AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

FACULTY OF COMPUTER SCIENCE, ELECTRONICS AND TELECOMMUNICATIONS

DEPARTMENT OF COMPUTER SCIENCE

Master of Science Thesis

Implementacja algorytmów dla systemów Monroe i Chamberlina-Couranta z nieliniową funkcją satysfakcji

Implementation of algorithms for Monroe and Chamberlin-Courant systems under nonlinear satisfaction function

Author: *Piotr Szmigielski*
Degree programme: *Computer Science*
Supervisor: *Piotr Faliszewski, PhD*

Kraków, 2016

Oświadczamy, świadomi odpowiedzialności karnej za poświadczenie nieprawdy, że niniejszą pracę dyplomową wykonaliśmy osobiście i samodzielnie (w zakresie wyszczególnionym we wstępie) i że nie korzystaliśmy ze źródeł innych niż wymienione w pracy.

Contents

1. Introduction

We study the effectiveness of algorithms for approximate winner determination under the Monroe [1] and Chamberlin-Courant [2] multiwinner voting rules using nonlinear satisfaction function. The purpose of both these rules is to select a group of candidates that best represent the voters. Having good voting rules and algorithms for them is important, because multiwinner elections are used both in human societies (e.g., for parliament elections) and particular software systems (e.g., in recommendation systems [3]). Rules studied in this paper are exceptionally interesting because they have two desired features of multiwinner rules: they provide accountability (there is a direct connection between the elected candidates and the voters, so each voter has a representative assigned to her and each candidate knows who she represents) and proportional representation of the voters' views.

We assume that candidates participate in the election with multiple winners (a committee with multiple members is selected) and they are elected by voters, each of whom ranks all the candidates (each voter provides a linear order over the set of candidates expressing her preferences). For each voter the Monroe and Chamberlin-Courant rules assign a single candidate as her representative (with some constraints, which are detailed in the further part of the thesis).

The candidates are selected and assigned to the voters optimally, by maximizing the total satisfaction of all the voters. The total satisfaction is calculated as a sum of individual satisfactions of the voters. We assume that there is a satisfaction function that measures how well a voter is represented by the candidate. The function is the same for each voter. It is a decreasing function, so a voter is more satisfied if the candidate assigned to her is ranked higher. In this thesis we study cases in which the satisfaction function is a nonlinear one.

Monroe and Chamberlin-Courant systems may be potentially very useful, because they are one of the few multiwinner election systems that provide both accountability and proportional results. Most of the currently used voting rules lack at least one of these properties. For example, D'Hondt method used to elect members of Polish lower house of parliament lacks accountability (specific parliament members are accountable to the political parties, not to the voters), while single-member constituency plurality system used for United Kingdom parliament elections lacks proportionality.

The main drawback of the aforementioned rules is that their winner determination problems are NP-hard [4, 3, 5] which makes them hard to use in practice, as it would force the use of algorithms that do not provide an optimal result for every data set. Therefore, using these systems for real-life elections may rise some difficulties. However, they can be used for the recommendation systems conveniently,

as a good but not optimal recommendation is still useful. Lu and Boutilier [3] and Skowron et al. [6] provided approximation algorithms for both the rules that return near-optimal results for various test cases (including real-life data and synthetic data), but for linear satisfaction functions only.

In this thesis we focus on providing several algorithms for the Monroe and Chamberlin-Courant rules using nonlinear satisfaction functions and evaluating them empirically against various data sets. For smaller data, results can be easily assessed by comparing them to the optimal result (calculated with the brute-force algorithm). For bigger data, the upper bound of the optimal result must be used for comparison. As finding optimal solution is NP-hard [4, 3, 5], there is a need to provide good algorithms which can compute results that are suboptimal, but still as close to optimal as possible. We implement and evaluate various heuristic algorithms, as well as the existing approximation algorithms for the linear satisfaction function, but applying them to the nonlinear cases.

Skowron et. al [6] have already provided approximation algorithms for these systems, but only under linear satisfaction function. In this thesis we focus on providing algorithms for non-linear satisfaction functions, as they can better reflect real preferences of the voters.

1.1. Results

TODO

2. Preliminaries

In the first part of this chapter we explain basic notions regarding multiwinner elections. Next, we present definitions of winner determination problems under Monroe and Chamberlin-Courant (abbreviated as CC) voting rules. Finally, we present some notions regarding approximation algorithms for these tasks.

2.1. Basic Notions

In these section we present essential definitions and notions concerning multiwinner elections.

Definition 1: Preferences. [6] Let's assume we have n agents (representing voters in the elections) and m alternatives (representing candidates in the elections). For each agent i, her *preference order* is a strict linear order $\succ_i$ over all the alternatives that ranks them from the most to the least desirable one. Collection V of the preference orders of all the agents is called a *preference profile*.

If A is the set of all the alternatives and B is a nonempty strict subset of A, then by $B \succ A - B$ we mean that all alternatives in B are preferred to those outside of B for the preference order $\succ$.

Definition 2: Positional scoring function. [6] Let's assume we have m alternatives. A function $\alpha^m : \{1, ..., m\} \to \mathbb{N}$ that assigns an integer value to each position in the agent's preference order is called a *positional scoring function* (PSF).

If α^m is a decreasing function (for each $i, j \in \{1, ..., m\}$, $i > j \implies \alpha^m(i) < \alpha^m(j)$), it is called a *decreasing positional scoring function* (DPSF) and it can represent an agent's satisfaction when an alternative from a particular position in her preference order is elected. For every DPSF $\alpha^m(m) = 0$, so an agent is not satisfied at all with her worst alternative. In some cases, we will write α instead of α^m to simplify notation.

A DPSF α^m is considered a *linear satisfaction function* if for each $i \in \{1, ..., m-1\}$ value $\alpha^m(i+1) - \alpha^m(i)$ is constant. Every DPSF not satisfying that constraint is considered a *nonlinear*

satisfaction function.

We define a family α of DPSFs as $\alpha = (\alpha^m)_{m\in\mathbb{N}}$, where α^m is a DPSF on $\{1, ..., m\}$, such that $\alpha^{m+1}(i+1) = \alpha^m(i)$ for all $m \in \mathbb{N}$ and $i \in \{1, ..., m\}$. Families of DPSFs are built iteratively by prepending values to functions with smaller domains (smaller m), leaving existing values from the previous function (previous family member) unchanged. Such families of DPSFs are called *normal* DPSFs.

Definition 3: Assignment functions. [6] Let's assume we have n agents and m alternatives. Let B be a set of all the agents ($B = \{1, ..., n\}$). A *K-assignment function* $\Phi : B \to \{a_1, ..., a_m\}$ is a function that assigns a single alternative to every agent in such way, that no more than K alternatives are selected ($\|\Phi(B)\| \leq K$). It is called a *Monroe K-assignment function* if it additionally satisfies the following constraint: For each alternative a we have that either $\left\lfloor \frac{\|B\|}{K} \right\rfloor \leq \|\Phi^{-1}(a)\| \leq \left\lceil \frac{\|B\|}{K} \right\rceil$ or $\|\Phi^{-1}(a)\| = 0$. It means that for Monroe K-assignment function, agents are assigned to exactly K alternatives and each of the alternatives has about $\frac{\|B\|}{K}$ agents assigned. If we have an assignment function Φ, alternative $\Phi(i)$ is called the *representative* of agent i.

Additionally, if we allow the K-assignment function to assign an empty alternative ($\perp$) to the agents, it is called a *partial K-assignment function*. It is also a *partial Monroe K-assignment function* if it can be extended to a regular Monroe K-assignment function by replacing all the empty alternatives with the regular alternatives ($\{a_1, ..., a_m\}$).

