VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

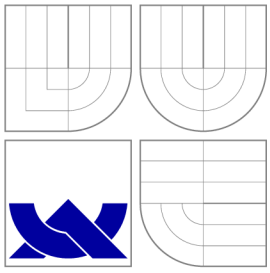
IMPLEMENTACE ALGORITMŮ TEORIE HER

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

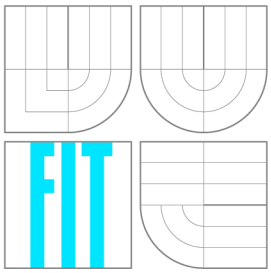
AUTOR PRÁCE
AUTHOR

STANISLAV ŽIDEK

BRNO 2009



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

IMPLEMENTACE ALGORITMŮ TEORIE HER

IMPLEMENATION OF A GAME THEORY LIBRARY

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

STANISLAV ŽIDEK

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. MARTIN HRUBÝ, Ph.D.

BRNO 2009

Abstrakt

Teorie her se během doby své existence stala vhodným nástrojem pro modelování různých situací, které obnášejí rozhodování racionálních entit – hráčů. Uplatnění v praxi je bohužel limitováno velikostí her, jež jsme schopni se současnou technikou spočítat. Tato diplomová práce se zaměřuje na korelované ekvilibrium v nekooperativních hrách a klade si za cíl vytvořit knihovnu, která bude schopna co nejefektivněji toto ekvilibrium hledat.

Abstract

Game theory has become very powerful tool for modelling decision-making situations of rational players. However, practical applications are strongly limited by the size of particular game, which is connected to the computational power of computers nowadays. Aim of this master's thesis is to design and implement a library, which would be able to find correlated equilibria in as complex non-cooperative games as possible.

Klíčová slova

Teorie her, korelované ekvilibrium, G-matice, eliminace dominovaných strategií, lineární programování, simplexová metoda, paralelizace, OpenMP.

Keywords

Game theory, correlated equilibrium, G-matrix, elimination of dominated strategies, linear programming, simplex method, parallelization, OpenMP.

Citace

Stanislav Židek: Implementace algoritmů Teorie her, diplomová práce, Brno, FIT VUT v Brně, 2009

Implementace algoritmů Teorie her

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Martina Hrubého, Ph.D.

.....
Stanislav Židek
26. května 2009

Poděkování

Rád bych na tomto místě poděkoval svému vedoucímu za inspirativní diskuse nad problematikou a testovací problémy (hry) z praktických modelů. Zároveň děkuji Dr. Ing. Petru Peringerovi za poskytnuté konzultace. Rovněž díky všem, kteří mě v průběhu psaní této práce podporovali.

© Stanislav Židek, 2009.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

Kapitola 1

Úvod

Teorie her se se zvyšováním výkonu moderních počítačů stává mocným nástrojem pro modelování a simulaci rozhodovacích situací v rozmanitých oborech lidského zkoumání, od ekonomie po biologii. Zároveň modely *reálných* problémů dosahují obrovské velikosti stavového prostoru, který se například vůbec nemusí vejít do paměti počítače, o časové náročnosti nemluvě.

Tato diplomová práce se konkrétně zaměřuje na hledání korelovaného ekvilibria, jednoho z konceptů rovnovážných bodů analyzovaných teorií her. Na toto ekvilibrium jsme ze zaměřili převážně z toho důvodu, že se jedná o koncept skutečně používaný v praxi a máme přístup ke hrám, které jsou součástí reálných modelů. Dále není tak náročné na výpočet a je pro hráče atraktivní z toho důvodu, že jim poskytuje lepší zisky oproti například ekvilibriu Nashovu.

Za konečný cíl si klademe vytvoření optimalizované, přenositelné a snadno použitelné knihovny či nástroje pro řešení těchto problémů. To může být pro řešení praktických modelů velmi užitečné, protože tyto problémy bývají tak rozsáhlé, že jsou v čisté podobě při výkonu současných počítačů v podstatě neřešitelné. Naštěstí existují metody redukce stavového prostoru her (kterými se budeme zabývat), jež zachovávají ekvivalenci vzhledem k chování hráčů, díky čemuž řešením redukované hry dostáváme stejné výsledky, jako u hry původní. Velký vliv rovněž má samotný způsob implementace, například použití efektivních datových struktur a algoritmů či paralelizace výpočtu. Každé zrychlení tak zvětšuje množinu her, jež jsme schopni vyřešit, a tím nám umožňuje zkoumat komplexnější rozhodovací situace.

V tomto textu se nejdříve v kapitole 2 zaměříme na popis oboru studia teorie her. Vymezíme si důležité pojmy a koncepty, které budeme dále používat ve zbytku práce (korelované ekvilibrium, dominovaná strategie...). Rozebereme princip metody eliminace dominovaných strategií, jež dokáže často velmi výrazně zredukovat velikost stavového prostoru hry, aniž by se rozhodování hráčů v redukované hře nějak lišilo od toho ve hře původní.

