Adapting Gurvich-Karzanov-Khachiyan's Algorithm for Parity Games

Implementation and Experimentation

Emmanuel Beffara

Project conducted at the Information Technology Department
in Uppsala University (Sweden) in summer 2001,
with Sergei Vorobyov

Adaptation de l'algorithme de Gurvich-Karzanov-Khachiyan aux jeux de parité

Implantation et expérimentation

[présentation succinte en français]

Emmanuel Beffara

Stage effectué au Département de Technologies de l'Information
de l'Université d'Uppsala (Suède) en été 2001,
sous l'encadrement de M. Sergei Vorobyov

Depuis sa fondation par von Neumann et Morgenstern en 1944, puis les travaux de Nash (prix Nobel d'économie en 1994) en 1950, la théorie des jeux est devenue un domaine classique et largement étudié, dépassant le cadre initial de l'économie, avec de nombreuses applications en mathématiques, en théorie des modèles, en logique, ainsi qu'en informatique théorique. Malgré l'existence de nombreux résultats dans la littérature, les retombées en complexité sont moins connues, et de nombreux problèmes importants restent ouverts.

La théorie des jeux infinis, adressée dans ce travail, est un champ d'étude important, car de tels jeux permettent de modéliser de nombreux problèmes de vérification de systèmes réactifs. Dans un tel formalisme, prouver la correction d'un système se traduit par la recherche de stratégies gagnantes pour l'un des joueurs. Par conséquent, il est crucial de disposer d'algorithmes efficaces pour déterminer l'existence de telles stratégies et pour les construire. Les *jeux de parité*, étudiés ici, sont considérés comme parmi les plus fondamentaux car ils sont équivalent aux problèmes de *model-checking* pour le μ-calcul, langage expressif permettant d'unifier toutes les logiques temporelles utilisées en vérification de programmes. La complexité exacte de ce problème reste une question ouverte posée en 1984 par Emerson, et elle est considérée comme « l'un des problèmes concrets les plus importants concernant la frontière de PTime » par Papadimitriou[16]. À ce jour, les meilleurs algorithmes sont exponentiels, ce qui les rend impraticables dans le cas général, ne permettant pas de vérifier des systèmes de grande taille. On conjecture cependant que la complexité de ce problème est polynomiale, ce qui motive la recherche de nouveaux algorithmes.

Fondements théoriques

On définit les jeux de parité de la façon suivante : deux joueurs, Alice et Bob, déplacent un jeton à tour de rôle sur les arêtes d'un graphe fini orienté, sans puits, où un rang entier est attribué à chaque sommet. Alice tente de faire en sorte que le plus grand des rangs qui apparaissent infiniment souvent au cours de la partie soit pair, et Bob essaie de le rendre impair. Il est connu que, pour chaque position initiale, l'un des deux joueurs possède une stratégie gagnante dite *stationnaire* (ou encore *sans mémoire*), c'est-à-dire qu'il peut choisir chaque coup sans tenir compte de l'histoire du jeu.

Plusieurs approches existent pour déterminer de telles stratégies, fondées par exemple sur la réduction vers le μ-calcul et la recherche de points fixes (Emerson, Clarke, Seidl) ou sur l'amélioration itérative de stratégies discrètes (Jurdziński), entre autres. Ce travail a pour but principal l'étude d'une méthode dérivée de la programmation linéaire, et moins connue du milieu du *model-checking*. En 1988, Gurvich, Karzanov et Khachiyan ont suggéré un algorithme (nommé GKK dans la suite) destiné à résoudre les *jeux de paiement moyen*.

Dans un jeu de paiement moyen, Alice et Bob déplacent à tour de rôle un jeton le long des arêtes d'un graphe fini, orienté et sans puits, dans lequel chaque arête possède un *coût* entier. On étudie alors l'évolution du coût moyen des arêtes traversées au cours d'une partie, et Alice tente de maximiser ce coût moyen, tandis que Bob tente de le minimiser. Dans ce modèle, il est connu que chaque position de départ v_0 a un coût *optimal*, c'est-à-dire qu'il existe un réel c tel qu'Alice dispose de stratégies assurant que le coût moyen de toute partie soit au moins égal à c, et que Bob a des stratégies assurant qu'il soit au plus égal à c. De plus, il est connu qu'il existe toujours des stratégies optimales stationnaires pour les deux joueurs.

L'algorithme proposé par Gurvich, Karzanov et Khachiyan [7] a pour but de déterminer ces coûts optimaux ainsi que des stratégies stationnaires associées. Il se fonde sur une notion de *transformation potentielle* permettant de modifier les coûts des arêtes au sein d'un graphe sans influencer la nature des solutions. De telles transformations sont appliquées itérativement un réseau pour parvenir à une forme dite canonique, ou les stratégies gloutonnes (consistant à choisir les arêtes de poids extrême à chaque pas) sont optimales.

Il s'avère que les jeux de parité peuvent être réduits en des jeux de paiement moyen, en définissant le coût des arêtes à partir du rang des sommets. Il est alors possible d'appliquer GKK pour résoudre le problème posé initialement. Le choix de ces coûts a une influence déterminante sur la complexité de l'algorithme, ainsi que sur la taille des réseaux qu'il est possible de traiter en pratique. On utilise ici une méthode fournissant des valeurs minimales possibles (sous certaines restrictions), ce qui permet d'élargir le domaine traité par nos applications.

Implantation

Une démarche expérimentale a été adoptée ici pour l'étude du comportement de GKK pour la résolution des jeux de parité. Une version de l'algorithme, adaptée à notre cas, est écrite dans le style de « programmation lettrée » (ou *literate programming*) décrit par Knuth [12, 13] à l'aide du système CWEB. Le langage choisi pour l'implantation est C++, pour des raisons liées à l'efficacité et à la connaissance préalable du langage. Cette démarche s'est avérée judicieuse, en effet le programme a pu rapidement être fonctionnel et fiable, et des modifications ont pu y être apportées aisément pour les besoins de l'expérience.

L'approche expérimentale se fonde sur l'étude de grandes quantités de réseaux de jeu générés aléatoirement. Une partie de l'implantation concerne donc la construction de réseaux, et l'effet sur l'algorithme de différents paramètres est étudié. Ces paramètres comportent en particulier le nombre de sommets, le degré sortant maximal des sommets, et le nombre le nombre de rangs différents qui leur sont attribués.

Simultanément, différentes méthodes sont explorées pour tenter de réduire l'amplitude des coûts associés aux arêtes des réseaux, car cette amplitude a un influence cruciale sur la complexité de l'algorithme. En effet, des réseaux de plus grande taille ont pu ainsi être étudiés. D'autre part, une étude a été menée sur l'influence d'éléments aléatoires sur le comportement de GKK. L'application de transformations aléatoires sur les coûts avant l'application de l'algorithme peut mener à une amélioration, permettant d'éviter des cas difficiles.

L'intégralité du programme est décrite dans le rapport, les choix d'implantation étant plus particulièrement développés dans le cas des structures de données et de la mise en œuvre de l'algorithme.

Observations expérimentales

Un grand nombre de réseaux aléatoires a été engendré (de l'ordre de quelques millions d'instances). Le nombre de sommets varie entre 1000 et 10000, et les degrés sortants sont variables, de même que le nombre des rangs attribués aux sommets (entre 2 et 7 principalement). Différentes mesures sont effectuées, en particulier le nombre moyen et maximal d'itérations nécessaires à la résolution de chaque instance. Ces données ont été analysées et représentées graphiquement (cf. rapport), et un certain nombre d'hypothèses s'en dégagent.

Il s'avère qu'aucun cas de complexité exponentielle n'a été rencontré, au contraire le nombre moyen d'itérations nécessaires semble croître de manière sous-linéaire, tant par rapport à la taille des réseaux que par rapport au nombre de rangs. De plus, les pires cas rencontrés ont un comportement similaire au cas moyen.

L'adaptation de l'algorithme étudié s'est avérée satisfaisante, et les résultats obtenus sont prometteurs. Bien que les études théoriques laissent supposer l'existence de comportements exponentiels, tous les exemples traités sont praticables, et montrent un comportement sous-linéaire (en nombre d'itérations), ce qui reste à expliquer d'un point de vue théorique.

Cette recherche se poursuit, et de plus amples expériences sont menées et planifiées, afin de confirmer les hypothèses formulées sur le comportement de GKK pour des jeux de plus grande taille. D'autres distributions aléatoires doivent de plus être explorées. On envisage ensuite de tester l'algorithme sur des cas obtenu à partir de problèmes de *model-checking* réputés difficiles (de telles données n'étaient cependant pas disponibles lors des études présentées ici).

Des études comparatives avec d'autres algorithmes connus seraient également intéressantes, ceci dit aucune donnée n'est disponible sur ce sujet, et la réalisation de telles études nécessiterait l'implantation de ces algorithmes, ce qui n'était pas le but de ce projet.

Contents

Introduction

Since its foundation by von Neumann and Morgenstern [22] in 1944, due to contributions by Nash [15] (Nobel prize in economics 1994), and by generations of bright mathematicians, game theory has become a classical and widely investigated mathematical discipline. Its numerous applications, ranging from economics to model theory, logic and model checking, show how rich and universal its principles may be. A large body of important results on game theory can be found in the literature, however its complexity-theoretic issues are less understood, and several important problems remain open.

Infinite games are especially important, since they provide a natural model for reactive systems. We concentrate here on the specific class of *parity games*, where the main problem can be formulated as follows:

> Bob and Alice alternate moves of a token on a finite directed bi-partite graph on n vertices painted in k colors, without sinks. Bob tries to ensure that in any infinite play the maximal color visited by the token infinitely often is even, whereas Alice tries to ensure it is odd. Is there a polynomial (in n and k) algorithm to decide a winner in such a game?

This problem is known to be equivalent to the so-called *propositional μ-calculus model checking*, a fundamental problem in program verification. The modal μ-calculus is a powerful logic expressively subsuming virtually all known temporal logics of programs [5, 4]. Therefore, settling this problem represents both theoretical and practical interests.

Solving parity games means finding the so-called *Nash equilibrium* (or *optimal strategies*) for this particular kind of games, a problem whose complexity is a fundamental open question in complexity theory, as Papadimitriou [16] stressed in a recent lecture:

> "The most interesting aspect of the Nash equilibrium concept to our community is that it is a most fundamental computational problem whose complexity is wide open. Suppose that two players have finitely many strategies each. Is there a polynomial algorithm for computing a (mixed) Nash equilibrium in such a game? [...] Together with factoring, the complexity of finding a Nash equilibrium is in my opinion the most important concrete open question on the boundary of P today."

Several approaches to solving parity games were extensively studied in the literature. The first one is based on the equivalence with the μ-calculus and computing fix points of monotonic operators in lattices. This approach is represented by [2, 9, 20], see also the survey [5]. The second approach uses the iterative discrete strategy improvement algorithms [21]. Other approaches are based on randomized simplex-like subexponential algorithms [18], on interior-point optimization schemes [17], or on pseudo-boolean optimization of completely unimodal functions [1]. In this paper we tried yet another approach, less familiar and traditional in the computer-aided verification and model-checking community.

In 1988, Gurvich, Karzanov and Khachiyan suggested an algorithm [7] (later referred to as GKK) for solving the so-called *mean-payoff game* problem, a slightly more general problem than parity games. However, their solution, quite different from known fixed-point algorithms for parity games, was ignored for a long time by the model-checking community. Recent works [19] have shown that it was actually pseudo-polynomial. Moreover, though the behavior of GKK is somewhat similar to the practical behavior of the Simplex method, worst cases are much harder to find, and no example of the theoretical exponential worst-case complexity are known.

In this work, we proceed to an experimental study of the behavior of GKK when adapted to parity games. An implementation is realized in the *literate programming* paradigm due to D. Knuth [11, 12]:

> "Computer programs are fun to write, and well-written computer programs are fun to read. One of the life's greatest pleasures can be the composition of a computer program that you know will be a pleasure for other people to read and for yourself to read."

The extensive experiments on a large number (millions of games of sizes 1000 – 20.000) of randomly generated networks demonstrate that, on the average, the number of steps required to solve parity games using GKK seems sub-linear, and that the most complicated examples are generated for small mean vertex outdegrees.

Chapter 1

Theoretical Foundations

1.1 Infinite games

In this part, we first state the definition of two kinds of infinite games. Afterwards, we recall a result about the existence of optimal stationary strategies for both forms of games, and finally prove a reduction theorem.

1.1.1 Definitions

Let us first state the definition of the objects we manipulate. We consider two different kinds of infinite games, namely *parity games* and *mean-payoff games*. Several definitions exist in the literature for both species, but all are mainly equivalent. In all forms of games considered here, there are two players with antagonistic interests. We will name the players **1** and **2**. When stating definitions in which we consider one of the players with no further precision, we will use the symbol $\boldsymbol{x}$ to refer to him.