Let S be a set of alternatives. By Φ^S we mean a K-assignment function (or a partial K-assignment function) that assigns agents only to alternatives from S.

Definition 4: Total satisfaction function. [6] We assume that α is a normal DPSF and $pos_i(x)$ represents position of alternative x in the i'th agent's preference order. Following function assigns a positive integer to a given assignment Φ:

$$l^{\alpha}_{sum}(\Phi) = \sum_{i=1}^{n} \alpha(pos_i(\Phi(i))) \tag{2.1}$$

This function combines satisfaction of the agents to assess the quality of the assignment for the entire society. It simply calculates the sum of the individual agents' satisfaction value and is used as the *total satisfaction function*.

For each subset of the alternatives $S \subseteq A$ that satisfies $\|S\| \leq K$, by Φ^S_α we mean the partial K-assignment (or the partial Monroe K-assignment) that assigns agents only to the alternatives from S and such that Φ^S_α maximizes the total satisfaction $l^{\alpha}_{sum}(\Phi^S_\alpha)$.

2.2. Monroe and Chamberlin-Courant Rules

We will now define the problems of winner determination under the Monroe and CC rules. The goal is to find an optimal assignment function, where by the optimal function we accept one that maximizes the total satisfaction.

Definition 5: Chamberlin-Courant and Monroe problems. [6] Let's assume we have n agents, m alternatives, preference profile V, $K \in \mathbb{N}$ ($K < m$) representing committee size and a normal DPSF α. The goal of the *Chamberlin-Courant problem* is to find a K-assignment function Φ for which the total satisfaction $l^{\alpha}_{sum}(\Phi)$ is maximal under preference profile V. The goal of the *Monroe problem* is similar, but it searches for a Monroe K-assignment function instead.

Intention of solving these problems is to find a (Monroe) K-assignment function which returns a set of K alternatives, who are viewed as the winners of the given multiwinner election (e.g. elected members of a committee).

2.3. Approximate Solutions

As for many normal DPSFs multiwinner election problems under both Monroe and CC rules are NP-hard [4, 3, 5], we are looking for approximate solutions.

Definition 6: Approximation algorithms. [6] Let r be a real number such that $0 < r \leq 1$, let α be a normal DPSF. An algorithm is an *r-approximation algorithm* for CC or Monroe problem if for every correct input it returns an assignment Φ such that $l^{\alpha}_{sum}(\Phi) \geq r \cdot s_{max}$, where s_{max} is the optimal total satisfaction $l^{\alpha}_{sum}(\Phi_{max})$.

3. State of the Art

The Chamberlin-Courant and Monroe multiwinner voting rules were introduced by Chamberlin and Courant [2] and Monroe [1] respectively. Despite having desired properties of multiwinner rules when we want to achieve a proportional representation [7], there has not been much research regarding these systems yet.

Complexity of both the systems was studied in the several papers and the conclusion is that they are both NP-hard in general case. Procaccia et al. [5] showed that these systems are NP-hard in the dissatisfaction-based framework in case of approval dissatisfaction function, Lu and Boutilier [3] presented Chamberlin-Courant rule hardness under linear satisfaction function, while Betzler et al. [4] studied the parameterized complexity of the rules. There were also papers that studied the complexity for some specific cases where preferences are either single-peaked (Yu et al. [8]) or single-crossing (Skowron et al. [9]).

Lu and Boutilier [3] were the first to study the approximability of the Chamberlin-Courant rule under linear satisfaction function, they presented an approximation algorithm for this case. Skowron et al. [6] gave more approximation algorithms for both the Monroe and Chamberlin-Courant rules (for the linear satisfaction function) and assessed their effectiveness against various data sets, but there is no work showing results for any nonlinear satisfaction function.

4. Implemented Algorithms

In this chapter we present implemented algorithms for the utilitarian versions of Monroe and Chamberlin-Courant multiwinner voting rules in the satisfaction-based framework.

Proposition 1 (Implicit in the paper of Betzler et al. [4]). Let α be a normal DPSF, N be a set of agents, A be a set of alternatives, V be a preference profile of N over A, and S a K-element subset of A (where K divides $\|N\|$). Then there is a polynomial-time-algorithm that computes a (possibly partial) optimal K-assignment Φ_α^S (Monroe K-assignment Φ_α^S) of the agents to the alternatives from S.

4.1. Existing Algorithms

In this section we present algorithms that already exist, but have never been applied to nonlinear cases before.

4.1.1. Algorithms A, B and C

Algorithm A was first presented by Skowron et al. [6] and tries to solve Monroe problem. It is a greedy algorithm that executes K iterations (where K is a size of the elected committee). In every iteration algorithm selects an alternative a_i that has not been assigned yet and assigns it to $\frac{N}{K}$ of the remaining agents whose satisfaction of being assigned to a_i is maximal (criterion for picking an alternative in each step is a sum of satisfaction of $\frac{N}{K}$ agents selected this way). This algorithm runs in polynomial time [6]. Pseudocode is presented in Algorithm 1.

Algorithm B is an extension to Algorithm A and was presented in the same paper. [6] In the first step, Algorithm B simply executes Algorithm A. Next, it uses algorithm from Proposition 1 to optimally assign alternatives from the winners set to the agents. As both the utilized algorithms run in polynomial time, so does Algorithm B.

Algorithm C is a further extension to Algorithm B, also presented by Skowron et al. [6]. While Algorithm B only keeps one partial assignment function Φ that is extended in each step until it

Algorithm 1 Algorithm A

procedure COMPUTEMONROEPROBLEMSOLUTION
  $\Phi \leftarrow$ a map defining a partial assignment, iteratively built by the algorithm
  $\Phi^{\leftarrow} \leftarrow$ the set of agents for which the assignment is already defined
  $\Phi^{\rightarrow} \leftarrow$ the set of alternatives already used in the assignment
  $\Phi = \{\}$
  for $i \leftarrow 1$ to K **do**
    $score \leftarrow \{\}$
    $bests \leftarrow \{\}$
    for all $a_i \in A \setminus \Phi^{\rightarrow}$ **do**
      $agents \leftarrow$ sort $N \setminus \Phi^{\leftarrow}$ so that if agent j preceeds agent j' then $pos_j(a_i) \leq pos_{j'}(a_i)$
      $bests[a_i] \leftarrow$ choose first $\frac{N}{K}$ elements from $agents$
      $score[a_i] \leftarrow \sum_{j \in bests[a_i]}(m - pos_j(a_i))$
    $a_{best} \leftarrow argmax_{a \in A \setminus \Phi^{\rightarrow}} score[a]$
    for all $j \in bests[a_{best}]$ **do**
      $\Phi[j] \leftarrow a_{best}$

becomes a complete solution, Algorithm C stores a list of d functions (d is provided as an algorithm parameter). In each step, for each alternative a with no agent assigned and for each Φ of the d functions stored, algorithm computes a greedy extension to Φ that assigns $\frac{N}{K}$ agents (that are not assigned to any other alternative yet) to a (the same way as in Algorithm A). For the next step, d functions that return the highest satisfaction are used. Finally, after the last iteration, winners are reassigned using algorithm from Preposition 1 in each of the stored functions. Function that gives the highest satisfaction is selected. If $d = 1$, Algorithm C is identical with Algorithm B. Pseudocode is presented in Algorithm 3.

Unlike previous algorithms, Algorithm C can be used for both Monroe and Chamberlin-Courant rules. To adapt it to the Chamberlin-Courant rule, we have to replace the entire first for all loop with the appropriate code, presented in Algorithm 2.