Kapitola 3 rozebírá teoretické předpoklady, ze kterých jsme při tvorbě výsledného nástroje vycházeli. Pojednává o lineárním programování, což je optimalizační metoda pro nás zajímavá z toho důvodu, že problém hledání korelovaného ekvilibria lze převést na problém lineárního programování. Dále se tato kapitola věnuje popisu *G-matice*, struktury, jež je užitečná jak pro reprezentaci vícehráčových her ve dvourozměrné podobě, tak pro zmíněnou transformaci na problém lineárního programování. Zabýváme se zde rovněž popisem systému *OpenMP*, který je vhodný pro paralelizaci programů v jazyce C/C++ na paralelních architekturách se sdílenou pamětí.

V kapitole 4 je popsána konkrétní implementace důležitých částí nástroje. Přesněji se zabývá jeho celkovou koncepcí, použitými datovými strukturami a také detailně popisuje

důležité algoritmy používané při řešení. Dále rozebírá způsob využití paralelizace a záležitosti s ní spojené, především synchronizaci vláken při výpočtu. Je zde také načrtnut způsob použití nástroje, konkrétně možnosti zadání vstupní hry, specifikace parametrů řešení, spuštění samotného výpočtu a získání spočítaných výsledků. Nakonec jsou popsány dva další jednoduché nástroje, které jsme během práce vytvořili, jeden pro analýzu her z hlediska dominance a druhý pro řešení problémů lineárního programování zaměřený na řídké problémy.

Kapitola 2

Teorie her a nekooperativní hry

Teorie her je disciplína aplikované matematiky, která se zabývá rozhodováním účastníků (hráčů) v konkrétní situaci. Hráči mají k dispozici určitou množinu strategií a užitek z hraní hry každého z nich závisí i na volbách všech ostatních hráčů.

Vzhledem k tomu, že jednou z jejích prvních a nejdůležitějších aplikací je modelování chování/rozhodování ekonomických subjektů v prostředí tržní ekonomiky, můžeme si hru představit například jako několik výrobců stejného (podobného) výrobku, kteří se rozhodují, kolik kusů mají vyrobit. Každý z výrobců může mít jinou výrobní kapacitu a jiné výrobní náklady, zároveň také všichni zohledňují cenu výrobku, která bude záviset na nabídce, tedy počtu kusů, jež vyrobí všichni dohromady.

Ukázalo se, že modelovací schopnosti teorie her jsou velmi rozsáhlé, a proto v této oblasti probíhal a probíhá intenzivní výzkum. Teorie her našla své uplatnění v tak odlišných oborech, jako jsou filosofie, biologie, sociologie, politické vědy, mezinárodní vztahy nebo už zmíněná ekonomie [7].

2.1 Historie

Vznik teorie her se datuje do roku 1944, kdy byla vydána kniha Theory of Games and Economic Behavior, jejímiž autory jsou John von Neumann a Oskar Morgenstern. V této knize se autoři zabývají především hrami dvou hráčů s nulovým součtem.

Jedním z nejdůležitějších momentů historie pak bylo vytvoření konceptu Nashova ekvilibria (podle Johna Nashe), které je aplikovatelné na mnohem širší škálu her než původní kritérum, navrhované ve zmíněné knize.

V padesátých letech dvacátého století zažila teorie her velmi rychlý vývoj, bylo představeno několik velmi důležitých konceptů, jako například extenzivní forma her, opakované hry nebo Shapleyho hodnota. Rovněž se objevily první aplikace v politice a filosofii.

V šedesátých letech Reinhard Selten objevil podherně dokonalé ekvilibrium (*subgame perfect ekvilibrium*) a John Harsanyi vyvinul a zkoumal Bayesovské hry.

Sedmdesátá léta přinesla první aplikace v oblasti biologie, především John Maynard Smith a jeho evolučně stabilní ekvilibrium. Dále se objevilo korelované ekvilibrium a byly analyzovány principy *common knowledge* (společné znalosti).

Za poznatky v oblasti teorie her byly uděleny již tři Nobelovy ceny za ekonomii, první v roce 1994 (pánové Nash, Selten a Harsanyi), druhá v roce 2005 (pánové Shelling a Aumann) a zatím poslední v roce 2007 (pánové Myerson, Hurwicz a Maskin) [7].

2.2 Typy her a jejich reprezentace

Za téměř století existence teorie her vzniklo velmi mnoho typů her a kritérií pro jejich klasifikaci, stejně jako i několik různých forem jejich zápisu. V této části si představíme alespoň ty nejvýznamnější z nich.