Parity games

This definition is similar to the one used by Emerson [6, 5] and Jurdziński [8, 10].

Definition 1 *Let $G = (V, E)$ be a finite directed graph without sinks, and let $\{V_{\mathbf{1}}, V_{\mathbf{2}}\}$ be a partition of its vertex set. Let p be a function from V to $\mathbf{N}$. Then the tuple $\mathcal{N} = (V_{\mathbf{1}}, V_{\mathbf{2}}, E, p)$ is called a* parity game network. *The function p is called the* priority function *of the network.*

Given such a network, the game consists in moving a token along the edges of the graph. First, an initial position v_0 is given in V. Then, at each step i, one of the players pushes the token along an edge starting at v_i (there is always at least one outgoing edge, since the graph is supposed to have no sinks). If $v_i \in V_{\mathbf{1}}$, then the edge is chosen by player **1**, otherwise it is chosen by **2**, which defines the next position v_{i+1}. This process defines an infinite sequence $\mathcal{V} = (v_i)_{i \geqslant 0}$ in V, and the definition of the game network gives an integer priority to each node on this sequence. Let us then denote by $\mathrm{Inf}(\mathcal{V})$ the set of values appearing an infinite number of times in $\mathcal{V}$:

$$\mathrm{Inf}(\mathcal{V}) = \big\{ p \mid p \in \mathbf{N}, \big|\{\, i \mid p(v_i) = p \,\}\big| = \infty \big\}$$

Then player **1** is declared to be the winner of the game if the largest integer in $\mathrm{Inf}(\mathcal{V})$ is even, and **2** is the winner if this number is odd.

Mean-payoff games

Let us now define the other model, namely mean-payoff games. Here the valuation does not act on the vertices but on the edges, and the winning condition is different. This definition is the one used by Gurvich et al. [7] and similar to the one used by Zwick and Paterson [24].

Definition 2 *Let $G = (V, E)$ be a finite directed graph without sinks, and let $\{V_{\mathbf{1}}, V_{\mathbf{2}}\}$ be a partition of its vertex set. Let c be a function from E to $\mathbf{Z}$. Then the tuple $\mathcal{N} = (V_{\mathbf{1}}, V_{\mathbf{2}}, E, c)$ is called a* mean-payoff game network*, and c is called the* cost function *of the network.*

Here again, the name of this class of games comes from the winning condition. The construction of the infinite sequence $\mathcal{V} = (v_i)_{i\geqslant 0}$ is done in the same way as for parity games. To decide the winner, we now consider the following *mean payoff* function for player **1**:

$$f_{\mathbf{1}}(c, \mathcal{V}) = \liminf_{n\to\infty} \frac{1}{n} \sum_{i=0}^{n-1} c(v_i, v_{i+1})$$

and $f_{\mathbf{2}}$ is defined the same way for player **2** using an upper limit. Then $f_{\mathbf{1}}(c, \mathcal{V})$ is considered to be the reward of the game for player **1**, and $-f_{\mathbf{2}}(c, \mathcal{V})$ to be the reward for player **2**, so **1** wants to maximize the mean cost while **2** wants to minimize it.

In order to get a definition of a winner for the game, we may then assume that **1** is declared to be the winner if $f_{\mathbf{1}}(c, v) > 0$, and **2** is declared to be the winner if $f_{\mathbf{1}}(c, v) \leqslant 0$. With this definition, the game seems unfair because the winning condition is not symmetric, but we will see later that in the context we consider, it has no influence, because we construct games in a way the value 0 cannot be obtained no mater how the players move.

1.1.2 Known results

The main problem we consider, in both kinds of games, is the following. Given a game network and an initial position, we want to know if one of the players has a *winning strategy*, i.e. , a way to choose his moves so that he is assured to win the game.

Stationary strategies

The first thing we have to do in order to solve this problem is to formalize it. Many definitions of strategies exist, with different notions of memory in particular [23], but we will see that in our case, we can restrict ourselves, without loss of generality, to the so-called *stationary strategies*, also known as *memoryless* strategies.

Definition 3 *Let $\mathcal{N}$ be a game network (either for a parity game or a mean-payoff game) and $G = (V_{\mathbf{1}} \cup V_{\mathbf{2}}, E)$ be the underlying graph. A stationary strategy for player $\boldsymbol{x}$ is any function s associating, to each vertex v in $V_{\boldsymbol{x}}$, a vertex v' in the set $V(v)$ of successors of v.*

Subsequently, if we consider a game network $\mathcal{N}$ and a stationary strategy s for a given player $\boldsymbol{x}$, a play $(v_i)_{i\in\mathbf{N}}$ is said to be consistent *with strategy s if, for each step i such that v_i is in $V_{\boldsymbol{x}}$, we have $v_{i+1} = s(v_i)$.*

Optimal strategies

One important known result is that, for both kinds of games, there always exists an optimal stationary strategy for both players. The meaning of this statement is not obvious, especially for parity games, so let us explain it more precisely.

When considering mean-payoff games, this means that for each starting position, there exists a constant p such that player **1** can force the game to have a payoff at least p and player **2** can force it to have a payoff at most p:

Theorem 1 *Let $\mathcal{N} = (V_1, V_2, E, c)$ be a mean-payoff game network. There exists a function p from V to $\mathbf{R}$ and a couple (s_1, s_2) of stationary strategies for $\mathbf{1}$ and $\mathbf{2}$ respectively such that for every play $\mathcal{V} = (v_i)_{i \in \mathbf{N}}$ in $\mathcal{N}$:*

- *if $\mathcal{V}$ is consistent with s_1, then $f_1(c, v) \geqslant p(v_0)$*
- *if $\mathcal{V}$ is consistent with s_2, then $f_2(c, v) \leqslant p(v_0)$*

Note that this gives a guarantee for one of the players without requiring the other one to play according to a stationary strategy.

The first non-constructive proof of this theorem (not included here) was published in 1976 by Moulin [14], and another one by Ehrenfeucht and Mycielski in 1979 [3]. However, the work we present here is based mainly on the constructive proof by Gurvich, Karzanov and Khachiyan [7] and the algorithm they provide is described in Section 1.2.

When considering parity games, the notion of optimal strategy is different. Actually, when starting from a given position, a given strategy may only be winning or not. A given strategy is said to be optimal if it is winning for every starting position for which a winning strategy exists, as clarified by the following:

Theorem 2 *Let $\mathcal{N} = (V_1, V_2, E, p)$ be a parity game network. For every starting vertex v, either $\mathbf{1}$ has a winning stationary strategy, or $\mathbf{2}$ has a winning stationary strategy from v.*

Moreover, there exists a partition of the vertex set $V_1 \cup V_2$ into winning sets W_1 and W_2 and stationary strategies s_1 and s_2 such that any play starting in W_1 and consistent with s_1 is winning for $\mathbf{1}$ and any play starting in W_2 and consistent with s_2 is winning for $\mathbf{2}$.

Once again, this always guarantees that one of the players can win by following his stationary strategy, while making no assumption on the way the opponent plays. This theorem will be proved as a consequence of Theorem 1 using the reduction of Section 1.1.3.

Now we know that for both models, there always exists optimal stationary strategies (more rigorously, we will show in the next section that for parity games this fact is implied by Theorem 1). From now on, we can therefore consider that both players always conform to stationary strategies. As a consequence, every play is eventually periodic, and the period is always a simple cycle. Another consequence is that with this restriction, the payoff functions f_1 and f_2 are now equal, and they can be written as

$$f_1(c, \mathcal{V}) = f_2(c, \mathcal{V}) = \lim_{n \to \infty} \sum_{k=0}^{n-1} c(v_k, v_{k+1})$$

since now this limit is always defined. The only unfairness remaining in the definition of the winner in mean-payoff games is then the case where the limit is zero (and even this case will vanish in the next section).

1.1.3 Reduction

Our purpose now is to demonstrate that parity games can be reduced to mean-payoff games, and then to study how known algorithms related to mean-payoff games [7, 19], and especially the constructive proof of Theorem 1, adapt to the resulting networks. There is a usual transformation which keeps the topology of the network and initially assigns costs to edges depending on the priority of their source vertex [8], and the method we use here is derived from it.

Let us consider a parity game network $\mathcal{N} = (V_1, V_2, E, p)$, with p taking its values in $\{0, \ldots, d\}$. We define the mean-payoff game network $\tilde{\mathcal{N}} = (V_1, V_2, E, c)$ by defining for all $(u, v) \in E$ the edge cost $c(u, v) = w_{p(u)}$ for some sequence of weights (w_i) growing in absolute value, to be defined later. The idea is to define the sequence (w_i) in such a way that for any (eventually cyclic) play $\mathcal{V}$, the sign of the mean payoff is determined only by the largest w_i on the period.

Considering an eventually cyclical path $\mathcal{V} = (v_i)_{i\geqslant 0}$ in the game network (as obtained if both players conform to stationary strategies), the mean cost of the edges along the path is

$$f(c, \mathcal{V}) = \lim_{n\to\infty} \sum_{k=0}^{n-1} c(v_k, v_{k+1}) = \frac{1}{|C|} \sum_{(u,v)\in C} c(u, v) = \frac{1}{|C|} \sum_{(u,v)\in C} w_{p(u)}.$$

In order to get a satisfactory equivalence relation, we then want $f(c, \mathcal{V})$ to be positive whenever the largest value of p on the cycle C is even, and negative otherwise. A usual method [8] consists in defining w_p to be $(-|V|)^p$ for every p, because there one edge with cost w_p dominates the sum of the costs of all edges with cost less than w_p.

However, there are obviously smaller values of (w_i) for which the reduction is valid. If we define n_i to be the number of vertices of priority i, then the validity condition is fulfilled if the following holds:

$$\begin{array}{ll} w_i \geqslant 1 - \sum_{0<2j+1<i} n_j w_j & \text{if } i \text{ is even,} \\ w_i \leqslant -1 - \sum_{0\leqslant 2j<i} n_j w_j & \text{if } i \text{ is odd,} \end{array} \tag{1.1}$$

because then the absolute value of each w_i is larger than the sum of the absolute value of all w_j of the opposite sign with $j < i$ multiplied by the number of vertices of priority j. If we define (w_i) from the formulas above using equalities, we can reduce the definition into[1]

$$w_0 = 1, \qquad w_1 = -n_0 - 1 \qquad \text{and} \qquad w_i = w_{i-2} - n_{i-1}w_{i-1} \quad \text{for all } i \geqslant 2, \tag{1.2}$$

and the resulting sequence (w_i) (resembling Fibonacci's) satisfies our condition (this is the method we use in Section 2.2 of the code).

As a consequence, $f(c, \mathcal{V})$ is never zero, and it is positive if and only if the highest value of $p(u)$ along the cycle is even. This proves that a cyclic play is winning for a given player in $\tilde{\mathcal{N}}$ if and only if it is also winning for him in $\mathcal{N}$.

The next thing to show is that winning strategies in $\tilde{\mathcal{N}}$ are actually winning strategies in $\mathcal{N}$. For this, let us consider the winning set $W_{\mathbf{1}}$ of player **1** in $\tilde{\mathcal{N}}$ and a winning strategy $s_{\mathbf{1}}$. Let $\mathcal{V}$ be a play starting in $W_{\mathbf{1}}$ and consistent with $s_{\mathbf{1}}$. Then, by definition, $\mathcal{V}$ is forced to stay inside $W_{\mathbf{1}}$. Since the sequence $\mathcal{V}$ takes its values in a finite set, it can be decomposed as a concatenation of cycles in $W_{\mathbf{1}}$. The above definition for the weights (w_i) then implies that, on every simple cycle in $W_{\mathbf{1}}$ consistent with $s_{\mathbf{1}}$, the maximum priority of vertices is even, since the cost of each simple cycle in $W_{\mathbf{1}}$ is positive. It follows that whenever some odd priority occurs infinitely often in $\mathcal{V}$, a larger even priority also occurs infinitely often, so the maximum of $\mathrm{Inf}(\mathcal{V})$ is even and **1** is the winner in $\mathcal{V}$. Therefore $s_{\mathbf{1}}$ is a winning strategy for **1** in $\mathcal{N}$.

Therefore, winning strategies in $\tilde{\mathcal{N}}$ are actually winning strategies in $\mathcal{N}$: a given player has a winning strategy in $\mathcal{N}$ starting at a given vertex if and only if he has one in $\tilde{\mathcal{N}}$ starting from the same vertex, and that the strategies are the same. The aim of defining the weights by (1.1), (1.2) is to keep them as small as possible, which is crucial for the GKK algorithm performance (both theoretically and practically, as we got convinced by experimenting).