Algorithm 2 Algorithm C - CC for all code replacement

for all $a_i \in A \setminus \Phi^{\rightarrow}$ **do**
  $\Phi' \leftarrow \Phi$
  for all $j \in N$ **do**
    if agent j prefers a_i to $\Phi'(j)$ **then**
      $\Phi'(j) \leftarrow a_i$
  $newPar.push(\Phi')$

Algorithm 3 Algorithm C

procedure COMPUTEMONROEPROBLEMSOLUTION
    $\Phi \leftarrow$ a map defining a partial assignment, iteratively built by the algorithm
    $\Phi^{\leftarrow} \leftarrow$ the set of agents for which the assignment is already defined
    $\Phi^{\rightarrow} \leftarrow$ the set of alternatives already used in the assignment
    $Par \leftarrow$ a list of partial representation functions
    $Par = []$
    $Par.push(//)$
    for $i \leftarrow 1$ to K **do**
        $newPar = []$
        for $\Phi \in Par$ **do**
            $bests \leftarrow \{\}$
            for all $a_i \in A \setminus \Phi^{\rightarrow}$ **do**
                $agents \leftarrow$ sort $N \setminus \Phi^{\leftarrow}$ (agent j preceeds agent j' implies that $pos_j(a_i) \leq pos_{j'}(a_i)$
                $bests[a_i] \leftarrow$ choose first $\frac{N}{K}$ elements of $agents$
                $\Phi' \leftarrow \Phi$
                for all $j \in bests[a_i]$ **do**
                    $\Phi'[j] \leftarrow a_i$
                $newPar.push(\Phi')$
        sort $newPar$ according to descending order of the total satisfaction of the assigned agents
        $Par \leftarrow$ choose first d elements of $newPar$
    for $\Phi \in Par$ **do**
        $\Phi \leftarrow$ compute the optimal representative function using an algorithm of Betzler et al. [4] for the set of winners $\Phi^{\rightarrow}$
    return the best representative function from Par

4.1.2. Algorithm R

As shown by Skowron et al. [6], algorithms A, B and C are potentially very useful when size of the committee is much lower than the number of alternatives (K is small compared to m), as under this condition they produce results with very high approximation ratio under linear satisfaction function. For the other cases (relatively large committee), a sampling-based randomized algorithm may be used. We call it Algorithm R. We expect that under nonlinear satisfaction function algorithms should behave similarly in relation to each other as under a linear one.

Algorithm R randomly picks K alternatives and then, using Proposition 1, assigns them to agents optimally. As a single execution of such an algorithm may simply pick only alternatives that are ranked low, random assignment should be computed a given number of times (which is provided as an algorithm parameter k), so there is a greater probability to attain a high quality solution. If size of the committee is only slightly lower than the number of alternatives (K is comparable to m) then the probability of finding a solution that is at least close to optimal is high [6]. Algorithm can naturally be used for both Monroe and Chamberlin-Courant systems.

4.1.3. Algorithm AR

Algorithm family A-C and Algorithm R are naturally suitable for different cases. Therefore, Skowron et al. [6] proposed to combine algorithms A and R into Algorithm AR. They also showed that under linear satisfaction function, Algorithm AR can achieve approximation ratio of $0.715 - e$ with probability λ. Both e and λ are provided as algorithm parameters. Naturally, for different satisfaction functions approximation ratio may vary, but we decided to test the algorithm in an unchanged version for comparison. Pseudocode is presented in Algorithm 4.

Algorithm 4 Algorithm AR

procedure COMPUTEMONROEPROBLEMSOLUTION
  H_j is the j'th harmonic number $H_j = \sum_{i=1}^{j}(\frac{1}{i})$
  $\lambda \leftarrow$ probability of achieving the approximation ratio $0.715 - e$ under linear satisfaction function
  if $\frac{H_K}{K} \geq \frac{e}{2}$ **then**
    compute the optimal solution using an algorithm of Betzler et al. [4] and return
  if $m \leq 1 + \frac{2}{e}$ **then**
    compute the optimal solution using a simple brute force algorithm and return
  $\Phi_1 \leftarrow$ solution computed by Algorithm A
  $\Phi_2 \leftarrow$ solution computed by Algorithm R (sampled $\log(1 - \lambda) \cdot \frac{2+e}{e}$ times)
  return the better assignment among Φ_1 and Φ_2

4.1.4. Algorithm GM

Algorithm GM (greedy marginal improvement) is an algorithm that was introduced by Lu and Boutilier [3] for the Chamberlin-Courant rule only. However, it was later generalized by Skowron et al. [6], so it can be applied to the Monroe rule as well. In the Monroe case, it can be considered as the Algorithm B improvement.

Algorithm starts with an empty set S. In each iteration of the algorithm an alternative a ($a \notin S$) is selected and added to S. Selected a maximizes the satisfaction value $l^{\alpha}_{sum}(\Phi^{S\cup\{a\}}_{\alpha})$. Iterations are executed until a complete committee is selected, so K iterations are required. For Monroe case, computing Φ^{S}_{α} is slow (it is achieved by using min-cost/max-flow algorithm [4]), which makes the algorithm execute for a relatively long time, as many computations are required. Pseudocode is presented in Algorithm 5.

Algorithm 5 Algorithm GM

1: **procedure** COMPUTECCORMONROEPROBLEMSOLUTION
2: Φ^{S}_{α} - the partial assignment that assigns a single alternative to at most $\frac{n}{K}$ agents, that assigns to the agents only the alternatives from S, and that maximizes the utilitarian satisfaction $l^{\alpha}_{sum}(\Phi^{S}_{\alpha})$
3: $S \leftarrow \emptyset$
4: **for** $i \leftarrow 1$ to K **do**
5: $a \leftarrow argmax_{a\in A\setminus S} l^{\alpha}_{sum}(\Phi^{S\cup\{\alpha\}}_{\alpha})$
6: $S \leftarrow S \cup \{a\}$
7: **return** Φ^{S}_{α}

4.1.5. Algorithm P

Algorithm P can only be applied to Chamberlin-Courant problem. It was first introduced by Skowron et al. [6]. In the beginning, it computes x (a non-negative integer). Next, it computes an assignment that should maximize the number of agents that have an alternative from the first x spots in their preferences assigned to them. This process is executed greedily. Afterwards, if there are still agents with no alternative assigned, the best alternative is picked from the ones already selected for at least one other agent.

$w(x)$ used in the algorithm is a Lambert's W-function, defined to be the solution of the equation $x = w(x)e^{w(x)}$. Algorithm runs in polynomial time. Pseudocode of Algorithm P is presented in Algorithm 6.

Algorithm 6 Algorithm P

procedure COMPUTECCPROBLEMSOLUTION
  $\Phi \leftarrow$ a map defining a partial assignment, iteratively built by the algorithm
  $\Phi^{\leftarrow} \leftarrow$ the set of agents for which the assignment is already defined
  $\Phi^{\rightarrow} \leftarrow$ the set of alternatives already used in the assignment
  $num\_pos_x(a) \leftarrow \|\{i \in [n] \setminus \Phi^{\leftarrow} : pos_i(a) \leq x\}\|$ - the number of not-yet assigned agents that rank alternative a in one of their first x positions
  $w(\cdot)$ - Lambert's W-function
  $\Phi = \{\}$
  $x = \left\lceil \frac{mw(K)}{K} \right\rceil$
  for $i \leftarrow 1$ to K **do**
    $a_i \leftarrow argmax_{a \in A \setminus \Phi^{\rightarrow}} num\_pos_x(a)$
    for all $j \in [n] \setminus \Phi^{\leftarrow}$ **do**
      if $pos_j(a_i) < x$ **then**
        $\Phi[j] \leftarrow a_i$
  for all $j \in A \setminus \Phi^{\leftarrow}$ **do**
    $a \leftarrow$ such alternative from $\Phi^{\rightarrow}$ that $\forall_{a' \in \Phi^{\rightarrow}} pos_j(a) \leq pos_j(a')$
    $\Phi[j] \leftarrow a$
  return Φ

4.2. New Algorithms

In this section we present algorithms that have been invented and implemented specifically for the purpose of this thesis.

