CE-solver manual

Stanislav Zidek

May 25, 2009

1 Introduction

CE-Solver is a tool for computing Correlated Equilibrium in possibly very large games. It useses slightly modified method proposed by dr. Hruby (see `http://perchta.fit.vutbr.cz/CE-Solver/1`), which is based on elimination of strictly dominated strategies using G-matrix.

2 Theory

2.1 Correlated equilibrium

Correlated equlilibrium (CE) is a solution concept in game theory that, unlike for example well known Nash equilibrium, allows some kind of simple synchronization between players.

It is useful for analyzing real market situations, because the synchronization in real world *is* possible.

In fact, CE is a probability distribution over the set of strategy profiles of the game.

2.2 G-matrix

G-matrix represents an elegant way to display multiplayer game (even games with more than two players) in a two dimensional table. It is also well suited for reduction of game's state space.

Rows are identified by three numbers: identification of player p and two different strategies of his (we will call them *sfrom* and *sto*). Colums are identified by strategy profiles of the game. Cells contain the difference

$$u_p(sfrom, s_{-p}) - u_p(sto, s_{-p}),$$

where s_{-p} are strategies played in strategy profile corresponding to the column of G-matrix and u_p is utitlity function for player p.

Note Cells in column identified by strategy profile which does not contain strategy *sfrom* are empty.

2.3 Reduction

Reduction of the game is quite straightforward once we construct the G-matrix. If we find a row that contains only negative values, we can eliminate corresponding strategy *sfrom*, which means that we can remove

- all rows that contain eliminated strategy and
- all colums with strategy profiles containing eliminated strategy.

After one elimination step it is possible that other strategy becomes dominated (its row contains only negative values), because we remove some colums that may have contained positive values which prevented its elimination earlier in the process.

As a consequence, the reduction is iterative procedure which can be performed as long as we are able to eliminate at leat one strategy. Eventually we may end up with single valid profile, after which the computing of CE is *very* simple.

2.4 Example

Consider game in table 1. Corresponding G-matrix is in table 2.

	D	**E**
A	1, 3	2, 5
B	2, 2	3, 1
C	3, 4	2, 3

Table 1: Example game

As we can see (in table 2), there is one negative row, **A**→**B** to be specific. After elimination of strtegy **A**, we get G-matrix in table 3.

After the first step of elimination, another strategy (**E**) can be eliminated, after which another strategy might be eliminated, and so on...

	AD	BD	CD	AE	BE	CE
A→B	−1			-1		
B→A		1			1	
B→C		−1			1	
C→B			1			−1
A→C	−2			0		
C→A		2			0	
D→E	−2	1	1			
E→D				2	−1	−1

Table 2: G-matrix corresponding to the game in 1 (we omit player identification – it is not neccessary since they both have distinct set of strategies).

	BD	CD	BE	CE
B→C	−1		1	
C→B		1		−1
D→E	1	1		
E→D			−1	−1

Table 3: G-matrix after one step of elimination

2.5 Computing CE as LP problem

If we are able to reduce whole game to single strategy profile, then the solution is easy. However, it might not be the case, so we need some tool to compute CE in any game.

The task of computing CE may be turned into solving LP problem. Its structure is very similar to structure of G-matrix. LP variables correspond to G-matrix columns (strategy profiles) and LP constraints correspond to G-matrix rows. One additional constraint must be added to make sum of probabilities equal to one (it assures unboundness of the solution).

Last issue remains: what are the coefficients of the objective function. There is no general rule, but two most logical and reasonably complex ways are as follows:

1. all coefficients are equal to one or
2. the coefficient of LP variable is the sum of payoffs of all players when playing associated strategy profile, more formally:

$$c_s = \sum_{i \in Q} u_i(s),$$

where s is the associated strategy profile, Q is set of players and u_i is payoff function of player i.

We decided to use the second approach, because it is able to reflect the prefference of equilibrium with higher payoffs for all players.

3 Implementation

CE-solver is a library with simple command line utility written in C++. It uses OpenMP to employ parallelism. LP problems are solved by GLPK library, but generally any tool can be used if you implement a simple interface.

3.1 Main ideas

3.1.1 CE-solver input

We tried to keep the interface as simple as possible to provide freedom to potential modellers. Creating game model is very easy in terms of interface with CE-solver (however complex it can be internally). You just have to create class inherited from `Game` and implement three methods. Details are described in part 4.

3.1.2 Storage of G-matrix

Since G-matrix is very huge, we decided to store just information that is essential. That is, all we need to know is the position in every row where we had to finish traversing, so after some eliminations we are able to skip forbidden profiles and continue traversing.

