Comparing exact and heuristic methods for the Electronic Board Drilling-Traveling Salesperson problem

Giovanni Mazzocchin

***Abstract*— This report outlines some aspects of our work on the implementation of well-known exact and heuristic methods for the classic *Electronic Board Drilling-Travelling Salesperson* problem. Implementation issues and evaluation outcomes will be included as well.**

I. INTRODUCTION

A. The problem

This assignment comes from a real-world scenario:
a company produces boards with holes used to build electric frames. Boards are positioned over a machine and a drill moves over them, stopping at some desired positions and drilling holes. Given the holes' positions on the board, we are supposed to determine the sequence of holes that minimises the total drilling time, taking into account that the time required for making a hole is constant.
This description's similarity to the more famous *Traveling Salesperson problem* is noteworthy: actually, they can be thought of as the same problem.

B. The project

As the title suggests, our work is made up of two main parts:

- the first part comprises a tiny C++ program that utilises *CPLEX*, a popular optimisation software which offers a library for this very language;
- the second part is made up of a quite larger program implementing a *Genetic Algorithm*.

II. USING *CPLEX* TO OBTAIN AN OPTIMAL SOLUTION

A. Data extraction

The simplest yet essential component of this program is a class containing some methods able to read data from plain text files. These files should be generated according to the following template:

- the very first line should contain just a number standing for the problem's size;
- the remainder of the file should contain a symmetric matrix of real numbers equipped with this meaning: the number in position *(i,j)* represents the cost needed to move from hole (or *city*) *i* to city *j*.

*This project has been done for a course on Combinatorial Optimisation held in Padua by professors Luigi De Giovanni and Marco Di Summa

B. Core components and remarkable issues

The *main* function works essentially as a starting point, namely, it does not perform anything remarkable except for calling our *setupProblem* function (in `TSPSolver.h-cpp`) and carrying out some sanity checks.

Listing 1: Visit constraint

```
for (int i = 0; i < data.n; i++){
 coefs[i] = 1.0;
 for (int j = 0; j < data.n; j++){
   idx[j] = i*data.n + j;
 }
 CHECKED_CPX_CALL(CPXaddrows,
    env, lp,
    0, 1, data.n, &rhs, &sense[0],
    &matbeg[0], &idx[0], &coefs[0],
    NULL, NULL);
}
```

With respect to variables and corresponding coefficients, we resorted to plain STL *vectors*. As regards variables' ordering, we arbitrarily put the *y*'s before the *x*'s; note that we must take this priority into account for the sake of a correct access to variables and coefficients. Listing 2 shows a sort of "preamble" which is quite common before any constraint.

Listing 2: Important data structure declarations

```
const int x_init_index = data.n*data.n;
std::vector<int> idx(data.n);
std::vector<double> coefs(data.n);
```

In the above code extract, `x_init_index` represents the index where the *x* variables start from: it is equal to the square of the problem's size due to the fact that every *y* has two indices. The second line declares a vector, `idx` (whose size corresponds to the problem's dimension), which will store all the variables the following constraint will make use of. On the other hand, `coefs` will store the coefficients related to the variables in question.

C. Outcomes: reached optima and execution times

In order to evaluate this first part, we used some reference random instances (which will be used by the second part as well) our program can be run against.
Table 1 gives insight into this program's efficiency and effectiveness.

TABLE I: Results for the first part

Size of the problem	Optimum	Time (in seconds)
12	66.4	0.38
26	937	0.85
42	699	14.5
48	10628	24.5
60	629.8	78.7
100(a)	21282	370
100(b)	22141	970
200	unknown	too large

III. Approximating the optimum through Genetic algorithms

The exact approach we described so far turns out to be quite unwieldy for attacking large instances, due to Simplex' inherent high time complexity. This sad news has prompted researchers to develop some *heuristics* which should be able to yield approximate yet good solutions. A general method among these heuristics is the one we encounter in *Genetic algorithms*.

A. Generic steps

A traditional Genetic Algorithm for optimisation problems can be summarised as follows:

1) devise a suitable **encoding** for solutions (a.k.a. individuals, hereafter there words will be interchangeable);
2) generate an initial **population**, i.e., a set of solutions;
3) `Loop`:
4) select couples (or groups) of **parent** solutions;
5) generate an **offspring** thanks to recombination;
6) evaluate new solutions' **fitness** (this concept will be clarified later on);
7) replace current population using the offspring;
8) `Until:` a sensible stopping criterion;
9) `return` the best solution altogether.