4.2.1. Genetic Algorithm

The idea of our algorithm is loosely based on metaheuristic genetic algorithms [10], such as the Firefly Algorithm [10, 11].

Our Genetic Algorithm starts with an initial set of creatures (each of them presenting a set of possible winners under Chamberlin-Courant or Monroe rule) which are later mutated and crossed over with each other. The best creatures (ones with the highest total satisfaction) are preferred for further mutation and crossover in order to better investigate the neighbourhood of local maxima, but on the other hand algorithm also produces new creatures by crossing over random existing ones to better explore the entire solution space, not limiting to local extrema.

In each iteration of the algorithm creatures are evaluated (winners are assigned to agents using algorithm from Preposition 1 and satisfaction is computed). Best creature in terms of total satisfaction

is compared with currently best found creature and takes its place if it is better. Half of the evaluated creatures (the best ones) are chosen for further propagation. Each of them is then mutated randomly. Remaining creatures are created by crossing over random creatures from the "better" half with each other. Resulting set of creatures is used for the next iteration. Number of iterations and number of creatures are the algorithm parameters. Pseudocode is presented in Algorithm 7.

Algorithm 7 Genetic Algorithm

procedure COMPUTECCORMONROEPROBLEMSOLUTION
  I - number of iterations
  c - number of creatures
  Φ_{best} - best creature (preference profile)
  $creatures \leftarrow$ generate initial random set of c creatures
  for $i \leftarrow 1$ to I **do**
    $creaturesSorted \leftarrow$ sort $creatures$ by total satisfaction
    if $satisfaction(creaturesSorted[1]) > satisfaction(\Phi_{best})$ **then**
      $\Phi_{best} = creaturesSorted[1]$
    $bestCreatures \leftarrow$ choose first $c/2$ elements from $creaturesSorted$
    $mutated \leftarrow$ mutate all creatures from $bestCreatures$ randomly
    $crossed \leftarrow$ crossover random creatures from $bestCreatures$ to produce $c/2$-element set
    $newCreatures \leftarrow mutated \cup crossed$
    $creatures \leftarrow newCreatures$
  return Φ_{best}

4.2.2. Simulated Annealing

The Simulated Annealing algorithm is inspired by physical annealing process, used in metallurgy. It involves heating and cooling a material to make it attain specific physical properties. Using simulated annealing for optimization of a complex function depending on many parameters was proposed by Kirkpatrick et al. [12]. We adapt it to the Chamberlin-Courant and Monroe problems.

In simulated annealing we use a temperature variable, which has a high value at the beginning and gets lower ('cools') during the execution. When the temperature is high, algorithm can accept solutions worse than the current one more frequently, so it is possible to leave a local optimum, as the global one may be in totally different area of the search space. When the temperature gets lower, algorithm focuses on the area where solution close to the optimum may lie.

To decide if the new solution should be accepted, the *acceptance function* is used. If the new solution is better, it is always accepted. If it is worse, acceptance function accepts the new solution with probability

p, which is calculated as follows:

$$p = \exp(\frac{E_c - E_n}{T}) \tag{4.1}$$

E_c is the current solution energy, E_n is the new solution energy and T is the temperature. Energy represents quality of the solution and, in our case, is proportional to the total satisfaction value of the solution.

The algorithm proceeds as follows. First, it generates a random initial set of winners. Initial temperature is given as a parameter. Then, in each iteration, a new solution is generated by replacing one of the winners in the current solution with a random alternative that is not a winner in the current solution (every time winners are assigned to agents using algorithm from Preposition 1). The newly created solution is then evaluated by the acceptance function. If it is accepted, it replaces the current solution. Otherwise, the current solution is kept. For the next iteration, temperature is decreased:

$$T \leftarrow T \cdot (1 - c) \tag{4.2}$$

c is the cooling rate, provided as an algorithm parameter. The algorithm stops when $T \leq 1$. Pseudocode is presented in Algorithm 8.

Algorithm 8 Simulated Annealing

procedure COMPUTECCORMONROEPROBLEMSOLUTION
    T_{start} - initial temperature
    c - cooling rate
    Φ_{curr} - current solution
    T - current temperature
    $E(\Phi)$ - energy of solution Φ
    $\Phi_{curr} \leftarrow$ generate initial random solution
    $T \leftarrow T_{start}$
    while $T > 1$ **do**
        $\Phi_{new} \leftarrow$ perform random replacement on Φ_{curr}
        $p \leftarrow \exp(\frac{E(\Phi_{curr}) - E(\Phi_{new})}{T})$
        $r \leftarrow$ generate random number in range $[0; 1)$
        if $E(\Phi_{new}) > E(\Phi_{curr})$ or $p > r$ **then**
            $\Phi_{curr} \leftarrow \Phi_{new}$
        $T \leftarrow T \cdot (1 - c)$
    return Φ_{curr}

5. Evaluation Results

In this chapter we present algorithm evaluation results and conclusions.

5.1. Testing rig

All algorithms were run on Lenovo W530 notebook with the following hardware and software:

– CPU: Intel Core i7-3740QM (2.7 GHz)

– RAM size: 16 GB

– Operating system: Windows 7 Enterprise 64-bit

All tests were performed under "Maximum Performance" setting.

5.2. Test data

Numerous test cases were generated, varying in problem size, number of winners, generation model and satisfaction function.

5.2.1. Problem size

Problem size is defined by number of agents (n) and number of alternatives (m). We selected three problem sizes for testing:

– Small instance: $n = 30$, $m = 10$

– Medium instance: $n = 400$, $m = 50$

– Large instance: $n = 400$, $m = 300$

5.2.2. Number of winners

For each problem size, we selected two different number of winners (K) - the first one to take a small part of alternatives as winners (around 15-20%) and the second one to take about half of alternatives as winners:

– Small instance: $K = 2$ or $K = 5$

– Medium instance: $K = 10$ or $K = 25$

– Large instance: $K = 50$ or $K = 200$

5.2.3. Data generation model

Test data was generated using two different models.

Polya urn model (Polya) This model assumes we have an urn with m balls in m colors, each representing an alternative. We generate preferences by drawing random balls from the urn. First, we draw alternatives for first rank in each agent's preference order, then for second ranks, and so on, until the entire preference profile is drawn. After the ball is drawn, it is returned to the urn and another ball of the same color is added to the urn too, so a probability of drawing the same color in the future is increased ("strong becomes stronger"). The only constraint is that an alternative cannot be drawn for an agent for which it has been already drawn before. This model generates preference profile where some relatively small number of alternatives is preferred to all the others by most agents.

Impartial culture model (IC) In this model, for each agent we generate preference order independently. For each agent, every preference order (every permutation of alternatives) has equal probability of being drawn. This model generates preference profile with no clearly dominating alternatives.