2.2.1 Způsoby reprezentace her

Normální (strategická) forma

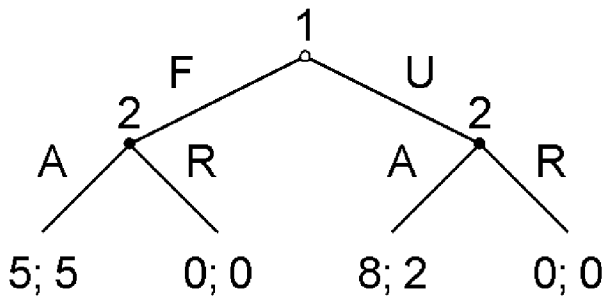
Tato forma je vhodná pro reprezentaci her, kde hráči volí své strategie všichni zároveň (nebo postupně, ale bez znalosti voleb ostatních hráčů). Hra dvou hráčů je v této formě většinou zobrazena jako matice, kde řádky reprezentují strategie prvního hráče, sloupce strategie druhého hráče a položky v matici seřazené užitky jednotlivých hráčů při daném strategickém profilu (definice strategického profilu později). Příklad můžeme vidět v tabulce 2.1. Jedná se o hru dvou hráčů, ve které každý hráč má na výběr dvě strategie – první hráč *Nahoru* a *Dolů*, druhý hráč *Doleva* a *Doprava*. Při jakékoliv volbě strategií obou hráčů má první hráč vždy užitek 1 a druhý 2.

	Hráč 2 hraje Doleva	**Hráč 2 hraje Doprava**
Hráč 1 hraje Nahoru	1, 2	1, 2
Hráč 1 hraje Dolů	1, 2	1, 2

Tabulka 2.1: Příklad hry dvou hráčů v normální (strategické) formě

Extenzivní forma

Pokud zkoumáme hry, ve kterých hráči volí své strategie v určitém pořadí a v okamžiku této volby mají určitou informaci o tom, jak volili předchozí, je vhodné použít právě extenzivní formu. Většinou se zobrazuje jako strom, ve kterém uzly jsou okamžiky rozhodnutí určitého hráče a hrany vedoucí k potomkům jsou jednotlivé strategie. Například na obrázku 2.1 vidíme, že v dané hře nejdříve první hráč zvolí jednu ze strategií F nebo U, a následně druhý hráč zvolí ze strategií A a R.



Obrázek 2.1: Hra v extenzivní formě (převzato z [7])

Forma daná charakteristickou funkcí

U kooperativních her je užitek koalic zadán charakteristickou funkcí tvaru $f : 2^Q \to \mathbb{R}$, kde Q je množina hráčů.

2.2.2 Typy her

Kooperativní vs. nekooperativní hry

U kooperativních her hráči tvoří koalice, ve kterých si rozdělí zisk daný charakteristickou funkcí. Základním předpokladem při řešení je vytvoření jedné velké koalice (*grand coalition*), ve které si hráči rozdělí zisk určitým způsobem tak, aby to bylo „pro všechny výhodné“.

Existuje několik způsobů, jak tuto výhodnost vnímat, respektive jak říci, zda dané rozdělení zisku je férové: [5]

- **Stabilní množina (stable set)** je množina vektorů rozdělení zisku, která splňuje kritéria vnitřní a vnější stability. Vektor rozdělení a dominuje b, pokud může skupina hráčů vytvořit vlastní menší koalici, ve které dosáhne každý většího zisku. Vnitřní stabilita označuje situaci, kdy žádný vektor rozdělení ve stabilní množině není dominován jiným vektorem z této množiny. Vnější stabilita říká to, že neexistuje vektor rozdělení, který by nebyl ve stabilní množině a zároveň nebyl dominován žádným vektorem ze stabilní množiny. Tento princip byl vytvořen právě J. von Neumannem a O. Morgensternem v jejich slavné knize [18], která dala základ celé disciplíně teorie her.

- **Jádro (core)** je množina vektorů rozdělení, pro kterou platí, že žádná odtrhnuvši se koalice nebude mít větší zisk, než je součet zisků jejich členů za současného rozdělení.

- **Silné ϵ-jádro** navíc počítá s určitou pokutou za opuštění velké koalice. Nutno podotknout, že tato pokuta může být i záporná. Můžeme pak hledat nejmenší jádro (least-core), což je neprázdné silné ϵ-jádro pro nejmenší hodnotu ϵ.

- **Shapleyho hodnota** vypočítává zisk jednotlivých hráčů ve velké koalici podle jejich přínosu k celkovému užitku.

- **Kernel** je množina rozdělení zohledňující jakousi vyjednávací sílu každého hráče vzhledem k jinému.

