Mixed-Integer Programming Methods for Finding Nash Equilibria

AMMM project

Diego Garcia-Olano, Jaime Andres, Jon Lejarreta, Elsa Mullor

CONTENTS :

I - Introduction

The purpose of this document is to explore different solutions based on Mixed Integer Programming and Metaheuristic (GRASP and Simulated Annealing) in order to solve a well known problem in Game theory, the *Nash Equilibrium*. The research document we have based our work on is called "*Mixed-Integer Programming Methods for Finding Nash Equilibria*" by Tuomas Sandholm, Andrew Gilpin and Vincent Conitzer. To get a general understanding of the problem to be solved, we briefly describe what Game Theory and Nash Equilibrium are.

Game Theory is the branch of mathematics concerned with the analysis of strategies for dealing with competitive situations where the outcome of a participant's choice of action depends critically on the actions of the other participants. Players will always try to get the best outcome after their decision in a non cooperative environment.

A game, in strategic or normal form, consists of three different elements:

- A set of players (two players in this case)
- A set of actions or strategies available to each user (pure or mixed strategies)
- A payoff or utility function for each player (gain after a decision is taken)

Strategies can be of two types: pure and mixed.

Pure Strategies contain a complete definition of how a player will play a game. That means it determines the move a player will carry out for any situation she could face.

Mixed Strategies, on the other hand, contain an assignment of probability to each pure strategy. This allows a player to randomly select a pure strategy. Pure strategies are a degenerate of a mixed strategy, where a particular strategy has probability 1 and any other probability 0.

Nash Equilibrium, named after mathematician John Nash, is a set of strategies, one for each player, such that no player has incentive to unilaterally change her action. Players are in equilibrium if a change in strategies by any one of them would lead that player to earn less than if she remained with her current strategy. That is, a situation is created where each of the players is better off not switching positions.

John Nash proved that at least one equilibrium can be found in any finite game, and also that different number of Nash equilibria are possible within a game. An optimal equilibrium is found where none of the players have incentives to move and each of the players is receiving maximum payoff value. Non optimal equilibrium reflects situations where no incentive exists to move from both sides, but sub-optimal payoff is achieved.

These games are displayed using a payoff matrix which shows the players, the strategies and the payoffs. Two numbers per cell represent both payoffs for both players. Moves along the matrix up/down, left/right give the range of options that can be chosen by the players.

In the following example, a payoff matrix is shown where three equilibria are possible.

	Option A	**Option B**	**Option C**
Option A	0, 0	**25, 40**	5, 10
Option B	**40, 25**	0, 0	5, 15
Option C	10, 5	15, 5	**10, 10**

Figure 1 – Game Rewards for players A and B

(B,A), (A,B), and (C,C) are the three equilibria.

Indeed, cell (B,A) contains an optimal Nash equilibrium. 40 is the maximum of the first column (for same player) and 25 is the maximum of the second player. Also 40 is the maximum payoff that could be received overall for that player (same applies to 25 to the other player). (A,B) it is also an optimal equilibrium. It reflects the same situation but in different order. (C,C) is also in equilibrium with (10,10). Moving to the left or up in the matrix would not provide a higher payoff for any of the players. Nevertheless, the payoffs are 30 units away from the maximum overall (40).

Note that an optimal solution is found where the so-called *regret*, difference between that maximum possible payoff and the one achieved in the suboptimal equilibria, is 0. Cases (B,A) and (A,B) show such situations.

As stated before, the mixed integer linear program will try to find optimal Nash equilibrium. The data sets will comprise 6 games with different complexity and size (60x60 and 150x150). The MLP program will be run with CPLEX Optimization Studio. We will see how the program evolves over time while trying to find better and better equilibria, each having smaller *regret.* We will see whether only Nash equilibrium or optimal solutions are found and how long it takes to achieve them.

Two Metaheuristics are then implemented to solve the same problem, GRASP and Simulated Annealing. This time we will see how these algorithms behave while searching the solution spaces and how fast they are able to find solutions.

II - Mathematical Model

The paper proposes different formulations depending on the objective value we want to calculate. In all of them, the following sets and variables are used.
We define *si* as each pure strategy that player *i* can take and *Si* as being the overall set of pure strategies for each player *i*. Each *si* has an associated binary variable *bsi* that corresponds to whether *si* has regret of being played (1) or not (0). The variable *ui* for each player *i* indicates the highest utility attainable given the others mixed strategy, that is, given all the strategies player *i* can take, check what's the one that has the maximum utility for all the other player possible moves. Variable *psi* is defined as the probability of pure strategy *si* being played as part of a mixed strategy for player *i.* Then for each strategy we have the utility for playing that single strategy *usi* and the regret of playing that strategy *rsi*. Finally the variable *Ui* is pre-calculated as the maximum difference between the best and the worst utility for each player.

In our case we seek the value that minimizes the amount of probabilities placed on those strategies that have some regret to be played. Then the intended objective is to minimize the overall sum of them in order to achieve a strategy with the least regret possible. For this, the authors define the auxiliary variable *gsi* to store the value of the probabilities placed and forced to be 1 if that strategy has no regret with a clever sum for not taking those probabilities into account.

Minimize $\sum_{i=0}^{1} \sum_{s_i \in S_i} g_{s_i} - (1 - b_{s_i})$

The following constraints enforce the restrictions of the previously introduced variables.

$$
\begin{aligned}
(\forall i) \sum_{s_i \in S_i} p_{s_i} &= 1 \\
(\forall i)(\forall s_i \in S_i) \quad u_{s_i} &= \sum_{s_{1-i} \in S_{1-i}} p_{s_{1-i}} u_i(s_i, s_{1-i}) \\
(\forall i)(\forall s_i \in S_i) \quad u_i &\geq u_{s_i} \\
(\forall i)(\forall s_i \in S_i) \quad r_{s_i} &= u_i - u_{s_i} \\
(\forall i)(\forall s_i \in S_i) \quad r_{s_i} &\leq U_i b_{s_i} \\
(\forall i)(\forall s_i \in S_i) \quad g_{s_i} &\geq p_{s_i} \\
(\forall i)(\forall s_i \in S_i) \quad g_{s_i} &\geq 1 - b_{s_i}
\end{aligned}
$$

With this model, an optimal solution for the Nash equilibrium problem can be found.