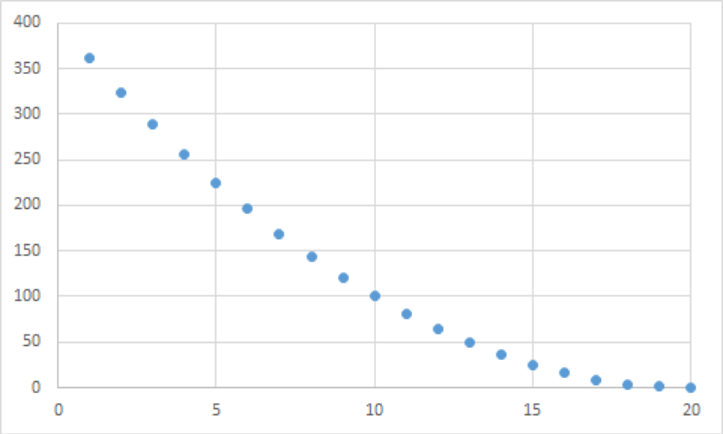
5.2.4. Satisfaction function

Two different satisfaction functions were used, each of them trying to model real preferences of the voters.

Square function This is simply a square function and it assumes that difference between voters in the first positions of the preference order should be higher than between the last positions (voter is more concerned with his favourite candidates, not the ones at the end of the list). The function is given as follows:

$$\alpha(i) = (m - i)^2 \tag{5.1}$$

Example of function for $m = 20$:



Strange function This function assumes that for the voter the difference between two candidates from the top is the same as the difference between two candidates from the bottom, while differences between candidates from the middle of the preference order are much less significant. The function is given as follows:

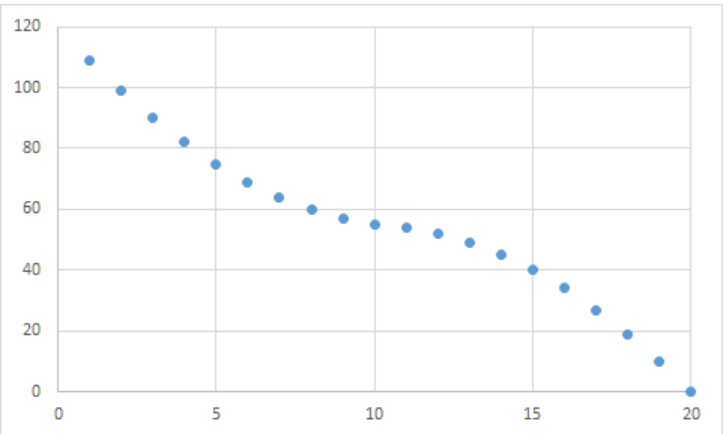
$$f(x) = \frac{x^2 + x}{2} \tag{5.2}$$

$$h = \frac{m}{2} \tag{5.3}$$

$$d = f(h) \tag{5.4}$$

$$\alpha(i) = \begin{cases} f(h - i + 1) + d - 1 : i < h \\ -f(i - h) + d : i \geq h \end{cases} \tag{5.5}$$

Example of function for $m = 20$:



5.2.5. Summary

Table below presents all generated test cases.

#	alternatives count	agents count	winners count	generation model	satisfaction function
1	10	30	2	Polya	Square
2					Strange
3				IC	Square
4					Strange
5			5	Polya	Square
6					Strange
7				IC	Square
8					Strange
9	50	400	10	Polya	Square
10					Strange
11				IC	Square
12					Strange
13			25	Polya	Square
14					Strange
15				IC	Square
16					Strange
17	300	400	50	Polya	Square
18					Strange
19				IC	Square
20					Strange
21			200	Polya	Square
22					Strange
23				IC	Square
24					Strange

5.3. Test Methodology

For each combination of problem size and generation model 100 files were generated independently. Algorithms were evaluated against all test files for each test case. We measured execution time and solution quality (total satisfaction) which was compared with brute-force (optimal) result for small instances and upper bound (total satisfaction if every agent has his top alternative assigned) for medium and large instances. Execution time and solution quality comparison with brute-force result or upper bound are presented as an average of 100 results (with standard deviation).

All test cases were used for both Chamberlin-Courant and Monroe problems.

5.4. Chamberlin-Courant Problem Evaluation

In this section we present evaluation results for Chamberlin-Courant problem.

5.4.1. Small instance

Results for small instance ($n = 30$, $m = 10$). Results are compared to optimal results.

Evaluated algorithms:

– Algorithm C ($d = 10$)

– Algorithm C ($d = 15$)

– Algorithm R ($k = 100$)

– Algorithm GM

– Algorithm P

– Genetic Algorithm ($I = 15, c = 5$)

– Simulated Annealing ($T_{start} = 100, c = 0.1$)

Results for 2 winners:

$n = 30$, $m = 10$, $K = 2$, Square function				
algorithm	Polya		IC	
	time [ms]	quality	time [ms]	quality
C (10)	0.66 ± 0.12	$100.000 \pm 0.000\%$	0.68 ± 0.14	$100.000 \pm 0.000\%$
C (15)	0.74 ± 0.16	$100.000 \pm 0.000\%$	0.74 ± 0.14	$100.000 \pm 0.000\%$
R (100)	0.75 ± 0.14	$99.710 \pm 1.162\%$	0.58 ± 0.05	$99.479 \pm 1.470\%$
GM	0.14 ± 0.03	$99.992 \pm 0.055\%$	0.10 ± 0.02	$99.690 \pm 0.856\%$
P	0.06 ± 0.01	$87.964 \pm 8.825\%$	0.05 ± 0.02	$90.096 \pm 6.482\%$
GA (15, 5)	0.68 ± 0.14	$99.633 \pm 1.261\%$	0.53 ± 0.06	$99.505 \pm 1.555\%$
SA (100, 0.1)	0.37 ± 0.07	$98.603 \pm 2.375\%$	0.28 ± 0.04	$98.693 \pm 2.549\%$

$n = 30$, $m = 10$, $K = 2$, Strange function				
algorithm	Polya		IC	
	time [ms]	quality	time [ms]	quality
C (10)	0.54 ± 0.07	$100.000 \pm 0.000\%$	0.56 ± 0.15	$100.000 \pm 0.000\%$
C (15)	0.56 ± 0.06	$100.000 \pm 0.000\%$	0.55 ± 0.06	$100.000 \pm 0.000\%$
R (100)	0.72 ± 0.15	$99.718 \pm 1.021\%$	0.59 ± 0.07	$99.947 \pm 0.240\%$
GM	0.16 ± 0.03	$99.994 \pm 0.044\%$	0.11 ± 0.03	$99.414 \pm 1.375\%$
P	0.06 ± 0.01	$91.785 \pm 6.058\%$	0.05 ± 0.01	$93.725 \pm 4.245\%$
GA (15, 5)	0.68 ± 0.15	$99.910 \pm 0.383\%$	0.52 ± 0.05	$99.670 \pm 0.780\%$
SA (100, 0.1)	0.32 ± 0.05	$98.820 \pm 2.256\%$	0.27 ± 0.04	$99.144 \pm 1.778\%$

Results for 5 winners:

$n = 30$, $m = 10$, $K = 5$, Square function				
algorithm	Polya		IC	
	time [ms]	quality	time [ms]	quality
C (10)	1.67 ± 0.30	$100.000 \pm 0.000\%$	1.59 ± 0.13	$99.993 \pm 0.073\%$
C (15)	2.29 ± 0.52	$100.000 \pm 0.000\%$	2.41 ± 0.48	$100.000 \pm 0.000\%$
R (100)	0.95 ± 0.09	$99.389 \pm 0.741\%$	0.94 ± 0.07	$99.218 \pm 0.842\%$
GM	0.16 ± 0.02	$99.980 \pm 0.108\%$	0.16 ± 0.01	$99.534 \pm 0.738\%$
P	0.06 ± 0.01	$91.789 \pm 4.927\%$	0.06 ± 0.01	$92.450 \pm 3.546\%$
GA (15, 5)	0.94 ± 0.04	$99.728 \pm 0.686\%$	0.97 ± 0.05	$99.349 \pm 1.014\%$
SA (100, 0.1)	0.53 ± 0.04	$98.185 \pm 1.550\%$	0.53 ± 0.03	$98.323 \pm 1.464\%$