- **Jadérko (nucleolus)** používá lexikografické seřazení rozdílů užitků, které získají koalice odtržením od velké koalice, k nalezení unikátního rozdělení.

Ve zbytku práce se budeme zabývat hrami nekooperativními, ve kterých se hráči rozhodují pro některou ze svých strategií tak, aby dosáhli co největšího zisku. Tento zisk je dán volbami strategií všech hráčů. Pokud se tedy budeme na kooperativní hry dívat jako na hledání jakéhosi konsensu o rozdělení celkového zisku mezi všemi hráči, nekooperativní hry můžeme vnímat jako individuální hledání nejvhodnějších strategií jednotlivými hráči za známých podmínek (zisky při volbách všech hráčů). Většinou neexistuje mechanismus, jenž by dokázal vynutit spolupráci, a pokud ano, hráči jej budou respektovat pouze tehdy, pokud to pro ně bude výhodné. Hráč tedy na spolupráci nemají přílišný zájem a každý hraje čistě za sebe tak, aby „urval co největší sousto“.

2.2.3 Symetrické vs. asymetrické hry

Symetrickou hru můžeme vnímat jako hru, kde jsou podmínky pro všechny hráče naprosto stejné. Definuje se jako hra, ve které může být libovolně změněno pořadí hráčů a užitek při tom zůstane pro jakoukoliv volbu strategií stejný.

Asymetrické hry tuto vlastnost nesplňují.

2.2.4 Simultánní vs. sekvenční (dynamické) hry

U simultánních her hráči volí svou strategii bez znalosti volby ostatních, kdežto u her sekvenčních nějakou informaci někteří hráči mají (ne nutně přesné tahy zvolené hráči před nimi).

Tento rozdíl se většinou projevuje i v zápisu, kdy pro hry simultánní se používá normální forma a pro hry sekvenční extenzivní forma.

2.3 Nekooperativní hra v normální formě

Definujme si formálně některé důležité pojmy, a posléze nekooperativní hru n hráčů v normální formě jako celek.

2.3.1 Hráči

Nechť Q je množina všech hráčů dané hry a $n = |Q|$ je počet hráčů. Většinou budeme označovat hráče přirozenými čísly, tedy $Q = \{1, 2, ..., n\}$.

2.3.2 Strategie

Nechť S_p ($p \in Q$) je množina ryzích strategií (někdy nazývána také množinou akcí [20])dostupných hráči p. Ryzí strategie (*pure strategy*) hráče p specifikuje jednu z možností chování hráče při hraní hry, jednoznačně určuje, kterou strategii zvolí. Pokud dále budeme hovořit o strategiích bez dalších upřesnění, budeme mínit právě strategie ryzí.

Mějme $|S_p| = k$. Smíšená strategie (*mixed strategy*) hráče p je pak k-tice obecně reálných čísel, $x = (x_1, \ldots, x_k)$ taková, že platí

$$\sum_{j=1}^{k} x_j = 1 \tag{2.1}$$

$$\forall j \in \{1, \ldots, k\} : 0 \leq x_j \leq 1. \tag{2.2}$$

Intuitivně lze smíšenou strategii také vnímat jako rozdělení pravděpodobnosti nad jednotlivými strategiemi ryzími.

Ryzí strategii můžeme tím pádem považovat za speciální případ strategie smíšené (pokud je pravděpodobnost hraní jedné ryzí strategie rovna jedné a pravděpodobnost hraní všech ostatních nulová).

Nazvěme

$$S = \prod_{i \in Q} S_i = S_1 \times S_2 \times ... \times S_n \tag{2.3}$$

strategickým prostorem hry a jeho prvky $s \in S$ strategickými profily (nebo také jen profily).

Často se nám ještě bude hodit značení

$$s_{-p} = \prod_{i \in Q, i \neq p} s_i, \tag{2.4}$$

jež značí kartézský součin strategií všech hráčů kromě p-tého ve strategickém profilu s. Intuitivně na tento součin můžeme pohlížet jako na kontext, v němž se p-tý hráč rozhoduje. Dále nechť (s_p^k, s_{-p}) značí strategický profil, který vznikne z profilu s náhradou strategie

p-tého hráče za s_p^k. Takto budeme značit situaci, kdy se hráč p odchýlí od strategického profilu tím, že hraje strategii s^k.

Označme dále $S_{-p} = \bigcup s_{-p}$ množinu situací, které mohou pro hráče p při rozhodování nastat.

2.3.3 Funkce užitku

Nechť u_p je funkce užitku hráče p, $u_p : S \to \mathbb{R}$. Tato funkce udává užitek (také nazývaný zisk) hráče p v závislosti na nastálém strategickém profilu (tedy v závislosti na strategiích zvolených jím i všemi ostatními hráči).