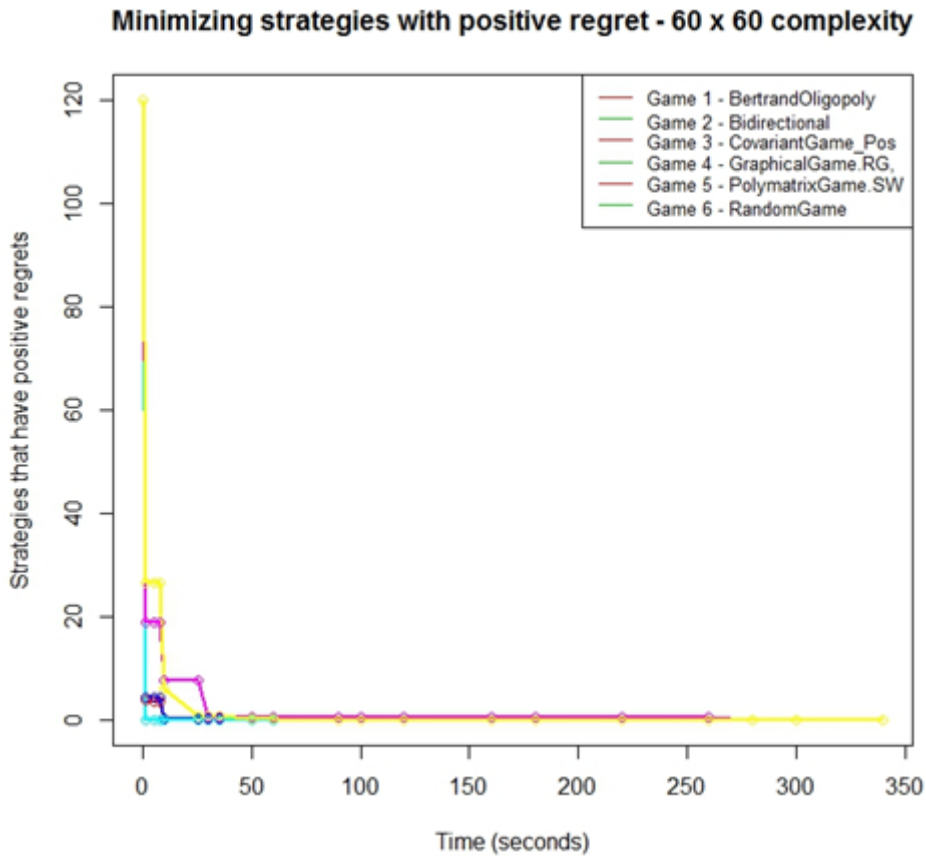
A - Mixed Integer Linear Program Implementation

In order to implement the mathematical model exactly, the IBM ILOG CPLEX for MILP solving is used. We include the code for the model in the addendum so it can be easily tested. In our implementation we preferred to differentiate variables for player 1 and player 2 as the paper only works with two players and this way the code can be better understood.

For our model we have three input variables. First, we introduce *nStrategiesPerPlayer* as an integer value defining the amount of strategies that each player can take, knowing that both players must have the same amount. Next, the payoffs for each combination of pure strategies, represented as a matrix of positive floating numbers, are introduced for player 1 and player 2 by the variables uP1 and uP2.

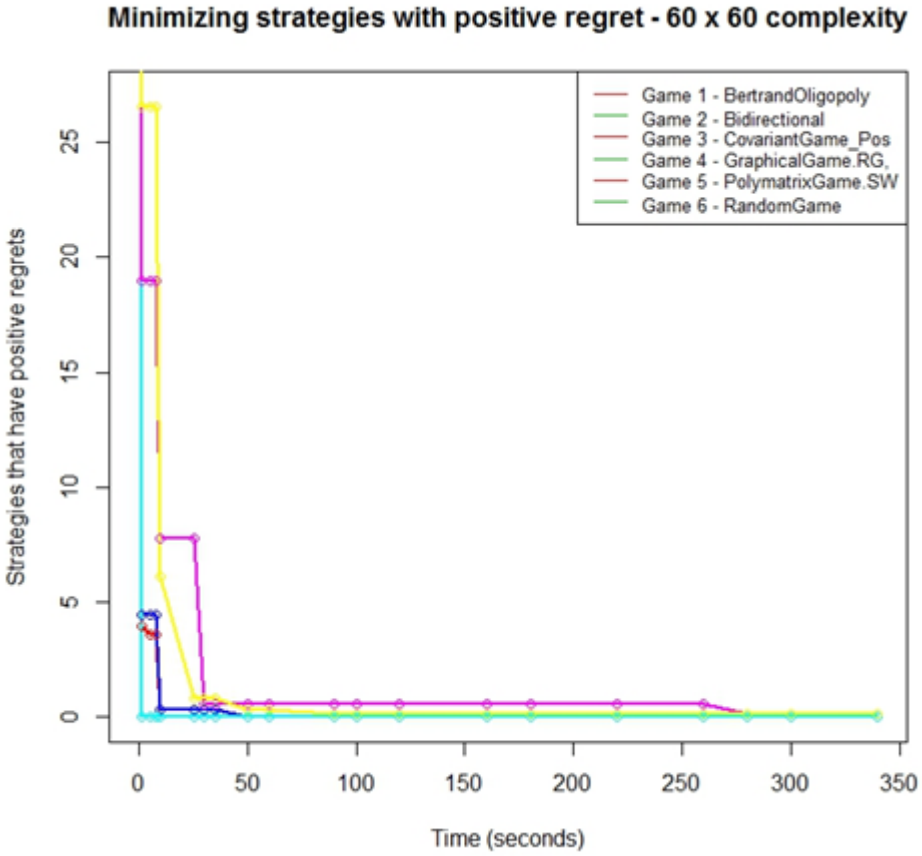
B – Results

First, we attempted to conduct the same simulations as presented in the paper. The paper lacks information on what machines were used for its tests and provides only the average time it took to solve different types of games. Thus we tried first to compute a solution for a 150x150 strategy game, the covariant game type in this instance, which is said to last on average 30 seconds according to the paper. However on our machines running the latest Intel core i7, it didn't terminate in less than an hour hence we assume they used some computer grid they had access to conduct their experiments. In order to reduce computation time while still being able to compare and contrast different game difficulties amongst different heuristics against exact solutions from the paper, we worked with 60 per 60 strategies situations. With this configuration we could complete some of the problems while others still didn't terminate. For the purpose of our work we selected a subset of the game types available to analyze and compare and we show the results below.



