An implementation of the Volume Algorithm

Francisco Barahona

June 22, 2000

1 Introduction

Here we describe an implementation of the Volume algorithm (VA) originally presented in [1]. We focus on the *uncapacitated facility location problem* (UFLP) as an example, see [2] for some of the theoretical issues. As a first step, a new user should be able to run our code "as is". This can also be used as a framework for Lagrangian relaxation. The user would have to modify the files `ufl.hpp`, `ufl.cpp`, `ufl.par`, and `data`, to produce an implementation for a different problem. The files `volume.hpp` and `volume.cpp` are specific to the VA, so the user should not have to change them. We hope to receive reports about bugs and/or successful experiences.

Initially this directory contains the files: `INSTALL, Makefile, volume.hpp, volume.cpp, ufl.hpp, ufl.cpp, ufl.par, data, doc.ps`. See the `INSTALL` file for instructions on installation.

Now we present the linear program used in [2]. This is

$$
\begin{aligned}
\min \sum c_{ij} x_{ij} \quad &+ \quad \sum f_i y_i && (1)\\
\sum_i x_{ij} \quad &= \quad 1, \text{ for all } j, && (2)\\
x_{ij} \quad &\leq \quad y_i, \text{ for all } i, j, && (3)\\
x_{ij} \quad &\geq \quad 0, \text{ for all } i, j, && (4)\\
y_i \quad &\leq \quad 1, \text{ for all } i. && (5)
\end{aligned}
$$

Here the variables y correspond to the locations, and the variables x represent connections between customers and locations. Let u_j be a set of Lagrange multipliers for equations (2). When we dualize equations (2), we obtain the *lagrangian problem*

$$
\begin{aligned}
L(u) \quad &= \quad \min \sum \bar{c}_{ij} x_{ij} + \sum \bar{f}_i y_i + \sum u_j,\\
x_{ij} \quad &\leq \quad y_i, \text{ for all } i, j,\\
x_{ij} \quad &\geq \quad 0, \text{ for all } i, j,\\
y_i \quad &\leq \quad 1, \text{ for all } i.
\end{aligned}
$$

Where the *reduced costs* $\bar{c}_{ij} = c_{ij} - u_j$, and $\bar{f}_i = f_i$. We apply the VA to maximize $L(\cdot)$ and to produce a primal vector $(\bar{x}, \bar{y})$ that is an approximate solution of (1)-(5). Using this primal information we run a heuristic that gives an integer solution.

In what follows we describe the different files in this directory.

2 ufl.par

This file contains a set of parameters that control the algorithm and contain some information about the data. Each line has the format

```
keyword=value
```

where `keyword` should start in the first column. If we add any other character in the first column, the line is ignored or considered as a comment. The file looks as below

```
fdata=data
*dualfile=dual.txt
dual_savefile=dual.txt
int_savefile=int_sol.txt
h_iter=100

printflag=3
printinvl=5
heurinvl=10

greentestinvl=1
yellowtestinvl=4
redtestinvl=10

lambdainit=0.1
alphainit=0.1
alphamin=0.0001
alphafactor=0.5
alphaint=50

maxsgriters=2000
primal_abs_precision=0.02
gap_abs_precision=0.
gap_rel_precision=0.01
granularity=0.
```

The first group of parameters are specific to the UFLP and the user should define them. `fdata` is the name of the file containing the data. `dualfile` is the name of a file containing an initial dual vector. If we add an extra character at the beginning (`*dualfile`) this line is ignored, this means that no initial dual vector is given. `dual_savefile` is the name of a file where we save the final dual vector. If this line is missing, then the dual vector is not saved. `int_savefile` is the name of a file to save the best integer solution found by the heuristic procedure, if this line is missing, then this vector is not saved. `h_iter` is the number of times that the heuristic is run after the VA has finished.

The remaining parameters are specific to the VA. See the `VOL_parms.html` file for their documentation (this file will be created after doing a 'make doc' or it's available on the Internet).