Označme pro jednoduchost n-tici funkcí užitku jako

$$U = (u_1, \ldots, u_n). \tag{2.5}$$

2.3.4 Definice hry

Nekooperativní hru v normální formě pak můžeme definovat jako trojici $G = (Q, S, U)$.

Definice 1 *Nekooperativní hru v normální formě definujeme jako trojici*

$$G = (Q, S, U), \tag{2.6}$$

kde

1. *Q je množina hráčů,*
2. *S je strategický prostor hry a*
3. *U je n-tice funkcí užitku.*

2.4 Ekvilibrium

Máme-li zadánu hru, například jako model reálné situace, jistě bychom rádi předpověděli, jak se jednotliví hráči budou v této situaci chovat, neboli hru vyřešili. Abychom to dokázali, musíme se rozhodnout, jakými principy se bude řídit rozhodování hráčů.

V teorii her se předpokládá racionálnost všech hráčů (někdy nazývaná racionálností voleb hráčů [20]). Ta říká, že každý hráč bude hrát tak, aby maximalizoval svůj užitek (nebude volit tak, aby jeho užitek byl menší než při jiné volbě), přičemž je schopen provádět jakkoliv složité úvahy, aby tohoto cíle dosáhl.

I při racionálnosti hráčů existuje mnoho přístupů, jak modelovat jejich rozhodování. Musíme si uvědomit, že volba přístupu je velmi podstatná záležitost a může silně ovlivnit nalezený výsledek.

Koncept ekvilibria představuje nejčastější způsob řešení matematických her. Ekvilibrium obecně značí stav systému, ve kterém jsou všechny síly v rovnováze. V kontextu her to můžeme vnímat jako situaci, v níž všichni hráči dosáhli bodu, kdy nemůžou dále zvýšit svůj užitek.

Představíme si několik nejznámějších ekvilibrií užívaných pro řešení nekooperativních her.

2.4.1 Nashovo ekvilibrium

Definice 2 *Strategický profil* $s^* = (s_1^*, \ldots, s_n^*)$ *je Nashovým ekvilibriem v ryzích strategiích hry* G*, pokud platí, že*

$$\forall p \in Q : \forall s' \in S_p : u_p(s_p^*) \geq u_p(s', s_{-p}^*). \tag{2.7}$$

Neformálně řečeno, Nashovo ekvilibrium je strategický profil, ve kterém si žádný hráč nemůže polepšit, pokud změní svou strategii a ostatní zůstanou u svých. Jinak řečeno, strategie hráče je nejlepší odpovědí (*best-response*) na strategie ostatních hráčů v tomto ekvilibriu.

Musíme si uvědomit, že v žádném případě nezaručuje hráčům nejlepší možné zisky. Pokud se podíváme například na jednu z nejznámějších her, Vězňovo dilema (2.2), jejímž Nashovým ekvilibriem je profil (Kolaborovat, Kolaborovat), vidíme, že by se dotyční hráči mohli mít mnohem lépe v případě, že by oba „zatloukali“. Avšak možnost zrady je příliš lákavá, při zatloukání si může kterýkoliv hráč polepšit kolaborací, a proto tento profil i přes lepší užitky Nashovým ekvilibriem *není*.

Vězňovo dilema	**Zatloukat**	**Kolaborovat**
Zatloukat	-1, -1	-10, 0
Kolaborovat	-10, 0	-5, -5

Tabulka 2.2: Vězňovo dilema – dva muži jsou zatčeni za zločin, nicméně policie nemá dostatek důkazů pro to, aby byli za tento zločin odsouzeni. Jsou drženi v oddělených celách a každý dostane nabídku: pokud bude svědčit proti druhému a ten bude zapírat, bude svědek propuštěn a kolega zavřen na 10 let. Pokud budou oba zatloukat, budou odsouzeni za drobný zločin na 1 rok. V případě, že by oba chtěli svědčit proti druhému, dostanou 5 let žaláře.

2.4.2 Podherně dokonalé ekvilibrium

Podherně dokonalé ekvilibrium, někdy také nazývané podherně dokonalé Nashovo ekvilibrium (*subgame perfect (Nash) equilibrium*), vzniklo zpřísněním podmínek Nashova ekvilibria pro sekvenční hry.

Toto ekvilibrium požaduje, aby strategie byly navíc v ekvilibriu i pro všechny podhry, a eliminuje tak možnost jakýchsi nelogických hrozeb, při kterých jsou strategie v ekvilibriu v určitou chvíli iracionální.

2.4.3 Korelované ekvilibrium

Korelované ekvilibrium navrhl v roce 1974 Robert Aumann. Na rozdíl od předchozích dvou, ve kterých neexistoval žádný mechanismus synchronizace voleb hráčů, u tohoto všichni hráči zpozorují signál, který jim určí, kterou strategii použít [1].