Here we compare the improvement of the solution found over time. It starts with the solution that all possible strategies are bad but played hence they all start at 120. Given the different difficulties we can see that there are some games that terminate faster than others. However as mentioned before, there are games that did not end in a short computation time on our machines. In

order to see how differently the games evolve over time we next show a close up of the y-axis from 0 to 25.



We can see that some games quickly reach the Nash equilibrium solution when none of the strategies taken by the players has regret while some take more time such as the one that terminates but takes the longest time, around 275 seconds specifically.

III - GRASP Meta Heuristic

In looking at which heuristics we could apply to the third formulation of the paper, we considered a few, including the biased random key genetic algorithm, and although the encoding of a game theory solution would be possible to use, it seemed a little overly burdensome for the purpose, and we decided to go with an algorithm that was more intuitive to implement given our problem. The code for this section may be found in the 2nd section of the addendum.

A - The Algorithm

The heart of greedy randomized adaptive search procedure essentially works via a two phase iterative process. In the first phase, candidate solutions are created. A solution in our case consists of two vectors, each representing the vector of possibilities of player 1 and player 2's respective strategies. For time considerations, we studied solely 60x60 games. Thus, we have two 60x1 probability vectors which each add up to 1.

To begin, we create an initial random solution for player 1 and player 2. We then create many additional permutations based on that initial solution by iteratively going through the values of player 1 ones vector and for each value, creating a candidate by swapping player 1's value with each value of player 2. Thus we create 60x60 = 3600 candidates which are each different a set of vectors for player 1 and player 2. This is coded in the ConstructGreedySolution() function.

Each set's costs is then assessed using a fitness function which takes a set of player strategy probability vectors and calculates the sum of the total regret of both players. As before, the regret of a player playing a given strategy is the best strategy he could have played given the payoff matrix of the given game and the probability vector of the other player minus what he actually played. This is spelled out in the Fitness() function.

After the costs of each candidate has been calculated, we then a restricted candidate list is constructed by selecting candidates which have a cost below Cmin - α(Cmin - Cmax) where Cmin and Cmax are the candidates with the least and most cost respectively, and α is a learning parameter, set to 0.5 in our case, which essentially defines the threshold for acceptance of candidates. After that, a candidate is selected from the restricted candidate list at random and passed on to phase 2.

In phase two, we conduct a local search around the candidate selected from phase 1. This time we create a new candidate by randomly shuffling the strategy probability vectors of each player, as opposed to swapping between them as we did in the prior phase. We then test the fitness of this new candidate similarly and if this cost is less than the initial candidate sent in, we assign him to be the new best candidate. We then continue iterating by shuffling on the current best candidate, and determining the new candidates' cost versus the current best until we hit a point where we have run 50 iterations in a row without changing the best candidate.

With this final best candidate from phase 2, we compare its cost to the current overall best solution to the problem (for the first iteration we just a random guess as a solution), and swap it to be the best if its cost is lower. We then send the best candidate back to phase 1 and continue the entire process until we have reached a maximum number of iterations (500 in our case) or we have found a solution which has zero regret (and thus cannot be improved).