B. Solution encoding and Fitness function

The chosen encoding was the so-called *path-representation*, according to which solutions can be seen as sequences of visited holes.
As for the *fitness* function, we decided not to use anything beyond the given objective function's value. Furthermore, we should say that we did not come across any viable alternative.

C. Generating an initial population

The first issue we face is generating a population as a good starting point for our algorithm. In this work we chose two strategies: the first leaves everything to chance, creating a random pool of feasible solutions the algorithm can start from, whereas the latter performs some *local search* (namely a certain amount of iterations with *Simulated Annealing*) on few (10%) individuals, so that hopefully (but we do know that Simulated Annealing accomplishes its goal) the "genetic loop" is provided with a higher-quality initial population. Sometimes the latter technique will be referred to as "smart initialisation".

Listing 3: Code within main annealing loop, from `TSPPopulation.cpp`

```
double neighbourVal=evaluate(neighbourSol);
double probability;
double diff = neighbourVal-currentSolVal;
if (neighbourVal < currentSolVal){
 double exponentialVal=exp(-(diff)/temp);
 double probability = 1;
 if (exponentialVal < 1){
   probability = exponentialVal;
 }
 probability = probability * 100;
 int prob_indicator = distr(rg);
 if (prob_indicator < probability){
   population[i] = neighbourSol;
 }
}
// temperature's drop
if (k == temp_thresh_1){
 temp = temp / 2;
}
```

D. Selecting individuals for mating

First of all, we should choose whether we want to perform mating on pairs or larger-sized groups of individuals. We decided to implement the former option due to its simplicity and overall effectiveness.
Plenty of selection criteria have been proposed in the literature: our choice fell on *Tournament Selection*, whose pseudocode is given below:

First step: choose k individuals from the population at random
Second step: choose the best individual from the tournament with probability p
Third step: choose the second best individual with probability $p * (1-p)$
N-th step: choose the n-th best individual with probability $p \times ((1-p)^{(n-2)})$

It can be shown that this strategy does not suffer from any bias towards "super-individuals" (that is, individuals endowed with better fitness), since it is easy to realise its independence from individuals' fitness values.

E. Recombining individuals through Crossover

As far as descendants' generation is concerned, we know we can carry it out thanks to various kinds of *Crossover operators*. Generally speaking, we could think of Crossover as a way to include a combination of parents' features into a child. As we have seen in the context of selection, there are lots of proposed operators here as well, which may vary in cost and effectiveness. The latter requirement matters particularly in the setting we are working on: in fact, any

operator that generates an offspring without being concerned about its feasibility would not be deemed as much effective. We wrote the code both for *Partially-Mapped Crossover* (Goldberg and Lingle) and *Order Crossover* (Oliver et al., Goldberg)), but since the former operator does not enforce feasibility, we are more amenable to go through the latter's steps, which are listed below:

1) randomly select a substring from a parent;
2) produce a proto-child by copying the substring into the its corresponding position;
3) discard (from the second parent) the holes which have already been put in the substring. The remaining sequence contains the holes eligible to enter the proto-child;
4) place the holes into the unfixed positions of the proto-child proceeding from left to right. The order of the holes stemming from the second parent must be preserved.

Listing 4: Order Crossover, in `TSPCrossover.cpp`

```
for (int i = 1; i < sol_size - 1; i++){
 if (counter1==fst_cut_ind_rnd){
  counter1=counter1+zoneSize;
 }
 if (counter2==fst_cut_ind_rnd){
  counter2=counter2+zoneSize;
 }
 it_indicator=find(conflictZone.begin(),
  conflictZone.end(),parent1[i]);
 if(it_indicator==conflictZone.end()){
  child1.sequence[counter1]=parent1[i];
 }
}
```

F. Generational replacement

One of the components needed by evolution is *Generational Replacement*. With lack of any sensible replacement, evolution would get stuck to inadequate individuals.
The kind of strategy we adopted is called *Best Individuals Replacement*. The aspect that drove us toward this method was its clear general description, which can be summarised straightforwardly with this sentence:
*generate **R** new individuals from **N** old ones, then keep the best **N** among the total **N + R**.*
We achieved this method's goal by means of plain vector concatenation and sorting. In particular, the concatenation consists in appending the current offspring to the current population (which are both represented with vectors). The resulting vector is then sorted using an appropriate comparator. Efficiency reasons forced us to dispose of the worst 3% of the resulting vector; this trick let us save up to minutes of computation time. Thus, the outcomes we will talk about will always refer to runs that take advantage of the trick mentioned above.