More formally, let us consider the winning set $W_{\mathbf{1}}$ of player **1** in $\tilde{\mathcal{N}}$ and a *stationary* optimal strategy $s_{\mathbf{1}}$ guaranteed by Theorem 1 (the case of the winning set $W_{\mathbf{2}}$ of player **2** is symmetrical). Suppose, towards a contradiction, that $s_{\mathbf{1}}$ is not winning for **1** in $\mathcal{N}$ on $W_{\mathbf{1}}$. This means that **2** has a (possibly non-stationary) strategy $s_{\mathbf{2}}$ winning against $s_{\mathbf{1}}$ from some starting vertex in $W_{\mathbf{1}}$, i.e., the maximal priority p of a vertex visited infinitely often is odd, when players follow strategies $s_{\mathbf{1}}$ and $s_{\mathbf{2}}$ respectively. It follows (since the graph is finite) that some vertex v of priority p in $W_{\mathbf{1}}$ is visited infinitely often. Consider a path $\mathcal{V}$ between any two consecutive visits of v in this play such that all intermediately visited vertices are of lower or equal priority (such a path should exist). Recall that **1** follows a stationary strategy, whereas **2** does not, in general. Clearly, **2** can modify his non-stationary strategy $s_{\mathbf{2}}$ on the path $\mathcal{V}$ from v to v so as to erase all intermediate simple cycles, since their cost is positive by the choice of weights. The resulting strategy $s'_{\mathbf{2}}$ will

[1] We could have started with $w_0 = 0$, but we wish to avoid 0 value cycles.

be stationary and the resulting simple cycle $\mathcal{V}'$ in $W_{\mathbf{1}}$ will have the vertex v with maximal odd priority p. Now the pair of stationary strategies $\{s_{\mathbf{1}}, s'_{\mathbf{2}}\}$ in $\tilde{\mathcal{N}}$ forms a loop in $W_{\mathbf{1}}$ with a *negative* cost (by the choice of weights in the reduction), which is a contradiction.

1.2 The algorithm

Our point is to study an algorithm finding, for each vertex of a given parity game network, which one of the players has a winning strategy. The algorithm we use is an adaptation to our case of an algorithm [7] used for solving mean-payoff games. We will first introduce some notation used to describe it, then we will sketch the original version, and finally we will study how the latter simplifies and how the restricted version can be formulated.

1.2.1 Definitions and notation

Given a mean-payoff game network $(V_{\mathbf{1}}, V_{\mathbf{2}}, E, c)$, the *extremal* value of a vertex v in V is defined as

$$\operatorname{ext}(c, v) = \begin{cases} \max\{c(v,u) \mid (v,u) \in E\} & \text{for } v \in V_{\mathbf{1}} \\ \min\{c(v,u) \mid (v,u) \in E\} & \text{for } v \in V_{\mathbf{2}} \end{cases}$$

Extremal edges are thus edges whose value is extremal for their origin vertex, and we define the *extremal successors* of a given vertex v as

$$V_{\text{ext}}(c, v) = \big\{u \bigm| (v,u) \in E, c(v,u) = \operatorname{ext}(c,v)\big\}$$

Finally, two cost functions c and c' are said to be *equivalent* if there exists a *potential function* ε from V to $\mathbf{R}$ such that for all edge (u,v),

$$c'(u,v) = c(u,v) + \varepsilon(u) - \varepsilon(v)$$

In this case we will write $c' = c_\varepsilon$. This notion of equivalence means that, for every finite path $(v_0, v_1, \ldots, v_p)$ the cost of the path for c' is equal to the cost of c plus a term $\varepsilon(v_0) - \varepsilon(v_n)$, which is clearly bounded. Therefore, the mean payoff of every play $(v_i)_{i\in\mathbf{N}}$, when it exists, is the same for c and c'. The transformation of c into c_ε is called a *potential transformation*, and it can be shown that such transformations are the only ones keeping every mean payoff invariant.

1.2.2 The method

The algorithm constructs an optimal stationary strategy for both players by reducing the initial game network $(V_{\mathbf{1}}, V_{\mathbf{2}}, E, c)$ into a so-called *canonical form* $(V_{\mathbf{1}}, V_{\mathbf{2}}, E, c')$ with some appropriate cost function c' equivalent to c. This canonical form is defined the following way:

Definition 4 *A mean-payoff game network $(V_{\mathbf{1}}, V_{\mathbf{2}}, E, c)$ is said to be in canonical form if there exists a function p from V to $\mathbf{R}$ such that for all v in V:*

i. $\operatorname{ext}(c, v) = p(v)$,

ii. $p(u) = p(v)$ *for all* u *in* $V_{\text{ext}}(c, v)$,

iii. *if* $v \in V_{\mathbf{1}}$ *then* $p(u) \leqslant p(v)$ *for all* u *in* $V(v)$,

iv. *if* $v \in V_{\mathbf{2}}$ *then* $p(u) \geqslant p(v)$ *for all* u *in* $V(v)$.

Let us consider a play $(v_i)_{i\in\mathbf{N}}$ in such a network, and let us assume that player $\mathbf{1}$ always greedily chooses extremal edges. Then at each step i the following holds:

- if $v_i \in V_{\mathbf{1}}$ then $c(v_i, v_{i+1}) = p(v_i)$,
- if $v_i \in V_{\mathbf{2}}$ then $c(v_i, v_{i+1}) \geqslant p(v_i)$.

By definition of the canonical form, the fact that **1** chooses extremal edges shows that the value of p is increasing along the path, therefore each cost along the path is at least $p(v_0)$. As a consequence, the payoff of this game is at least $p(v_0)$. Using the same method, we can also prove that if player **2** always greedily chooses extremal edges, then the payoff of the game is at most p. Therefore, in networks in canonical form, any strategy consisting in choosing extremal edges is optimal.

Original version

This algorithm was first published by Gurvich, Karzanov and Khachiyan in 1988 [7] as a constructive proof of Theorem 1, and later reformulated and somewhat generalized by Pisaruk in 1999 [19] for a wider class of games. We first present a formulation of the algorithm here, afterwards we will see how it adapts to reductions of parity games.

Throughout the algorithm's description, we will use the following notation. Given a subset X of V, we write $X_{\mathbf{1}} = X \cap V_{\mathbf{1}}$ and similarly $X_{\mathbf{2}} = X \cap V_{\mathbf{2}}$. At any time, K, S and T are defined as follows:

$$\begin{array}{ll} K(c) = \{v \mid \mathrm{ext}(c,v) = p\} & \textit{critical} \text{ vertices,} \\ S(c) = \{v \mid \mathrm{ext}(c,v) < p\} & \textit{insufficient} \text{ vertices,} \\ T(c) = \{v \mid \mathrm{ext}(c,v) > p\} & \textit{redundant} \text{ vertices,} \end{array}$$

where p is a given threshold value between $m = \min_{v \in V} \mathrm{ext}(c,v)$ and $M = \max_{v \in V} \mathrm{ext}(c,v)$. Finally, *shifting* a cost function c by an amount δ with respect to a subset L of V means substituting c' for c where, for each edge (u,v), the cost $c'(u,v)$ is defined by

$$c'(u,v) = \begin{cases} c(u,v) + \delta & \text{if } u \in L \text{ and } v \notin L, \\ c(u,v) - \delta & \text{if } u \notin L \text{ and } v \in L, \\ c(u,v) & \text{otherwise.} \end{cases}$$

Thus c' is equivalent to c because it corresponds to the application of the potential function $\delta \cdot \chi_L$ (where χ_L naturally denotes the indicator function of L in V).

The set L above is defined as a set of vertices starting from which player **2** has a stationary strategy forcing player **1** into S, when both players use only extremal edges.

The algorithm keeps doing the above cost shifts with $\delta > 0$ while it is possible. When it stops, either 1) L is empty, or 2) $\bar{L}$ is empty, or 3) $(L, \bar{L})$ is a *regular* partition of the game network. The latter means that player **1** cannot escape from L and player **2** cannot escape from $\bar{L}$, when their opponents follow extremal edges. Morever, the player **2** guaranteed payoff in L is at most p and the player **1** guaranteed payoff in $\bar{L}$ is at least p.

In cases 1) and 2), the new threshold value p is selected in the middle of $[m, p]$ or $[p, M]$, respectively, and the algorithm iterates. In case 3), one obtains two smaller networks to which the algorithm applies recursively. The full details and correctness proofs are contained in the article [7].

Reduced version

The GKK algorithm thus provides for computing the optimal payoff for each starting position in a mean-payoff game. What we want, on the other hand, is the winner for each starting position in a parity game. Therefore, several simplifications may be made to the algorithm.

First of all, in the reduction we use of parity games, deciding the winner from a given starting position simply means finding the sign of the optimal mean payoff starting from this position. Therefore when using 0 as the threshold value p, exactly one execution of the auxiliary algorithm is needed in order to get the desired result.

After simplification, our algorithm can be formulated the following way, provided that $(V_{\mathbf{1}}, V_{\mathbf{2}}, E, c)$ is a mean-payoff game network reduced from a parity game network. It produces a potential function a such that any strategy choosing always extremal edges is an optimal strategy.

1: **for all** $v \in V$ **do** $a(v) \leftarrow 0$

while $S(c_a) \neq \emptyset$ and $T(c_a) \neq \emptyset$ **do**
  $L \leftarrow S(c_a)$
  repeat
    $X \leftarrow \{\, v \mid v \in K(c_a)_{\mathbf{1}} \setminus L, V_{\text{ext}}(c_a, v) \subseteq L \,\}$
    $Y \leftarrow \{\, v \mid v \in K(c_a)_{\mathbf{2}} \setminus L, V_{\text{ext}}(c_a, v) \cap L \neq \emptyset \,\}$
    $L \leftarrow L \cup X \cup Y$
  until $X = Y = \emptyset$
  $E_1 \leftarrow \{\, (u,v) \mid (u,v) \in E, u \in L_{\mathbf{1}}, v \in \bar{L} \,\}$
  $E_2 \leftarrow \{\, (u,v) \mid (u,v) \in E, u \in \bar{L}_{\mathbf{2}}, v \in L \,\}$
  $\hat{S} \leftarrow \{\, v \mid v \in S(c_a)_{\mathbf{2}}, V_{\text{ext}}(c_a, v) \subseteq \bar{L} \,\}$
  $\hat{T} \leftarrow \{\, v \mid v \in T(c_a)_{\mathbf{1}}, V_{\text{ext}}(c_a, v) \subseteq L \,\}$
  if $E_1 = E_2 = \emptyset$ and $\hat{S} = \hat{T} = \emptyset$ **then return** a
  $\delta \leftarrow \min\Big(\big\{|c_a(e)| \mid e \in E_1 \cup E_2\big\} \cup \big\{|\operatorname{ext}(c_a, v)| \mid v \in \hat{S} \cup \hat{T}\big\}\Big)$
  for all $v \in L$ **do** $a(v) \leftarrow a(v) + \delta$
end while
return a

Let us now prove the correctness of this algorithm. Two exit conditions represent different situations:

- If $S(c_a)$ is empty then the extremal value of each vertex is non-negative, therefore all edges leaving vertices that belong to **2** have a non-negative cost. Consequently, if **1** always chooses extremal edges, every game has a positive payoff. In this case **1** is always the winner. If $T(c_a)$ is empty, then the situation is symmetric and **2** is always the winner.

- If at some point the sets E_1, E_2, $\hat{S}$ and $\hat{T}$ are all empty, then starting in L, **2** can guarantee that the game never exits L, and that starting in $\bar{L}$ **1** can force the game to stay in $\bar{L}$. For the same reason as in the previous cases, if the game stays in L, then **2** is the winner, and if it stays in $\bar{L}$, then **1** is the winner. Therefore, this provides optimal strategies for both players.

So whenever the algorithm terminates, the obtained potential a provides the expected result.

To demonstrate that the algorithm always terminates Pisaruk [19] introduced the function Φ defined as

$$\Phi(a) = \sum_{v \in V} \min\Big(\|a\|_\infty - a(v),\ \big(|V|-1\big)\|c\|_\infty\Big),$$

where $\|f\|_\infty$ denotes the maximum absolute value of the function f. He demonstrated that the value of this function Φ is monotonically increasing, as the algorithm proceeds, and is bounded.

Remarking that at least one element of this sum is always equal to zero (since $\|a\|_\infty$ is reached by a in at least one point), we get the bound

$$\Phi(a) \leqslant \big(|V|-1\big)^2 \|c\|_\infty$$

on Φ, and subsequently on the number of iterations. Considering the reduction we use, this bounds the number of iterations to the order n^{2+k} if n is the number of vertices and k the number of colors, which is thus a pseudo-polynomial worst case bound. However, experimenting on reductions of parity games shows much better results on the average, as we will see later.

Chapter 2

Implementation

A large part of the work consisted in experimenting with this algorithm. For this purpose, an implementation was made in order to study its behavior, and various kinds of input data were tested. This lead to the observation of several interesting properties, not all of which we managed to prove.