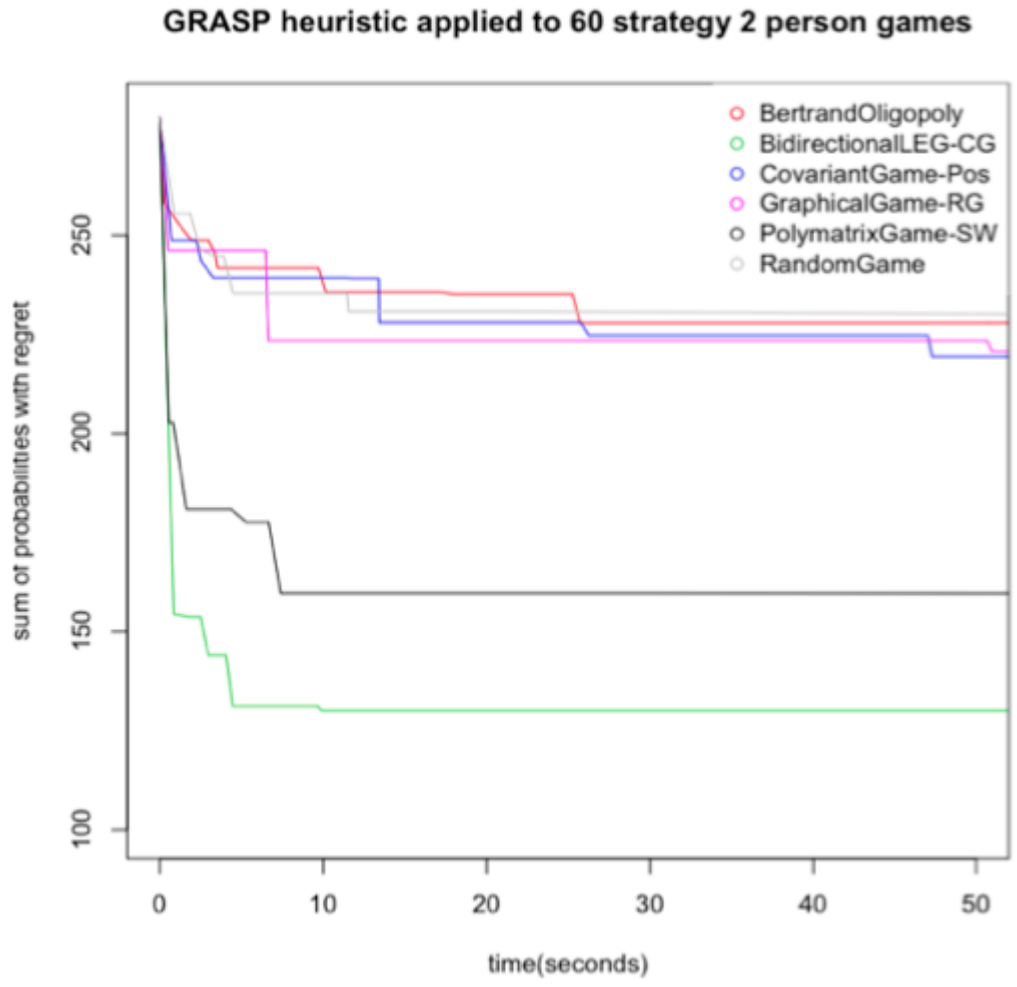
B - Process Notes

As before we are using the GAMUT game generator for 6 different types of 60x60 games. For each game, we randomly generate three different game files using the following command:

java -jar gamut.jar -g BidirectionalLEG-CG -players 2 -actions 60 -normalize -min_payoff 0 -max_payoff 10 -random_params 1 -f BidirectionalLEG-CG-1.game

The list of games used includes BertrandOligopoly, BidirectionalLEG-CG, CovariantGame-Pos, GraphicalGame-RG, PolymatrixGame, and RandomGame. We then converted the files themselves (via vim and grep) directly into formatted multi-dimensional array variables which could be used one by one as input for different runs with our program, which was itself coded in PHP. This language choice was simply based on prior coding experience and the heuristic would be better optimized using C. Additionally, the first phase of the implemented algorithm included two additional ways of generating additional varied candidates in hopes of having a better chance at escaping local optima, but in the end, these added a great deal of extra time to the algorithm and did not improve the results obtained dramatically so they were discarded.

C - Results



The graph above shows the run time averages over three runs for each of the 6 game types to which we applied our GRASP heuristic. It shows how the beginning random solution at time 0 for the average of each game was bad, 300 being the worst possible cost of a solution, and how over time a better solution with a lower cost would be obtained, but that they tended to reach a solution that they could not then better. The graph only shows a minutes worth of execution time, but the behavior of the graphs would continue until the simulations ended. This result implies that after a solution with a certain level of cost was obtained, the initial random construction of a new possible solution at the beginning of phase 1 was not sufficient enough in successive iterations, each with their own permutations, restricted candidate lists and local searches, to better it.

IV – Simulated Annealing

The idea of simulated annealing is to perform a local search around a point p and to accept any point p' that verifies $f(p') \leq f(p)$ but also to be able to accept a point p'', with a non null probability, even if $f(p'') \geq f(p)$. As the probability to accept « worst » position is low, the global behavior of the algorithm is a reduction of the objective function, but this non null probability enables not to be stuck in local minima.

The probability to accept worst position is inspired from Thermodynamics. The formula is

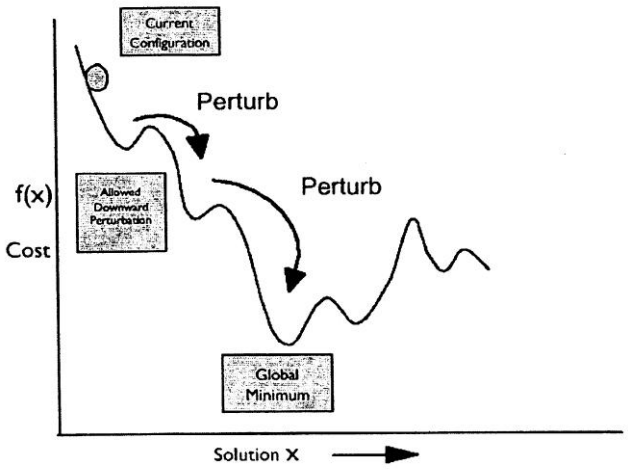
$p = e^{-\frac{\Delta E}{k*T}}$ where $\Delta E = f(p'') - f(p)$. $\Delta E \geq 0$ in the case of a worst position.

k is the Boltzmann constant equal to $1.3806488 \times 10^{-23}$ m^2 kg s^{-2} K^{-1} in the actual Boltzmann law.

And T is the Temperature of the system (in Kelvin). The higher the temperature is, the more agitated the system becomes. This is represented by a higher probability to accept worst position.

The algorithm proceeds as follow:

1) Initialize (randomly)
2) Select neighbor
3) evaluate ΔE
4) if ≤0 accept move
5) If ≥0 accept move with probability P.
6) Loop to 2)



In our problem the fitted value that needs to be minimized is the regret.

The implementation is done with MATLAB.

The code can be found in the appendix 3.

A – Initialization

The first part is the initialization. p1 and p2 are randomly initialized, and then we compute the expected utility, the regret and the normalized regret for both players. K is defined as $1.38 * 10^{-10}$. T is defined as n*N .

N is the input variable, n is the number of strategies. We need T to be large enough so that the program doesn't stop too soon (T is the stopping criteria), but not too large or it twill too easily accept a worse position as next step. If we choose to allow that situation, we just have to increase N. Then the program will also run for a longer time so that it can reach a stable state.

T is decreased every time the loop is run. With time the system goes colder and the probability to accept worst strategies as next step decreases.

B - Loop

The loop is divided in two, one for each player. First we fix the strategies of player two and perform changes on player one 'strategies. Then we reverse the roles.

The neighbor selection follows this rule:

First it's a neighbor of the previous step so one third comes from the current array.

Then we need a random part in the case that no changes are accepted, so that in the following run of the loop the proposed neighbor won't be the same (otherwise the algorithm is stuck in a position).

Then we want to influence the choice in a particular way. The aim is to reduce the regret. So we make the following assumption: The probability to choose a strategy should be higher as the expected utility of this strategy is high. According to that the last third of the new proposed array of strategy will be equal to the normalized array of expected utilities.

