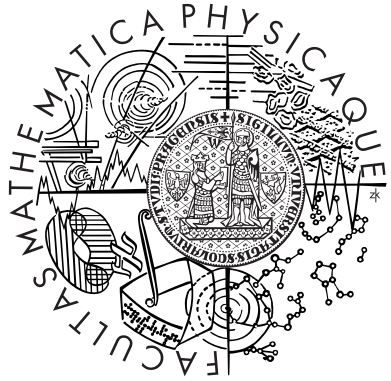
Charles University in Prague
Faculty of Mathematics and Physics

BACHELOR THESIS



Kateřina Böhmová

Heuristiky pro problém výměny ledviny

Heuristics for the Kidney Exchange Problem

Department of Applied Mathematics

Supervisor: Mgr. Eva Jelínková
Study programme: Computer Science,
General Computer Science

Prague, 2009

I would like to thank Eva Jelínková for her supervising and for a lot of time, advice and ideas she gave me. I really enjoyed the work with you. I would also like to thank Petr Škovroň for proofreading, correcting mistakes, and a lot of encouragement.

I declare that I have written all of the thesis on my own and that I cited all used sources of information. I agree with public availability and lending of the thesis.

Prague, 24 May 2009 Kateřina Böhmová

Table of contents

Název práce: Heuristiky pro problém výměny ledviny
Autor: Kateřina Böhmová
Katedra: Katedra aplikované matematiky
Vedoucí bakalářské práce: Mgr. Eva Jelínková
E-mail vedoucího: eva@kam.mff.cuni.cz
Abstrakt: V této práci se zabýváme problémem výměny ledviny. Jedná se o kombinatorický model problému rozdělení žijících dárců ledvin pacientům. Přesněji řečeno, máme množinu nekompatibilních dvojic pacient-dárce a snažíme se permutací dárců získat páry vhodné k transplantaci. Požadujeme, aby výsledné řešení bylo stabilní, což ve zkratce znamená, že nepřipouštíme výskyt skupiny dvojic, jejíž všichni členové by si polepšili vytvořením jiné permutace jen mezi sebou.

V práci vysvětlíme známé metody pro hledání řešení (algoritmus Top Trading Cycles a heuristiky) a pro testování stability řešení. Popíšeme předchozí známé výsledky pojednávající o těžkosti problému.

Navrhujeme hledat dobré stabilní řešení tak, že opakovaně aplikujeme heuristiky na počáteční řešení vygenerované pomocí TTC. Používáme několik známých heuristik spolu se dvěma novými. Předvedeme výsledky provedených testů, které ukazují vylepšení dosažené heuristikami oproti počátečnímu řešení.

Také předvedeme nový algoritmus pro testování stability řešení, který běží výrazně rychleji než původní algoritmus.

Klíčová slova: Výměna ledviny, Top Trading Cycles, testování stability

Title: Heuristics for the Kidney Exchange Problem
Author: Kateřina Böhmová
Department: Department of Applied Mathematics
Supervisor: Mgr. Eva Jelínková
Supervisor's e-mail address: eva@kam.mff.cuni.cz
Abstract: In this thesis we are dealing with the Kidney exchange game, which is a combinatorial model of the problem of distribution of living donors of kidneys to patients. More specifically, having a set of incompatible recipient-donor pairs we want to create a permutation of the donors to obtain pairs compatible for a transplantation. We require that the solution is stable, which essentially means that there is not a group of pairs such that it would be better for all of them to create another permutation just among themselves.

We give an overview of known methods for finding solutions (the Top Trading Cycles algorithm and heuristics) and for testing the stability of a solution. We describe previously known results concerning the hardness of the problem.

We propose to seek for a good stable solution by starting with the result of the TTC algorithm and then applying heuristics repeatedly. We use several known heuristics together with two new ones. We present results of a series of tests to show the improvement achieved by the heuristics.

We also present a new algorithm for testing the stability of a solution. This algorithm runs significantly faster than the previously known one.

Keywords: Kidney exchange, Top Trading Cycles, stability testing

1. Introduction

A patient that suffers a kidney failure needs a transplantation. There is a couple of possibilities how a suitable life-saving organ can be obtained. He may wait until a kidney of a deceased donor becomes available. In this case the patient will be probably added to the end of a long list of candidates. Alternatively, he may have a living donor (e.g., a family member) willing to give him an organ of his own.

Unfortunately, it may happen that the patient and his donor are not compatible for some immunological reason (e.g., a blood type). Consequently the donor cannot donate his kidney to his recipient because it would be rejected and the important organ would be in vain.

As this problem is not so rare, there are a lot of incompatible pairs of recipient-donor. Therefore a lot of countries (e.g., USA, the Netherlands, Romania, the United Kingdom) try to help these people by a systematic kidney exchange program.

The main idea of these programs is to take several incompatible pairs and permute the donors in such a way that as many recipients as possible will obtain a feasible kidney and their donors will donate their kidney. There are algorithms that are used in practice that reflect that the program of kidney exchange is beneficial (see [KdK$^+$05] and [AB$^+$07]).

However, various requirements can be placed on the desired permutation and it is not simple to create a solution that has the parameters as good as possible. The problem of kidney exchange is currently studied. The research concerns a description of diverse aspects of the problem and a searching for efficient algorithms to obtain good permutations.

Several mathematical models of the problem exist. The model that we deal with was studied for example in the paper [CL06]. It assumes that every recipient has an linearly ordered list of other participants' donors that are acceptable for him. The resulting permutation consists of several cycles and we require that the solution is stable. The stability of a permutation basically means that there cannot be a group of patients such that it would be better for them to create a permutation just among themselves. We believe that this requirement of stability is important, because a discovery of the existence of such a group would decrease the trust in the system and could be destructive to the whole program of cooperation.

1.1. Structure of the work and a summary of results

In this thesis, we study the heuristics for improving a permutation while preserving stability.

In Chapter 2 we describe the studied model in more detail and more formally. We give the necessary definitions. Then we give an overview of an important algorithm called Top Trading Cycles, and its shortcomings. Afterwards we give several previously known results showing that most aspects of the problem are NP-complete.

Chapter 3 gives more information about the stability testing algorithms. We describe a previously known algorithm. Afterwards we introduce two new algorithms that are based on the ideas taken from the previous one. First we modify the original algorithm to obtain one that has essentially the same time complexity $O(n^3)$ but that significantly differs in real running time in our implementation. Finally we describe another modification called the Stable Local algorithm. It needs some additional assumptions, and by this loss of generality we gain a time complexity $O(n^2\sqrt{n})$.

Chapter 4 concerns the heuristics that try to improve the solution generated by the TTC algorithm. We describe two heuristics previously known and we add two more. One of our heuristics is similar to the first two, while the other is based on a different idea. We prove that all these heuristics satisfy the necessary conditions to be able to use our Stable Local to test the stability of a permutation generated by them.

Chapter 5 contains the results of tests we have performed. We compare the real runtime of the stability testing algorithms and we show how successful the heuristics are.

The appendices A and B contain the programmer and user documentation of the program we implemented to find a solution using heuristics.

The enclosed CD contains an electronic version of the thesis and documentations, the source code of the program, and inputs for the simulations used in Chapter 5.

2. The Kidney exchange game

In this chapter we describe in more detail the combinatorial model of kidney exchange that we use. We examine the same model as Cechlárová and Lacko [CL06] and we use also the definitions from that paper (we have just slightly reformulated them).

The model assumes that all donors can be evaluated for each patient. Those that are acceptable for him are put into a sorted list (the best donor is the first in the list). We refer to this list as a preference list. We assume that the preference lists do not contain ties, that is, for each two donors in a list we can decide which one is better.

The model is called a Kidney exchange game. It is represented by a set of players (each player corresponds to a pair recipient-donor) and their preference lists. We sometimes call the players vertices, because we can view the Kidney exchange game as a directed graph, where players are vertices and each donor j in the preference list of recipient i corresponds to the directed edge (i, j).

Definition 2.1. *A* kidney exchange game *(KEG for short) is a pair (V, Φ), where V is a finite set of players and Φ is a mapping that to each player $i \in V$ assigns a subset Φ_i of V completely ordered by a relation $\prec_i$. By $a \prec_i b$ we mean that player i prefers player a to player b. We assume that $i \notin \Phi_i$. If $j \in \Phi_i$ we say that the player j is an* acceptable *donor for the player i.*

Definition 2.2. *A* solution *of a KEG (V, Φ) is a permutation π of V such that for each $i \in V$, we have $\pi(i) \in \Phi_i$ or $\pi(i) = i$. In the case $\pi(i) = i$ we say that the player i is* uncovered, *otherwise i is* covered.

The resulting permutation assigns to each recipient either a donor from his preference list or a donor of his own. Thus the permutation consists of several cycles. Whenever a player is in a cycle of length 1 it means that the permutation does not assign him any better possibility that the donor of his own and we say that he is uncovered.

Our aim is finding permutations that cover as many players as possible and contain as short cycles as possible. The motivation of the first effort is clear because if the player is covered he has a feasible kidney assigned. The short cycles are better because in fact it is necessary to operate all people that participate in a cycle simultaneously. Otherwise a donor whose recipient has already obtained a kidney could change his mind and say that he does not want to donate his kidney anymore.

Definition 2.3. *Let π be a solution of a KEG (V, Φ). For each i we define $C^\pi(i) = (i, \pi(i), \pi^2(i), \ldots, \pi^{s-1}(i))$ to be the cycle containing player i, where π^k is repeated application of π and s is the smallest $s \in \mathbb{N}$ such that $\pi^s(i) = i$. By $l^\pi(i)$ we denote the length of the cycle $C^\pi(i)$. As an exception, if $\pi(i) = i$ we let $l^\pi(i) = \infty$. We also analogically define $C^\pi(e)$ and $l^\pi(e)$ where e is and edge $(i, \pi(i))$ for an arbitrary covered player i.*

A player can evaluate a permutation according to several measures. The player prefers an assignment that gives him a better donor. The second, less

important criterion is the length of the cycle that he participates in. If two assignments give him the same donor, he prefers the one putting him into a shorter cycle. Therefore every player can prefer one permutation to another.