Zmíněný signál tedy v podstatě vybere strategický profil, který se bude hrát, a rozešle příslušné strategie příslušným hráčům. Ekvilibrium potom tkví v pravděpodobnosti výběru jednotlivých profilů – pro všechny hráče nesmí existovat výhodnější strategie než obdržená v signálu.

Definice 3 *Mějme hru $G = (Q, S, U)$. Rozdělení pravděpodobnosti P nad strategickými profily S je korelovananým ekvilibriem právě tehdy, když*

$$\forall p \in Q : \forall s^i, s^j \in S_p : \sum_{s_{-p} \in S_{-p}} P(s^i, s_{-p}) \cdot u_p(s^i, s_{-p}) \geq \sum_{s_{-p} \in S_{-p}} P(s^i, s_{-p}) \cdot u_p(s^j, s_{-p}), \quad (2.8)$$

kde $P(s)$ je pravděpodobnost hraní profilu s při rozdělení pravděpodobnosti P.

Nashovo vs. korelované ekvilibrium

Korelované ekvilibrium představuje obecnější koncept, než Nashovo ekvilibrium (ve smíšených strategiích) [6]. Platí, že každé Nashovo ekvilibrium (ve smíšených strategiích) je zároveň korelovaným ekvilibriem (po nezbytném přepočtu pravděpodobností hraní strategií na pravděpodobnosti hraných profilů). Výhodou korelovaného ekvilibria je rovněž nižší výpočetní složitost jeho hledání [19].

Korelované ekvilibrium umožňuje, aby hráči dosáhli celkově většího zisku než u ekvilibria Nashova. Důvodem je existence důvěryhodného synchronizačního mechanismu (zmíněného signálu), jenž může částečně eliminovat nutnost vysoké podezíravosti hráčů vůči sobě navzájem a z toho plynoucí hraní „bezpečnějších“ strategií, které nepřinesou tak vysoký užitek, ale hráčům dostatečně „kryjí záda“.

Vezměme známou hru „Chicken“ jako příklad (viz tabulku 2.3). V této hře existují tři Nashova ekvilibria: dvě ryzí (Uhnout, Neuhnout) a (Neuhnout, Uhnout) a jedno smíšené, ve kterém každý z hráčů neuhne s pravděpodobností 1/3 [9]. Můžeme spočítat, že průměrný užitek hráčů je 4,5 u ryzích a $32/9 \doteq 3{,}55$ u smíšeného ekvilibria.

Hra „Chicken“	**Neuhnout**	**Uhnout**
Neuhnout	0, 0	7, 2
Uhnout	2, 7	6, 6

Tabulka 2.3: Hra „Chicken“ – dva hráči se proti sobě rozjedou v autech; pokud jeden z nich uhne druhému, je nazván zbabělcem (v originále „chicken“, odtud název hry), když neuhne žádný, zabijí se.

Nyní vezměme korelované ekvilibrium, ve kterém se bude hrát profil (Neuhnout, Neuhnout) s pravděpodobností 1/2 a profily (Uhnout, Neuhnout), (Neuhnout, Uhnout) s pravděpodobností 1/4. Průměrný užitek pak bude $1/4 \cdot 7 + 1/4 \cdot 2 + 1/2 \cdot 6 = 5{,}25$.

Vidíme, že průměrný užitek hráče je v tomto korelovaném ekvilibriu větší, než v ekvilibriích Nashových.

Ještě si pro jistotu ověřme, že toto rozdělení pravděpodobnosti je skutečně korelovaným ekvilibriem:

1. Hráči přijde signál *Uhnout* – pokud se jím bude řídit, jeho průměrný užitek bude $1/3 \cdot 2 + 2/3 \cdot 6 = 14/3$, pokud ne, zisk bude stejný ($7 \cdot 2/3 + 0 \cdot 1/3 = 14/3$).

2. Hráči přijde signál *Neuhnout* – pokud se jím bude řídit, jeho průměrný užitek bude 7, pokud ne, získá průměrně 6.

Vidíme, že podmínka (2.8) platí, dané rozdělení je tedy korelovaným ekvilibriem.

2.5 Dominance strategií a redukce velikosti hry

V této části se budeme zabývat využitím racionality hráčů a principu dominance strategií ke zmenšení velikosti hry.

2.5.1 Striktní a slabá dominance

Definice 4 *Říkáme, že strategie s_p^j hráče p* striktně dominuje *jeho strategii s_p^k právě tehdy, když*

$$\forall s_{-p} \in S_{-p} : u_p(s_p^j, s_{-p}) > u_p(s_p^k, -p). \tag{2.9}$$

Strategii s_p^j pak nazýváme striktně dominantní *a strategii s_p^k* striktně dominovanou.