This is why

```
p1_new = transpose(expected_u1 / sum(expected_u1));
p1_new = (p1_new+p_random+p1)/3;
```

When the neighbor is constructed, the program just evaluates both regret and normalized regret with this new proposition.

Then ΔE is computed. In this case it's (R_new – R) - and (– ΔE) is (R – R_new).

Here we choose to run the algorithm on the regret. But if we wanted to run it on the normalized regret we just would have to change ΔE to (E_new – E). In this case it is more relevant to plot the normalized regret.

Once ΔE is evaluated we compute the probability p and accordingly move to the next step or not. If the next step is accepted, we add the new values (regret and normalized regret) to their respective array. If the next step isn't accepted, no changes are done and we move to the second player.

The operations are the same for the second player.

In this case we choose to change the names of the variables and not to over-write them (R_new_new and not R_new…) but it is just to facilitate the debugging process.

When both players have tried the next move, we decrease the temperature and loop again.

C – Results

This program is run with six samples of different games theories.

We plot the regret's evolution of all of them:

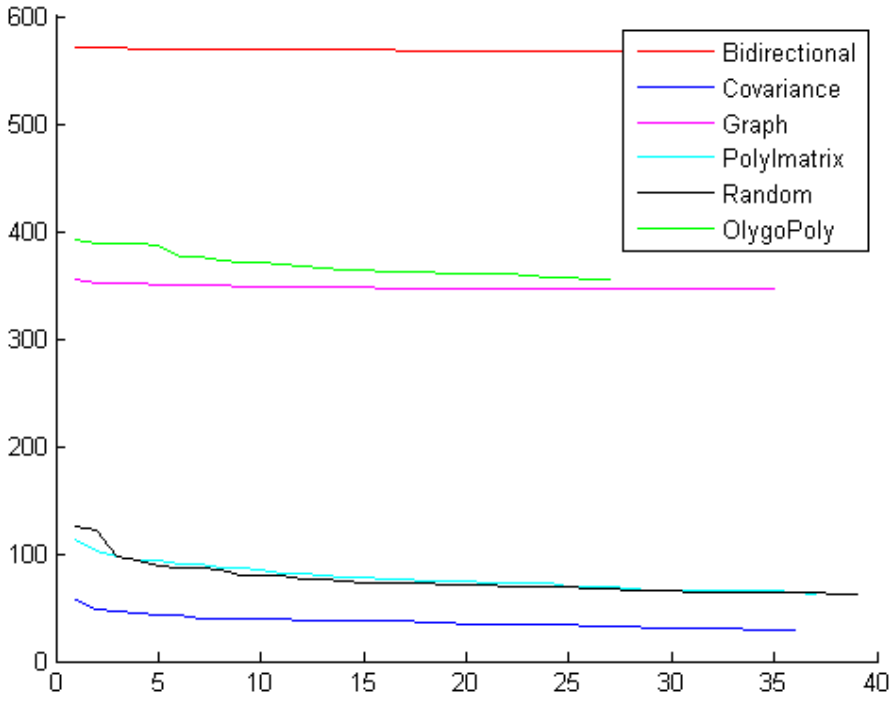


Figure 2 – Regret for all the six games

Here is what we obtained with normalized regret (the code is changed as explained above)

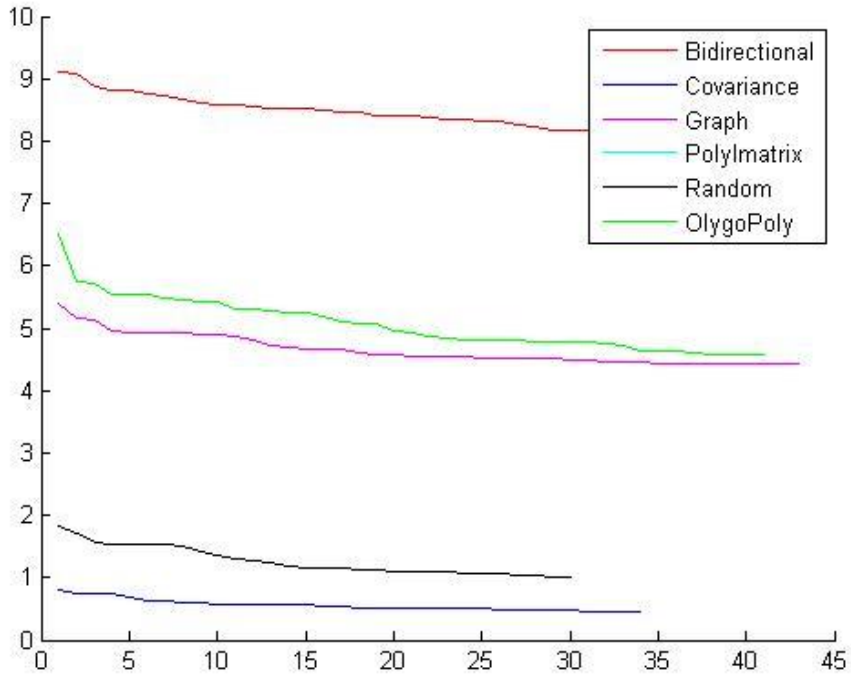


Figure 3 – normalized regret for the 6 games

In all cases we note that the regret decreases globally, but also, that it can accept small increases. The time of computation for those results is around 45s for each game.

To accept a « worse » case more often we can increase the temperature (here the value of N). In that case the graph would be more chaotic.

Playing with the input parameter, we can choose to accept more or less “worst” cases. We could also modify k or the stopping parameter if for instance we want to reach a certain value for the regret.