2.1 The implementation

The algorithm was implemented in the reduced version, in C++ using the literate-programming tool CWEB [13]. This document is actually the "weaved" version of the program.[1]

The choice was made to use a procedural language like C++ rather than, for instance, functional languages such as SML, because the algorithm is strongly imperative, as are all algorithms manipulating graphs, and because it is much easier to represent and manipulate graphs in an imperative way. C++ was preferred to other imperative languages because of prior knowledge of the language and because of the existence of CWEB, which allows writing clean and well-documented programs with C and C++. The Standard Template Library STL (`http://www.sgi.com/tech/stl/`) was used extensively.

In this section we present the main points of the implementation of the algorithm. Implementation details of less importance can be found in Section A.

2.1.1 Data structures

We start by defining the data structures needed to represent game networks. We first define a generic class *graph* to represent directed graphs, and then include data in the vertices and in the edges to make a game network out of it.

Graph manipulation

Representing graphs in any programming language is always a difficult problem because there is no universally efficient data structure for it. In the case of game networks, we will assume that the graphs usually have a large number n of vertices but a small (compared to n^2) number of edges. Therefore our representation will consist of an array containing the vertices, and each vertex will contain the list of its successors (this is the so-called *incidence list* representation).

19 ⟨ Classes for game networks 19 ⟩ ≡
  struct graph {
    ⟨ Definition of **vertex** 20 ⟩

[1] Some knowledge of CWEB may be useful to understand in full detail some elements of the presentation of this document. However, readers unfamiliar with it should not be disturbed too much by the code fragments.

```
    vector⟨vertex⟩ V;
    int size;
    ⟨ Additional data for graph 29 ⟩
    graph(int the_size) { V.resize(size = the_size); }
    ⟨ Methods for graph 30 ⟩
  };
```
This code is used in chunk 76.

A vertex will be identified by its index in the main vector, and as stated above, it will contain the list of all edges leaving it. We also use a **vector** for this list for efficiency reasons. The only topological information an edge contains is thus its destination vertex, and we use an integer to represent it. In both cases, we defer the declaration of non-topological data for later, because the need for such data arises when defining the algorithms themselves.

20 ⟨ Definition of **vertex** 20 ⟩ ≡
```
  struct edge {
    int dest;
    ⟨ Data contained in each edge 24 ⟩
    ⟨ Methods for edge 25 ⟩
  };
  struct vertex {
    vector⟨edge⟩ E;
    int degree;
    ⟨ Data contained in each vertex 27 ⟩
    vertex( ) { degree = 0; ⟨ Initialize data in a vertex 33 ⟩ }
    ⟨ Methods for vertex 28 ⟩
  };
```
This code is used in chunk 19.

For readability, we provide generic macros for browsing the vertices in a graph and for browsing the outgoing edges in a vertex, in the form of a **for** loop. They are meant to be used in methods of the **graph** class and of the **vertex** class respectively. These macros include the declaration of their index variable, because most of the time this variable is not used elsewhere.

21
```
#define for_vertex(i)  for (int i = 0; i < size; i++)
#define for_edge(i)  for (int i = 0; i < degree; i++)
```

Game networks

Now that we have a topological representation of graphs, we want to include material in the structures so that they represent game networks.

In mean-payoff game networks, all edges have a cost. We introduce a type for these costs, so it can be easily modified to another type. This may be used, for instance, with a type of integers with arbitrary precision, in order to study cases where large costs appear.

23 ⟨ Auxiliary data types 23 ⟩ ≡
```
  typedef long long cost_type;      /* warning: this type is a GNU extension */
```
See also chunks 26, 31, 35, and 60.
This code is used in chunk 76.

24 ⟨ Data contained in each edge 24 ⟩ ≡ `cost_type cost;`
This code is used in chunk 20.

The constructor for edges thus takes an integer (for the destination) and a **cost_type** (for the cost) as its arguments.

25 ⟨ Methods for **edge** 25 ⟩ ≡

edge(**int** *dest_vertex*, **cost_type** *initial_cost*)
{
  dest = *dest_vertex*;
  cost = *initial_cost*;
}

See also chunk 101.
This code is used in chunk 20.

Every vertex belongs to a specific player. The players will be named **1** and **2** here (their actual names in the code is *White* and *Black* respectively). For type safety, we define them as the two members of an enumerate type.

26 **format** *Black* *TeX*
format *White* *TeX*

⟨ Auxiliary data types 23 ⟩ +≡ **enum player** { **1**, **2** };

We thus include a field *who* in the **vertex** class to represent the owner of each vertex, and we do not initialize it, because anyway the network has to be constructed after the default constructor is called. The algorithm also introduces a function named ext, which associates to every vertex the extremal cost of edges leaving it, depending on the owner of this vertex. For **1**, it is the maximum, and it is the minimum for **2**. In the implementation we just represent this using another field in every vertex.

27 ⟨ Data contained in each vertex 27 ⟩ ≡

player *who*;
cost_type *ext*;

See also chunks 32 and 59.
This code is used in chunk 20.

We provide a function *update_ext* to compute this value. This function assumes that the vertex has at least one outgoing edge, because the algorithms assumes it also. In this method, it will be useful to perform a validity check on the new value of *ext* during the algorithm, so we introduce it.

28 ⟨ Methods for **vertex** 28 ⟩ ≡

cost_type *update_ext*( )
{
  cost_type *new_ext* = $E[0].cost$;
  if ($who \equiv$ **1**) **for_edge** (i) $new\_ext = max(new\_ext, E[i].cost)$;
  else for_edge (i) $new\_ext = min(new\_ext, E[i].cost)$;
  ⟨ Check the validity of the new value of *ext* 109 ⟩
  return *ext* = *new_ext*;
}

See also chunks 43, 49, 50, 53, 58, 62, 72, 73, and 100.
This code is used in chunk 20.

For a given graph, the algorithm defines m and M to be, respectively, the lowest and highest value of this function over the graph. We also include these as variables in the graph object, as well as the threshold value p used by the algorithm.

29 ⟨ Additional data for **graph** 29 ⟩ ≡ **cost_type** m, M, p;

See also chunks 55 and 57.
This code is used in chunk 19.

It is also useful to have a method for graph objects that applies this to every vertex. This function also updates the variables m and M that hold the minimum and the maximum of ext over all vertices.

30 ⟨Methods for **graph** 30⟩ ≡

```
void update_ext()
{
  for_vertex (i) {
    cost_type ext = V[i].update_ext();
    if (i ≡ 0) m = M = ext;
    else if (ext < m) m = ext;
    else if (ext > M) M = ext;
  }
}
```

See also chunks 34, 36, 56, 61, 71, 99, and 107.
This code is used in chunk 19.

2.1.2 The main algorithm

The algorithm introduces various sets of vertices. First, a partition is made, by comparing the *ext* value to the threshold p, into three sets S, K and T. Moreover, a set L is constructed in steps, using two subsets X and Y of K. We thus introduce two enumerate types and two fields in the **vertex** class. We use only one enumerate value for X and Y because these sets do not need to be distinguished.

The partition in S, K and T is done using a field of the enumerated type **part**, and we also introduce a fourth value *Undef* for debugging purposes. This is the value the field gets on creation of the vertices.

31 ⟨Auxiliary data types 23⟩ +≡

```
enum part { S, K, T, Undef };
enum status { L, XY, L̄ };
```

Each vertex will thus contain a field *pt* indicating in which part it belongs (S, K or T) and a field *st* representing its status with respect to L. Since the algorithm is based on the notion of potential transformation, we also include a field *potential* that will accumulate all the transformations so far.

32 ⟨Data contained in each vertex 27⟩ +≡

```
part pt;
status st;
cost_type potential;
```

For the purpose of validity checking, the *pt* field has to be initialized to the special value *Undef*. The potential has to be initialized to zero.

33 ⟨Initialize data in a vertex 33⟩ ≡

```
pt = Undef;
potential = 0;
```

This code is used in chunk 20.

The following method resets the value of *pt* to *Undef* and the value of *potential* to 0 for every vertex, in order to make several tries with one game network.

34 ⟨Methods for **graph** 30⟩ +≡

```
void clear()
{
  for_vertex (i) {
    V[i].pt = Undef;
    V[i].potential = 0;
  }
}
```

The algorithm may end for two different reasons: Either S or T happens to get empty after an iteration, in which case a dichotomy of the interval in which extremal costs lie is done, or a regular partitioning of the vertices is found. Since the behavior of the main algorithm depends on why the auxiliary one ended, we let our function return a value to represent this.

35 ⟨ Auxiliary data types 23 ⟩ +≡ **enum issue** { *S_or_T_empty*, *Partition* };

Now we can write the outline of the auxiliary algorithm. We call the associated function *reduce*. The algorithm is explained in the report. In the original form, the process is supposed to return a new cost function, but here we will simply make the transformation in place. Another modification of the original algorithm is that the threshold p used do decide whether vertices are insufficient, redundant or critical, is now a parameter, which does not have to be $(m + M)/2$. Actually, for parity games this parameter is always zero.

36 ⟨ Methods for **graph** 30 ⟩ +≡
```
issue reduce(const cost_type threshold)
{
  reduce_calls++;
  ⟨ Variables for reduce 37 ⟩
  ⟨ Compute the sets S, K and T 38 ⟩
  while (⟨ Neither S nor T is empty 39 ⟩) {
    ⟨ Display the sizes 105 ⟩
    reduce_iterations++;
    ⟨ Define L as equal to S 40 ⟩
    do {     /* loop to compute L */
      label_iterations++;
      ⟨ Compute X and Y 42 ⟩
      ⟨ Include X and Y into L 46 ⟩
    } while (⟨ X or Y is not empty 47 ⟩);
    ⟨ Dump the sets on the standard output 106 ⟩
    ⟨ Identify E1, E2, Ŝ and T̂ and compute δ 48 ⟩
    if (⟨ All four sets are empty 51 ⟩) {
      ⟨ Update m and M 54 ⟩
      return Partition;
    }
    ⟨ Shift the costs by δ with respect to L 52 ⟩
    ⟨ Compute the sets S, K and T 38 ⟩
  }
  ⟨ Dump the sets on the standard output 106 ⟩
  ⟨ Update m and M 54 ⟩
  return S_or_T_empty;
}
```

Computing the three parts of the graph implies knowing the bounds m and M first, which is assumed to be done already using *update_ext*, then defining the threshold p, and then browsing the set of states. In this modified version, the threshold may be imposed, so we it is not a variable of the algorithm anymore.

In order to check the number of elements of S and T, we also introduce a counter for each, so the test is easy to do.

37 ⟨ Variables for *reduce* 37 ⟩ ≡ **int** *size_of_S*, *size_of_T*;
See also chunks 41 and 103.
This code is used in chunk 36.

So the first computation of the partition of the vertex set V also includes the initialization of m, M (which are computed when evaluating ext) and p.

38 ⟨ Compute the sets S, K and T 38 ⟩ ≡
$p =$ *threshold*;

$size\_of\_S = size\_of\_T = 0;$
for_vertex (i) {
  if $(V[i].ext < p)$ { $V[i].pt = S;$ $size\_of\_S$++; }
  else if $(V[i].ext > p)$ { $V[i].pt = T;$ $size\_of\_T$++; }
  else { $V[i].pt = K;$ }
}
This code is used in chunk 36.

Therefore, testing whether S or T is empty means only comparing their sizes with zero.

39 ⟨Neither S nor T is empty 39⟩ ≡ $(size\_of\_S > 0) \wedge (size\_of\_T > 0)$
This code is used in chunk 36.

Defining the initial value of the set L as the set S could have been done during the first computation of the partition, but we have to include it inside the loop because L cannot be computed on-the-fly when costs are updated at the end of the loop.

40 ⟨Define L as equal to S 40⟩ ≡
for_vertex (i)
  if $(V[i].pt \equiv S)$ $V[i].st = L;$
  else $V[i].st = \bar{L};$
This code is used in chunk 36.

The sets X and Y are distinct in the algorithm because their meaning is opposed, but both are added to L after each iteration of the inner loop, and only the emptiness of their reunion is checked, so we do not need to differentiate them here. For checking their emptiness, we use the same technique as for S and T.

41 ⟨Variables for *reduce* 37⟩ +≡ **int** $size\_of\_XY;$

42 ⟨Compute X and Y 42⟩ ≡
$size\_of\_XY = 0;$
for_vertex (i) {
  if $((V[i].pt \neq K) \vee (V[i].st \equiv L))$ **continue**;
  if $(V[i].who \equiv \mathbf{1})$ { ⟨Check if all extremal edges in $V[i]$ lead to L 44⟩ }
  else { ⟨Check if an extremal edge in $V[i]$ leads to L 45⟩ }
}
This code is used in chunk 36.

In order to check for the conditions required to belong to X or Y, we have to browse the edges of a given vertex. This deserves new methods in the **vertex** class to be defined. However, because of our representation of edges, these methods must have knowledge of the whole graph in order to check if the destination vertices belong to L or not.