Definition 2.4. *Let $i \in V$ be an arbitrary player. Let π and σ be permutations that assign to i either himself or an acceptable donor. We say that a player i* prefers *the permutation π to the permutation σ if at least one of the following conditions holds:*
(i) $\pi(i) \prec_i \sigma(i)$ *or*
(ii) $\pi(i) = \sigma(i)$ *and* $l^\pi(i) < l^\sigma(i)$.
If the player i prefers *π to σ we write $\pi \sqsubset_i \sigma$.*

The model we use also requires that the obtained solution is stable. The solution is not stable if there is a group of players that can form a permutation just among themselves that gives to each of them a better donor or at least a shorter cycle. If such a group discovered that its members have better options than those given by the program, it would impair the trust and it could destroy the whole program of cooperation.

Definition 2.5. *A subset $A \subseteq V$ forms a* blocking cycle *of a solution π if there exists a permutation ρ of A such that both of the following conditions hold:*
(i) the group A of players forms a cycle in the permutation ρ;
(ii) $\rho \sqsubset_i \pi$ for each $i \in A$.

Definition 2.6. *A solution π is* stable *if there is no blocking cycle of π.*

There is an algorithm that can be used to find a stable solution in a linear time with respect to the sum of lengths of the preference lists. It is called TTC and we give a basic information on it in the following section.

2.1. The Top Trading Cycles algorithm

The Top Trading Cycles algorithm (TTC for short) is a very effective algorithm to find a stable solution to a KEG (V, Φ). Although the solution generated by TTC has some disadvantages (as we will see later), we use it a lot, because with its time complexity of $(O(m + n))$ it is the fastest algorithm we have to generate any stable solution, where $n = |V|$ and $m = \sum_{i=1}^{n} |\Phi_i|$.

The TTC algorithm was originally proposed by Gale and published by Shapley and Scarf [SS74]. We are using the algorithm as it is stated in [CL06].

Algorithm.
Input: A KEG (V, Φ).
Output: A permutation π that is a stable solution to the KEG.

- Set $N := V$, select an arbitrary player $v \in N$ and set $current := v$.
- `REPEAT`
  - `WHILE` *current* has an acceptable donor in N and $\pi(current)$ is not set yet `DO`
    - Set $\pi(current)$ to be its most preferred donor $j \in N$ and set *current* to be j.

- ◇ `IF` $\pi(current)$ is already set `THEN`
  - ∗ a cycle $C^{\pi}(current)$ was obtained. $N := N - C^{\pi}(current)$.
- ◇ `ELSE`
  - ∗ Set $\pi(current) := current$, remove $current$ from the set N.
- ◇ `IF` exists a vertex $v \in N$ such that $\pi(v) = current$ `THEN`
  - ∗ Set $current$ to be v and set $\pi(v)$ to undefined.
- ◇ `ELSE`
  - ∗ Set $current$ to be an arbitrary vertex from N.

- `UNTIL` the set N is empty.

The TTC algorithm is very quick and as Cechárová and Medina [CRM00] proved, it generates a stable solution. However, TTC was devised for another problem. Thus the solution generated by it may be for our purposes far from optimal.

Examples of cases when the TTC algorithm gives a suboptimal solution are given in [CL06]. The first example shows a case when TTC generates one long cycle although the solution where every player is in a cycle of length 2 is also stable. Consider a KEG (V, Φ) such that:

- $V := \{a_1, a_2, \ldots, a_{2n}\}$
- $\Phi(a_{2i-1}) := \{a_{2i}\}$ and $\Phi(a_{2i}) := \{a_{2i+1}, a_{2i-1}\}$ for each $i \in \{1, 2, \ldots, n\}$, where $a_{2n+1} = a_1$.

The TTC generates a solution containing just a cycle $(a_1, a_2, \ldots, a_{2n})$. However the solution consisting of cycles (a_{2i-1}, a_{2i}) for each $i \in \{1, 2, \ldots, n\}$ is also stable and contains much shorter cycles.

Another example shows that sometimes TTC generates a solution that covers less players than possible. Consider a KEG (V, Φ) such that:

- $V := \{a_1, a_2, \ldots, a_n, b_1, b_2, \ldots, b_n, c_1, c_2, \ldots, c_n\}$
- $\Phi(a_i) := \{b_i\}$, $\Phi(b_i) := \{a_{i+1}, c_i\}$ and $\Phi(c_i) := \{a_i\}$ for each $i \in \{1, 2, \ldots, n\}$

The TTC generates just a cycle $(a_1, b_1, a_2, b_2 \ldots, a_n, b_n)$ and leaves all the players c_i uncovered. However the solution containing n cycles (a_i, b_i, c_i) for each $i \in \{1, 2, \ldots, n\}$ and covering all the players with short cycles is also stable.

As our motivation of the problem of kidney exchange is to find a donor to as many patients as possible and to allow for having them operated in the same time, we try to find a solution that covers as many players as possible and contains as short cycles as possible. Hence, there may be solutions that we like more than the one generated by the TTC algorithm.

2.2. Hard cases of KEG

We would like to find the "best" solution to a given KEG, but it is not easy at all. The first problem is that it is not clear what shall this "best" mean. We prefer solutions that cover more players and also we like more those that contain shorter cycles. However as we can see in the following example, these two simple requirements may be already in a contradiction. We define the following KEG (V, Φ).

- $V := \{a, b, c\}$

- $\Phi_a := \{b\}$, $\Phi_b := \{c, a\}$ and $\Phi_c := \{a\}$.

Two stable solutions for this input are possible and we cannot say which one is "better":

- The solution consisting of the cycle (a, b, c) that covers all vertices but its length is 3.
- The solution containing the cycle (a, b) that has the length 2, but the vertex c remains uncovered.

Another problem is that even if we decide which property is more important to us, we cannot find the optimal solution with respect to this parameter in a polynomial time. There is a couple of results that document this situation. Cechlárová and Lacko [CL06] formulated the following decision problems and proved their hardness.

Theorem 2.7. *Let π be a solution of KEG (V, Φ) generated by TTC. The decision problem whether there exists a stable solution σ such that $l^\sigma(i) < l^\pi(i)$ for each $i \in V$ is NP-complete.*

Theorem 2.8. *Let (V, Φ) be a KEG. The decision problem whether there exists a stable solution σ that covers all players in V is NP-complete.*

Theorem 2.9. *Let (V, Φ) be a KEG. The decision problem whether there exists a stable solution σ such that $l^\sigma(i) \leq 3$ for each $i \in V$ is NP-complete.*

Irving [Irv07] proved the NP-completeness of the Cycle stable roommates problem. The statement can be slightly reformulated for the kidney exchange problem, resulting in the following theorem.

Theorem 2.10. *Let (V, Φ) be a KEG such that $|V|$ is even and Φ_i contains all players $v \in V - i$ for each $i \in V$. The decision problem whether there exists a stable solution σ that $l^\sigma(i) = 2$ for each $i \in V$ is NP-complete, even if blocking cycles are restricted to be of length 3.*

When we know that the problem of finding the best solution is NP-hard, we may try to find a polynomial time approximation algorithm. However, the following result proved by Biró and Cechlárová [BC07] shows that even this is not possible (at least for the case when we try to minimise the number of uncovered donors).

Theorem 2.11. *Let (V, Φ) be a KEG. The problem of finding a stable solution that covers as many players in V as possible is not approximable within $e^{1-\varepsilon}$ for any $\varepsilon > 0$ unless $P = NP$.*

These results left a very little hope for an efficient algorithm for finding an optimal solution to a KEG. However, we have also seen that the output of the TTC algorithm may be improved. In the following chapters we are describing our effort to find a better solution in bearable time than the one given by TTC.

3. Stability testing algorithms

Once we have a solution to the Kidney exchange game, we are interested whether it is stable. For this we need an efficient algorithm for testing the stability.

A polynomial-time algorithm for testing the stability of the given solution π was introduced by Cechlárová and Hajduková [CH99]. As the aim of the authors of the paper was not the optimality of the algorithm, it can be improved. Our aim is to modify the algorithm and obtain a better complexity.

The paper [CH99] brought a characterisation which reduces the original problem of testing stability to the problem of finding short cycles in a set of auxiliary graphs. All algorithms in this chapter are based on this idea. The auxiliary graphs contain edges that correspond to the more preferred donors of every recipient than those used in the solution and edges of long cycles of the solution. These graphs are introduced formally by the following definition.

Definition 3.1. *Let π be a solution of KEG (V, Φ). For each $k \in \{1, \ldots, |V|\}$ we define the oriented graph $G_k(\pi) = (V, E_k)$ such that the edge (i, j) is in E_k if one of the following conditions holds:*

(i) $j \prec_i \pi(i)$ *or*

(ii) $\pi(i) = j$ *and* $k \leq l^{\pi}(i)$.

We also define the oriented graph $G_\infty(\pi) = (V, E_\infty)$. The edge (i, j) is in E_∞ if $j \prec_i \pi(i)$.

The algorithms test every auxiliary graph G_k whether it contains shorter cycles. Formally we have the following Theorem stated equivalently in [CH99] as Theorem 3.

Theorem 3.2. *Let π be a solution of KEG (V, Φ). Let $\mathcal{D}$ be the set of graphs $G_1(\pi), G_2(\pi), \ldots G_n(\pi)$ for π. Then the solution π is stable if and only if both of the following conditions hold:*

(i) G_∞ does not contain a directed cycle;

(ii) For every k, the graph G_k does not contain a directed cycle of length less than k.

In the following sections we describe the algorithms we use for testing the stability, all based on the idea of identifying short cycles in the graphs G_k. First, in Section 3.1 we give a short overview of the method used in the paper [CH99] and the way we have implemented it.

In the remaining sections we introduce another approach we came up with. In Section 3.2 we explain the idea of looking for short cycles in graphs using the Depth-First-Search and Breadth-First-Search algorithms (for short DFS and BFS respectively). In Section 3.3 we use the knowledge of the way how the heuristics do changes to the solution and with this and the previously introduced idea of using DFS and BFS we describe the Stable Local algorithm. This algorithm is intended to check the stability after calling one of the heuristics, but by this loss of generality we gain a better time complexity. For the reader's convenience and for completeness we add also the DFS and BFS algorithms that belong to

the computer science folklore. The form in which we use them is described in Sections 3.4 and 3.5.