$n = 30$, $m = 10$, $K = 5$, Strange function				
algorithm	Polya		IC	
	time [ms]	quality	time [ms]	quality
C (10)	1.54 ± 0.13	$100.000 \pm 0.000\%$	1.68 ± 0.45	$100.000 \pm 0.000\%$
C (15)	2.35 ± 0.64	$100.000 \pm 0.000\%$	2.27 ± 0.13	$100.000 \pm 0.000\%$
R (100)	0.93 ± 0.08	$99.331 \pm 0.768\%$	0.94 ± 0.08	$99.425 \pm 0.632\%$
GM	0.16 ± 0.02	$99.982 \pm 0.091\%$	0.16 ± 0.00	$99.672 \pm 0.615\%$
P	0.06 ± 0.01	$90.235 \pm 3.734\%$	0.06 ± 0.01	$94.353 \pm 2.656\%$
GA (15, 5)	0.96 ± 0.10	$99.710 \pm 0.607\%$	0.99 ± 0.06	$99.621 \pm 0.544\%$
SA (100, 0.1)	0.54 ± 0.04	$98.431 \pm 1.350\%$	0.53 ± 0.03	$98.741 \pm 0.959\%$

5.4.1.1. Conclusions

Conclusions from small instance results:

1. Results show no significant differences between Square and Strange satisfaction functions.

2. Algorithm C seems to be the best one in the terms of solution quality. It returned an optimal solution for all cases except one, in which increasing number of stored functions (d) from 10 to 15 allowed it to return an optimal solution anyway.

3. Algorithm GM is also returning very good results, although not optimal, while being much faster than Algorithm C. It remains to be seen how these algorithms behave in relation to each other for larger instances.

4. Algorithm R performs surprisingly well, but for every case either Algorithm C or Algorithm GM is faster and gives better results.

5. Algorithm P is the fastest one, but it also returns the worst results. It has very large standard deviation, which means that for some instances it returns much better result than average, but for some much worse.

6. Genetic Algorithm and Simulated Annealing do not perform well, as they are both significantly slower than Algorithm GM, while returning worse results. For larger instances, we tried different input parameters to see if it makes these algorithms perform better.

7. Results seem to be very similar for Polya and IC for such small instance. Only algorithm P performs better under IC, which is rather counter-intuitive. However, it may be caused by high standard deviation and therefore too small number of test files for this result to be representative.

5.4.2. Medium instance

Results for medium instance ($n = 400$, $m = 50$). Results are compared to upper bound.

Evaluated algorithms:

– Algorithm C ($d = 10$)

– Algorithm C ($d = 15$)

– Algorithm R ($k = 100$)

– Algorithm GM

– Algorithm P

– Genetic Algorithm ($I = 100, c = 20$)

– Genetic Algorithm ($I = 200, c = 25$)

– Simulated Annealing ($T_{start} = 100, c = 0.01$)

– Simulated Annealing ($T_{start} = 100, c = 0.005$)

Results for 10 winners:

$n = 400$, $m = 50$, $K = 10$, Square function				
algorithm	Polya		IC	
	time [ms]	quality	time [ms]	quality
C (10)	251.85 ± 11.73	$96.517 \pm 0.553\%$	267.20 ± 9.65	$89.757 \pm 0.294\%$
C (15)	379.52 ± 9.60	$96.517 \pm 0.553\%$	402.30 ± 9.43	$89.769 \pm 0.286\%$
R (100)	938.14 ± 11.43	$94.589 \pm 0.727\%$	941.29 ± 12.78	$88.540 \pm 0.224\%$
GM	25.45 ± 0.96	$96.508 \pm 0.591\%$	27.24 ± 2.11	$89.494 \pm 0.377\%$
P	2.62 ± 0.07	$89.795 \pm 2.308\%$	2.68 ± 0.22	$86.599 \pm 0.711\%$
GA (100, 20)	696.82 ± 15.81	$96.256 \pm 0.563\%$	761.59 ± 11.25	$89.412 \pm 0.315\%$
GA (200, 25)	1723.27 ± 23.67	$96.404 \pm 0.553\%$	1894.81 ± 19.53	$89.562 \pm 0.292\%$
SA (100, 0.01)	169.32 ± 7.39	$93.589 \pm 0.866\%$	169.55 ± 7.59	$88.132 \pm 0.302\%$
SA (100, 0.005)	335.91 ± 7.52	$94.013 \pm 0.911\%$	334.44 ± 6.37	$88.207 \pm 0.234\%$

$n = 400$, $m = 50$, $K = 10$, Strange function				
algorithm	Polya		IC	
	time [ms]	quality	time [ms]	quality
C (10)	251.14 ± 9.65	$96.733 \pm 0.510\%$	266.23 ± 9.37	$90.719 \pm 0.255\%$
C (15)	380.68 ± 10.18	$96.733 \pm 0.510\%$	404.76 ± 11.29	$90.731 \pm 0.247\%$
R (100)	941.99 ± 16.86	$95.084 \pm 0.624\%$	951.27 ± 15.43	$89.698 \pm 0.188\%$
GM	25.56 ± 1.46	$96.725 \pm 0.516\%$	27.16 ± 1.86	$90.529 \pm 0.323\%$
P	2.61 ± 0.07	$90.769 \pm 1.985\%$	2.68 ± 0.28	$88.062 \pm 0.600\%$
GA (100, 20)	696.42 ± 15.04	$96.518 \pm 0.526\%$	760.18 ± 12.71	$90.423 \pm 0.247\%$
GA (200, 25)	1738.66 ± 27.14	$96.631 \pm 0.515\%$	1922.78 ± 24.20	$90.554 \pm 0.261\%$
SA (100, 0.01)	167.29 ± 5.80	$93.988 \pm 0.739\%$	167.73 ± 7.18	$89.329 \pm 0.292\%$
SA (100, 0.005)	338.42 ± 10.02	$94.450 \pm 0.719\%$	342.32 ± 11.21	$89.450 \pm 0.230\%$

Results for 25 winners:

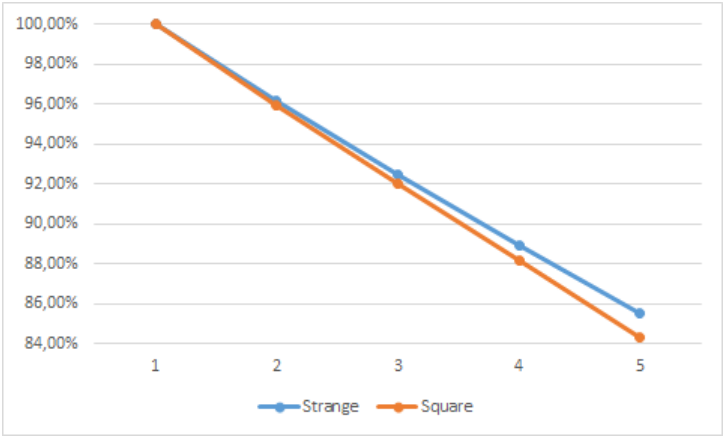
$n = 400$, $m = 50$, $K = 25$, Square function				
algorithm	Polya		IC	
	time [ms]	quality	time [ms]	quality
C (10)	541.20 ± 12.76	$99.374 \pm 0.015\%$	569.84 ± 12.51	$97.585 \pm 0.119\%$
C (15)	832.17 ± 16.83	$99.374 \pm 0.015\%$	869.71 ± 14.88	$97.597 \pm 0.118\%$
R (100)	2094.20 ± 15.70	$98.574 \pm 0.204\%$	2142.40 ± 22.10	$97.041 \pm 0.084\%$
GM	51.70 ± 3.53	$99.373 \pm 0.146\%$	54.80 ± 3.58	$97.518 \pm 0.126\%$
P	4.40 ± 0.14	$96.740 \pm 0.855\%$	4.42 ± 0.09	$96.235 \pm 0.255\%$
GA (100, 20)	1652.20 ± 28.08	$99.138 \pm 0.183\%$	1781.69 ± 21.33	$97.335 \pm 0.102\%$
GA (200, 25)	4096.02 ± 60.15	$99.188 \pm 0.179\%$	4483.16 ± 43.37	$97.383 \pm 0.100\%$
SA (100, 0.01)	399.24 ± 8.47	$98.170 \pm 0.302\%$	399.77 ± 8.48	$96.859 \pm 0.113\%$
SA (100, 0.005)	813.16 ± 15.30	$98.326 \pm 0.252\%$	813.26 ± 14.00	$96.898 \pm 0.098\%$