43 ⟨Methods for **vertex** 28⟩ +≡
bool *vext_in*(**const graph** &G, **status** *stat*) **const**
{
  for_edge (i) **if** $((E[i].cost \equiv ext) \wedge (G.V[E[i].dest].st \neq stat))$ **return** *false*;
  return *true*;
}
bool *vext_hits*(**const graph** &G, **status** *stat*) **const**
{
  for_edge (i) **if** $((E[i].cost \equiv ext) \wedge (G.V[E[i].dest].st \equiv stat))$ **return** *true*;
  return *false*;
}

Using these methods, the conditions are easily expressed, and they just have to update the field *st* in the vertex and to increment the counter.

44 ⟨Check if all extremal edges in $V[i]$ lead to L 44⟩ ≡
if $(V[i].vext\_in(*\mathbf{this}, L))$ { $V[i].st = XY;$ $size\_of\_XY$++; }
This code is used in chunk 42.

45 ⟨Check if an extremal edge in $V[i]$ leads to L 45⟩ ≡

```
if (V[i].vext_hits(*this, L)) { V[i].st = XY; size_of_XY++; }
```

This code is used in chunk 42.

After these two sets are computed and their size is known, we can include them into L if they are not empty, before entering the next iteration of the construction of L.

46 ⟨Include X and Y into L 46⟩ ≡

```
if (size_of_XY > 0) for_vertex (i) if (V[i].st ≡ XY) V[i].st = L;
```

This code is used in chunk 36.

The loop condition is trivial.

47 ⟨X or Y is not empty 47⟩ ≡ *size_of_XY* > 0

This code is used in chunk 36.

The next thing to do is to identify the four sets defined in the paper [7] (see Section 1.2.2). This includes the set E_1 of edges allowing **1** to leave L and the set E_2 of edges allowing **2** to enter L. This also includes the set $\hat{S}$ of insufficient edges belonging to **2** for which all extremal edges lead to $\bar{L}$ and the set $\hat{T}$ of redundant edges belonging to **1** for which all extremal edges lead to L. At the same time, we want to compute the distance from the cost of these edges (or the extremal cost of the vertices) and get the minimum of all of them. This will also use additional methods in the **vertex** class. Moreover, we define the boolean *found* to remember if one of these sets was not empty.

48 **format** *delta* *TeX*

⟨Identify E_1, E_2, $\hat{S}$ and $\hat{T}$ and compute δ 48⟩ ≡

```
cost_type δ = max(M − p, p − m);      /* this cannot be exceeded */
bool found = false;
for_vertex (i)
  if (V[i].who ≡ 1) {
    if (V[i].st ≡ L) found = V[i].edges_out_of_L(*this, p, δ) ? true : found;
    if ((V[i].pt ≡ T) ∧ (V[i].vext_in(*this, L))) {
      δ = min(δ, V[i].ext − p);
      found = true;
    }
  } else {
    if (V[i].st ≡ L̄) found = V[i].edges_into_L(*this, p, δ) ? true : found;
    if ((V[i].pt ≡ S) ∧ (V[i].vext_in(*this, L̄))) {
      δ = min(δ, p − V[i].ext);
      found = true;
    }
  }
```

This code is used in chunk 36.

The method *edges_out_of_L* browses the edges leaving the vertex, and if an edge e of them leads out of L, then δ is compared to $p - c(e)$ and the minimum is kept as δ. That is why δ is passed to the method as a reference. The methods returns a boolean indicating if some appropriate edge was found.

49 ⟨Methods for **vertex** 28⟩ +≡

```
bool edges_out_of_L(const graph &G, cost_type p, cost_type &δ)
{
  bool found = false;
  for_edge (i)
    if (G.V[E[i].dest].st ≡ L̄) {
      δ = min(δ, p − E[i].cost);
      found = true;
    }
```

```
    return found;
  }
```

The complementary method *edges_into_L* does the same, excepts that it checks whether the edges lead into L and that the value they compare to δ is the opposite of the previous one.

50 ⟨Methods for **vertex** 28⟩ +≡

```
  bool edges_into_L(const graph &G, cost_type p, cost_type &δ)
  {
    bool found = false;
    for_edge (i)
      if (G.V[E[i].dest].st ≡ L) {
        δ = min(δ, E[i].cost − p);
        found = true;
      }
    return found;
  }
```

The exit condition here is simply the negation of the variable *found* that indicates if an element has been found in one of the four sets.

51 ⟨All four sets are empty 51⟩ ≡ ¬*found*

This code is used in chunk 36.

Finally, shifting the costs (potential transformation) is a simple browsing of all edges in order to increase by δ the costs of all edges leaving L and to decrease by δ the costs of those entering L. We use yet another method of vertices for this purpose. This process also updates the value of *ext* for each vertex, and it updates the partition accordingly. The value of δ is displayed here if necessary, and the number of modified *ext* values is counted.

52 ⟨Shift the costs by δ with respect to L 52⟩ ≡

```
  if (verb_sets) cout ≪ "␣␣␣delta␣=␣" ≪ δ ≪ endl;
  else if (verb_delta) { cout ≪ "␣" ≪ δ; cout.flush(); }
  modified_ext = 0;
  for_vertex (i) V[i].shift(*this, δ);
  if (verb_delta ∧ modified_ext ≡ 0) cout ≪ vt.red ≪ "␣EXT" ≪ vt.reset;
```

This code is used in chunk 36.

The method for one vertex browses the list of edges and modifies the cost. It only computes the new value of *ext* if at least one edge was modified. Moreover, this method updates the value of th potential of the vertices.

53 ⟨Methods for **vertex** 28⟩ +≡

```
  void shift(const graph &G, cost_type δ)
  {
    bool modif = false;
    if (st ≡ L) {
      potential += δ;
      for_edge (i)
        if (G.V[E[i].dest].st ≡ L̄) {
          E[i].cost += δ;
          modif = true;
        }
    } else {
      for_edge (i)
        if (G.V[E[i].dest].st ≡ L) {
          E[i].cost −= δ;
          modif = true;
        }
```

```
    }
    if (modif) update_ext();
}
```

Before returning from the algorithm, we end by computing the value of m and M, which were not updated, though *ext* has been modified for many vertices.

54 ⟨Update m and M 54⟩ ≡

```
m = M = V[0].ext;
for_vertex (i) {
    m = min(m, V[i].ext);
    M = max(M, V[i].ext);
}
```

This code is used in chunk 36.

2.2 Random generation of networks

To test the algorithm on a considerable number of large problem instances (game networks)[2], we provide a form of random generation of game networks with different parameters (maximum outdegrees and number of colors). In this generation, we select uniformly at random:

1. the owner of each vertex,
2. its outdegree (between one and a specified upper bound, or equal to a specific value),
3. its successors,
4. its priority (between zero and a specified upper bound).

Other natural methods, and distributions, could be easily incorporated, and may be investigated in further works.

We only generate game networks that are reductions of parity game networks. This means that all vertices have a given priority and that all edges leaving a given edge receive the *same* initial cost, depending on this priority. Different methods are possible for choosing these costs, so we implement various ways of doing it. To compute these costs, we may have to know all priorities in advance, so we have to proceed in two steps. I order to allow comparison between different methods, we implement each step as a separate function, so randomization may be done once and costs chosen several times differently.

55 ⟨Additional data for **graph** 29⟩ +≡

```
vector⟨int⟩ frequency;      /* the frequency of each priority */
vector⟨cost_type⟩ weight;      /* the weights used for reduction */
```

The *randomize* function creates edges and randomizes priorities. It is always used on newly created graphs, so once again we assume that all vertices in the table belong to the object (and thus $inf \equiv 0$).

56 ⟨Methods for **graph** 30⟩ +≡

```
void randomize(int max_edges, int max_prio, bool fixed_degree)
{
    for_vertex (i) V[i].randomize(max_edges, max_prio, size, fixed_degree);
    max_priority = max_prio;
}
```

57 ⟨Additional data for **graph** 29⟩ +≡ `int max_priority;`

Once again, the actual work is done by methods of the **vertex** class. The function *rand* is used to provide the basic random numbers. The role of the *randomize* method is to decide randomly the owner of a vertex and to create edges (with zero as their initial cost).

[2]In our experiments, we manipulate about 10^6 games of size of order 10^4.

58 ```
#define RAND(m) ((int)((double)(m) * rand()/(RAND_MAX + 1.0)))
⟨Methods for vertex 28⟩ +≡
  void randomize(int max_edges, int max_prio, int net_size, bool fixed_degree)
  {
    if (rand() > (RAND_MAX/2)) who = 1;
    else who = 2;
    prio = RAND(max_prio);
    if (fixed_degree) degree = max_edges; else degree = RAND(max_edges) + 1;
    for (int i = 0; i < degree; i++) E.push_back(edge(RAND(net_size), 0));
  }
```

59 ```
⟨Data contained in each vertex 27⟩ +≡ int prio;
```

So we provide different methods for choosing the costs in the network according to the priorities. We first introduce a new enumerated type to represent the methods.

60 ```
⟨Auxiliary data types 23⟩ +≡
  enum cost_method {
    Exponential,   /* if the priority is k, the costs are (−n)^k */
    Smallest   /* compute the smallest valid value for each color */
  };
```

Then we provide a method in the **graph** class to initialize the costs with one of these methods.

61 ```
⟨Methods for graph 30⟩ +≡
  void set_initial_costs(cost_method method)
  {
    ⟨Compute the frequency of each priority 63⟩
    weight.resize(max_priority);
    switch (method) {
    case Exponential: ⟨Compute weights using the Exponential method 64⟩ break;
    case Smallest: ⟨Compute weights using the Smallest method 65⟩
    }
    ⟨Display the weights if required 102⟩
    reset_costs();
  }
  void reset_costs() { for_vertex (i) V[i].set_all_costs(weight[V[i].prio]); }
```

62 ```
⟨Methods for vertex 28⟩ +≡
  void set_all_costs(cost_type value) { for_edge (i) E[i].cost = value; }
```

When computing the frequency of each pritority, we first initialize the table named *frequency* and then we browse the vertices.

63 ```
⟨Compute the frequency of each priority 63⟩ ≡
  frequency.resize(max_priority, 0);
  for (int i = 0; i < max_priority; i++) frequency[i] = 0;
  for_vertex (i) frequency[V[i].prio]++;
```
This code is used in chunk 61.

For the *Exponential* method, the computation is fairly easy.

64 ```
⟨Compute weights using the Exponential method 64⟩ ≡
  weight[0] = 1;
  for (int i = 1; i < max_priority; i++) weight[i] = −size * weight[i − 1];
```
This code is used in chunk 61.

For the *Smallest* method, the process is not so simple. Here we use the method described in Section 1.1.3: Let us name $w_i$ the minimum valid value to be assigned to out-going edges for vertices of color $i$. The sequence $(w_i)$ can be defined by recurrence by noticing that $w_0 = 1$ and
```

that the following holds for every integer i:

$$w_{2i} = 1 - \sum_{0<2j-1<2i} n_{2j-1} \cdot w_{2j-1} \qquad\qquad w_{2i+1} = -1 - \sum_{0\leqslant 2j<2i+1} n_{2j} \cdot w_{2j}$$

This is the smallest possible value if we only consider the number of vertices of one color in the whole graph. More precise results could be obtained by studying elementary cycles in the graph, but this is not our point here. To implement this, we observe that the recurrence can be rewritten in the form similar to a Fibonacci sequence. We have $w_0 = 1$, $w_1 = -n_0 - 1$, and for all $i \geqslant 2$ we have $w_i = w_{i-2} - n_{i-1} \cdot w_{i-1}$. This assumes that the number of priorities used is at least 2 (because of the size of the *weight* table), but this restriction characterizes interesting cases.

65 ⟨Compute weights using the *Smallest* method 65⟩ ≡

```
weight[0] = 1;
weight[1] = -frequency[0] - 1;
for (int i = 2; i < max_priority; i++)
    weight[i] = weight[i - 2] - frequency[i - 1] * weight[i - 1];
```

This code is used in chunk 61.

Chapter 3

Experimental observations

In this part we present and study experimental results obtained by running the previously described algorithm (with the test program described in Section A), showing how many iterations of the algorithm are needed to decide the winner in every position of a game network.

3.1 Number of iterations

3.1.1 Varying color numbers and outdegrees

Figure 3.1 shows a surface plot made from the average number of iterations in randomly generated networks of size 1000, with varying outdegrees and numbers of colors, analyzing 100 randomly generated networks in each case. It appears that the highest number of iterations occurs for small outdegrees, namely 4 or 5 depending on the number of colors.