Neformálně řečeno, strategie je striktně dominována v případě, že existuje jiná strategie, která *v každé situaci* přináší hráči větší zisk.

Definice 5 *Strategie s_p^j hráče p slabě dominuje strategii s_p^k právě tehdy, když*

$$\forall s_{-p} \in S_{-p} : u_p(s_p^j, s_{-p}) \geq u_p(s_p^k, -p), \tag{2.10}$$

a zároveň

$$\exists s_{-p} \in S_{-p} : u_p(s_p^j, s_{-p}) > u_p(s_p^k, -p). \tag{2.11}$$

Strategii s_p^j nazýváme slabě dominantní *a strategii s_p^k* slabě dominovanou.

2.5.2 Eliminace dominovaných strategií

V této části se seznámíme s důležitou technikou používanou pro „zmenšení“ rozsáhlé hry, kterou chceme vyřešit.

Racionalita hráčů

Za předpokladu racionálních hráčů, který jsme si vyslovili, je jasné, že žádný hráč nebude hrát striktně dominovanou strategii – v každé situaci, kdy by měl hrát dominovanou strategii, si může polepšit tím, že zvolí strategii dominantní.

Redukce velikosti hry

Předpoklad racionality všech hráčů nám může velmi výrazně pomoci zredukovat velikost hry, kterou chceme vyřešit.

Vezměme si například hru v tabulce 2.4. Pokud se na ni zaměříme podrobněji, zjistíme, že strategie A řádkového hráče je striktně dominována strategií B – ať zvolí sloupcový hráč jakkoliv, vždy má řádkový hráč větší užitek z hraní B ($2 > 1 \wedge 3 > 2$). Oba hráči si tohoto faktu jsou vědomi, proto řádkový hráč nikdy nebude hrát strategii A a sloupcový s tím počítá. Strategie A nemá vliv na výsledek a můžeme ji s klidem v duši vypustit.

Hra se nám zmenšila do podoby zachycené v tabulce 2.5. V této chvíli řádkový hráč nemá žádnou dominantní strategii, nicméně sloupcový ano – strategie D striktně dominuje strategii E.

Všimněme si, že strategie E v původní hře rozhodně nebyla dominovanou, v případě, kdy by řádkový hráč hrál strategii A, by sloupcovému přinesla větší užitek. Možnost její

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

Tabulka 2.4: Příklad redukce hry eliminací striktně dominovaných strategií (krok 1)

	D	E
B	2, 2	3, 1
C	3, 4	2, 3

Tabulka 2.5: Příklad redukce hry eliminací striktně dominovaných strategií (krok 2)

eliminace vyplynula až z faktu, že racionální řádkový hráč svou strategii A hrát nikdy nebude.

Můžeme tedy hru dále zredukovat, a to do podoby zachycené v tabulce 2.6.

Předchozí eliminací jsme způsobili to, že nyní (tabulka 2.6) je strategie B řádkového hráče striktně dominována, opět tedy můžeme hru redukovat. Výslednou hru již nebudeme uvádět v tabulce, považujeme to za zbytečné.

	D
B	2, 2
C	3, 4

Tabulka 2.6: Příklad redukce hry eliminací striktně dominovaných strategií (krok 3)

Dostali jsme ekvivalentní hru, která namísto původních šesti strategických profilů obsahuje profil jediný. Vidíme, že se nám situace s ohledem na řešení hry značně zjednodušila a nalezení ekvilibria v této hře bude o dost snadnější – při jednom zbylém profilu velmi triviální.

Velmi důležitým poznatkem je, že redukce stavového prostoru hry na základě eliminace dominovaných strategií je ve své podstatě procesem iterativním, neboť tím, že ze hry odstraníme dominovanou strategii, se může jiná strategie jiného hráče stát dominovanou, ačkoliv jí zprvu nebyla.

2.5.3 Common knowledge

Jde o důsledek toho, že první hráč ví, v jakých podmínkách se rozhoduje druhý. Navíc tento druhý ví, že první ví, v jakých podmínkách se rozhoduje, a tak bychom mohli pokračovat do nekonečna. Této situaci říkáme *common knowledge* (zjednodušeno na dvouhráčovou hru).

Kapitola 3

Analýza východisek návrhu knihovny

V této kapitole se zaměříme na důležité aspekty návrhu implementovaných nástrojů. Budeme se zabývat jak teoretickými základy některých problémů, tak i některými ryze praktickými záležitostmi.

Naším úkolem je navrhnout knihovnu, která bude schopna v „rozumném“ čase vyřešit velmi rozsáhlé hry. Zaměříme se na výpočet korelovaného ekvilibria, které podle [14] představuje vhodný koncept řešení her, souvisejících s chováním hráčů (ekonomických subjektů) v prostředí tržní ekonomiky.