V – Conclusion

All three of our implementations give similar outcomes. Indeed, the plots showing the evolution of the regret along the time are comparable. In spite of that, we cannot properly compare the results as in the three situations we imposed a stopping criterion and in most cases we didn't reach a Nash equilibrium. With the Cplex implementation some games reached a Nash equilibrium while others didn't terminate and to be able to exploit results we had to make the program stop running. For both GRASP implementation and Simulated Annealing the stopping criterion was the number of iterations of the main loop. Therefore the process time cannot be strictly compared from one implementation to another. What we were able to compare though is the efficiency of these algorithms (How fast the regret is reduced). From the previous plots we can point out that the efficiency of the Cplex program and the Grasp heuristic are comparable. On the other hand the Simulated Annealing implementation seems less efficient. Therefore we can conclude that, in the case of our six games, the Grasp Heuristic or a Cplex implementation are more appropriate to get the best results in a shorter time. From a different point of view, we could also want to determine which of these three implementations comes up with the best results. Although the Simulated Annealing program is less efficient, it is in theory the only one that can bypass local optima. Taking this into account it could be interesting to let the three programs run until they reach a Nash Optima and compare the quality of the strategies proposed. To do so we would have needed more performing computers.

Appendix

1. Code for the .mod file for IBM ILOG CPLEX

```
/*********************************************
* LP implementation of Formulation 3 from the paper
* Mixed-Integer Programming Methods for Finding Nash Equilibria
*********************************************/

/* Ss is the finite range of possible strategies a player can take. Each player has the same amount of strategies*/
int nStrategiesPerPlayer = ...;
range nStrategies = 1.. nStrategiesPerPlayer;

/* s[i] is the set of strategies of player i=1,2.
It basicallly is a lsit of numbers from 1 to nStrategiesPerPlayer */
// splayer1=splayer=nStrategies;

/*b[si] is a binary variable for a strategy si for player i,
    which if set to 1, makes p[si] = 0
      and if set to 0, makes r[si] = 0    */
dvar int+ bP1[s in nStrategies];
dvar int+ bP2[s in nStrategies];

/*p[si] is the probability that strategy s is taken by player i*/
dvar float+ pP1[sP1 in nStrategies];
dvar float+ pP2[sP2 in nStrategies];

/*u[si] is the utility value associated with strategy si being taken by player i
   it is the highest possible expected utility that that player can obtain
    given the other players mixed strategy.

   the agents utility is ui(si , si-1 ), where si is the agents chosen pure strategy (from set Ss),
   and si-1 is the other agents chosen pure strategy
   They actually are float+.
*/
float uP1[sP1 in nStrategies][sP2 in nStrategies] = ...;
float uP2[sP1 in nStrategies][sP2 in nStrategies] = ...;

/* Mean value of utility for a certains trategy based on all the possible strategies played by the other */
dvar float uMeanP1[sP1 in nStrategies];
dvar float uMeanP2[sP2 in nStrategies];

/* Best utility for player i=1,2*/
dvar float+ uBestP1;
dvar float+ uBestP2;

/* g[si] is used to penalize probability placed on strategies with positive
    regret in formulation 3*/
dvar float gP1[sP1 in nStrategies];
dvar float gP2[sP2 in nStrategies];
/* the regret r[si] of pure strategy si is the difference in utility
    for player i between playing an optimal strategy (given the other
    player's mixed strategy) and playing si */
dvar float+ rP1[sP1 in nStrategies]; //explanation: Whats the regreat of p1 playing strat1 from all possible plays that the adversary can do?
dvar float+ rP2[sP2 in nStrategies];

dvar float+ z; //We define the z variable.
//Ui Variables to measure the best situation minus the worst situation for each player.
// initialization of variables used on the constraints. They will be setted on the preprocessing block.
float uDiffP1;
```

```
float uDiffP2;

execute{
 //uDiffP1 init
 var maxP1 = uP1[1][1];
 var minP1 = uP1[1][1];
 //uDiffP2 init:
 var maxP2 = uP2[1][1];
 var minP2 = uP2[1][1];
 for(var sP1=1;sP1<=nStrategiesPerPlayer; sP1++)
 {
     for(var sP2=1; sP2<=nStrategiesPerPlayer; sP2++)
     {
       //Player 1 check.
          if(uP1[sP1][sP2]>maxP1) maxP1=uP1[sP1][sP2];
          if(uP1[sP1][sP2]<minP1) minP1=uP1[sP1][sP2];
        //Player 2 check
        if(uP2[sP1][sP2]>maxP2) maxP2=uP2[sP1][sP2];
          if(uP2[sP1][sP2]<minP2) minP2=uP2[sP1][sP2];
  }
 }
 uDiffP1 = maxP1-minP1;
 uDiffP2 = maxP2-minP2;
}

/*objective function for F3*/
minimize z;

subject to{

  /*constraint 1, probablity of all possible strategies for a player must sum to 1 */
  sum(sP1 in nStrategies)pP1[sP1] == 1;
  sum(sP2 in nStrategies)pP2[sP2] == 1;

  /*constraint 2, the utility of si for player i is equal to
       the summation over the possible strategies s1-i of the other players probability of
       taking that strategy p[s1-i] multiplied by
       the utility player i receives from his play si given the others s1-i */

    forall(sP1 in nStrategies)
         uMeanP1[sP1] == (sum(sP2 in nStrategies) (pP2[sP2] * uP1[sP1][sP2])) ;
  forall(sP2 in nStrategies)
         uMeanP2[sP2] == sum(sP1 in nStrategies) (pP1[sP1] * uP2[sP1][sP2]) ;
  /* constraint 3 defines u[i] as being the highest possible value player can get from
     all possible strategies si he can take */
  forall(sP1 in nStrategies) uBestP1 >= uMeanP1[sP1];
  forall(sP2 in nStrategies) uBestP2 >= uMeanP2[sP2];

  /*Constraint 4 defines the regret value*/
  forall(sP1 in nStrategies) rP1[sP1] == uBestP1 - uMeanP1[sP1];
  forall(sP2 in nStrategies) rP2[sP2] == uBestP2 - uMeanP2[sP2];


  //Constraint 6
  //UP1 and UP2 are calculated on the preprocessing block. (Read Class exercises)
  forall(sP1 in nStrategies) rP1[sP1] <= uDiffP1*bP1[sP1];
  forall(sP2 in nStrategies) rP2[sP2] <= uDiffP2*bP2[sP2];


  //Constraint 9
```

```
  forall(sP1 in nStrategies) gP1[sP1] >= pP1[sP1];
  forall(sP2 in nStrategies) gP2[sP2] >= pP2[sP2];


  //Constraint 10

  forall(sP1 in nStrategies) gP1[sP1] >= 1-bP1[sP1];
  forall(sP2 in nStrategies) gP2[sP2] >= 1-bP2[sP2];


  //aux Constraint for objective function
  z==sum(sP1 in nStrategies, sP2 in nStrategies)
     (gP1[sP1] - ( 1 - bP1[sP1]) + gP2[sP2]-(1-bP2[sP2]));
}
```