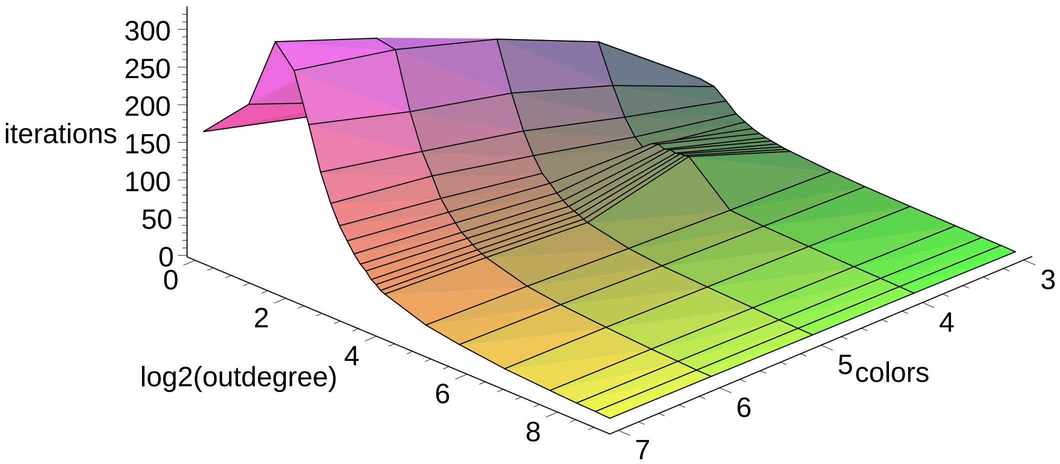

Figure 3.1: average number of iterations for networks of size 1000 and variable outdegrees and numbers of colors

Similar results appear for other network sizes. For instance, on networks of 4000 vertices, the algorithm behaves as shown on Figure 3.2, which is very similar. An interesting thing to consider is also the number of iterations required by the worst cases we encounter in the experiments. As Figure 3.3 shows, in these cases, GKK behaves in a similar way. The number of iterations is at most twice the one in the average case, which is still under the number of vertices. No exponential behavior is observed with respect to the number of colors, in particular. However, this requires some discussion, because we are only considering randomly generated problem instances. This

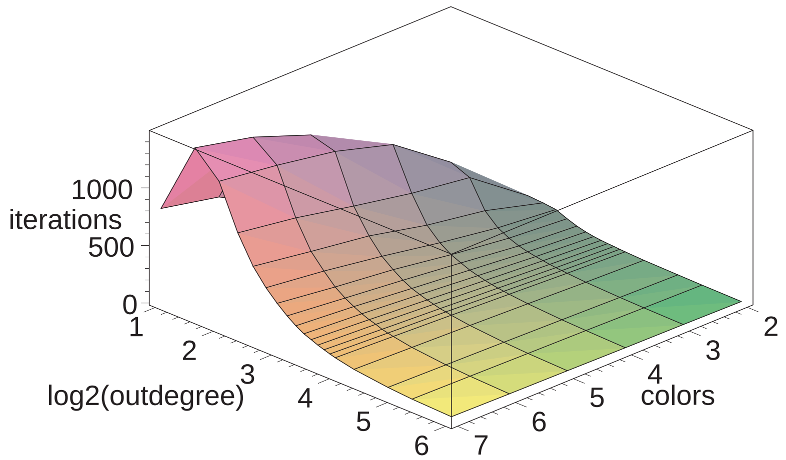

Figure 3.2: average number of iterations for networks of size 4000 and variable outdegrees and numbers of colors

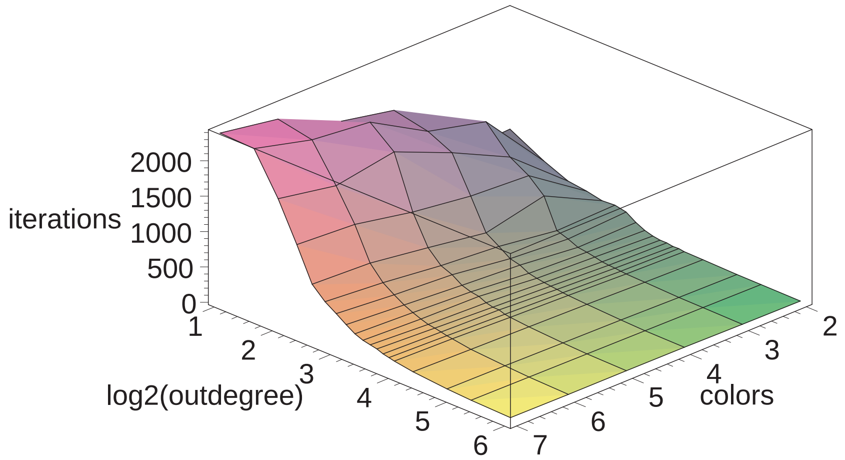

Figure 3.3: maximum number of iterations for networks of size 4000 and variable outdegrees and numbers of colors

does not mean that actual worst cases are not exponential, but no such instance was generated. Moreover, we never managed to exhibit any bad case specifically (the worst we constructed show at most a linear behavior with respect to the number of vertices, independently of the number of colors).

A point of interest, in the study of the complexity of the algorithm, is the maximum number of consecutive iterations where the S and T are not modified. Since they may only decrease, the number of iterations where at least one of them is modified can not exceed the number of vertices. Therefore, finding a good bound on the number of consecutive iterations with no change on these sets would also provide a good bound on the overall complexity. Figure 3.4 shows the maximum encountered value of this parameter in the same cases as the previous two examples, and it appears

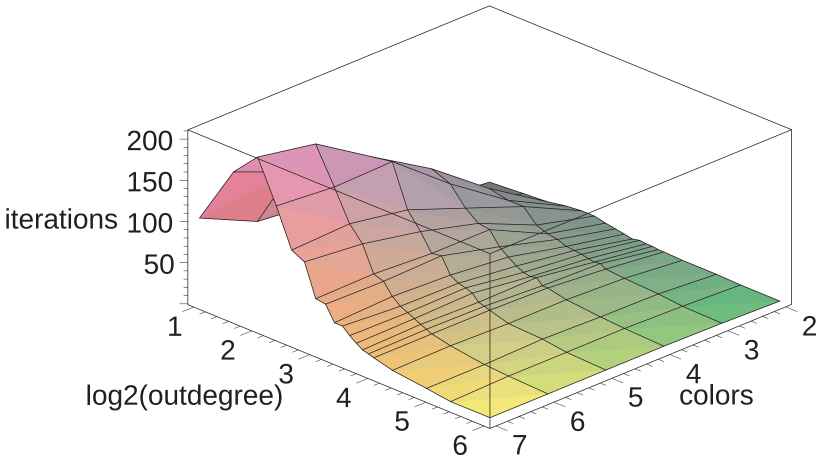

Figure 3.4: maximum number of iterations without change for networks of size 4000 and variable outdegrees and numbers of colors

that the behavior here is, again, very similar to the number of iterations, at a lower scale. The same restrictions apply here as for maximum complexity in Figure 3.3, but this gives trustable ideas of the value in average cases. Unfortunately, our studies of this parameter did not lead to any precise explanation nor hypothesis on this behavior.

More experimental data are available, with other network sizes up to 20.000, and the results are similar in every case, showing a peak for expected outdegrees 2 or 3, with a number of iterations seemingly sub-linear with respect to the number of colors. Experiments with constant outdegrees confirm this behavior.

3.1.2 Worst-case for the outdegree

Considering the previous results, we may want to only consider the behavior on small fixed outdegrees. Figure 3.5 shows the average number of iterations in the same context as Figure 3.1, and the maximum number of iterations encoutered by the test program.

3.2 Potential transformations

The programs allows applying potential transformations to the networks before running the algorithms. This way it is possible to study the influence of the actual edge costs on several mean-payoff game networks equivalent to the same parity game.

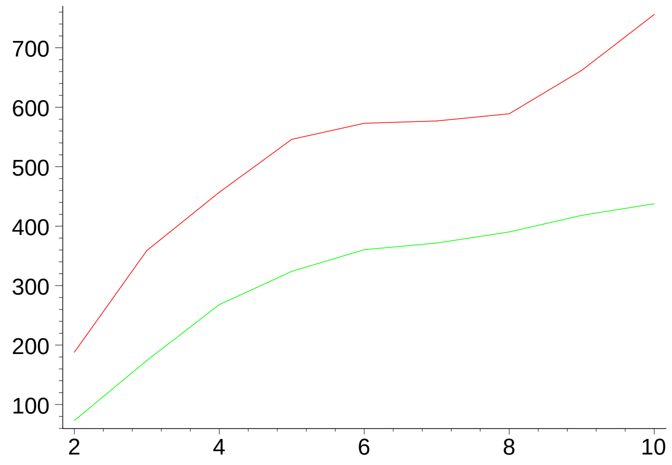

Figure 3.5: average and maximum number of iterations for networks of size 1000, maximum outdegree 4 and variable numbers of colors

3.2.1 The transformations

The transformations we apply are defined as follows. Let $\mathcal{N}$ be a mean-payoff game network obtained as the reduction of a parity game network with priority function p. Let (w_i) be the sequence of costs used in the reduction (as explained in Section 1.1.3). Given an integer k_0, we apply the transformation associated with the potential function a defined as:

$$\text{for all } v \in V, \qquad a(v) = \begin{cases} 0 & \text{if } p(v) < k_0 \\ -w_{p(v)} & \text{if } p(v) \geqslant k_0 \end{cases}$$

This transformation may only decrease the amplitude of the cost function. Experiments actually show that when k_0 is in the middle of the set of priorities, the average number of required iterations is slightly less.

We provide this transformation in the form of a new method for the class **graph**. This will first define potentials, as a new field in **vertex** objects, and then shift the costs accordingly. The method needs to know the number of vertices because it is the base for the potential values.

71 ⟨Methods for **graph** 30⟩ +≡

```
void apply_potential(int n, int k0)
{
  for_vertex (i) V[i].def_potential(*this, k0);
  for_vertex (i) V[i].apply_potential(*this, n);
}
```

The definition of the potential is done simply by comparing the priority of the vertex (as stored into the *prio* field) to k_0 and setting the value accordingly.

72 ⟨Methods for **vertex** 28⟩ +≡

```
void def_potential(const graph &G, int k0)
{
  if (prio ≥ k0) potential = −G.weight[prio];
  else potential = 0;
}
```

Applying the potential transformation redefines the value of the costs according to the priority of the vertices, rather than using the current value, so the method *apply_potential* for the graph can be used to try with different values of k_0 on the same graph.

73 ⟨Methods for **vertex** 28⟩ +≡

$$\textbf{void } apply\_potential(\textbf{const graph } \&G, \textbf{int } n)$$
$$\{$$
$$\quad \textbf{for\_edge } (i)\ E[i].cost \mathrel{+}= potential - G.V[E[i].dest].potential;$$
$$\}$$

3.2.2 The results

As stated above, these transformations do not increase the amplitude (norm) of the initial cost function. Therefore, they do not influence the worst-case behavior of the algorithm. Considering the ratio between the number of iterations of GKK required with such transformations and without them (see Figure 3.6) shows that the behavior is rather significantly improved when choosing values of k_0 around the middle of the color interval.

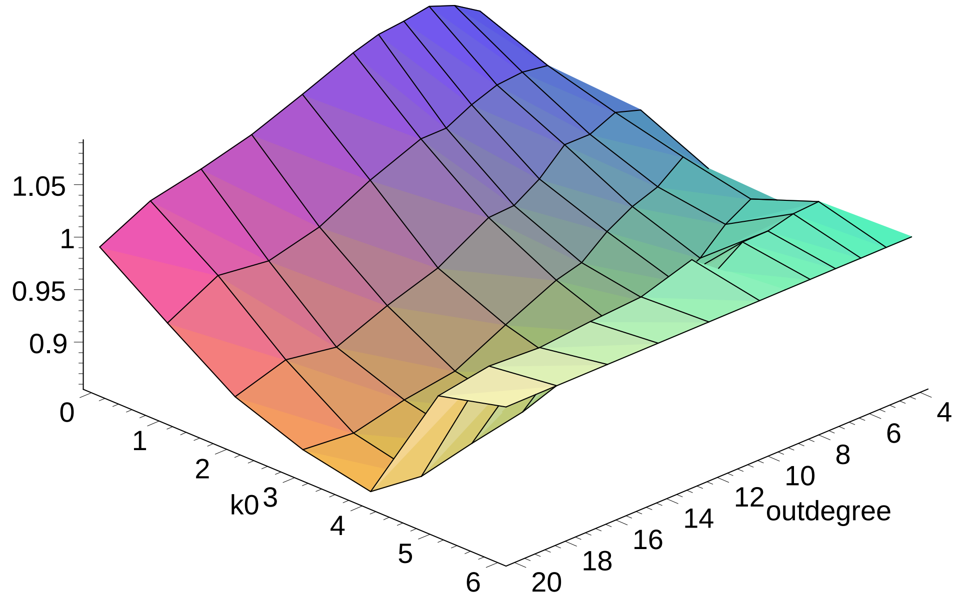

Figure 3.6: ratio between the number of iterations used after applying initial potential transformations and the number required without it (average data for 100 networks of size 1000 with 6 colors and variable outdegrees)

Conclusions and Further Works

The approach investigated in this work proved to be successful. The GKK algorithm could actually be adapted to parity games using our reduction, and the implementation could be made running and reliable thanks to the CWEB technology of literate programming. It also appeared to be sufficiently maintainable, so that additional features could be included easily whenever required by experiments.