3 data

The file `data` has the following format. On the first line we have the number of possible locations and the number of customers. On the next lines, the cost of opening each location appears, one cost per line. Then each of the remaining lines is like

$i \quad j \quad d_{ij}$,

where i refers to a location, j refers to a customer, and d_{ij} is the cost of serving customer j from location i. The indices i and j start from 1. If a pair i, j is missing then the cost d_{ij} is set to 10^7.

4 ufl.hpp

This file contains C++ classes specific to the UFLP.

First we have a class of parameters specific to the UFLP. The description of these parameters appears in the preceding section.

```
class UFL_parms {
public:
   string fdata;         // file with the data
   string dualfile;      // file with an initial dual solution
   string dual_savefile; // file to save final dual solution
   string int_savefile;  // file to save primal integer solution
   int h_iter;           // number of times that the primal heuristic will be
                         // run after termination of the volume algorithm

   UFL_parms(const char* filename);
   ~UFL_parms() {}
};
```

Before the next class we should mention the classes `VOL_dvector` and `VOL_ivector` defined in `Volume.hpp`. The pseudo-code below illustrates their use.

```
int n=100;
VOL_dvector x(n);  // a double vector with n entries
x=0.;              // sets to 0. all entries of x
VOL_dvector y;     // a double vector, it size remains to be set
y.allocate(n);     // size is set
y=x;               // copy each entry of x into y
VOL_dvector z(y);  // a double vector of the same size as y,
                   // all entries of y are copied into z
x[0]=-1;           // first entry of x is set to -1
y[0]=x[0];         // copy first entry of x into first entry of y
```

The class `VOL_ivector` is used for vectors of `int`. One can do the same operations as for `VOL_dvector`.

Then we have a class containing the data for the UFLP.

```
class UFL\_data { // original data for uncapacitated facility location
public:
   VOL_dvector fcost; // cost for opening facilities
```

```
   VOL_dvector dist;  // cost for connecting a customer to a facility
   VOL_dvector fix;   // vector saying if some variables should be fixed
                      // if fix=-1 nothing is fixed
   int ncust, nloc;   // number of customers, number of locations
   VOL_ivector ix;    // best integer feasible solution so far
   double      icost; // value of best integer feasible solution
public:
   UFL_data() : icost(DBL_MAX) {}
   ~UFL\_data() {}
};
```

Then we have

```
class UFL_hook : public VOL_user_hooks {
public:
   // for all hooks: return value of -1 means that volume should quit
   // compute reduced costs
   int compute_rc(void * user_data,
                  const VOL_dvector& u, VOL_dvector& rc);
   // solve lagrangian problem
   int solve_subproblem(void * user_data,
                        const VOL_dvector& u, const VOL_dvector& rc,
                        double& lcost, VOL_dvector& x, VOL_dvector&v,
                        double& pcost);
   // primal heuristic
   // return DBL_MAX in heur_val if feas sol wasn't/was found
   int heuristics(void * user_data, const VOL_problem& p,
                  const VOL_dvector& x, double& heur_val);
};
```

Here the function `compute_rc` is used to compute reduced costs. In the function `solve_subproblem` we solve the lagrangian problem. In `heuristics` we run a heuristic to produce a primal integer solution.

Finally in this file we have `UFL_parms::UFL_parms(const char *filename)`, where we read the values for the members of `UFL_parms`.

5 ufl.cpp

This file contains several functions that we describe below.

First we have `int main(int argc, char* argv[])`. In here we initialize the classes described in `ufl.hpp`, and read the data. Then `volp.psize()` is set to the number of primal variables, and `volp.dsize()` is set to the number of dual variables. Then we check if a dual solution is provided and if so we read it.