2. Code for the Grasp heuristic. RunGrasp.php

```
<?php
// depending on your machines php.ini configuration this may or may not be necessary
ini_set('memory_limit', '9999999999999999999M');

/* sample procedure:
1. select game type you want to run Grasp with by commenting out the appropriate "require_once" line in
 _construct() and saving file.
2. from commandline in folder, type: php RunGrasp.php
3. results will be output directly to screen
*/

class GRASP
{
    public $payoff_matrix;
    public $strategies;
    public function __construct()
    {
        //uncomment out the game data you want to run Grasph with.
                        // only one input file may be used at a time.
        require_once("test_data/BertrandOligopoly-60-1.php");
        /*
        require_once("test_data/BidirectionalLEG-CG-60-1.php");
        require_once("test_data/CovariantGame-Pos-60-1.php");
        require_once("test_data/GraphicalGame-RG-60-1.php");
        require_once("test_data/PolymatrixGame-SW-60-1.php");
        require_once("test_data/RandomGame-60-1.php");
        */

        $this->payoff_matrix = $tmp;
        $this->strategies = 61;
    }

    // Randomized Initial Solution.
    public function ConstructRandomSolution($payoff_matrix){
        //generate random solution UP =
        //candidates are 2 rows by 150 columns
        $p1 = $this->sub_candidate();
        $p2 = $this->sub_candidate();
        $as = array_sum($p1);
        $candidate = array($p1,$p2);

        return $candidate;
    }

    // GreedyConstruction Phase
    public function ConstructGreedySolution($payoff_matrix, $greediness_factor){
        $strategies = $this->strategies;
        $p1 = $this->sub_candidate();
        $p2 = $this->sub_candidate();
        $original_candidate = array($p1,$p2);
        $original_cost = $this->Fitness($original_candidate,$payoff_matrix);

        //player 1 swaps with other player2
        $a2max = 0; $a2min = 100000000000;
        $between_swaps = array();
        for($r=1; $r<$strategies; $r++)
        {
            $between_swaps[$r] = array();
```

```
        for($x=1; $x<$strategies; $x++)
        {
            $temp1 = $p1;
            $temp2 = $p2;
            $temp1[$r] = $p2[$x];
            $temp2[$x] = $p1[$r];
            $temp_candidate = array($temp1,$temp2);
            $temp_cost = $this->Fitness($temp_candidate,$payoff_matrix);
            $between_swaps[$r][$x] = array($temp_candidate, $temp_cost);
            if($temp_cost > $a2max) $a2max = $temp_cost;
            if($temp_cost < $a2min) $a2min = $temp_cost;
        }
    }

    $max_cost = $a2max;
    $min_cost = $a2min;
    $max_minus_min = $max_cost - $min_cost;
    $rcl = array();
    $threshhold = $min_cost + ( $greediness_factor * $max_minus_min);
    $c = 0;
    for($n = 1; $n<$strategies;  $n++)
    {
        for($m = 1; $m<$strategies; $m++){
            if($between_swaps[$n][$m][1] <=  $threshhold)
            {
                $rcl[$c] = array();  $rcl[$c] = $between_swaps[$n][$m];
                $c++;
            }
        }
    }

    //select one from RCL at random and return to then compare against prior Sbest
    $select_index = mt_rand(0,count($rcl)-1);
    $candidate = $rcl[$select_index];
    return $candidate;
}

public function sub_candidate(){
    //generate random solutions which add to 1
    $strategies = $this->strategies;
    $player = array();
    for($i=1; $i < $strategies; $i++){ $player[$i] = mt_rand(0,1000); }
    $as = array_sum($player);
    foreach($player as $k=>$v){ $player[$k] = $v / $as; }
    return $player;
}

public function Fitness($solution,$payoff_matrix){
    //is based on the move both players made and what would have been the best move
    $strategies = $this->strategies;
    $p1 = 2;
    $p2 = 1;
    $fitness = 0;
    $p1_expected_u = array();
    $p2_expected_u = array();

    //solution[0]  is player one's probabilites solution
    //solution[1]  is player two's probabilities solution

    $u1_best = 0;
```

```
    $u2_best = 0;
    for($r=1; $r< $strategies; $r++)
    {
        $s1 = 0;
              $s2 = 0;
        for($c=1; $c<$strategies; $c++)
        {
          //player 1
          $cur_pp1 = $solution[1][$c];          ///ps-1
          $payoff  = $payoff_matrix[$r][$c][0];  //u1(s1,s2)
          $s1 += $cur_pp1 * $payoff;

          //player 2
          $cur_pp2 = $solution[0][$c];
          $payoff2  = $payoff_matrix[$r][$c][1];
          $s2 += $cur_pp2 * $payoff2;

        }

        // usi for every strategy
        $p1_expected_u[$r] = $s1;
        $p2_expected_u[$r] = $s2;

        //final u1 and u2 for both players
        if($u1_best < $s1){ $u1_best = $s1;}
        if($u2_best < $s2){ $u2_best = $s2;}
    }
    $regrets_p1 = array();
    $regrets_p2 = array();

                //regrets for each strategy of the players
    for($x=1; $x<$strategies; $x++){
        $regrets_p1[$x] = $u1_best - $p1_expected_u[$x];
        $regrets_p2[$x] = $u2_best - $p2_expected_u[$x];
    }

    $total_regret = array_sum($regrets_p1)+array_sum($regrets_p2);
    return $total_regret;
}

public function LocalSearch($newSolution,$max_no_improve,$payoff_matrix){
    $no_improve = 0;
    $current_best_cost = $newSolution[1];
    $current_best_candidate = $newSolution[0];

    while($no_improve < $max_no_improve)
    {
        $t1 = $current_best_candidate[0]; shuffle($t1);
        $t2 = $current_best_candidate[1]; shuffle($t2);

        $potential_candidate = array($t1,$t2);
        $potential_cost = $this->Fitness($potential_candidate,$payoff_matrix);
        if($potential_cost < $current_best_cost) {
            $current_best_cost = $potential_cost;
            $current_best_candidate = $potential_candidate;
        }
        else{
            $no_improve++;
        }
    }
```

```
        return $current_best_candidate;
    }

    public function RunGrasp()
    {
        $payoff_matrix = $this->payoff_matrix;
        $max_iter = 500;
        $max_no_improve = 50;
         $greediness_factor = 0.3;

        $newSolution = $this->ConstructRandomSolution($payoff_matrix);
        $newFitness = $this->Fitness($newSolution,$payoff_matrix);

        $BestSolution = $newSolution;
        $BestFitness = $newFitness;

        $iteration = 0;
        while ($iteration < $max_iter && $BestFitness > 0) {
            $PotentialSolution =
                                        $this->ConstructGreedySolution($payoff_matrix,$greediness_factor);

                $PotentialSolutionAfterSearch=
                    $this->LocalSearch($PotentialSolution,$max_no_improve,$payoff_matrix);

            $newFitness = $this->Fitness($PotentialSolutionAfterSearch,$payoff_matrix);

            if ($newFitness < $BestFitness) {
                $BestSolution = $nextnewSolution;
                $BestFitness = $newFitness;
            }
            $iteration++;
        }

        $endTime = microtime();
        $totaltime = $endTime - $startTime;
        print("Best Solution: \n");
        print_r($BestSolution);
        print("Final Regret: $BestFitness\n");
        print("\nTerminated after:  totaltime: $cumulative_total_time\n");
        print("$cumulative_total_time , $BestFitness\n");
    }

}

$grasp = new GRASP();
$grasp->RunGrasp();
?>
```