Listing 5: Best individual replacement, in `TSPPopulation.cpp`

```
TSPSolution repPop(vector offspring){
 population.insert(population.end(),
    offspring.begin(),offspring.end());
 TSPSolutionComparator tspSolComp;
 sort(population.begin(),population.end(),
    tspSolComp);
}
```

G. Exploiting mutation to avoid premature convergence

It is well known that lack of *mutation* occurrence would make our algorithm doomed to be characterized by poor diversification. As a matter of fact, *diversification* becomes necessary in order to escape from premature convergence, that is to say, ending up with a population made up of excessively similar individuals (*genetic drift*).
Therefore we are forced to introduce mutations, which obviously should occur seldom enough. Thanks to much painstaking tuning and testing, we found out that a fixed 0.2% mutation probability (then multiplied by three during the very last 5% of total generations) drives to decent results. Thus, hereafter we will stick to this value.
As in the case with other genetic steps, there are many ways to perform mutation on an individual. Trial and error led us to choose *2-opt* moves (namely, reversal of random substrings); however, our first choice consisted in performing random swaps between couples of holes (with very poor results which will not be reported).
At last, it is worthwhile to mention that mutations could occur both on the initial population ("initial" is referred to a specific generation) and on its offspring.

Listing 6: 2-opt move, in `TSPCrossover.h`

```
void doMutation(TSPSolution& individual){
 TSPSolution tmpSol(individual);
 for(int i=fstIndex;i<=sndIndex;i++){
   individual.sequence[i]=tmpSol[...];
 }
}
```

IV. Outcome analysis

In this section we will focus on outcomes and data representation. In the last section of the paper, called *Plots*, we will show two graphs per instance, whose aim is to show how quality and running time vary depending on the number of generations, given a fixed chosen number of individuals. Medium-sized instances have been attacked using both pure randomness and some local search in initialisation. On the other hand, bigger instances have been attacked only by means of smart initialisation.
We remark on the fact that all the plotted data come from *averages*, that is, the program has been run several times after fixing some parameters and then an average of the results has been calculated. Further details about testing can be found in the appendix titled *Practical Notes*.
The analysed instances are exactly the same as the ones that have been discussed in the first part of the paper.

From now on, we proceed with a thorough analysis of the graphs.

1) Small instances: fig. a), b), c), and d) show the average outcomes for two specific 12×12 and 26×26 instances. Figures show evident quality improvements as long as the number of generations increases, with some exceptions that are due to (we believe) the stochastic nature of genetic algorithms. Up to now, running time is not an issue.

2) Medium-sized instances: figures e) and f) come from a more complex background: this 42×42 instance has been tested both with and without Simulated Annealing. Thus, two lines have been plotted. Using Simulated Annealing (1000 iterations) to initialise even a small fraction of the starting population gave us remarkable improvements, notwithstanding a reduced population size. In fact, the 800-individual population was tested without Simulated Annealing, whereas the 500-individual one did take advantage of it.

3) Large instances: figures g) and h) refer to the instance that had been named *100a* in the first part. Here we plotted only the results coming from a Simulated Annealing initialisation (6000 iterations), because its absence provided us with very poor results. These plots show that quality and running time start to become an issue.

A table with quality variances for some instances is given below:

TABLE II: Quality variance on different instances

Size	Smart init. iter.	Population-Gener.	Variance
42	0	800-400	1.37%
42	1000	500-300	0.63%
60	0	1500-500	1.77%
60	4000	1500-500	0.17%
100b	6000	1500-500	1.07%

V. CONCLUSIONS

This practical work on combinatorial problems allowed us to grasp many fundamental concepts. First of all, we realised that exact methods do not always manage to give good solutions in an acceptable amount of time. Seeing the striking surge in running time beyond instances with 60 holes has been interesting as well.

As far as genetic algorithms are concerned, the results for smaller instances have been excellent both in term of quality and computation time. On the other hand, bigger instances required much more time to approach the optimum. Initialising some individuals by way of Simulated Annealing gave us a dramatic improvement for these instances.