3.1.3 Payoff cache

Computing payoffs is computationally hardest part of real-world models. Because of that, we created so called *payoff cache*, which is used to store computed payoffs. When they are needed again, which might happen several times during solving, we do not have to compute them and just read them from this cache (see 3.2.4).

3.1.4 Parallelization

Parallelism is a natural choice to speed up computing. Because G-matrix reduction can be quite easily done in parallel, we designed the whole library with parallelism in mind. For this purpose, we chose *OpenMP* system because

it is easy to use and provides good results on parallel architectures with shared memory.

3.2 Structure

In this part, we are going to describe how does CE-solver work internally. We will outline what happens when a game is being solved and describe some importatnt classes. Other classes will described in part 4.

Basic scheme of the library could be seen in class diagram 1.

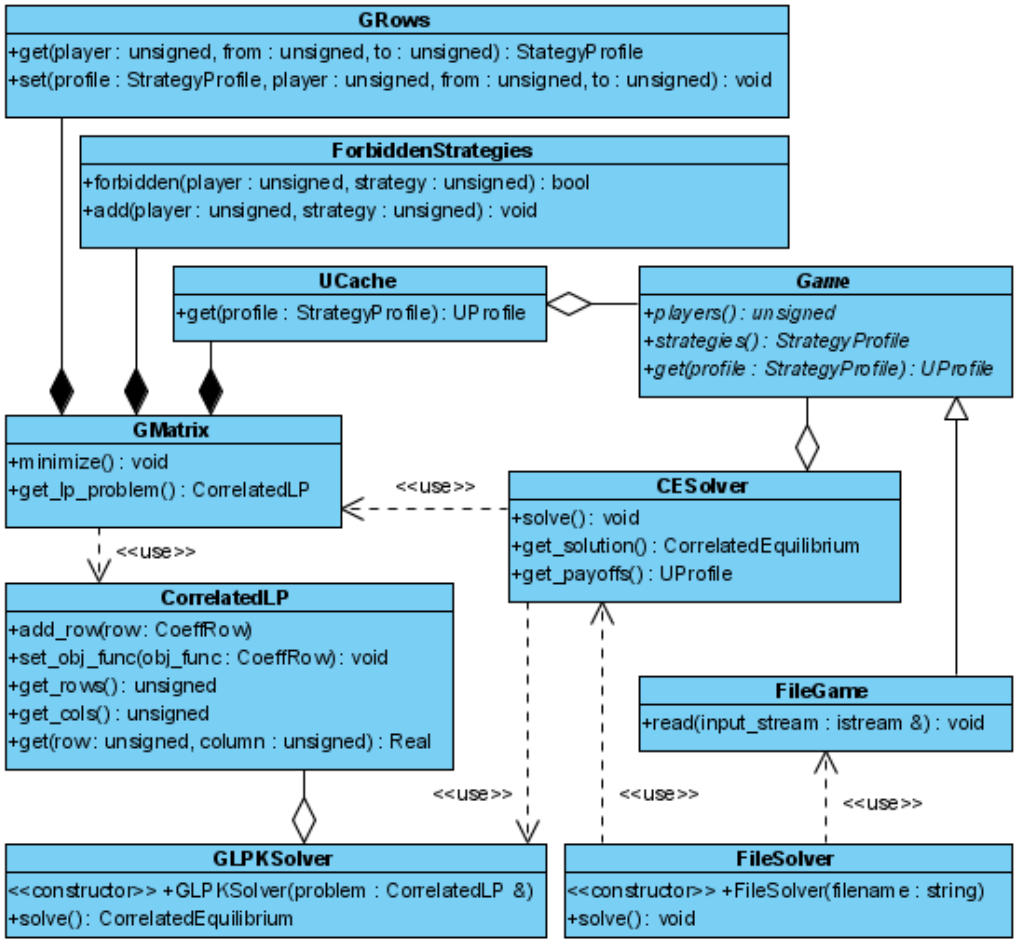

Figure 1: Class diagram of CE-solver

3.2.1 Solving game

When method `solve()` of `CESolver` is invoked, it does the following things:

1. Creates `GMatrix object`.
2. Calls `minimize()` method of `GMatrix`.
3. Creates object of `CorrelatedLP` by invoking method `get_lp_problem()` of `GMatrix`.
4. Solves LP problem (`CorrelatedLP`) by using `GLPKSolver` and stores the solution.

Because we use GLPK as a "black-box" component, the main algorithmic complexity lies in the method `minimize` of `GMatrix`, so we are going to describe this class more thoroughly.

`GMatrix` can be considered as computational heart of whole CE-solver. Minimization is done by iterating through G-matrix until no more strategies can be eliminated. One iteration is done in method `iterate`, which assigns G-matrix rows to threads. Every thread then goes through its row (skipping undefined columns) until it reaches the end (then $sfrom$ can be eliminated) or encounters nonnegative value.