$n = 400$, $m = 50$, $K = 25$, Strange function				
algorithm	Polya		IC	
	time [ms]	quality	time [ms]	quality
C (10)	540.44 ± 12.59	$99.404 \pm 0.138\%$	567.23 ± 12.07	$97.726 \pm 0.108\%$
C (15)	831.26 ± 18.11	$99.404 \pm 0.138\%$	863.77 ± 16.02	$97.732 \pm 0.103\%$
R (100)	2136.59 ± 23.57	$98.667 \pm 0.186\%$	2091.77 ± 16.60	$97.214 \pm 0.085\%$
GM	51.68 ± 2.23	$99.403 \pm 0.138\%$	54.74 ± 3.07	$97.662 \pm 0.118\%$
P	4.39 ± 0.25	$96.939 \pm 0.790\%$	4.45 ± 0.54	$96.472 \pm 0.233\%$
GA (100, 20)	1649.14 ± 28.84	$99.180 \pm 0.179\%$	1769.20 ± 19.84	$97.491 \pm 0.096\%$
GA (200, 25)	4104.78 ± 61.44	$99.227 \pm 0.172\%$	4422.15 ± 27.53	$97.548 \pm 0.099\%$
SA (100, 0.01)	397.18 ± 8.62	$98.280 \pm 0.268\%$	396.13 ± 6.25	$97.034 \pm 0.111\%$
SA (100, 0.005)	799.15 ± 11.14	$98.397 \pm 0.233\%$	799.15 ± 9.75	$97.090 \pm 0.105\%$

5.4.2.1. Conclusions

Conclusions from medium instance results:

1. Solution quality compared to upper bound is generally a bit better for Strange satisfaction function. This is because for top alternatives Square function decreases faster compared to the first alternative (see chart below), so algorithms score is relatively worse when selecting second or third alternative in the preference order.



2. The difference between Polya and IC is clearly visible now (Polya could still be too random for smaller instance to see the diference). It is much easier to find a solution close to an upper bound for a Polya generation model, as there is a group of alternatives that are ranked high by most agents, while IC is totally random.

3. Algorithm C is still the best algorithm in terms of solution quality. For Polya, it finds solutions very close to an upper bound, especially with $K = 25$, which means that most agents have their favourite alternative assigned. For IC using $d = 15$ instead of $d = 10$ slightly improves the solution quality, so if we seek the best solution possible, setting higher d is reasonable.

4. Algorithm GM performs excellently, attaining only slightly lower solution quality than Algorithm C in most cases, while executing much faster.

5. Algorithm P is still very fast, and additionally standard deviation of its quality is acceptable for this problem size (there should be no instances for which the solution quality is much worse than average). Quality is much lower than for Algorithms C or GM, but if we need a decent solution that does not have to be optimal and we want to calculate it fast, this algorithm may be used.

6. Algorithm R does not seem to be useful for this problem size. It executes longer than Algorithm C and returns much worse results.

7. Genetic Algorithm generates solutions of very good quality (comparable to Algorithm GM), but executes much too long to be useful in both tested configurations. Simulated Annealing also seems to be useless, as Algorithm GM executes much faster and gives better results.

5.4.3. Large instance

Results for large instance ($n = 400$, $m = 300$). Results are compared to upper bound.

Evaluated algorithms:

– Algorithm C ($d = 10$)

– Algorithm C ($d = 15$)

– Algorithm R ($k = 100$)

– Algorithm GM

– Algorithm P

– Genetic Algorithm ($I = 100, c = 20$)

– Genetic Algorithm ($I = 200, c = 25$)

– Simulated Annealing ($T_{start} = 100, c = 0.01$)

– Simulated Annealing ($T_{start} = 100, c = 0.005$)

Results for 50 winners:

$n = 400$, $m = 300$, $K = 50$, Square function				
algorithm	Polya		IC	
	time [s]	quality	time [s]	quality
C (10)	15.66 ± 0.11	$99.418 \pm 0.051\%$	16.33 ± 0.10	$98.466 \pm 0.056\%$
C (15)	24.12 ± 0.12	$99.420 \pm 0.051\%$	25.04 ± 0.14	$98.472 \pm 0.061\%$
R (100)	0.337 ± 0.011	$97.787 \pm 0.140\%$	0.337 ± 0.011	$97.197 \pm 0.075\%$
GM	1.46 ± 0.02	$99.412 \pm 0.052\%$	1.53 ± 0.02	$98.400 \pm 0.071\%$
P	0.0585 ± 0.0045	$97.029 \pm 0.416\%$	0.0588 ± 0.0049	$96.804 \pm 0.154\%$
GA (100, 20)	6.96 ± 0.11	$99.039 \pm 0.084\%$	8.41 ± 0.05	$97.822 \pm 0.073\%$
GA (200, 25)	16.81 ± 0.29	$99.125 \pm 0.069\%$	$20,88 \pm 0.14$	$97.927 \pm 0.062\%$
SA (100, 0.01)	1.91 ± 0.03	$97.662 \pm 0.213\%$	1.92 ± 0.02	$97.197 \pm 0.087\%$
SA (100, 0.005)	3.81 ± 0.05	$97.794 \pm 0.172\%$	3.85 ± 0.03	$97.221 \pm 0.082\%$

$n = 400$, $m = 300$, $K = 50$, Strange function				
algorithm	Polya		IC	
	time [s]	quality	time [s]	quality
C (10)	15.68 ± 0.16	$99.425 \pm 0.050\%$	16.32 ± 0.10	$98.495 \pm 0.059\%$
C (15)	24.07 ± 0.14	$99.426 \pm 0.050\%$	25.05 ± 0.13	$98.502 \pm 0.060\%$
R (100)	0.329 ± 0.002	$97.838 \pm 0.129\%$	0.329 ± 0.002	$97.264 \pm 0.073\%$
GM	1.42 ± 0.01	$99.419 \pm 0.052\%$	1.49 ± 0.01	$98.438 \pm 0.075\%$
P	0.0575 ± 0.0006	$97.098 \pm 0.400\%$	0.0583 ± 0.005	$96.881 \pm 0.148\%$
GA (100, 20)	6.79 ± 0.10	$99.054 \pm 0.081\%$	8.25 ± 0.03	$97.872 \pm 0.069\%$
GA (200, 25)	16.61 ± 0.23	$99.131 \pm 0.073\%$	20.50 ± 0.42	$97.952 \pm 0.059\%$
SA (100, 0.01)	1.89 ± 0.03	$97.741 \pm 0.197\%$	1.89 ± 0.01	$97.239 \pm 0.092\%$
SA (100, 0.005)	3.77 ± 0.04	$97.840 \pm 0.175\%$	3.78 ± 0.01	$97.286 \pm 0.072\%$