3.1. Using the Floyd-Warshall algorithm

The method to identify cycles in every graph G_k in the paper [CH99] uses the Floyd-Warshall algorithm (FW for short) with $O(n^3)$ time complexity. The FW algorithm computes a matrix of graph distances $A(G_k)$ for each k. The i-th element in the diagonal of the matrix is equal to the length of the shortest cycle containing i. Since the construction of a graph G_k can be done in $O(n^2)$ steps, the stability testing algorithm introduced in the paper is polynomial.

In our implementation we are trying to improve solution to a KEG by calling a lot of heuristics (more about this in Chapter 4) and after every improvement done by a heuristic we want to check the stability. Thus we perform the stability tests very often and the time complexity of stability testing becomes a bottleneck. Therefore we focus on improving the asymptotic time complexity and the real running time of the stability testing algorithm.

We implemented the stability test using the FW algorithm. To save some time the graph G_k is constructed from G_{k+1} by adding edges that are in the solution in cycles of length k. By this we achieve the time complexity $O(n^4)$. The algorithm may be improved in the following way. Do not compute the matrix $A(G_k)$ for each k from the beginning and only recalculate the data that were affected by adding the edges from cycles of length k. By this we achieve the time complexity $\Theta(n^3)$. However, we found that for an input $n = 1000$ the algorithm takes already too long to be of practical use. Therefore our aim was to find a faster stability testing algorithm.

3.2. Using the Stable DFS-BFS algorithm

Since we think that the stability tests using FW algorithm can not be improved much more, we tried to find a different way to determine the length of the shortest cycles in the graphs G_k.

The idea of the following algorithm is to use the DFS algorithm to test the acyclity of the graph G_∞. The DFS algorithm works in time $(O(n + m))$ and testing the acyclity of given graph is its natural application.

Testing the rest of the graphs G_k cannot be done by a straightforward application of DFS. However, related algorithm the BFS algorithm can tell us the length of the shortest cycle that contains a given vertex or a given edge. It is not so obvious how to use this, but we only change the way we think of the graphs G_k. We know that that among the graphs G_k the two most different are G_∞ and G_1 and even these differ just by at most n edges that G_1 has in addition.

The main idea of the Stable DFS-BFS algorithm is to test the acyclity of G_∞ first and then successively add edges from the cycles (from the longest to the shortest) of the given solution σ. After every added edge e, we test whether e does not participate in a cycle shorter than $l^\sigma(e)$ because such a cycle would be blocking and according to Theorem 3.2 the tested solution wouldn't be stable.

Algorithm.
Input: A permutation σ that is a solution of KEG (V, Φ) that we want to test for stability.
Output: A logical value determining whether the solution σ is stable or not.

- Construct $G_\infty(\sigma) := (V, E_\infty)$, where an edge (i, j) is in E_∞ if $j \prec_i \sigma(i)$.
- Test the acyclity of G_∞ by the DFS algorithm (see Section 3.4). `IF` G_∞ contains a cycle `THEN`
  - ◇ `RETURN FALSE`.
- Set an auxiliary graph G to be G_∞.
- `FOR` each length k `FROM` $|V|$ `DOWNTO` 1 `DO`
  - ◇ `FOR` each player $v \in V$ such that $l^\sigma(v) = k$ `DO`
    - ∗ Add the edge from v to $\sigma(v)$ to G.
    - ∗ Using the BFS algorithm (see Section 3.5) find the length l of the shortest path from $\sigma(v)$ to v in the graph G.
    - ∗ `IF` $l < k - 1$ `THEN`
      - ○ `RETURN FALSE`.
- `RETURN TRUE`.

Now we prove the correctness and the time complexity of the Stable DFS-BFS algorithm.

Lemma 3.3. *The Stable DFS-BFS algorithm returns* `TRUE` *if the given solution is stable, otherwise returns* `FALSE`.

Proof. Let σ be a permutation that is a solution to the KEG (V, Φ) that we want to test for stability.

- Assume that the solution σ is not stable. Then there is a blocking cycle C which is witnessed by a permutation ρ of C.
  - ◇ If $\rho(c) \prec_c \sigma(c)$ for each $c \in C$, then the graph G_∞ contains the whole cycle C and the DFS algorithm (Lemma 3.13) will find such a cycle C and return `FALSE`.
  - ◇ Otherwise there exists a nonempty set $D \subseteq C$ such that for each $d \in D$: $\rho(d) = \sigma(d)$ and $l^\rho(d) < l^\sigma(d)$ (from Definitions 2.5, 2.4). As $(d, \sigma(d))$ for each $d \in D$ are edges in the solution σ, in some step they will be present in the auxiliary graph G. Let $c \in D$ be the vertex for which $(c, \sigma(c))$ is the last added edge from C to G. After the adding of $(c, \sigma(c))$ when $k = l^\sigma(c)$ we use the BFS algorithm (Lemma 3.13) to find the length of the shortest path from $\sigma(c)$ to c which is $l^\rho(c) - 1$. Thus we find the blocking cycle C and return `FALSE`.
- To prove the other implication, assume that the algorithm returns `FALSE`. It means that either $G_\infty(\sigma)$ is not acyclic or there exists a k such that in the $|V| - k + 1$ iteration of the outer for-cycle the graph G already contains a cycle shorter than k. As G is in that moment in fact a subgraph of G_k, also the graph G_k contains a cycle shorter than k. Then according to Theorem 3.2 the solution σ is not stable.

Thus when the given solution is stable, the Stable DFS-BFS algorithm returns `TRUE`, otherwise it returns `FALSE`. ∎

Before proving the time complexity, we state and prove some auxiliary observations that will be used also in Section 3.3.

Observation 3.4. *The graph $G_i(\sigma)$ is different from the graph $G_{i+1}(\sigma)$ if and only if the solution σ contains at least one cycle of length i.*

Proof. From Definition 3.1 we see that the graph $G_i(\sigma)$ is either exactly the same as the graph $G_{i+1}(\sigma)$ or it has additional edges from all cycles of length i in the permutation σ. ∎

Observation 3.5. $|E(G_1(\sigma))| \leq n + |E(G_\infty(\sigma))|$.

Proof. The number of edges of $G_1(\sigma)$ is at most the number of edges of $G_\infty(\sigma)$ plus the number of edges in σ.

All graphs $G_k(\sigma)$ and $G_\infty(\sigma)$ are defined on the set V of vertices where $|V| = n$. In the permutation σ every vertex $v \in V$ occurs at most in one cycle. Therefore the sum of degrees over all vertices in the cycles of σ is at most $2n$ which gives us at most n edges used in the solution σ. ∎

Lemma 3.6. *The time complexity of the Stable DFS-BFS algorithm is $O((m+n)n)$, where $n = |V|$ and $m = \sum_{i=1}^{n} |\Phi_i|$.*

Proof. In the algorithm we run once the DFS algorithm to test the acyclity of the graph G_∞ and this takes time $O(m+n)$.

Then we add at most n edges (Observation 3.5) and we test each of them whether it participates in a shorter cycle using the BFS algorithm and this takes time $O((m+n)n)$.

Thus the time complexity of the algorithm is $O((m+n)n)$, where $n = |V|$ and $m = \sum_{i=1}^{n} |\Phi_i|$. ∎

3.3. Stable Local algorithm

We test the stability after every change in the solution — after every successful reconnection of the cycles in the solution caused by a heuristic (see Chapter 4). As we will see, all our heuristics make just small local changes to the solution, they reconnect just a small set of vertices.

Therefore we came up with the Stable Local algorithm which uses the knowledge of the local changes and which has the time complexity $O(n^2\sqrt{n})$ which is an improvement compared to the original complexity $\Theta(n^3)$.

By the Stable Local algorithm we want to test a stability of the given permutation σ when we have an additional information about blocking cycles that may violate the stability of σ. We are given a set P of vertices whose size is bounded by a constant and we know that every cycle that blocks the solution σ has to contain at least one of the vertices from P. We call the vertices from P *problematic*.

This requirement might seem a bit strong, but in fact this is exactly what we need to test the stability of heuristics. Thanks to the way the heuristics do changes to the given solution, we can provide a small set P of problematic vertices and guarantee that every blocking cycle contains at least one vertex from P. We

state and prove this later in Chapter 4 where we give more details about the used heuristics.

The main idea of the algorithm is to treat the vertices from P somehow specially, it is sufficient to test just vertices from P whether they participate in a blocking cycle.

Algorithm.
Input: A permutation σ that is a solution of KEG (V, Φ) that we want to test for stability and a set P of problematic vertices.
Output: A logical value determining whether the solution σ is stable or not.

- Construct $G_\infty(\sigma) := (V, E_\infty)$, where an edge (i, j) is in E_∞ if $j \prec_i \sigma(i)$.
- Test the acyclity of $G_\infty(\sigma)$ by the DFS algorithm (see Section 3.4). `IF` $G_\infty(\sigma)$ contains a cycle `THEN`
  - `RETURN FALSE`.
- $G_{|V|+1} := G_\infty$.
- `FOR` each length k `FROM` $|V|$ `DOWNTO` 1 `DO`
  - Construct $G_k(\sigma)$ from $G_{k+1}(\sigma)$ by adding edges $(v, \sigma(v))$ for each v such that $l^\sigma(v) = k$.
  - `IF` $G_k(\sigma) \neq G_{k+1}(\sigma)$ (`IF` an edge from σ was added) `THEN`
    - `FOR` every $v \in P$ `DO`
      - Using the BFS algorithm (see Section 3.5) find the length l of the shortest nontrivial path from v to v in the graph G_k.
      - `IF` $l < k$ `THEN`
        - `RETURN FALSE`.
- `RETURN TRUE`.

Theorem 3.7. *Let σ be an arbitrary permutation that is a solution to the KEG (V, Φ). Let P be a set of problematic vertices such that every blocking cycle in σ contains at least one vertex from P. The Stable Local algorithm determines the stability of the solution σ in time $O((m + n)\sqrt{n}|P|)$, where $n = |V|$ and $m = \sum_{i=1}^{n} |\Phi_i|$.*

We prove this theorem at the end of this section. First we prove a few lemmas, starting with the correctness of the Stable Local algorithm.