However, we should always be aware of the fact that these heuristics give no optimality guarantee.

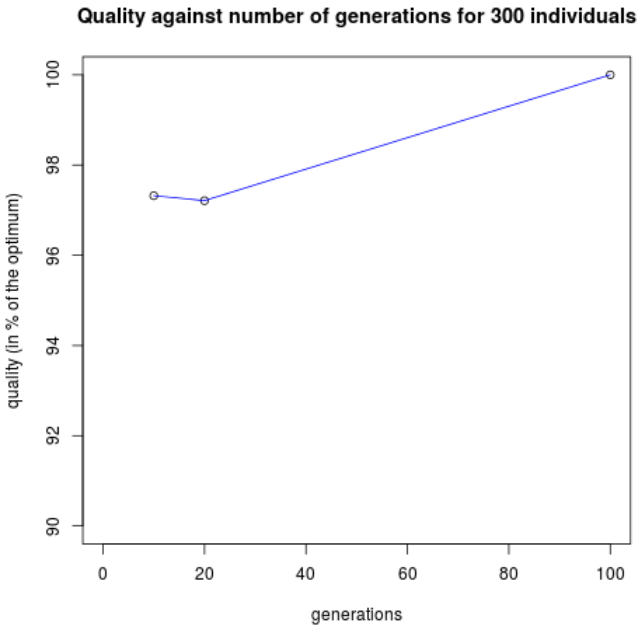
PRACTICAL NOTES

Here the reader can find some important information regarding compilation and execution of the software.

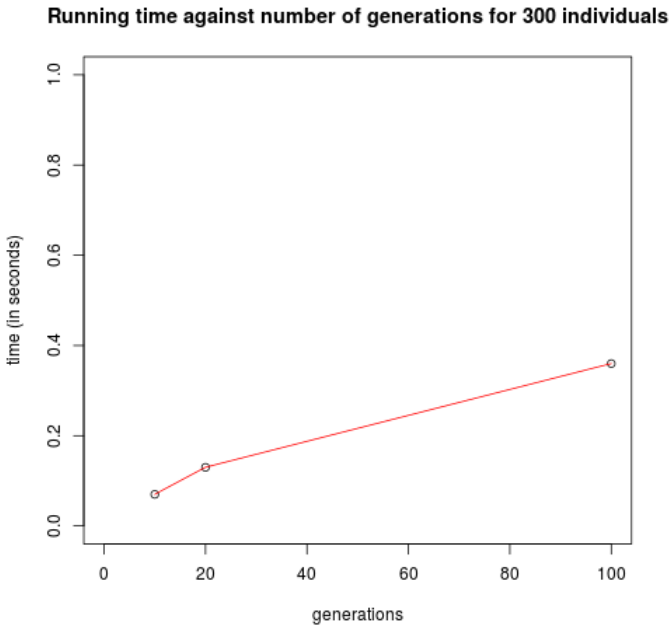
1) **Compilation and execution**: compilation should be as easy as running `make` in the project's root folder from the command line.
   In order to get some hints on how to run the program, launch `./main` from the same directory; anyway, we have provided you with some handy scripts which will be discussed below.
2) **Instances**: the directory named *instance_generator* contains some instances we found in *TSPLIB*, a library of sample instances for well-known combinatorial problems.
3) **Testing**: some *shell scripts* are present under the directory named `results`. There exists a script named `launchX.sh` (where *X* is the instance's size) for each instance. In order to run them, you are obliged to give them execution permissions running the following command: `chmod +x script_name`.
   These scripts accept two parameters from the command line: first, they require the problem's known optimum, then the name of an output file which will store their output.
   We decided to leave some parameters such as *population's size* and *number of generations* hard-coded.
   After receiving all the data, these scripts run our program several times (up to 49 for smaller instances, whereas we stopped to 30 for the bigger ones), so that we can calculate statistical parameters (such as mean and variance) afterwards. This large amount of testing is required since the algorithms' behaviour is stochastic in nature.

