing and scheduling optimizer," *Interfaces*, vol. 13, pp. 4–23, 1983.

[2] G. G. Brown, C. J. Ellis, G. W. Graves, and D. Ronen, "Real-time wide area dispatching of mobil tank trucks," *Interfaces*, vol. 17, pp. 107–120, 1987.

[3] L. Davis, *Handbook of Genetic Algorithms*. New York: Van Nostrand Reinhold, 1991.

[4] J. Desrosiers, Y. Dumas, M. M. Solomon, and F. Soumis, "Time constrained routing and scheduling," in *Network Routing*, M. O. Ball, T. L. Magnanti, C. L. Monma, and G. L. Nemhauser, Eds. Amsterdam, The Netherlands: North-Holland, 1995, pp. 35–139.

[5] E. Dijkstra, "A note on two problems in connection with graphs," *Numerische Mathematik*, vol. 1, pp. 269–271, 1959.

[6] M. Gendreau, F. Guertin, J. Y. Potvin, and E. Taillard, "Tabu search for real-time vehicle routing and dispatching," Tech. Rep. CRT-96-47, Centre de recherche sur les transports, Université de Montréal, Montréal, P.Q., Canada, 1996.

[7] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*. Reading, MA: Addison-Wesley, 1989.

[8] J. H. Holland, *Adaptation in Natural and Artificial Systems*. Ann Arbor, MI: Univ. of Michigan Press, 1975.

[9] J. R. Koza, *Genetic Programming: On the Programming of Computers By Means of Natural Selection*. Cambridge, MA: MIT Press, 1992.

[10] A. Kumar and Y. P. Gupta, Eds., *Comput. Oper. Res.*, (special issue on genetic algorithms), vol. 22 pp. 1–157, 1995.

[11] J. Y. Potvin, G. Dufour, and J. M. Rousseau, "Learning vehicle dispatching with linear programming models," *Comput. Oper. Res.*, vol. 20, pp. 371–380, 1993.

[12] W. B. Powell, "A comparative review of alternative algorithms for the dynamic vehicle allocation problem," in *Vehicle Routing: Methods and Studies*, B. L. Golden and A. A. Assad, Eds. Amsterdam, The Netherlands: North-Holland, 1988, pp. 249–291.

[13] H. N. Psaraftis, "A dynamic programming solution to the single vehicle many-to-many immediate request dial-a-ride problem," *Transp. Sci.*, vol. 2, pp. 130–154, 1980.

[14] H. N. Psaraftis, J. B. Orlin, D. Bienstock, and P. M. Thompson, "Analysis and solution algorithms of sealift routing and scheduling problems: Final report," Working Paper 1700-85, Sloan School of Management, Mass. Inst. Technol., Cambridge, 1985.

[15] H. N. Psaraftis, "Dynamic vehicle routing problems," in *Vehicle Routing: Methods and Studies*, B. L. Golden and A. A. Assad, Eds. Amsterdam, The Netherlands: North-Holland, 1988, pp. 223–248.

[16] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," in *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*. Cambridge, MA: MIT Press, 1986, pp. 318–364.

[17] M. W. P. Savelsbergh, "The general pickup and delivery problem," *Transp. Sci.*, vol. 29, pp. 17–29, 1995.

[18] Y. Shen, J. Y. Potvin, J. M. Rousseau, and S. Roy, "A computer assistant for vehicle dispatching with learning capabilities," *Ann. Oper. Res.*, vol. 60, pp. 189–212, 1995.

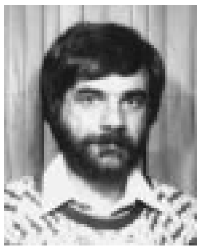
[19] P. Trudeau, J. M. Rousseau, J. A. Ferland, and J. Choquette, "An operations research approach for the planning and operation of an ambulance service," *INFOR*, vol. 27, pp. 95–113, 1989.

[20] N. H. M. Wilson, J. M. Sussman, H. K. Wang, and B. T. Higonett, "Scheduling algorithms for dial-a-ride systems," Tech. Rep. USL-TR-71-13, Urban Syst. Lab., Mass. Inst. Technol., Cambridge, 1971.

[21] N. H. M. Wilson and N. H. Colvin, "Computer control of the Rochester dial-a-ride system," tech. rep. R-77-30, Dept. Civil Eng., Mass. Inst. Technol., Cambridge, 1977.

Ilham Benyahia received the engineering diploma from the University of Constantine, Constantine, Algeria, in 1986, the M.Sc. degree from the University of Nice, Nice, France, in 1988, and the Ph.D. degree in computer science from the University Pierre et Marie Curie, Paris, France, in 1993. She then held a postdoctoral position with the Center for Research on Transportation, University of Montréal, Montréal, P.Q., Canada.

Since 1995, she has been a Researcher with the Research Institute of Hydro-Québec, Varennes. Her research interests include object architectures and frameworks to build and test complex applications, and the development of problem solving techniques based on genetic algorithms, expert systems, and real-time scheduling. She has been involved in French and Canadian R&D projects. The most recent one deals with real-time information management for the Hydro-Québec intelligent network project.



Jean-Yves Potvin received the B.Sc. degree in mathematics and the M.Sc. and Ph.D. degrees in computer science from Montréal University, Montréal, P.Q., Canada, in 1981, 1983, and 1987, respectively. He completed a postdoctoral fellowship at Carnegie-Mellon University, Pittsburgh, PA, in 1988.

From 1988 to 1993, he was Assistant Researcher with the Department of Computer Science and Operations Research, Montreal University. Since 1993, he has been Associate Professor in this same department. His research interests include heuristic search techniques such as tabu search, genetic algorithms, and neural networks for solving different types of vehicle routing and dispatching problems. He has published more than 50 papers in refereed journals and conference proceedings on these topics.

Dr. Potvin is a member of the Institute for Operations Research and the Management Sciences (INFORMS).