Lemma 3.8. *The Stable Local algorithm returns* `TRUE` *if the given solution is stable, otherwise returns* `FALSE`.

Proof. Let σ be a permutation that is a solution to the KEG (V, Φ) that we want to test for stability and let P be a set of problematic vertices.

- Assume that the solution σ is not stable. Then there is a blocking cycle C which is witnessed by a permutation ρ of C.
  - If the blocking cycle C uses just edges that are in the preference lists more preferred to those used in the permutation σ, it means that $\rho(c) \prec_c \sigma(c)$ for each $c \in C$, and the whole cycle C is in the graph G_∞. The algorithm tests the acyclity of G_∞ using the DFS algorithm correctly (Lemma 3.13), so it will find such a cycle C and return `FALSE`.

- ◇ Otherwise C contains at least one edge from the solution σ. It means that there exists a $c \in C$ such that $\rho(c) = \sigma(c)$ and $l^\rho(c) < l^\sigma(c)$ (Definitions 2.5, 2.4). Hence for some k the graph G_k contains a cycle of length less than k. After a set of edges that forms the cycles of length k in the permutation σ is added (the graph G_k was constructed) the algorithm tests whether there appears a shorter cycle that would block σ. We know that it has to test just the cycles containing a problematic vertex. It finds the length l of the shortest cycle containing v for all $v \in P$ (Lemma 3.13) and tests whether it is lower than k. Since the length of C has to be lower than k, the cycle C will be found and the algorithm returns `FALSE`.

- To prove the other implication, assume that the algorithm returns `FALSE`. It means that either $G_\infty(\sigma)$ is not acyclic or there exists a k such that $G_k(\sigma)$ contains a cycle shorter than k. Then according to Theorem 3.2 the solution σ is not stable.

Thus when the given solution is stable, the Stable Local algorithm returns `TRUE`, otherwise it returns `FALSE`. ∎

Lemma 3.9. *Let σ be an arbitrary permutation that is a solution to KEG. Then among the graphs $G_k(\sigma)$, we have at most $O(\sqrt{n})$ different ones.*

Proof. According to Observation 3.4, whenever the graphs $G_i(\sigma)$ and $G_{i+1}(\sigma)$ are different it means that σ contains at least one cycle of length i. In other words, the observation says that to know the number of different graphs $G_k(\sigma)$, it is sufficient to count the number of different lengths of cycles in the solution σ.

On the other hand, we know that the cycles in the solution σ are formed by at most n edges. Now we will count how many different cycles can be in σ in the worst case (what is the maximum).

We take l cycles of different lengths. The lowest sum of numbers of edges of these cycles is attained if we take the sequence $\{1, 2, 3, \ldots, l\}$ as their lengths. Thus they have $l(l+1)/2$ edges in total and every l cycles of different lengths have at least that many edges.

As we know the number of edges that can be used to form the cycles in the solution σ, we can write $l(l+1)/2 \leq n$. By a few simple algebraic manipulations we get

$$\begin{aligned} n &\geq l(l+1)/2 \\ 2n &\geq l^2 + l \\ 2n + 1/4 &\geq l^2 + l + 1/4 = (l + 1/2)^2 \\ \sqrt{2n + 1/4} &\geq l + 1/2 \end{aligned}$$

we obtain $l \leq \sqrt{2n + 1/4} - 1/2$. Thus the maximum number of the different lengths of cycles in σ is at most $\sqrt{2n + 1/4} - 1/2$.

Therefore there are at most $O(\sqrt{n})$ different graphs $G_k(\sigma)$. ∎

Now we calculate the time complexity of Stable Local.

Lemma 3.10. *The time complexity of the Stable Local algorithm is* $O((m+n)\sqrt{n}|P|)$.

Proof. First we run once the DFS algorithm to test the acyclity of the graph G_∞, and this takes $O(m+n)$.

Then for all different graphs G_k we run the BFS algorithm to find the shortest path from every problematic vertex v to itself. As we have $O(\sqrt{n})$ different graphs G_k and $O(|P|)$ different problematic vertices we obtain that the time complexity of the second part is $O((m+n)\sqrt{n}|P|)$.

Thus the time complexity of the algorithm is $O((m+n)\sqrt{n}|P|)$, where $n = |V|$ and $m = \sum_{i=1}^{n} |\Phi_i|$. ∎

Now we have all we need to prove the Theorem 3.7.

Proof. (Of Theorem 3.7)

The theorem follows directly from Lemma 3.8 and Lemma 3.10. ∎

3.4. The DFS algorithm

We use the widely known Depth-First Search algorithm (DFS, see [Knu97]) to find out whether the given oriented graph G is acyclic.

We start in an arbitrary vertex, follow its edges "down" the oriented tree and mark vertices. If we are lead by an edge back to an ancestor, we have found a cycle and the graph is not acyclic.

Informally, we walk on edges of G and as we are visiting vertices we assign states to them. There are three possible states: $state(u)$ is "unvisited" when we haven't visited the vertex u yet. If $state(u)$ is "active" it means that u is an ancestor of a vertex that we are currently dealing with. If $state(u)$ is "done" it means that we have already searched all the branches that lead from the vertex u. Whenever we are lead to an "active" vertex it means we have found a cycle and the graph G is not acyclic. Otherwise when all vertices are "done" it means that the graph is acyclic.

Algorithm.
Input: An oriented graph $G = (V, E)$.
Output: A logical value determining whether the graph is acyclic.

- `FOR` every $u \in V$ `DO`
  - ◇ Set $state(u)$:= "unvisited"
- Choose an arbitrary vertex $v \in V$, set $state(v)$:= "active" and $current := v$
- `REPEAT`
  - ◇ `IF` there exists $u \in V$ such that $(current, u) \in E$ and $state(u) \neq$ "done" `THEN`
    - ∗ `IF` $state(u)$ = "active" `THEN`
      - ○ `RETURN FALSE`.
    - ∗ `ELSE` set $state(u)$:= "active" and $current := u$
  - ◇ `ELSE`
    - ∗ $state(current)$:= "done"
    - ∗ `IF` $current$ has an "active" parent a `THEN`
      - ○ Set $current := a$

    - ∗ `ELSE`
        - ∘ Choose an arbitrary "unvisited" vertex $v \in V$, set $state(v) :=$ "active" and $current := v$
- `UNTIL` all vertices are "done"
- `RETURN TRUE`.

In our implementation we store ancestors in a stack.

Lemma 3.11. *The DFS algorithm tests the acyclity of the given graph G.*

Proof. Assume that G contains a cycle $C = (V_C, E_C)$. In some step the algorithm has to take a vertex $v \in V_C$, mark it as "active" and follow its outgoing edges, in particular follow the edges of the cycle C. In some step it gets back to v and because $state(v)$ is "active" it returns `FALSE`.

On the other hand, assume that G does not contain a cycle. Hence the algorithm is never lead back to an "active" vertex (otherwise we could go on edges in the opposite direction to its ancestors and find a cycle because this "active" vertex would be its own ancestor). Hence it never returns `FALSE`. In every step of the `REPEAT` cycle at least one vertex changes its state. We can see that an "unvisited" vertex can change its state only to "active" and an "active" vertex can not become anything but "done". As there is no cycle in G. Thus after finitely many steps all vertices are "done" and the algorithm returns `TRUE`. ∎

Lemma 3.12. *The time complexity of the DFS algorithm for testing the acyclity of the graph G_∞ is $O(n + m)$.*

Proof. During the DFS algorithm every vertex changes its state at most three times. If we work out the implementation details it is possible to change and test the state of a vertex in a constant time and it is enough to use every edge just once. This gives us at most $O(n + m)$ operations for the DFS algorithm. ∎

3.5. The BFS algorithm

The widely known Breadth-First Search algorithm (BFS, see [Knu97]) may be used to find the length of the shortest path from a vertex v to any other vertex in the given graph G, in particular, to a given vertex w. We use the BFS algorithm to determine the length of the shortest non-trivial path from v to v in the Stable Local algorithm.

During the run of the algorithm we set up a graph distance $\varphi(u)$ from v for every visited vertex u and we group the vertices with the same φ into a set A_φ. If $\varphi(w)$ is determined we are done.

Algorithm.
Input: An oriented graph $G = (V, E)$ and vertices $v, w \in V$.
Output: The length of the shortest non-trivial path from v to w.

- `FOR` every $v \in V$ `DO`
    - ⋄ Set $\varphi(u) := \infty$
- Set $A_0 := \{v\}$ and $i := 0$

- `REPEAT`
  - $\diamond$ inc(i)
  - $\diamond$ `FOR` every edge $(t, u) \in E$ such that $t \in A_{i-1}$ and $\varphi(u) = \infty$ `DO`
    - $*$ Set $\varphi(u) := i$
    - $*$ Insert u into the set A_i
- `UNTIL` $\varphi(w) < \infty$ or $A_i = \emptyset$
- `RETURN` $\varphi(w)$

In our implementation we keep the unprocessed vertices of A_{i-1} and so far visited vertices from A_i in a queue.

Lemma 3.13. *The BFS algorithm returns the length of the shortest path between given vertices v and w of a given graph G. If $v = w$ then it returns the length of the shortest non-trivial cycle containing v.*

Proof. In the i-th iteration of the `REPEAT` cycle we construct a set A_i containing exactly the vertices of G with the graph distance from v equal to i, and for every $u \in A_i$ we set $\varphi(u)$ to i.

In every step at least one vertex is put into A_i otherwise the `UNTIL` clause is satisfied. Thus after finitely many steps the algorithm finishes and returns $\varphi(w)$ which is either the graph distance from v to w or ∞ in case there is no path from v to w.

In the case $v = w$ it returns the length of the shortest non-trivial path from v to v (in fact the shortest cycle containing the vertex v). ∎

Lemma 3.14. *The time complexity of the BFS algorithm for finding the shortest path between two vertices of the graph G_k is $O(n + m)$.*

Proof. During the BFS algorithm every vertex is inserted into A_i and taken from A_i once and $\varphi(u)$ is set at most twice.

If we work out the implementation details it is possible to set $\varphi(u)$ and operate with A_i in $O(1)$ time and it is enough to use every edge from E just once.