PLOTS

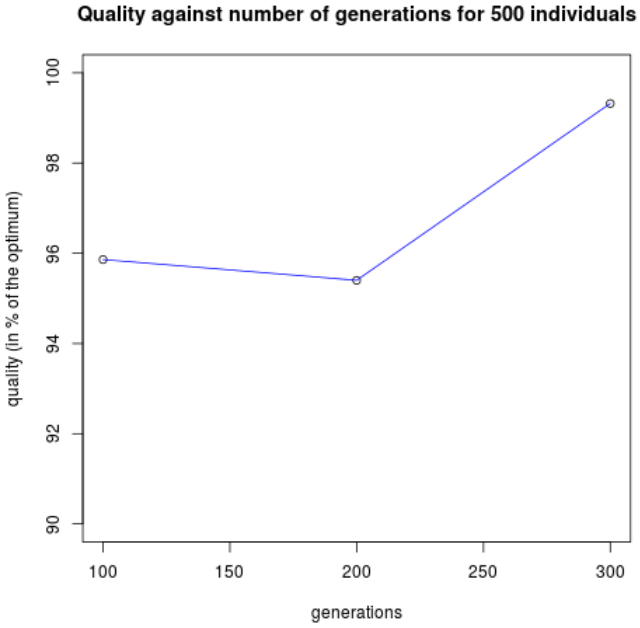
Some plots (along with understandable captions) are provided in the next pages.