Important algorithms are presented in figures 2 and 3 using C++/Python inspired pseudocode.

```
players = number of players
//for all players
for (unsigned p = 0; p < players; p++):
    p_strats = number of strategies of player p
    for (unsigned sto = 0; sto < p_strats; sto++):
        if (strategy sto of player p is forbidden):
            continue
        for (unsigned sfrom = 0; sfrom < p_strats; sfrom++):
            //now we are in G-matrix row identified by:
            //  (p: sfrom -> sto)
            if (strategy sfrom of player p is forbidden):
                continue
            if (sfrom != sto):
                //check according row
                ...
```

Figure 2: Iterating through rows of G-matrix

3.2.2 G-matrix positions – `GRows`

Class `GRows` is used to store positions in G-matrix rows where traversing had to stop. Initially, it is empty and when a position for a row is requested and nothing has been previously stored for this row, it just returns first strategy profile – $(0, \ldots, 0)$.

3.2.3 Eliminated strategies – `ForbiddenStrategies`

Since we do not want to store explicitly all strategy profiles which have already been eliminated, we just store eliminated strategies in this particular class. Internally it contains a vector of boolean values, which is stored effectively.

This class is used every time we need to advance in traversing row from one profile to another. We have to check that all strategies in next profile are valid (have not been eliminated). If it is not the case, we must proceed to nex profile until we reach valid profile or the end of the row.

3.2.4 Cache of computed payoffs – `UCache`

This is very important class for solving games where payoff computation takes considerable amount of time. Performed tests show that this amount is about 5–10 μs on computer with 16 processors Intel Xeon X5355 (2,66 GHz).

3.2.5 Correlated LP problem – `CorrelatedLP`

This class provides some kind of link between minimization of game and solving LP problem. It is in fact the output of minimization, provided by `G-matrix`, and the input of `GLPKSolver`.

It might be important only in situation when you want to create you own LP solver, in which case it is its input.

3.2.6 Interface to GLPK – `GLPKSolver`

`GLPKSolver` provides an interface for using GLPK to solve LP problem in `CorrelatedLP`. It has been designed to make it as simple as possible to "switch" to another LP solving tool. If you are interested in this way, you should implement class similar to `GLPKSolver` with methods.

Generally the interface is so simple that all you have to do is to create similar constructor and method `solve()`. For additional details, please see the source code.

4 Usage

4.1 Library interface

The only thing modeller has to do is to create his game model, which must be inherited from abstract class `Game`. Then he can create object `CEsolver`, that coputes the CE.

4.1.1 Game model – class `Game`

There are three methods a modeller has to implement. Two of them are rather simple. The method `unsigned players()` just returns the number of players playing the game and method `StrategyProfile strategies()` returns the number of strategies each player has in object of class `StrategyProfile`.

Class `StrategyProfile` This class represents a strategy profile of the game. The easiest way to use it is probably to call constructor which takes number of players as an argument and then access the elements using the `operator[]`. For performance reasons, it is implementes as static array of size `MAX_PLAYERS`, which is 16 by default. In case of games with more players, modeller has to specify appropriate value of this macro at compile time. It is also possible to specify value lesser than 16, which might make sense if the modeller wants to save some memory.

The third method is the most complex and also the most important, it does the actual computation of the payoffs. Its signature is:

```
UProfile<class T=float> get(StrategyProfile prof)
```

It takes a strategy profile as an argument and returns corresponding payoffs.

Class `UProfile` This class is very similar, almost identical to `StrategyProfile` in its behaviour. It is usually used to store the payoffs computed for every player in specific strategy profile.

Note on parallelism Main payload of computation is considered to be just mentioned method `get`. CE-solver tries to utilize parallelism in computing payoffs, which means invoking this method in parallel. So it must be reentrant (along with other two methods, by the way). We try to enforce this by using the `const` modifier on all these methods.

4.1.2 Computing equilibrium – class `CESolver<bool B, class T=float>`

Now modeller is in situation that he has just created the model of the game and wants to compute the CE. The simple way to do that is to construct `CESolver` object and call his method `solve()`. The method `get_solution()` serves to get the computed solution in `std::map<StrategyProfile, double>` and the method `get_payoffs()` returns players' expected payoffs in case of playing the CE.

The first template argument (`bool B`) is of cardinal importance. It says whether to use the payoff cache or not. Setting it to *true* has great impact on performance when computing of the payoffs takes significant time (about 10 μs). However, if it is not the case, then using the cache is useless and the argument should be set to *false*.

4.2 Correlated Equilibrium File Solver

CEFS is a simple command line utility that is able to solve games saved in files.