Thus we need $O(n + m)$ operations in total. ∎

4. Heuristics

In the KEG we want to find a stable solution that covers as many players as possible and that has as short cycles as possible. Obviously these conditions may be contradictory, but even if we state them more clearly then finding such a solution is NP-complete (Theorems 2.7, 2.8, 2.9, 2.10 and 2.11).

We already have the TTC algorithm that generates a stable solution very quickly, but there may be a better solution than that and the TTC algorithm is not able to find it because it does not care about non-covered players and long cycles.

To find out how "optimal" is the permutation obtained by the TTC algorithm, Cechlárová and Lacko [CL06] do TTC first, and then they perform heuristics and they watch how often they succeed. We use this idea of running heuristics upon the result of TTC in order to find as good solution as possible in a reasonable time.

Two heuristics were introduced in [CL06]:

- **Cut cycle heuristic**. It tries to shorten a cycle from the given solution by reconnecting vertices of the cycle and creating two cycles.
- **Cut and add heuristic**. It tries to cover more vertices than the given solution does. It takes a cycle, cuts it into two and to one of the newly obtained cycles it adds a non-covered vertex.

We propose two new heuristics:

- **Cut two to three heuristic**. This one is similar to the heuristics mentioned above but instead of cutting one cycle, it takes two and tries to reconnect them into three.
- **Forbidden edges heuristic**. This heuristic is significantly different from all the heuristics mentioned above. While the previous ones change just a small part of the given solution, Forbidden edges may generate a solution that is completely different from the given one. In short, it takes a stable solution, forbid constantly many edges used in the given solution, and by TTC it generates a new solution that can not use the forbidden edges. For more information about Forbidden edges heuristics see Section 4.2.

A solution generated by these heuristics may be unstable and it is necessary to test it for stability. Since we have the changes made by the heuristics under control and we know that they are just local, we can together with the solution σ deliver also a small set P of problematic vertices and guarantee that every blocking cycle contains at least one of them. This is exactly the input the Stable Local algorithm needs to work correctly and we can determine the stability of σ using Stable Local as it is stated in Section 3.3.

In the following sections we give more details about heuristics and we prove the necessary condition for using the Stable Local algorithm. In Section 4.1 we deal with all three Cut heuristics, because they have a lot of things in common. In Section 4.2 we describe the Forbidden edges heuristic.

In [CL06] the authors by Cut cycle or Cut and add heuristics tried to improve the solution systematically by trying every pair of vertices in every cycle. We do

not do it this way, because we are afraid of bad time complexity, so we try to obtain a better solution by mixing various heuristics and by choosing the cycles and vertices to change randomly.

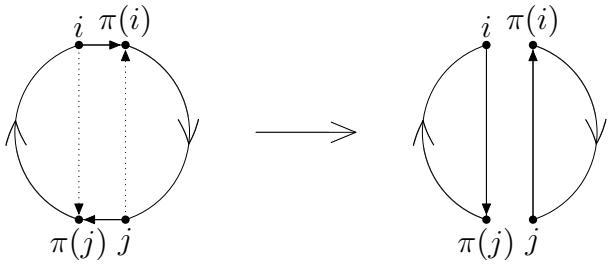
4.1. Cut algorithms

The Cut cycle algorithm

This algorithm tries to improve the given solution by cutting a cycle into two smaller ones.

Algorithm.
Input: A solution π.
Output: A solution ϱ and a logical value determining whether ϱ differs from π or not (that is, whether ϱ is a better solution than π).

- Randomly select a cycle to cut.
- Randomly select two distinct vertices i, j of the cycle.
- Try to cut the cycle in two by reconnecting i and j. Let $\sigma(i) := \pi(j)$, $\sigma(j) := \pi(i)$ and for every other vertex m set $\sigma(m) := \pi(m)$.
- `IF` it is not possible to reconnect (that is, $\pi(j) \notin \Phi_i$ or $\pi(i) \prec_i \pi(j)$ or $\pi(i) \notin \Phi_j$ or $\pi(j) \prec_j \pi(i)$) `THEN`
  - ◇ `RETURN` (π,`FALSE`).
- Test the stability of σ. `IF` σ is not stable `THEN`
  - ◇ `RETURN` (π,`FALSE`).
- `ELSE`
  - ◇ `RETURN` (σ,`TRUE`).



(We implemented this heuristic in such a way that more than two cycles may be created instead of only two. However, it seems to us too complicated and that the chance to choose a right set of vertices to reconnect is small, so it is not used in the program.)

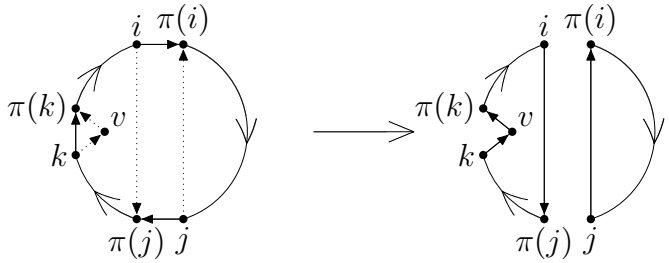
The Cut and add algorithm

The algorithm tries to improve the given solution by cutting a cycle into two smaller ones and adding an isolated vertex to one of the shorter cycles.

Algorithm.
Input: A solution π.
Output: A solution ϱ and a logical value determining whether ϱ differs from π or not (that is, whether ϱ is a better solution than π).

- Randomly select one cycle to cut.
- Randomly select one isolated vertex v.

- Randomly select three vertices i, j, k of the cycle.
- Try to cut the cycle in two by reconnecting the vertices i, j and k and adding the vertex v. Let $\sigma(i) := \pi(j)$, $\sigma(j) := \pi(i)$, $\sigma(k) := v$, $\sigma(v) := \pi(k)$ and for every other vertex m set $\sigma(m) := \pi(m)$.
- `IF` it is not possible to reconnect (that is, $\pi(j) \notin \Phi_i$ or $\pi(i) \prec_i \pi(j)$ or $\pi(i) \notin \Phi_j$ or $\pi(j) \prec_j \pi(i)$ or $v \notin \Phi_k$ or $\pi(k) \prec_i v$ or $\pi(k) \notin \Phi_v$) `THEN`
  - ◇ `RETURN` (π,`FALSE`).
- Test the stability of σ. `IF` σ is not stable `THEN`
  - ◇ `RETURN` (π,`FALSE`).
- `ELSE`
  - ◇ `RETURN` (σ,`TRUE`).

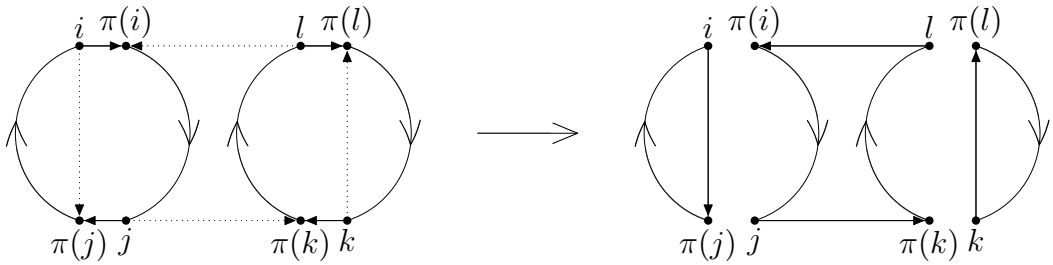


The Cut two to three algorithm

This algorithm is our variation of the previously introduced ones. It tries to improve the given solution by cutting two cycles into three shorter ones.

Algorithm.
Input: A solution π.
Output: A solution ϱ and a logical value determining whether ϱ differs from π or not (that is, whether ϱ is a better solution than π).

- Randomly select two cycles to cut.
- Randomly select two vertices i, j of the first cycle and two vertices k, l of the second cycle.
- Try to cut the cycles in three by reconnecting i, j, k and l. Let $\sigma(i) := \pi(j)$, $\sigma(j) := \pi(k)$, $\sigma(k) := \pi(l)$, $\sigma(l) := \pi(i)$ and for every other vertex m set $\sigma(m) := \pi(m)$.
- `IF` it is not possible to reconnect `THEN`
  - ◇ `RETURN` (π,`FALSE`).
- Test the stability of σ. `IF` σ is not stable `THEN`
  - ◇ `RETURN` (π,`FALSE`).
- `ELSE`
  - ◇ `RETURN` (σ,`TRUE`).



Note that in every Cut algorithm when we were creating a new solution σ from π we changed just the out-coming edges of a constantly small set P of vertices, and for every vertex v such that $v \notin P$ is $\pi(v) = \sigma(v)$. The set P will be the one that is necessary in Theorem 3.7 but we still have to prove that it satisfies the condition that every blocking cycle in σ contains at least one vertex from P.

Let V be the set of vertices, let π be a stable solution to a KEG (V, Φ) and let σ be a solution to (V, Φ) that differs from π only for vertices from P where $P \subseteq V$. Thus the permutation σ is as follows.

- For all $u \notin P$, $\sigma(u) = \pi(u)$.
- For all $v \in P$, $\sigma(v) = \pi(\xi(v))$, where the function ξ is a bijection from P to P that is determined by the the reconnecting performed by the heuristic.

Lemma 4.1. *Any blocking cycle in σ contains at least one vertex from the set P.*

Proof. Assume for contradiction that there is a set of vertices $C \subseteq V - P$ that forms a blocking cycle witnessed by a permutation ρ of C. Thus for every $c \in C$ we have $c \notin P$, hence $\sigma(c) = \pi(c)$.