3. Code for Simulated Annealing

```
function [p1,p2,G,P] = simulatedanealing2(u,v,N)
%p1 and p2 are the final strategies of player 1 and 2
%G is the vector of normalized regret
%P is the vector of regret
%u and v are the utility matrix for the two players
%N is a constant that impacts T
%% Initialisation
tic;   % To measure the time of compilation
k = 1.38*10^(-10);
[n,m] = size(u);
p1 = rand(n,1);
p2 = rand(n,1);
p1 = p1/sum(p1);    %the first p1 and p2 are randomly generated
p2 = p2/sum(p2);
expected_u1 = zeros (1,n);
expected_u2 = zeros (1,n);
expected_u1_new = zeros (1,n);
expected_u2_new = zeros (1,n);
%The expected utility are required to calculate the regret
for i=1:n
   expected_u1(i) = u(i,:)*p2;
   expected_u2(i) = transpose(p1) * v(:,i);
end
expected_max1 = max(expected_u1);
expected_max2 = max(expected_u2);
max1 = ones(1,n)*expected_max1;
max2 = ones(1,n)*expected_max2;
regret1 = (max1-expected_u1)*p1;     % Normalized regret
regret2 = (max2-expected_u2)*p2;
R = sum(max1-expected_u1) + sum(max2-expected_u2);  % Regret
P = [R];
E = regret1 + regret2;
G = [E];
%Annealing variables
T=N*n;
%/Annealing variables
%% loop begins
while (T>1)  % T is the stopping criterion

p_random = rand(n,1);
p_random = p_random/sum(p_random);
           %CHANGEP1
           %select a new p1 in neighborhood
           p1_new = transpose(expected_u1 / sum(expected_u1));
           p1_new = (p1_new+p_random+p1)/3;


           for i=1:n
               expected_u2_new(i)= transpose(p1_new)*v(:,i);
           end
           max2_new = ones(1,n)* max(expected_u2_new);
           regret2_new = (max2_new - expected_u2_new)*p2;
           regret1_new = (max1 - expected_u1)*p1_new;
           E_new = regret1_new + regret2_new;
           R_new = sum(max1 - expected_u1)+ sum(max2_new -
expected_u2_new);

           %Annealing variables
           p=exp((R - R_new)/(k*T));
```

```
            chance = rand(1,1);
            %/Annealing variables

            % evaluate Delta(R) and proceed next step
            if (R_new < R)|((R_new>R)&(chance<p))
                p1 = p1_new;
                expected_u2=expected_u2_new;
                max2 = max2_new;
                regret2 = regret2_new;
                regret1 = regret1_new;
                E = E_new;
                R = R_new;
                P = horzcat(P,[R]);
                G=horzcat(G,[E]);


            end

            %CHANGEP2
p_random = rand(n,1);
p_random = p_random/sum(p_random);
            p2_new = transpose(expected_u2 / sum(expected_u2));
            p2_new = (p2_new+p_random+p2)/3;

            for i=1:n
                expected_u1_new(i)=u(i,:)*p2_new;
            end

            max1_new = ones(1,n)* max(expected_u1_new);
            regret1_new_new = (max1_new-expected_u1_new)*p1;
            regret2_new_new = (max2-expected_u2)*p2_new;
            E_new_new = regret1_new_new + regret2_new_new;
            R_new_new = sum(max1_new - expected_u1_new) + sum(max2 -
expected_u2);

            %Annealing variables
            p=exp((R - R_new_new)/(k*T));
            chance = rand(1,1);
            %/Annealing variables
            if (R_new_new < R)|((R_new_new>R)&(chance<p))
                p2 = p2_new;
                expected_u1=expected_u1_new;
                max1 = max1_new;
                regret2 = regret2_new_new;
                regret1 = regret1_new_new;
                R = R_new_new;
                P = horzcat(P,[R]);
                E = E_new_new;
                G=horzcat(G,[E]);

            end

            %reduce T
            T=T*(1-1/(2*N*log(2)));
end
toc;
end
```