For the UFLP all relaxed constraints are equations, so the dual variables are unrestricted. In this case we do not have to set bounds for the dual variables. If we have inequalities of the type $ax \geq b$, then we have to set the lower bounds of their dual variables equal to 0. If we had constraints of the type type $ax \leq b$, then we have to set the upper bounds of their variables equal to 0. This would be done as in the pseudo-code below.

```
// first the lower bounds to -inf, upper bounds to inf
volp.dual_lb.allocate(volp.dsize);
volp.dual_lb = -1e31;
volp.dual_ub.allocate(volp.dsize);
volp.dual_ub = 1e31;
// now go through the relaxed constraints and change the lb of the ax >= b
// constrains to 0, and change the ub of the ax <= b constrains to 0.
for (i = 0; i < volp.dsize; ++i) {
   if ("constraint i is '<=' ") {
      volp.dual_ub[i] = 0;
   }
   if ("constraint i is '>=' ") {
      volp.dual_lb[i] = 0;
   }
}
```

The function `volp.solve` invokes the VA. After completion we compute the violation of the fractional primal solution obtained. This vector is `psol`. Then we check if the user provided the name of a file to save the dual solution. If so, we save it. Then we run the primal heuristic using `psol` as an input. Notice that this heuristic has also been run periodically during the execution of the VA. Then if the user has provided the name of a file to save the integer heuristic solution, we do it. Finally the values of the solutions and some statistics are printed.

The next function is `void UFL_read_data(const char* fname, UFL_data& data)`, where we read the data. `data.nloc` is the number of locations, `data.ncust` is the number of customers. `data.fcost` is a vector containing the cost of opening each location. `data.dist` is a vector containing the cost of serving customers from facilities. All entries are initialized to 10^7 and then particular entries are being set with the statement

`dist[(i-1)*ncust + j-1]=cost;`

where `i` is the index of a location and `j` is the index of a customer. Here the indices start from 1. Finally we have a vector `data.fix` associated with the locations. A particular entry is set to 0 if the location should be closed, it is set to 1 if it should be open, and it is set to -1 if this variable is free. Initially all entries are set to -1.

In the function

`double solve_it(void * user_data, const double* rdist, VOL_ivector& sol)`

we solve the lagrangian problem. We receive the data and reduced costs as input and return a primal vector. The solution is in the vector `sol`. Its first `n` entries correspond to the locations, then all remaining entries correspond to connections between locations and customers.

In the function

`int UFL_hook::compute_rc(void * user_data, const VOL_dvector& u, VOL_dvector& rc)`

we compute the reduced costs. They will be used to solve the lagrangian problem.

In the function

```
int
UFL_hook::solve_subproblem(void *user_data,
                           const VOL_dvector& u, const VOL_dvector& rc,
                           double& lcost, VOL_dvector& x,
                           VOL_dvector& v, double& pcost)
```

we compute the lagrangian value, we call `solve_it`, we compute the objective value and the vector v defined as follows. If $\hat{x}$ is the primal solution given by `solve_it`, and $Ax \sim b$ is the set of relaxed constraints, then the difference v is

$$v = b - A\hat{x}.$$

The last function in this file is

```
int
UFL_hook::heuristics(void * user_data, const VOL_problem& p,
                     const VOL_dvector& x, double& icost)
```

where we run the following simple heuristic. Given a fractional solution $(\bar{x}, \bar{y})$, let $\bar{y}_i$ be the variable associated with location i. We pick a random number $r \in [0, 1]$ and if $r < \bar{y}_i$ facility i is open, and closed otherwise. We repeat this for every facility, then given the set of open facilities we find a minimum cost assignment of customers. This function is invoked periodically in the VA and by the main program after the VA has finished.

References

[1] F. Barahona and R. Anbil, *The volume algorithm: producing primal solutions with a subgradient method*, Research Report RC 21103, IBM, T. J. Watson Research Center, 1998.

[2] F. Barahona and F. Chudak, *Solving large scale uncapacitated facility location problems*, Research Report RC 21515, IBM, T. J. Watson Research Center, 1999.