When C forms a blocking cycle it means that for every $c \in C : \rho \sqsubset_c \sigma$. But because $\sigma(c) = \pi(c)$, we also have $\rho \sqsubset_c \pi$ for every $c \in C$. Thus the set C is also a blocking cycle for the permutation π. This is a contradiction with the assumption that π is stable. Hence the lemma holds. ∎

Lemma 4.2. *The Cut algorithms end after finitely many steps and return a stable solution to the KEG.*

Proof. We try to create a new solution σ by reconnecting a randomly selected vertices in a given stable solution π. If the reconnection is possible, we test σ for stability using Stable Local algorithm that we can use to determine the stability according to the Lemma 4.1. If we were successful and we have obtained a new stable solution, we return (σ,`TRUE`), otherwise we return (π,`FALSE`). ∎

Lemma 4.3. *The time complexity of the Cut algorithms is $O((m+n)\sqrt{n}|P|)$.*

Proof. The algorithms select the vertices and do the reconnection in the time $O(n+m)$. To test the stability of σ the time $O((m+n)\sqrt{n}|P|)$ is needed. Thus the time complexity of the Cut algorithms is $O((m+n)\sqrt{n}|P|)$. ∎

4.2. The TTC with forbidden edges algorithm

The previous heuristics have a disadvantage that they cannot do bigger changes that just divide cycles into smaller ones. Unfortunately, it may happen that there are no longer any cycles to cut and the heuristics can not improve the solution further although it is not so "good" yet. Also, since the TTC algorithm generates always the same output for a given input (Theorem 1 in [CRM00]) the situation will be similar even if we try to run the program repeatedly.

Thus we were curious whether we could force the TTC algorithm to give us an alternative solution. From this follows an idea of randomly changing the input

of TTC and by this obtaining an alternative solution. Since this new solution is not necessary stable, we test it for the stability afterwards. Since we want to use the Stable Local algorithm to test the stability, we do the changes of the input carefully. We choose constantly many edges from the cycles in the original solution π, "forbid" them (remove them from the preference lists that are the input for TTC) and generate a new solution σ. As we will see later, we can use the Stable Local algorithm to determine the stability. We implemented the TTC with forbidden edges algorithm as a heuristic, so if the obtained solution σ is stable we decide between the solution σ and π which one we like more and we throw the other away.

It may seem that with this forbidding strategy we are disallowing the players their preferred donors. However, we hope that by this we are able to prevent a situation where the TTC algorithm had assigned a donor to a player in such a bad way that he prevented a solution much better in the global point of view.

Algorithm.
Input: A solution π
Output: A solution ϱ and a logical value determining whether ϱ differs from π or not (that is, whether ϱ is a better solution than π).

- Randomly select constantly many edges from the solution π and put them into the set P_E. Put the heads of the edges from P_E into a set P of vertices.
- Run the TTC algorithm on the preference lists without the edges from P_E and in this way generate a solution σ.
- Using Stable Local test σ for stability. `IF` it is not stable `THEN`
  - ◇ `RETURN` (π,`FALSE`).
- Test which one of the solutions σ and π is better. `IF` the better solution is σ `THEN`
  - ◇ `RETURN` (σ,`TRUE`).
- `ELSE`
  - ◇ `RETURN` (π,`FALSE`).

Now we prove the correctness and the time complexity of the Forbidden edges algorithm. As we want to use the Stable Local algorithm to test the stability of the generated solution σ, we determine the set P of problematic vertices and we prove that they satisfy the necessary condition that every blocking cycle of σ contains at least one vertex from P.

During the algorithm we have "forbidden" edges P_E from the solution π to generate the solution σ. As the necessary set of problematic vertices we take the set P such that $p \in P$ if and only if $(p, \pi(p)) \in P_E$. Now we prove that this set satisfies the condition.

Let π be a solution to the KEG (V, Φ) and let $P \subseteq V$ be a small set of vertices. We create an auxiliary KEG (V, Ψ) by forbidding edges from the solution π. More formally, let Ψ be as follows:
(i) $\Psi_i := \Phi_i$ for each $i \in V - P$ and
(ii) $\Psi_i := \Phi_i - \{\pi(i)\}$ for each $i \in P$.

Observation 4.4. *Let σ be the solution to the KEG (V, Ψ) generated by the TTC algorithm. The solution σ is both a stable solution to the KEG (V, Ψ) and a solution (not necessarily stable) to the the KEG (V, Φ).*

Proof. The solution σ is a stable solution to the KEG (V, Ψ) because it is created by TTC that generates stable solutions. Since σ is a solution to the KEG (V, Ψ), we have $\sigma(i) \in \Psi_i$ or $\sigma(i) = i$ for each $i \in V$. As $\Psi_i \subseteq \Phi_i$ for each $i \in V$, we also have $\sigma(i) \in \Phi_i$ for each $i \in V$. Therefore σ is also a solution to the KEG (V, Φ) ■

Lemma 4.5. *Let σ be the solution to the KEG (V, Ψ) generated by the TTC algorithm. The stability of the solution σ can be violated only by a cycle containing at least one vertex from P.*

Proof. For a contradiction assume that there is a blocking cycle $C \subseteq V - P$ witnessed by a permutation ρ of C that blocks the solution σ with respect to KEG (V, Φ).

The cycle C is blocking, so $\rho \sqsubset_c \sigma$ for each $c \in C$ with respect to Φ. As $C \cap P = \emptyset$ then $\Psi_c = \Phi_c$ for each $c \in C$ which means that also $\rho \sqsubset_c \sigma$ for each $c \in C$ with respect to Ψ. Thus the cycle C is also a blocking cycle of σ with respect to KEG (V, Ψ). However this is a contradiction because according to Observation 4.4, σ is a stable solution to the KEG (V, Ψ). Therefore every blocking cycle of the solution σ contains at least one vertex from P. ■

Lemma 4.6. *The Forbidden edges algorithm ends after finitely many steps and returns a stable solution to the KEG.*

Proof. According to Observation 4.4, Forbidden edges generates a solution σ to the KEG. The problem is that σ may not be stable. Therefore we test σ for stability using Stable Local algorithm that we can use to determine the stability according to Lemma 4.5. After that we just decide which one of the solutions π and σ is "better" and return it. ■

Lemma 4.7. *The time complexity of the Forbidden edges algorithm is $O((m+n)\sqrt{n}|P|)$.*

Proof. The algorithm copies the preference lists and forbids $|P|$ edges from the solution π in time $O(m+n)$. To generate the solution σ the TTC algorithm is used, which works in $O(m+n)$. To test the stability of σ the time $O((m+n)\sqrt{n}|P|)$ is needed. The "better" solution is determined in time $O(n)$. Thus the total time complexity of the Forbidden edges algorithm is $O((m+n)\sqrt{n}|P|)$. ■

5. Simulations and tests

In this chapter, we present the results of several tests we performed. As we do not have any real data, we generated a couple of samples using the method described in [CL06].

We use the Blood model to have a better approximation of real data than just random data. The model gives for each pair recipient-donor a pair of the blood types that is chosen according to the probabilities given in Table 1. The data in the table are taken from [SG+05].

Blood type	Donor O	Donor A	Donor B	Donor AB
Patient O	14.0	37.8	12.0	2.0
Patient A	6.3	6.8	5.1	2.8
Patient B	2.4	6.1	1.2	2.1
Patient AB	0.5	0.5	0.2	0.1

Table 1.

When the blood types are determined, we generate the preference lists. For each recipient i we randomly ordered the donors of the same blood type and with the probability $1 - r$, where r is the probability of rejection, we put each of them into the preference list of i.

We generated several input sets using the Blood model with the probability of rejection equal to 0.2 and 0.4 and for the number of players n equal to $30, 100, 500, 1000$ and 10000.

We used this data sets to:

- test how successful are our heuristics in comparison to all possible stable solutions (Section 5.1);
- test how fast is the Stable DFS-BFS in comparison to the algorithm based on the Floyd-Warshall algorithm (see Section 5.2);
- examine the improvement done by heuristics in comparison to the solution generated by TTC (Section 5.3);
- see more details on how our heuristics work on various inputs (Section 5.4).

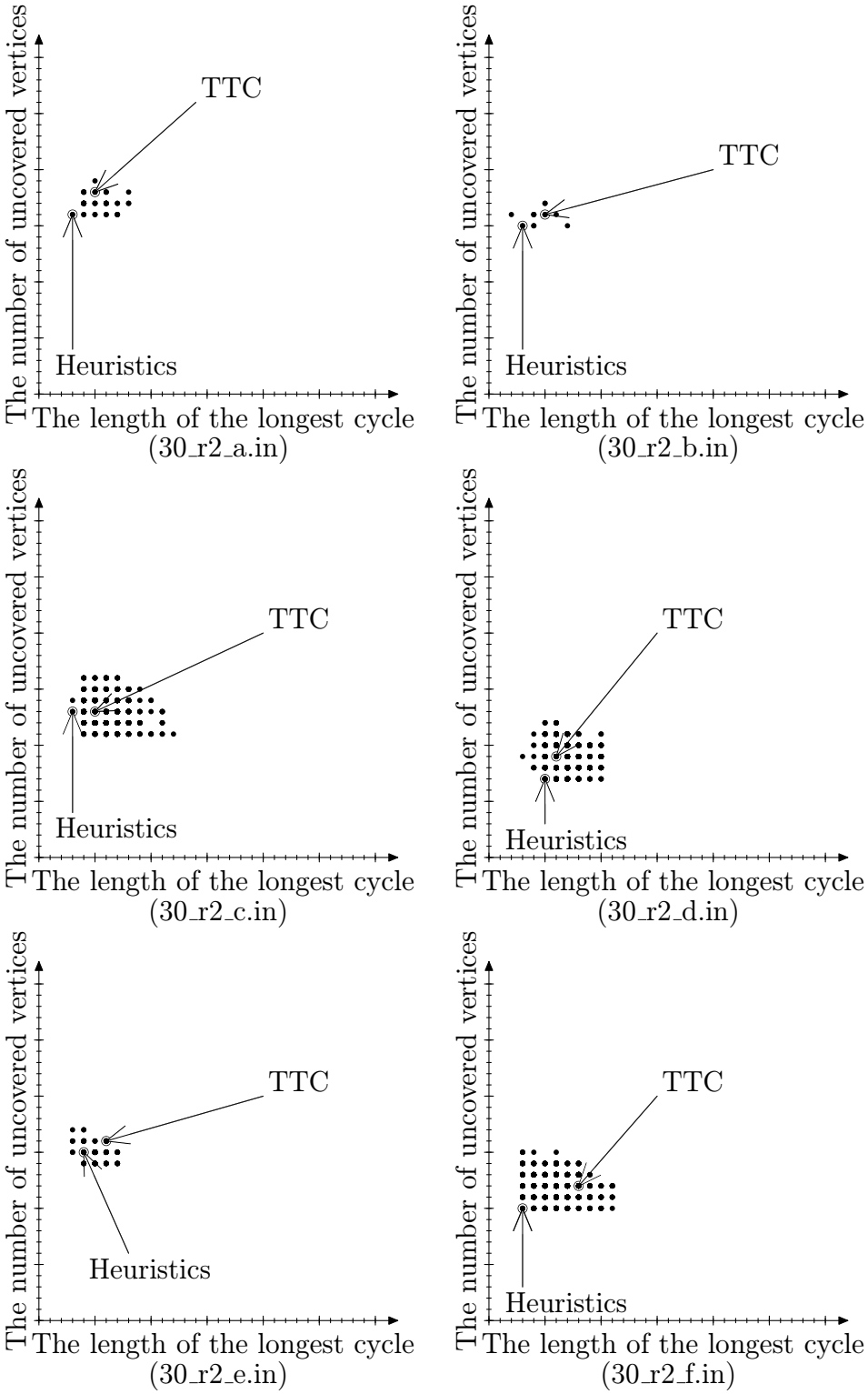
The "quality" of the solution is measured by the length of the longest cycle (the less the better), the number of uncovered vertices (again the less the better) and the number of cycles longer than one (the more cycles the better).