4.2.1 File formats

There are two file formats accepted by CEFS.

Native file format CEFS's native file format is very simple. It can be describe in notation inspired by Extended Backus-Naur Form (EBNF) as:

```
game = players, ws, strategies list, payoffs;
players = natural number (* number of players*);
strategies list = players * (strategies, ws);
strategies = natural number (* number of strategies *);
payoffs = product of strategies * uprofile;
uprofile = players * (number, ws);
number = integer | real number;
ws = ? one or more white space characters ?;
integer = ? obvious meaning ?;
natural number = ? integer greater than zero ?;
real number = ? real number in standard format ?;
```

EBNF does not allow to specify number of repetitions by non-constant expression, but we use it. Since `players` is a natural number, we use the notation

```
strategies list = players * (strategies, ws);
```

to express that `(strategies, ws)` must be repeated exactly n times, where n is natural number yielding the value that was covered by nonterminal `players`. Informally, this game format represents a serialized table of payoffs, preceeded by short header containing its size (number of players and their strategies). Table of payoffs is serialized "lexicographically", by which we mean that payoffs are ordered by strategy profiles as $(1,1)$, $(1,2)$, $(2,1)$, $(2,2)$ (last player's strategy changes most frequently). For example, game in table 4 is saved as file in figure 4.

	D	**E**
A	1, 3	2, 5
B	2, 2	3, 1
C	3, 4	2, 3

Table 4: Game to be saved in file (see figure 4)

Gambit normal form games (`.nfg`) Another formats supported by CEFS are two slightly different Gambit (well known free software library for solving mathematical games) formats for normal form games. They are well described on Gambit website, so we will not go into them here. Let us just discourage you from using gambit format. Native format is much more simple and although Gambit's format is well specified, almost no game bundled with it respects it.

4.2.2 Invocation

CEFS can be invoked without any arguments. That means it reads input game from `stdin` and does all actions (export and solve reduced game).

If argument `-s` is used, CEFS only minimizes and solves reduced game, it does not export it.

Argument `-e` means that CEFS only minimizes and exports reduced game. Default behaviour is to export to file `reduced.game` with mappings in `reduced.map`.

You can change the file name of reduced game by specifying argument `-r filename`. So instead of exporting to `reduced.game` and `reduced.map` CEFS exports minimized game to `filename.game` and filename.map.

Help can by obtained using argument `-h`.

4.2.3 Mappings file

To preserve some relation between orginal and reduced game we introduce so called "mappings" file. It is created together with file with reduced game and contains information about correspondence of strategies between both games.

The header is similar to our native file format, it consists of number of players followed by number of strategies that were not eliminated for each player. Then there are lists of indices of strategies in original game that correspond to strategies in reduced game.

You can see an example in figure 5. It says that 4-player game was reduced (first row) so that zeroth player has two strategies, first player has just one remaining strategy, second player has five strategies and three strategies of fourth were not eliminated (second row). Obviously we are indexing from zero. Then you can see that two strategies that remained for zeroth player originally had indices two and four (third row) and so on.

5 Conclusion

CE-solver is powerful library for finding Correlated Equilibium. It is very simple to use, while carefully designed algorithms and data structures preserve efficiency of computation.

In this manual, we described the theoretical background, implementation and usage of CE-solver library and CEFS utility. It explains the main ideas and functionality and contains some useful directions for potential modellers.

```
//checking G-matrix row for player p and strategies sfrom and sto
Queue<StrategyProfile> skipped
end=false, step=false
StrategyProfile prof = rows.get(p, sfrom, sto)
if (!valid_profile(prof, p)): //reached end of the row
    make strategy sfrom of player p forbidden
    change = true
    continue
//main cycle of iterating through G-matrix row
while (!end && difference(prof, player, sfrom, sto, skipped) < 0):
    step = true
    end = !next_profile(prof, p)
if (step || end):
    while (!skipped.empty()):
        prof = skipped.front()
        skipped.pop()
        //now we use the method difference with active waiting
        //(payoffs should be already computed)
        if (difference(prof, p, sfrom, sto) >= 0):
            end = false
            break
if (end):
    make strategy sfrom of player p forbidden
    change = true
    continue
if (step):
    rows.set(prof, p, sfrom, sto)
```

Figure 3: Algorithm of traversing G-matrix row – object `rows` (of class `GRows`) stores positions where traversing finished last time, variable `change` says whether there have been any change in forbidden strategies since the beginning of the iteration

```
2
3 2
1 3     2 5
2 2     3 1
3 4     2 3
```

Figure 4: Game from table 4 saved in file

```
4
2 1 5 3
2 4
0
2 3 5 11 17
0 1 2
```

Figure 5: Game from table 4 saved in file