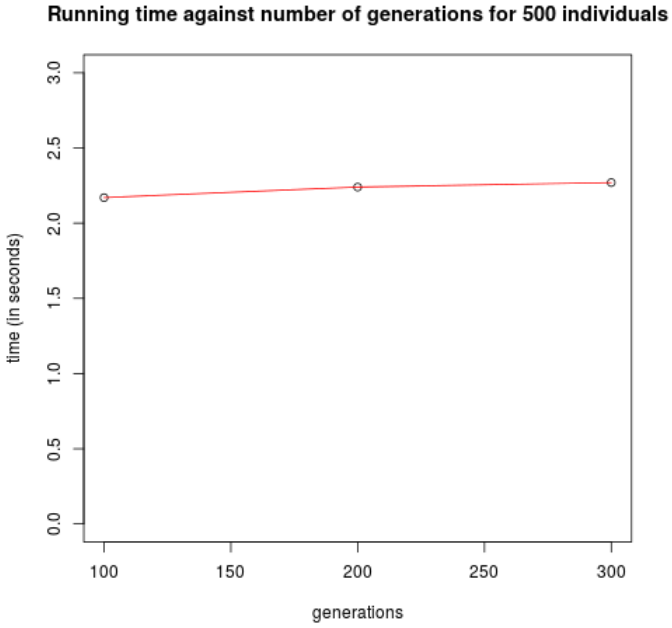
(a) Quality for a 12×12 instance



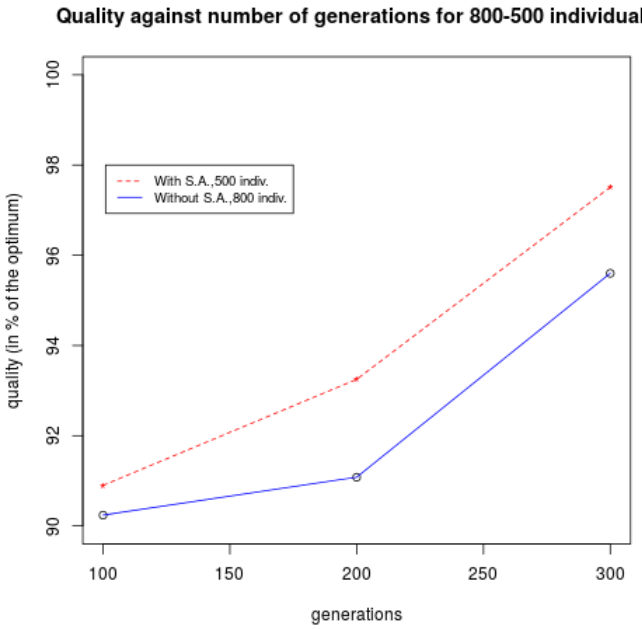
(b) Running time for a 12×12 instance



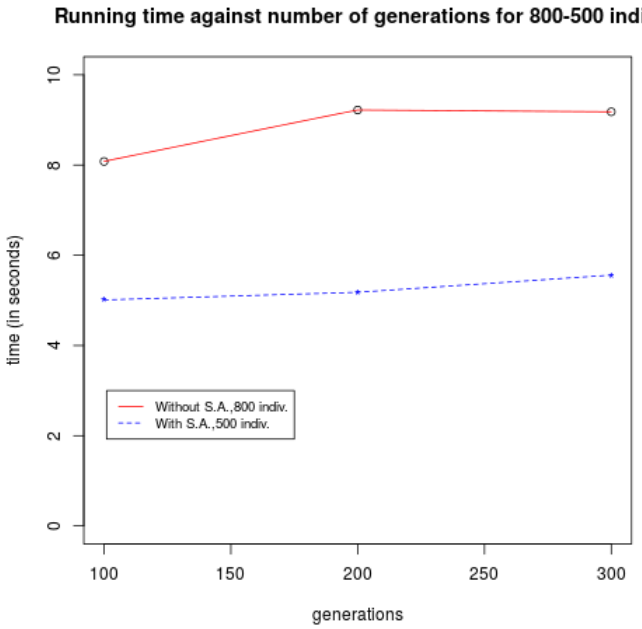
(c) Quality for a 26×26 instance



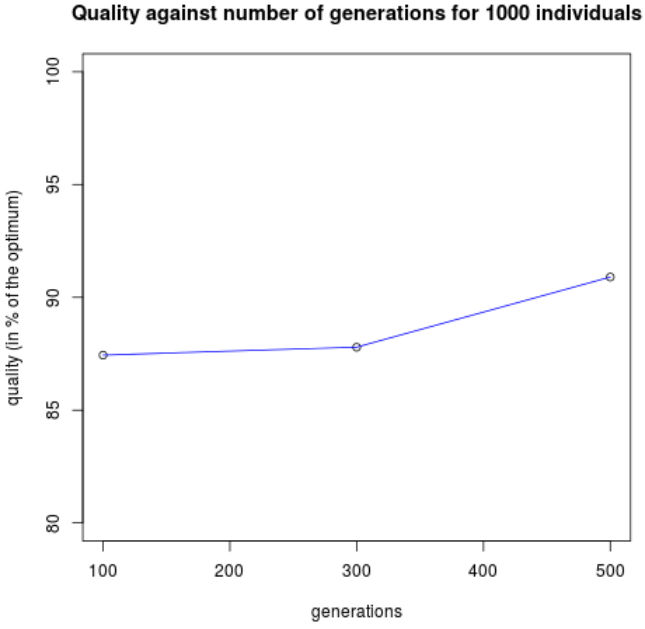
(d) Running time for a 26×26 instance



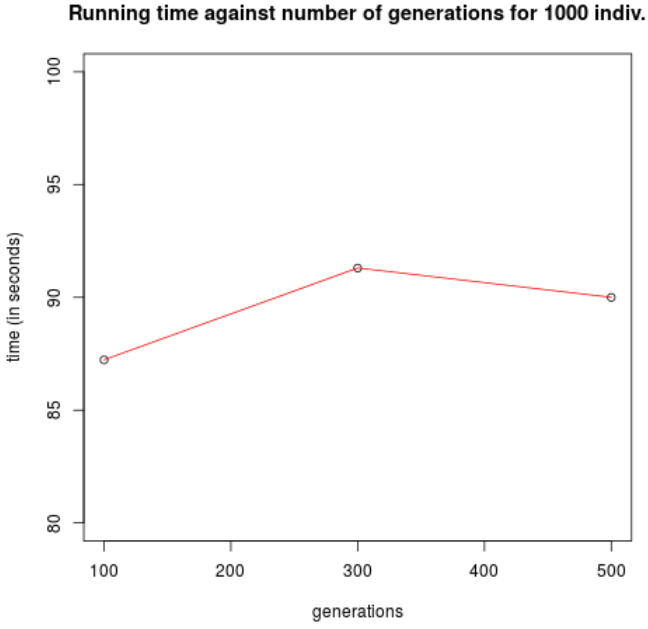
(e) Quality for a 42×42 instance



(f) Running time for a 42×42 instance



(g) Quality for a 100×100 instance



(h) Running time for a 100×100 instance

REFERENCES

[1] Jean-Yves Potvin, Genetic algorithms for the Traveling Salesman Problem, Centre de Recherche sur les Transports, Université de Montréal, C.P. 6128, Succ. Centre-Ville, Montréal, Québec, Canada H3C 3J7