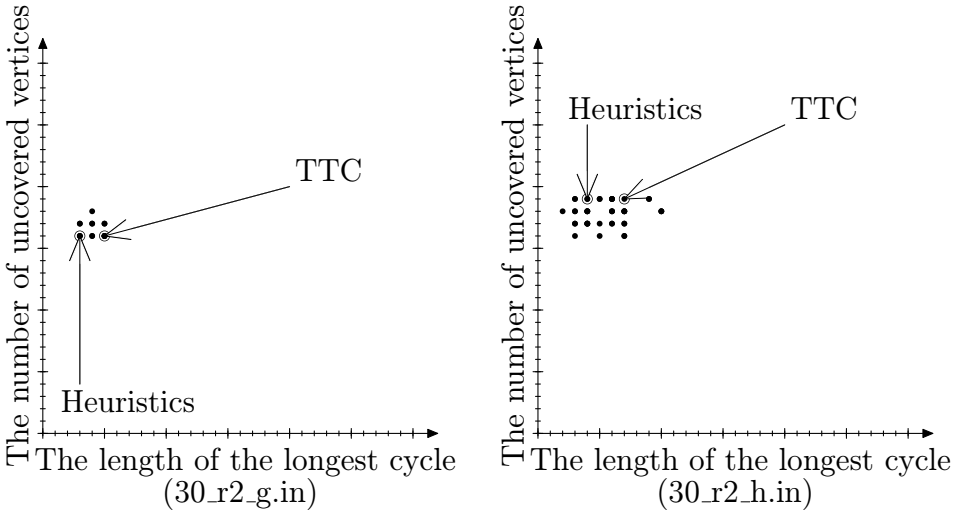
5.1. All stable solutions vs. those found by heuristics

We performed several tests to find out how good is the solution generated by TTC and those generated by the heuristics in comparison with all possible stable solutions. As we are only able to find all stable solutions for a KEG that does not contain too many players, we generated several input sets with 30 players and the probability of rejection equal to 0.2. For this data we found all stable solutions and compared them with the solutions obtained by TTC and the heuristics.

The sets of all stable solutions for these inputs were generated by exhaustive search done by the program jadro [Jel09]. The input instances with 30 patients were the largest for which the program was able to finish in bearable time.

In the following pictures all solutions with respect to two different parameters are depicted: the length of the longest cycle in the solution and the number of the vertices not covered by the solution. Several stable solutions usually correspond to a single point, because they have the same values of the examined parameters. The solution generated by the TTC algorithm is marked in the picture as well as the best solution obtained by our heuristics after approximately 10 executions of the program, where every execution calls all heuristics 20000 times.





In the cases when our solution was relatively bad, the number of solutions was smaller in general. On the other hand, when the number of solutions was higher, our heuristics performed very well. For example, `30_r2_h.in` has 45 stable solutions and `30_r2_f.in` has 1695 stable solutions.

5.2. Floyd-Warshall versus Stable DFS-BFS

Although the Stable DFS-BFS algorithm has the same asymptotical time complexity ($\Theta(n^3)$) as the algorithm for testing the stability using the Floyd-Warshall algorithm, there is a big difference in the real running time.

We performed a set of simulations on various samples. We generated a stable solution using the TTC algorithm and than we measured how long it takes to test it for the stability using FW and then using Stable DFS-BFS.

We put the obtained times into Table 2. The first column gives the number of vertices of the sample, the second gives the value of the parameter r that determines the probability of rejection, the third column shows how long does FW took to test the stability and the last column shows the time needed by Stable DFS-BFS.

# vertices	Value of r	FW time	DFS-BFS time
10000	0.2	247m12.822s	0m23.523s
10000	0.2	246m55.227s	0m23.391s
10000	0.2	246m49.659s	0m23.673s
1000	0.2	0m14.328s	0m00.142s
1000	0.2	0m14.378s	0m00.147s
1000	0.2	0m14.391s	0m00.146s
1000	0.2	0m14.389s	0m00.147s
1000	0.2	0m14.380s	0m00.137s

Table 2. (part a.)

# vertices	Value of r	FW time	DFS-BFS time
1000	0.4	0m14.333s	0m00.108s
1000	0.4	0m14.314s	0m00.116s
1000	0.4	0m14.317s	0m00.114s
1000	0.4	0m14.319s	0m00.116s
1000	0.4	0m14.317s	0m00.099s
500	0.2	0m01.781s	0m00.032s
500	0.2	0m01.811s	0m00.032s
500	0.2	0m01.810s	0m00.031s
500	0.2	0m01.809s	0m00.032s
500	0.2	0m01.807s	0m00.031s
500	0.4	0m01.798s	0m00.022s
500	0.4	0m01.794s	0m00.021s
500	0.4	0m01.795s	0m00.024s
500	0.4	0m01.796s	0m00.022s
500	0.4	0m01.797s	0m00.025s

Table 2. (part b.)

We suppose that the reason why DFS-BFS is faster than FW is that the running time of DFS-BFS depends on the number of edges, while FW always has to process the whole matrix $n \times n$. While the number of edges is of the order n^2, the multiplicative constant is much smaller than 1. Also the work with big matrices may be demanding on the cache.

5.3. The improvement done by heuristics vs. TTC

We have done a couple of tests to find out how much our heuristics can improve the solution generated by TTC. As the TTC algorithm is deterministic and our heuristics are not, we put into Table 3 the solution generated by TTC together with solutions obtained by five executions of the program, where every execution calls all heuristics 20000 times.

# vert	Value of r	TTC longest	TTC # uncov	TTC # cycles	Heur. longest	Heur. # uncov	Heur. cycles
1000	0.4	35	468	56	35	466	67
					30	467	72
					30	467	73
					35	467	69
					35	467	66
1000	0.4	33	492	52	25	492	68
					24	492	63
					30	491	67
					24	492	63
					27	492	64

Table 3. (part a.)

# vert	Value of r	TTC longest	TTC # uncov	TTC # cycles	Heur. longest	Heur. # uncov	Heur. cycles
500	0.4	16	266	32	15	264	44
					14	265	43
					14	265	42
					15	263	49
					15	264	50
500	0.4	22	246	36	21	246	43
					22	246	45
					22	245	45
					22	246	45
					22	246	46
1000	0.2	44	476	58	42	475	72
					42	475	74
					42	475	73
					35	475	74
					42	475	74
1000	0.2	42	477	57	41	477	81
					31	476	76
					41	476	82
					41	476	81
					41	477	74
1000	0.2	46	472	49	35	468	68
					35	468	67
					25	468	67
					29	468	64
					40	469	65
500	0.2	27	217	37	27	216	47
					22	216	48
					23	216	47
					23	216	46
					23	216	45
500	0.2	27	227	39	23	226	54
					23	226	52
					23	226	52
					23	226	51
					23	226	54
500	0.2	19	249	43	17	247	53
					17	245	54
					19	247	51
					19	247	52
					19	246	49

Table 3. (part b.)

From Table 3 we can see that our heuristics always were able to do some improvement. Although the number of covered vertices never changes a lot, the improvement of the length of the longest cycle and the number of cycles is interesting.

5.4. The examples of progress of heuristics

We also give a couple of examples of executions of the program on various inputs, where every execution calls all heuristics 20000 times.

1. $n = 30$, $r = 0.2$, input: `30_r2_a.in`

Type of heuristic	Longest	# uncovered	# cycles	Time
TTC algorithm	5	18	3	0h 00m 00.017s
Forbidden edges	5	18	4	0h 00m 00.018s
Forbidden edges	5	17	4	0h 00m 00.018s
Forbidden edges	4	17	5	0h 00m 00.018s
Final results	4	17	5	0h 00m 01.121s

2. $n = 30$, $r = 0.2$, input: `30_r2_b.in`

Type of heuristic	Longest	# uncovered	# cycles	Time
TTC algorithm	5	16	5	0h 00m 00.020s
Forbidden edges	4	16	5	0h 00m 00.022s
Cut cycle	4	16	6	0h 00m 00.022s
Final results	4	16	6	0h 00m 01.225s

3. $n = 30$, $r = 0.2$, input: `30_r2_c.in`

Type of heuristic	Longest	# uncovered	# cycles	Time
TTC algorithm	5	13	5	0h 00m 00.017s
Cut cycle	5	13	6	0h 00m 00.017s
Cut cycle	3	13	7	0h 00m 00.017s
Final results	3	13	7	0h 00m 00.996s

4. $n = 100$, $r = 0.2$, input: `100_r2_a.in`

Type of heuristic	Longest	# uncovered	# cycles	Time
TTC algorithm	10	54	10	0h 00m 00.002s
Forbidden edges	9	54	11	0h 00m 00.019s
Forbidden edges	7	52	12	0h 00m 00.024s
Cut cycle	7	52	13	0h 00m 00.027s
Cut cycle	7	52	14	0h 00m 00.053s
Cut cycle	7	52	15	0h 00m 00.119s
Final results	7	52	15	0h 00m 10.194s