The results we obtained are over our initial expectations. Though theoretical studies of the algorithm predicted a number of iterations polynomial in the size of the game networks and exponential in the number of colors, the considerable number of runs we made (of order millions) on randomly generated instances of the problem (on instances of size up to 20000, with up to 6 or 7 colors) always require small numbers of iterations and the behavior seems sub-linear in both parameters. Moreover, examples of the (not yet completely excluded) exponential behaviors were never found, neither by random generation nor by explicit research[1]. Though no conclusions may be drawn about the actual worst-case behavior of GKK applied to parity games, these results let us think that average-case cost is actually polynomial with respect to all parameters.

Our study of this problem an ongoing work. More experiments are led, planned, and the implementation is improved to allow for studying a wider range of problems, especially in order to know what kind of behavior appears for large numbers of priorities.

While these observations are already satisfactory by themselves, further work has to be done about this problem. A first discussion point is the method used to generate our random data: Our probability distribution is hard to study from a mathematical point of view, and it may not be satisfactory with respect to practical cases. Since the problem of finding winning strategies in parity games is known to be equivalent to μ-calculus model-checking, it could be tried, for instance, to consider random instances of such model-checking problems, or even to study real cases. However, this argument equally applies to any *a priori* fixed probability distribution. The problem here is that no real-life data of known bad cases are available for study.

Another point of interest, when trying to qualify our algorithm as a *good* solution to the initial problem, would be to compare its behavior to the actual behavior of other known algorithm on the same problem instances. However, no similar results on experiments with other algorithms were known about, so such a study would have required implementing other algorithms ourselves for the purpose of comparison, and we chose not to proceed in this direction.

Certainly, it is necessary to continue a more precise theoretical study of the problem and the algorithm in order to provide sufficient explanations of the behaviors we observed.

[1] Incidentally, we learned from the authors of the original paper that they never found explicit worst cases for their algorithm neither.

Appendix A

The program

This section contains auxiliary parts of the implementation of the algorithm. This mainly includes a test program, functions used to generate outputs and a series of validity checkings. The material in this section is not of primary importance for the purpose of this presentation but is included to make the document self-contained. Besides, this part provides for a comfortable environment for experimentations.

A.1 The test program

The general layout of the code is rather standard.

76 ⟨Include files 79⟩
⟨Preprocessor definitions⟩
⟨Auxiliary data types 23⟩
⟨Global variables and functions 77⟩
⟨Classes for game networks 19⟩
⟨Test program 80⟩

Various progress reports may be produced during the execution of the algorithms, and various validity checks may be performed. Since we do not want to always display all kinds of possible debugging data, we provide a series of settings to control verbosity.

77 ⟨Global variables and functions 77⟩ ≡
bool *verb_sets* = *false*; /* dump the value of S, K, T and L at each iteration */
bool *verb_result* = *false*; /* dump the final value of L in *reduce* */
bool *verb_delta* = *false*; /* dump the value of δ at each iteration */
bool *verb_weights* = *false*; /* print out the weights used in cost initialization */
bool *verb_sizes* = *false*; /* print out size changes for S and T */
bool *verb_k0* = *false*; /* print out each value of k_0 when applying potentials */
bool *verb_file* = *false*; /* print raw tuples for reading with Maple */

See also chunks 78, 94, 104, and 110.
This code is used in chunk 76.

For readability, the VT100 standard allows us to generate a fancy, colorful display on a terminal, so we just use this feature, using a few constant definitions.

78 ⟨Global variables and functions 77⟩ +≡
struct vt_info {
const char **reset*;
const char **bold*;
const char **red*;
const char **green*;
const char **blue*;

```
};
const vt_info dumb = {"","","","",""};
const vt_info vt100 = {"\x1B[0m","\x1B[1m","\x1B[31m","\x1B[32m","\x1B[34m"};
vt_info vt = vt100;
```

We put all include files right here to save some space and readability afterwards.

79 ⟨ Include files 79 ⟩ ≡

```
#include <stdarg.h>
#include <stdlib.h>
#include <time.h>
#include <iostream>
#include <vector>
```

This code is used in chunk 76.

We now provide a program to experiment with this algorithm. Its purpose will be to set up the verbosity parameters, and to perform series of test runs with different kinds of variations.

80 ⟨ Test program 80 ⟩ ≡

```
int main(int argc, char *argv[])
{
  bool fixed_degree = false;    /* should all vertices have the same outdegree ? */
  bool init_potentials = false;
     /* do we apply various potentials before running the algorithm ? */
  cost_method method = Smallest;    /* how do we define the costs ? */
  ⟨ Read the command line 81 ⟩
  ⟨ Initialize global variables 85 ⟩
  ⟨ Variables in the main function 87 ⟩
  vector⟨status⟩ result(size);    /* the state of L at the end */
  long sw2 = 0;    /* sum of the sizes of the winning set for player 2 (i.e. L) */
  try {
    for (int i = 0; i < tries; i++) {
      ⟨ Create a random network G 84 ⟩
      if (¬init_potentials) {
        ⟨ Run the algorithm on G 86 ⟩
        ⟨ Update the global statistics 88 ⟩
      } else
        for (int k0 = 0; k0 ≤ max_prio; k0++) {
          if (verb_k0) {
            cout ≪ vt.bold ≪ "[k0␣=␣" ≪ k0 ≪ "]" ≪ vt.reset;
            cout.flush();
          }
          ⟨ Reinitialize the graph G 90 ⟩
          G.apply_potential(size, k0);    /* apply the initial potential transformation */
          ⟨ Run the algorithm on G 86 ⟩
          ⟨ Update the statistics for k0 and define result 91 ⟩
          if (verb_k0) cout ≪ endl;
        }
      if (verb_result) { ⟨ Display L if needed and update sw2 92 ⟩ }
    }
  } catch(const char *pb) {
    cout.flush();
    cerr ≪ endl ≪ vt.red ≪ vt.bold ≪ "***␣" ≪ pb ≪ vt.reset ≪ endl;
    return −1;
  }
  ⟨ Display average results 96 ⟩
```

```
    return 0;
  }
```
This code is used in chunk 76.

A.1.1 The command line

On the command line we specify: 1) the number of test-runs to do, 2) the size of the graphs, 3) the maximum degree, and 4) the number of priorities used for randomization.

81 ⟨Read the command line 81⟩ ≡
```
  if (argc < 5) {
    ⟨Print a list of allowed options 83⟩
    return 1;
  }
  int tries = atoi(argv[1]);
  int size = atoi(argv[2]);
  int degree = atoi(argv[3]);
  int max_prio = atoi(argv[4]);
```
See also chunk 82.
This code is used in chunk 80.

The “options” on the command line are actually letters in the fifth and following arguments. Each letter specifies a particular verbosity level to be switched, or a particular feature to activate.

82 ⟨Read the command line 81⟩ +≡
```
  for (int i = 5; i < argc; i++)
    for (const char *p = argv[i]; *p ≠ 0; p++)
      switch (*p) {
      case 'D': fixed_degree = true; break;
      case 'P': init_potentials = true; break;
      case 'X': method = Exponential; break;
      case 'c': verb_sizes = ¬verb_sizes; break;
      case 'd': verb_delta = ¬verb_delta; break;
      case 'k': verb_k0 = ¬verb_k0; break;
      case 'f': verb_file = ¬verb_file; break;
      case 'r': verb_result = ¬verb_result; break;
      case 's': verb_sets = ¬verb_sets; break;
      case 'v': vt = dumb; break;
      case 'w': verb_weights = ¬verb_weights; break;
      default: cerr ≪ "I␣don't␣understand␣the␣switch␣'" ≪ *p ≪ "'." ≪ endl;
      }
```

Anyway, it is always useful to help the user remember these switches, so we have to provide a help message.

83 ⟨Print a list of allowed options 83⟩ ≡
```
  cerr ≪ "syntax:␣" ≪ argv[0] ≪ "␣<tries>␣<size>␣<max␣degree>␣<max␣prior\
      ity>␣[options]" ≪ endl;
  cerr ≪ "execution␣options:" ≪ endl;
  cerr ≪ "␣␣D␣=␣let␣all␣vertices␣have␣the␣maximum␣outdegree" ≪ endl;
  cerr ≪ "␣␣P␣=␣apply␣initial␣potential␣tranformations" ≪ endl;
  cerr ≪ "␣␣X␣=␣use␣exponential␣costs␣in␣PG/MPG␣reduction" ≪ endl;
  cerr ≪ "display␣options:" ≪ endl;
  cerr ≪ "␣␣c␣=␣display␣the␣size␣of␣S␣and␣T␣when␣they␣change" ≪ endl;
  cerr ≪ "␣␣d␣=␣display␣the␣value␣of␣delta␣at␣each␣iteration" ≪ endl;
  cerr ≪ "␣␣f␣=␣display␣raw␣results␣suitable␣for␣reading␣with␣Maple" ≪ endl;
  cerr ≪ "␣␣k␣=␣display␣each␣k0␣when␣'P'␣is␣turned␣on" ≪ endl;
```

```
cerr << "  r = display the result of the algorithm (final size of L)" << endl;
cerr << "  s = display the sets S, K, T, L at each iteration" << endl;
cerr << "  v = do not use vt100 escape codes" << endl;
cerr << "  w = display the weights used in PG/MPG reduction" << endl;
```
This code is used in chunk 81.

A.1.2 Execution of the algorithm

The first thing to do is, of course, to create a game network. This implies creating a new **graph** object, then randomizing it in order to create ownerships, priorities and edges, and then initialize the edge costs using the method previously chosen.

84 ⟨ Create a random network G 84 ⟩ ≡
```
graph G(size);
G.randomize(degree, max_prio, fixed_degree);
G.set_initial_costs(method);
```
This code is used in chunk 80.

This randomization implies initializing a random seed at the beginning of the program, since we do not want to always get the same test graphs. For this, we use the usual “time” method:

85 ⟨ Initialize global variables 85 ⟩ ≡ `srand(time(Λ));`

See also chunk 95.
This code is used in chunk 80.

After that, running the algorithm simply means calling the *reduce* method, with 0 as the threshold value. However, this requires computing the extremal value of the vertices before, and reseting *reduce_iterations* to get a valid value.

86 ⟨ Run the algorithm on G 86 ⟩ ≡
```
reduce_iterations = 0;
G.update_ext();
G.reduce(0);
```
This code is used in chunk 80.

Updating the statistics means adding the number of iterations in the last call to *reduce* to an accumulator, and updating the maximum value in another variable.

87 ⟨ Variables in the *main* function 87 ⟩ ≡
```
int total_iterations = 0, max_iterations = 0;
```
See also chunk 89.
This code is used in chunk 80.

88 ⟨ Update the global statistics 88 ⟩ ≡
```
if (verb_weights) {
  if (verb_delta ∨ verb_sizes) cout << endl << vt.bold;
  cout << "-- " << reduce_iterations << " iterations" << vt.reset << endl;
}
total_iterations += reduce_iterations;
max_iterations = max(max_iterations, reduce_iterations);
```
This code is used in chunk 80.

When using a potential variation, we let the threshold k_0 range from 0 to *max_prio*, so that the last case is what would be obtained if no potential was applied at all. In this case, statistics are done for each value of k_0 separately, so we have to use tables to store the number of iterations for each value.

89 ⟨ Variables in the *main* function 87 ⟩ +≡
```
vector⟨int⟩ total_iter(max_prio + 1);    /* sets all elements to 0 */
vector⟨int⟩ max_iter(max_prio + 1);
```

In this case, the same graph is used for every value of k_0, of course, and so the costs have to be reinitialized for each value.

90 ⟨Reinitialize the graph G 90⟩ ≡

```
G.clear();    /* reset all pt fields to Undef */
G.reset_costs();
```

This code is used in chunk 80.

Therefore, updating the statistics means adding the value of *reduce_iterations* to *total_iter*[k_0]. At the same time, we check that the results we obtained are the same as for the previous value of k_0.

91 ⟨Update the statistics for k_0 and define *result* 91⟩ ≡

```
total_iter[k0] += reduce_iterations;
max_iter[k0] = max(max_iter[k0], reduce_iterations);
if (k0 == 0)
  for (int j = 0; j < size; j++) result[j] = G.V[j].st;
else
  for (int j = 0; j < size; j++)
    if (result[j] != G.V[j].st) throw "results changed";
```

This code is used in chunk 80.

At each iteration, whether we apply initial potential transformations or not, we end up counting the number of elements in L (and displaying them if we displayed the sets previously). We add this number to the sum *sw2* in order to get an average value of the size of L at the end.