Results for 200 winners:

$n = 400$, $m = 300$, $K = 200$, Square function				
algorithm	Polya		IC	
	time [s]	quality	time [s]	quality
C (10)	50.19 ± 0.38	$100.000 \pm 0.000\%$	50.54 ± 0.27	$99.9536 \pm 0.0099\%$
C (15)	79.44 ± 0.60	$100.000 \pm 0.000\%$	81.67 ± 0.62	$99.9537 \pm 0.0097\%$
R (100)	1.29 ± 0.01	$99.776 \pm 0.016\%$	1.29 ± 0.00	$99.737 \pm 0.011\%$
GM	4.04 ± 0.02	$100.000 \pm 0.000\%$	4.00 ± 0.01	$99.952 \pm 0.010\%$
P	0.134 ± 0.002	$99.669 \pm 0.044\%$	0.134 ± 0.002	$99.672 \pm 0.031\%$
GA (100, 20)	27.20 ± 0.97	$100.000 \pm 0.000\%$	30.67 ± 0.08	$99.950 \pm 0.015\%$
GA (200, 25)	67.72 ± 2.41	$100.000 \pm 0.000\%$	76.45 ± 0.21	$99.953 \pm 0.014\%$
SA (100, 0.01)	7.08 ± 0.04	$99.757 \pm 0.021\%$	7.05 ± 0.02	$99.724 \pm 0.017\%$
SA (100, 0.005)	14.16 ± 0.06	$99.773 \pm 0.024\%$	14.10 ± 0.03	$99.736 \pm 0.014\%$

$n = 400$, $m = 300$, $K = 200$, Strange function				
algorithm	Polya		IC	
	time [s]	quality	time [s]	quality
C (10)	49.33 ± 0.35	$100.000 \pm 0.000\%$	50.04 ± 0.20	$99.9538 \pm 0.0091\%$
C (15)	78.57 ± 0.50	$100.000 \pm 0.000\%$	80.79 ± 0.33	$99.9542 \pm 0.0097\%$
R (100)	1.30 ± 0.00	$99.776 \pm 0.014\%$	1.30 ± 0.00	$99.741 \pm 0.011\%$
GM	4.05 ± 0.01	$100.000 \pm 0.000\%$	4.00 ± 0.01	$99.953 \pm 0.010\%$
P	0.134 ± 0.001	$99.672 \pm 0.043\%$	0.134 ± 0.001	$99.676 \pm 0.031\%$
GA (100, 20)	27.29 ± 0.97	$100.000 \pm 0.000\%$	30.77 ± 0.05	$99.950 \pm 0.015\%$
GA (200, 25)	67.79 ± 2.41	$100.000 \pm 0.000\%$	77.74 ± 0.41	$99.954 \pm 0.014\%$
SA (100, 0.01)	7.11 ± 0.04	$99.755 \pm 0.026\%$	7.24 ± 0.05	$99.727 \pm 0.017\%$
SA (100, 0.005)	14.22 ± 0.07	$99.773 \pm 0.022\%$	14.32 ± 0.11	$99.736 \pm 0.016\%$

5.4.3.1. Conclusions

Conclusions from large instance results:

1. These test cases turned out to be very easy for tested algorithms - all solutions are of very high quality compared to upper bound. The reason of this is very high number of winners, which allows many agents to have their favourite alternative assigned, even for data generated with IC model. For Polya and 200 winners, many algorithms always return an optimal solution which has the same total satisfaction as upper bound - allowing 200 out of 300 candidates to win and using Polya model allows for every agent to be maximally satisfied in virtually every case.

2. Relation between Algorithm C, Algorithm GM and Algorithm P remains the same as for the middle instance.

3. For such large data sets, Algorithm R may become an option, as it runs relatively fast (not as fast as Algorithm P though) and returns solution of quality between Algorithm GM and Algorithm R. It is possible that this algorithm gets relatively better for even larger data sets.

4. Genetic Algorithm and Simulated Annealing are not a viable option for small number of winners ($K = 50$), as they both return worse results than Algorithm GM and they run slower. However, Genetic Algorithm may be useful for large data sets where there is a lot of winners. For $K = 200$, it becomes similar to Algorithm GM in means of both execution time and solution quality.

5.4.4. Conclusions summary

These are final conclusions regarding evaluation of algorithms for Chamberlin-Courant Problem based on all executed test cases.

1. There is no significant difference between results for Square and Strange satisfaction functions. It seems that the most important part of the satisfaction function is near the top of the preference order, because middle and bottom alternatives are rarely selected and differences between them are not very important, and both Square and Strange functions are similar for the top alternatives. Results could be much different for some kind of exponential function.

2. Algorithm C and Algorithm GM are viable for every tested case. One should use Algorithm C to get solution with the best quality possible (potentially increasing input parameter d for a chance to find an even better solution). Algorithm GM always returns slightly weaker result than Algorithm C, but it runs much faster, so it may be used as a compromise between execution time and solution quality.

3. Algorithm P returns substantially weaker solutions than Algorithm GM, but it runs extremely fast. It may be used if solution does not have to be good but not necessarily close to optimal (e.g., in recommendation systems), but there is a need to obtain it as fast as possible. This algorithm should not be used for small data sets, because the solution may be of very low quality, but this should not be a problem, because for small instances Algorithms GM runs very fast too.

4. For large cases, Algorithm R can provide good results with moderate execution time. However, due to high level of randomization, this algorithm should only be used to supplement another one, so it can possibly find a better solution.

5. It is possible that Genetic Algorithm may be a good option for large cases with high winner count, but it requires further research.

6. Summary

References

[1] B. Monroe. Fully proportional representation. *American Political Science Review*, 89(4):925–940, 1995.

[2] B. Chamberlin and P. Courant. Representative deliberations and representative decisions: Proportional representation and the borda rule. *American Political Science Review*, 77(3):718–733, 1983.

[3] T. Lu and C. Boutilier. Budgeted social choice: From consensus to personalized decision making. *IJCAI*, 11:280–286, 2011.

[4] N. Betzler, A. Slinko, and J. Uhlmann. On the computation of fully proportional representation. *Journal on Artificial Intelligence Research*, 47:475–519, 2013.

[5] A.D. Procaccia, J.S. Rosenschein, and A. Zohar. On the complexity of achieving proportional representation. *Social Choice and Welfare*, 30(3):353–362, April 2008.

[6] P. Skowron, P. Faliszewski, and A. Slinko. Achieving fully proportional representation, approximability results. *Artificial Intelligence*, 222:67–103, May 2015.

[7] E. Elkind, P. Faliszewski, P. Skowron, and A. Slinko. Properties of multiwinner voting rules. *Proceedings of the 13th International Conference on Autonomous Agents and Multiagent Systems*, pages 53–60, May 2014.

[8] L. Yu, H. Chan, and E. Elkind. Multiwinner elections under preferences that are single-peaked on a tree. *Proceedings of IJCAI-13*, pages 425–431, 2013.

[9] P. Skowron, L. Yu, P. Faliszewski, and E. Elkind. The complexity of fully proportional representation for single-crossing electorates. *Proceedings of the 6th International Symposium on Algorithmic Game Theory*, pages 1–12, October 2013.

[10] X.S. Yang. *Nature-inspired Metaheuristic Algorithms*. Luniver Press, 2010.

[11] S. Łukasik and S. Żak. Firefly algorithm for continuous constrained optimization. *Lecture Notes in Artificial Intelligence*, 5796:97–106, 2009.

[12] S. Kirkpatrick, C.D. Gelatt, and M. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

List of Tables