5. $n = 100$, $r = 0.2$, input: `100_r2_b.in`

Type of heuristic	Longest	# uncovered	# cycles	Time
TTC algorithm	9	53	13	0h 00m 00.001s
Forbidden edges	8	53	13	0h 00m 00.005s
Cut cycle	7	53	14	0h 00m 00.059s
Cut cycle	7	53	15	0h 00m 00.136s
Final results	7	53	15	0h 00m 10.496s

6. $n = 100$, $r = 0.2$, input: `100_r2_c.in`

Type of heuristic	Longest	# uncovered	# cycles	Time
TTC algorithm	13	49	13	0h 00m 00.001s
Cut cycle	11	49	14	0h 00m 00.007s
Cut cycle	11	49	15	0h 00m 00.008s
Cut cycle	11	49	16	0h 00m 00.039s
Forbidden edges	7	45	16	0h 00m 09.377s
Cut cycle	7	45	17	0h 00m 09.394s
Cut cycle	7	45	18	0h 00m 09.428s
Final results	7	45	18	0h 00m 11.933s

7. $n = 500$, $r = 0.2$, input: `500_r2_a.in`

Type of heuristic	Longest	# uncovered	# cycles	Time
TTC algorithm	27	217	37	0h 00m 00.078s
Forbidden edges	27	217	39	0h 00m 00.604s
Forbidden edges	22	217	40	0h 00m 00.839s
Cut cycle	22	217	41	0h 00m 01.935s
Cut cycle	22	217	42	0h 00m 06.697s
Cut cycle	22	217	43	0h 00m 12.863s
Cut cycle	22	217	44	0h 00m 35.483s
Cut cycle	22	217	45	0h 00m 51.279s
Cut cycle	22	217	46	0h 01m 25.479s
Final results	22	217	46	0h 06m 52.914s

8. $n = 500$, $r = 0.2$, input: `500_r2_b.in`

Type of heuristic	Longest	# uncovered	# cycles	Time
TTC algorithm	27	227	39	0h 00m 00.086s
Forbidden edges	27	227	40	0h 00m 00.833s
Cut cycle	27	227	41	0h 00m 00.947s
Forbidden edges	23	226	41	0h 00m 01.341s
Cut cycle	23	226	42	0h 00m 02.544s
Cut cycle	23	226	43	0h 00m 05.604s
Cut cycle	23	226	44	0h 00m 18.345s
Cut cycle	23	226	45	0h 00m 44.582s
Cut cycle	23	226	46	0h 00m 59.304s
Cut cycle	23	226	47	0h 03m 06.455s
Cut cycle	23	226	48	0h 03m 34.989s
Final results	23	226	48	0h 06m 53.763s

9. $n = 500$, $r = 0.2$, input: `500_r2_c.in`

Type of heuristic	Longest	# uncovered	# cycles	Time
TTC algorithm	19	249	43	0h 00m 00.085s
Forbidden edges	19	247	43	0h 00m 00.864s
Forbidden edges	17	247	43	0h 00m 03.275s
Cut cycle	17	247	44	0h 00m 15.843s
Cut cycle	17	247	45	0h 00m 23.327s
Cut cycle	17	247	46	0h 00m 28.292s

Cut cycle	17	247	47	0h 00m 37.171s
Cut cycle	17	247	48	0h 00m 38.062s
Cut cycle	17	247	49	0h 00m 55.819s
Cut cycle	17	247	50	0h 01m 24.521s
Cut cycle	17	247	51	0h 01m 25.070s
Cut cycle	17	247	52	0h 02m 23.050s
Cut cycle	17	247	53	0h 02m 36.701s
Final results	17	247	53	0h 06m 23.307s

10. $n = 1000$, $r = 0.2$, input: `1000_r2_a.in`

Type of heuristic	Longest	# uncovered	# cycles	Time
TTC algorithm	44	476	58	0h 00m 00.226s
Forbidden edges	44	475	59	0h 00m 04.944s
Forbidden edges	44	475	62	0h 00m 07.203s
Cut cycle	44	475	63	0h 00m 30.855s
Cut cycle	44	475	64	0h 00m 50.203s
Cut cycle	42	475	65	0h 00m 52.512s
Cut cycle	42	475	66	0h 02m 01.338s
Cut cycle	42	475	67	0h 05m 43.378s
Forbidden edges	30	475	69	0h 06m 47.021s
Cut cycle	30	475	70	0h 07m 28.107s
Cut cycle	30	475	71	0h 07m 44.364s
Cut cycle	28	475	72	0h 07m 44.478s
Cut cycle	28	475	73	0h 08m 03.818s
Cut cycle	28	475	74	0h 08m 44.376s
Cut cycle	28	475	75	0h 10m 12.318s
Cut cycle	28	475	76	0h 10m 22.501s
Cut cycle	28	475	77	0h 12m 01.400s
Cut cycle	28	475	78	0h 13m 15.296s
Cut cycle	28	475	79	0h 13m 21.425s
Cut cycle	28	475	80	0h 13m 31.444s
Cut cycle	28	475	81	0h 13m 33.260s
Cut cycle	28	475	82	0h 16m 51.480s
Cut cycle	28	475	83	0h 17m 03.776s
Cut cycle	28	475	84	0h 17m 10.922s
Final results	28	475	84	0h 32m 52.688s

11. $n = 1000$, $r = 0.2$, input: `1000_r2_b.in`

Type of heuristic	Longest	# uncovered	# cycles	Time
TTC algorithm	42	477	57	0h 00m 00.385s
Forbidden edges	42	477	59	0h 00m 09.177s
Forbidden edges	36	477	59	0h 00m 28.716s
Forbidden edges	31	476	68	0h 00m 43.524s
Cut cycle	31	476	69	0h 00m 57.149s
Cut cycle	31	476	70	0h 01m 17.747s
Cut cycle	31	476	71	0h 01m 19.545s
Cut cycle	31	476	72	0h 01m 24.224s

Cut cycle	31	476	73	0h 01m 36.851s
Cut cycle	31	476	74	0h 03m 59.316s
Cut cycle	31	476	75	0h 05m 01.335s
Cut cycle	31	476	76	0h 06m 02.766s
Cut cycle	31	476	77	0h 07m 41.802s
Cut cycle	31	476	78	0h 10m 04.314s
Cut cycle	31	476	79	0h 12m 03.200s
Cut cycle	31	476	80	0h 18m 16.152s
Cut cycle	31	476	81	0h 28m 31.571s
Final results	31	476	81	0h 34m 24.390s

12. $n = 10000$, $r = 0.2$, input: `10000_r2_a.in`

Type of heuristic	Longest	# uncovered	# cycles	Time
TTC algorithm	136	4762	211	0h 00m 16.313s
Forbidden edges	136	4761	213	0h 20m 22.156s
Forbidden edges	101	4761	225	2h 02m 38.672s
Cut cycle	101	4761	226	13h 54m 11.965s
Cut cycle	101	4761	227	31h 18m 07.058s
Cut cycle	101	4761	228	33h 03m 07.318s
Cut cycle	101	4761	229	36h 31m 37.672s
Cut cycle	101	4761	230	87h 02m 14.197s
Final results	101	4761	230	99h 19m 37.756s

We were surprised to see that the process of improving always follows the similar pattern. First, Forbidden edges succeeds a few times which does more or less all the interesting changes, and all the following improvements are done by Cut cycle and they are just increasing the number of cycles. We suppose that Forbidden edges is very good at making greater changes, but it is more difficult for it to do some smaller ones.

6. Conclusion

We dealt with the problem of kidney exchange. We proposed two new heuristics, one of which seems to bring significant improvement. We improved the stability testing algorithm to obtain better running time. In case of testing changes performed by the heuristics, we also have better asymptotical time complexity.

It may be interesting to explain the behaviour of the heuristics on various data sets. As the Forbidden edges heuristic seems to be more successful than others, we may ask whether one can find other radical heuristics.

In real data, there may be some patterns that could be used to create more effective heuristics. It could be worth while to find such patterns.

7. References

[AB+07] David J. Abraham, Avrim Blum, and Tuomas Sandholm. Clearing algorithms for barter exchange markets: enabling nationwide kidney exchanges. In *Proc. 8th ACM conference on Electronic commerce (EC '07)*, pages 295–304, 2007. (Cited on page 1.)

[BC07] Peter Biró and Katarína Cechlárová. Inapproximability of the kidney exchange problem. *Information Processing Letters*, 101:199–202, 2007. (Cited on page 6.)

[CH99] Katarína Cechlárová and Jana Hajduková. Stability testing in coalition formation games. In *Proc. 5th international symposium on Operational research (SOR)*, pages 111–116, 1999. (Cited on pages 7, 8.)

[CL06] Katarína Cechlárová and Vladimír Lacko. The kidney exchange problem: How hard is it to find a donor? In *IM Preprint 4/2006 (2006)*, 2006. (Cited on pages 1, 3, 4, 5, 6, 16, 22.)

[CRM00] Katarína Cechlárová and Antonio Romero-Medina. Stability in coalition formation games. *Int. J. Game Theory*, 29:487–494, 2000. (Cited on pages 5, 19.)

[Irv07] Robert W. Irving. The cycle roommates problem: a hard case of kidney exchange. *Information Processing Letters*, 103:1–4, 2007. (Cited on page 6.)

[Jel09] Eva Jelínková. Jadro. Private communication, 2009. (Cited on page 22.)

[KdK+05] K. Keizer, M. de Klerk, B. Haase-Kromwijk, and W. Weimar. The Dutch algorithm for allocation in living donor kidney exchange. *Transplantation Proceedings*, 37:589–591, 2005. (Cited on page 1.)

[Knu97] Donald E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms (3rd Edition)*. July 1997. (Cited on pages 13, 14.)

[SG+05] Dorry L. Segev, Sommer E. Gentry, Daniel S. Warren, Brigitte Reeb, and Robert A. Montgomery. Kidney paired donation and optimizing the use of live donor organs. *Journal of American Medical Association*, 293/15:1883–1890, 2005. (Cited on page 22.)

[SS74] Lloyd Shapley and Herbert E. Scarf. On cores and indivisibility. *Journal of Mathematical Economics*, 1:23–37, 1974. (Cited on page 4.)