92 ⟨Display L if needed and update *sw2* 92⟩ ≡

```
int w2 = 0;
for (int j = 0; j < size; j++)
  if (G.V[j].st == L) {
    if (verb_sets) cout << j << " ";
    w2++;
  }
if (verb_sets) cout << "=> 2nd wins in " << w2 << " vertices out of " << size << endl;
sw2 += w2;
```

This code is used in chunk 80.

A.1.3 Profiling

One of the main purposes of this implementation is to study the actual complexity of the process on specific networks. Therefore we want profiling to be done during the execution of the algorithm. For this purpose, we introduce a series of global variables, which get incremented during the actual execution of the algorithm.

94 ⟨Global variables and functions 77⟩ +≡

```
int reduce_calls;    /* number of calls to the auxiliary algorithm */
int reduce_iterations;    /* total number of iterations of the auxiliary algorithm */
int label_iterations;    /* total number of iterations in the construction of L */
```

These variables also have to be initialized.

95 ⟨Initialize global variables 85⟩ +≡

```
reduce_calls = 0;
reduce_iterations = 0;
label_iterations = 0;
```

At the end of the process, we display the average and the maximum of the number of iterations. When using different initial potentials, we print one result for each value of k_0.

96 ⟨ Display average results 96 ⟩ ≡

```
if (verb_sets ∨ verb_delta ∨ verb_sizes) cout ≪ endl;
if (¬verb_file) cout ≪ endl ≪ vt.bold ≪ "average␣result:" ≪ vt.reset;
if (init_potentials)
  for (int k0 = 0; k0 ≤ max_prio; k0++)
    cout ≪ endl ≪ "␣␣k0␣=␣" ≪ k0 ≪ "␣␣=>␣␣" ≪ (double)
        total_iter[k0]/tries ≪ "␣iterations␣(max:␣" ≪ max_iter[k0] ≪ ")";
else if (¬verb_file)
  cout ≪ "␣" ≪ (double) total_iterations/tries ≪ "␣iterations␣(max:␣" ≪
      max_iterations ≪ ")";
else cout ≪ max_prio ≪ "␣" ≪ degree ≪ "␣" ≪ (double)
      total_iterations/tries ≪ "␣" ≪ max_iterations ≪ "␣" ≪ max_nb_iter ≪ endl;
```

See also chunk 97.
This code is used in chunk 80.

After that, if the corresponding switch was turned on, we also print the average size of the final value of L.

97 ⟨ Display average results 96 ⟩ +≡

```
if (verb_result) {
  if (¬init_potentials) cout ≪ ",␣";
  else cout ≪ endl;
  cout ≪ "2nd␣wins␣on␣" ≪ (double) sw2/tries ≪ "␣vertices";
}
if (¬verb_file) cout ≪ endl;
```

A.2 Output functions

A.2.1 Dumping structures

Every good data structure in an experimental program has methods to be printed out on a terminal for various purposes. We provide one for each of our structures, and we name it *printOn*. A graph is just a list of vertices, so we just browse the table and print each vertex with its index. We also dump the value of m and M.

99 ⟨ Methods for **graph** 30 ⟩ +≡

```
void printOn(ostream &s)
{
  s ≪ "--␣m␣=␣" ≪ m ≪ ",␣M␣=␣" ≪ M ≪ endl;
  for_vertex (i) {
    s ≪ i ≪ ":␣";
    V[i].printOn(s, *this);
    s ≪ endl;
  }
}
```

A vertex has an owner and a list of edges, so the printing function is simple.

100 ⟨ Methods for **vertex** 28 ⟩ +≡

```
void printOn(ostream &s, const graph &G) const
{
  if (who ≡ 1) s ≪ "white,";
  else s ≪ "black,";
  for_edge (i) E[i].printOn(s);
  s ≪ ",␣ext␣=␣" ≪ ext ≪ ",␣vext␣=␣{";
  for_edge (i)
    if (E[i].cost ≡ ext) s ≪ "␣" ≪ E[i].dest;
  s ≪ "␣}";
```

```
}
```

Finally, printing an edge means printing the destination vertex and the cost of the edge.

101 ⟨Methods for **edge** 25⟩ +≡

```
void printOn(ostream &s) const
{
  s ≪ "␣(" ≪ dest ≪ ":␣" ≪ cost ≪ ")";
}
```

A.2.2 Output data

When creating networks, we may want to display how the weights of edges are chosen when reducing a parity games into a mean-payoff game.

102 ⟨Display the weights if required 102⟩ ≡

```
if (verb_weights) {
  if (verb_delta ∨ verb_sizes) cout ≪ endl ≪ vt.bold;
  cout ≪ "w0␣=␣" ≪ weight[0];
  for (int i = 1; i < max_priority; i++) cout ≪ ",␣w" ≪ i ≪ "␣=␣" ≪ weight[i];
  cout ≪ vt.reset ≪ endl;
}
```

This code is used in chunk 61.

A.2.3 Progress reports

In the algorithm, we only display the size of S and T if they have changed since the last iteration, therfore we introduce the *previous_S* and *previous_T* variables. We also want to know the maximum number of iterations in a row where the sets S and T are not modified, so we introduce other variables and test to update this.

103 ⟨Variables for *reduce* 37⟩ +≡

```
int previous_S = 0, previous_T = 0;
```

104 ⟨Global variables and functions 77⟩ +≡ **int** *nb_iter* = 0, *max_nb_iter* = 0;

105 ⟨Display the sizes 105⟩ ≡

```
if (verb_sizes) {
  if (previous_S ≠ size_of_S) cout ≪ vt.green ≪ "␣S:" ≪ size_of_S ≪ vt.reset;
  if (previous_T ≠ size_of_T) cout ≪ vt.blue ≪ "␣T:" ≪ size_of_T ≪ vt.reset;
  cout.flush();
}
if (previous_S ≡ size_of_S ∧ previous_T ≡ size_of_T) nb_iter++;
else {
  if (nb_iter > max_nb_iter) max_nb_iter = nb_iter;
  nb_iter = 0;
}
previous_S = size_of_S;
previous_T = size_of_T;
```

This code is used in chunk 36.

The purpose of our program is the study of the algorithm, so we may want dump the values of the sets at every iteration, when verbosity is required.

106 ⟨Dump the sets on the standard output 106⟩ ≡

```
if (verb_sets) {
  cout ≪ endl;
  dump_part("S", S);
  dump_part("K", K);
```

```
    dump_part("T", T);
    cout ≪ "L␣=␣{";
    for_vertex (i)
      if (V[i].st ≡ L) cout ≪ "␣" ≪ i;
    cout ≪ "␣}" ≪ endl;
  }
```

This code is used in chunk 36.

The method *dump_part* prints the contents of the specified part (S, K or T) on the standard output as a list of integers surrounded by curly braces, and prefixed by a string of the form "S␣=␣".

107 ⟨ Methods for **graph** 30 ⟩ +≡

```
  void dump_part(const char *name, part p)
  {
    cout ≪ name ≪ "␣=␣{";
    for_vertex (i)
      if (V[i].pt ≡ p) cout ≪ "␣" ≪ i;
    cout ≪ "␣}" ≪ endl;
  }
```

A.3 Validity checking

The purpose of the introduction of the *Undef* value besides S, K and T is to do a check of validity when updating the *ext* values: The monotonicity property guarantees that *ext* only gets closer to the threshold value p, which is a field of the graph object. When the value of *pt* is *Undef*, no check is to be done. If the value of *ext* is changed, we increment a global counter, in order to check if there are cases where none of the values of *ext* are changed at one iteration.

109 ⟨ Check the validity of the new value of *ext* 109 ⟩ ≡

```
  switch (pt) {
  case S:
    if (new_ext < ext) throw "non-monotonicity␣in␣S";
    break;
  case K:
    if (new_ext ≠ ext) throw "K␣decreased";
    break;
  case T:
    if (new_ext > ext) throw "non-monotonicity␣in␣T";
  }
  if (pt ≠ Undef ∧ new_ext ≠ ext) modified_ext++;
```

This code is used in chunk 28.

110 ⟨ Global variables and functions 77 ⟩ +≡ **int** *modified_ext*;

Index

The following index includes a list of the identifiers used, and where they appear. Underlined entries indicate the chunk of definition (according to CWEB). The index also includes entries for key terms used throughout the document.

List of Refinements

Bibliography

[1] H. Björklund, V. Petersson, and S. Vorobyov, *Experiments with iterative improvement algorithms on completely unimodal hypercubes*, Technical Report 2001-017, Information Technology/Uppsala University, September 2001.

[2] A. Browne, E. M. Clarke, S. Jha, D. E. Long, and W. Marrero, *An improved algorithm for the evaluation of fixpoint expressions*, Theoretical Comput. Sci., 178 (1997), pp. 237–255. Preliminary version in CAV'94, LNCS'818.

[3] A. Ehrenfeucht and J. Mycielski, *Positional strategies for mean payoff games*, International Journal of Game Theory, 8 (1979), pp. 109–113.

[4] E. A. Emerson, *Temporal and modal logic*, in Handbook of Computer Science, J. van Leeuwen, ed., vol. B, Elsevier, 1990, pp. 996–1072.

[5] ——, *Model checking and the Mu-calculus*, in DIMACS Series in Discrete Mathematics, N. Immerman and P. G. Kolaitis, eds., vol. 31, 1997, pp. 185–214.

[6] E. A. Emerson, C. Jutla, and A. P. Sistla, *On model-checking for fragments of μ-calculus*, in Computer Aided Verification, Proc. 5th Int. Conference, vol. 697, Lecture Notes in Computer Science, 1993, pp. 385–396.

[7] V. A. Gurvich, A. V. Karzanov, and L. G. Khachiyan, *Cyclic games and an algorithm to find minimax cycle means in directed graphs*, USSR Computational Mathematics and Mathematical Physics, 28 (1988), pp. 85–91.

[8] M. Jurdziński, *Deciding the winner in parity games is in UP $\cap$ co-UP*, Information Processing Letters, 68 (1998), pp. 119–124.

[9] M. Jurdzinski, *Small progress measures for solving parity games*, in 17th STACS, vol. 1770 of Lecture Notes in Computer Science, Springer Verlag, 2000, pp. 290–301.

[10] M. Jurdziński, *Small progress measures for solving parity games*, in 17th Annual Symposium on Theoretical Aspects of Computer Science (STACS), vol. 1770 of Lecture Notes in Computer Science, Springer-Verlag, February 2000, pp. 290–301.

[11] D. E. Knuth, *Literate programming*, The Computer Journal, 27 (1984), pp. 97–11.

[12] ——, *Literate Programming*, vol. 27 of CSLI Lecture Notes, Center for the Study of Language and Information, 1992.

[13] D. E. Knuth and S. Levy, *The CWEB System of Structured Documentation*, Addison Wesley, Reading, MA, USA, 1993. Version 3.0.

[14] H. Moulin, *Prolongement des jeux à deux joueurs de somme nulle*, in Bulletin de la Société Mathématique de France, SMF, 1976, mémoire 45.

[15] J. Nash, *Equilibrium points in n-person games*, Proceedings of the National Academy of Science, 36 (1950), pp. 48–49.

[16] C. H. Papadimitriou, *Algorithms, games, and the Internet*, in 33rd Annual Symposium on Theory of Computing (STOC), ACM, July 2001, pp. 749–753.

[17] V. Petersson and S. Vorobyov, *Interior-point approach to parity games*, in IEEE Annual Symposium on Logic in Computer Science (LICS'2001), Boston, Massachusetts, June 2001. Short communication, full version: Technical Report #008, 2001, Department of Information Technology, Uppsala Unitersity.

[18] ——, *A randomized subexponential algorithm for parity games*, Nordic Journal of Computing, (2001). To appear. Preliminary version: Technical Report #1, 2001, Department of Information Technology, Uppsala Unitersity. Also: Proceedings of the 12th Nordic Workshop on Programming Theory.

[19] N. Pisaruk, *Mean cost cyclical games*, Mathematics of Operations Research, 24 (1999).

[20] H. Seidl, *Fast and simple nested fixpoints*, Information Processing Letters, 59 (1996), pp. 303–308.

[21] J. Vöge and M. Jurdzinski, *A discrete strategy improvement algorithm for solving parity games*, in CAV'00: Computer-Aided Verification, vol. 1855 of Lecture Notes in Computer Science, Springer Verlag, 2000, pp. 202–215.

[22] J. von Neumann and O. Morgenstern, *Theory of Games and Economic Behavior*, Princeton University Press, 1944.

[23] W. Zielonka, *Infinite games on finitely coloured graphs with applications to automata on infinite trees*, Theoretical Computer Science, 200 (1998), pp. 135–183. Fundamental Study.

[24] U. Zwick and M. Paterson, *The complexity of mean payoff games on graphs*, Theoretical Computer Science, 158 (1996), pp. 343–359.