3.1 Lineární programování

Lineární programování (*linear programming*) je optimalizační technika, která hledá extrém lineární funkce při dodržení jistých omezení. Patří do skupiny metod používaných v oblasti operačního výzkumu (*operations/operational research*).

Lineárním programováním se zde zabýváme proto, že velmi úzce souvisí s metodou hledání korelovaného ekvilibria s pomocí takzvané G-matice (viz 3.2).

Programování Termín „programování“ se v tomto kontextu používá spíše z historických důvodů, s programováním číslicových počítačů, jak jej chápeme dnes, nemá pranic společného. Jde o programování ve smyslu „plánování“, kdy se například problém co nejefektivnější výroby převede na LP problém a výsledkem je jakýsi „plán“, kolik kterých výrobků vyprodukovat.

3.1.1 Problém lineárního programování (LP problém)

Lineární programování hledá extrém (maximum nebo minimum) lineární funkce n proměnných (kriteriální funkce) při dodržení m omezení ve formě lineárních rovnic nebo nerovnic [8]. LP problém se dvěma proměnnými a dvěma omezeními může vypadat například takto:

$$\begin{aligned} \text{Maximalizujte} \quad & 3x - 5y \\ \text{při dodržení} \quad & x + y \leq 5 \\ & -x + 2y \geq 12 \\ & 5x - 6y = 23 \end{aligned} \tag{3.1}$$

Řešením LP problému je vektor hodnot proměnných v kriteriální funkci; obecně řešením nazýváme jakýkoliv takovýto vektor.

Pokud dané řešení splňuje všechna omezení, jde o tzv. *přípustné řešení* (*feasible solution*). Pokud navíc neexistuje přípustné řešení s větší hodnotou kriteriální funkce, jde o řešení *optimální*.

Existují dvě nejznámější třídy algoritmů pro řešení LP problémů, a to algoritmy založené na simplexové metodě, které fungují na principu procházení po hranách konvexního mnohostěnu ohraničujícího prostor přípustných řešení (řešení splňující všechny omezení), a metody založené na vnitřních bodech (*interior point methods*), které procházejí vnitřkem tohoto mnohostěnu. Oba způsoby v současné době dosahují obecně srovnatelných výsledků [8]. Dále se zaměříme na simplexovou metodu z důvodu její relativní jednoduchosti a dobré zdokumentovanosti.

3.1.2 Simplexová metoda

Tato metoda, kterou v roce 1947 vytvořil George Dantzig, využívá konvexnosti prostoru přípustných řešení k jejich jednoduchému procházení (přičemž se kvalita aktuálního řešení nezhoršuje) a testování na lokální/globální optimum. Hledá optimum tak, že prochází prostor přípustných řešení daného LP problému (tj. řešení splňujících všechna omezení) po hranách konvexního mnohostěnu, který jej ohraničuje [2].

Dále budeme přepokládat, že řešíme maximalizační problém. Můžeme si to dovolit bez újmy na obecnosti, neboť každý minimalizační LP problém můžeme jednoduše převést na problém maximalizační (prostou inverzí koeficientů kriteriální funkce).

V každém vrcholu se algoritmus musí rozhodnout, kam pokračovat dále. Směr se volí tak, aby pokud možno rostla a rozhodně neklesala. V určitých případech, pokud narazíme na takzvané *degenerované* řešení, se může stát, že se při kroku metody hodnota nezmění.

Vzhledem k linearitě kriteriální funkce i jednotlivých omezení (a z toho vyplývající konvexnosti prostoru) má řešení LP problému důležitou vlastnost: Každé lokální optimum je zároveň optimem globálním [12]. Díky tomu můžeme poměrně jednoduše zjistit, že se nacházíme ve vrcholu reprezentujícím optimální řešení; v optimu jsme, pokud všechny hrany vedoucí z příslušného vrcholu vedou ke snížení hodnoty kriteriální funkce – v tom případě jsme v lokálním (a pro LP problém tedy i v globálním) maximu.

Simplexová tabulka

Geometrický pohled na řešení je sice použitelný k vysvětlení základních principů díky své názornosti, nicméně pro počítání rozsáhlejších problémů na počítači příliš užitečný není. Mnohem lepší je „algebraický“ přístup, kdy se místo pohybu v n rozměrném prostoru ohraničeném hyperrovinami zaměříme na operace lineární algebry s maticí reprezentující aktuální stav řešení. Proto se většinou používají různé modifikace takzvané *simplexové tabulky* (*simplex tableau/table*), jež reprezentuje problém a jeho aktuální řešení (vrchol konvexního mnohostěnu) ve formě dvourozměrné tabulky.

Uvažujme standardní tvar LP problému:

$$\begin{aligned} \text{Maximalizujte funkci} \quad & \mathbf{c}^{\mathrm{T}} \cdot \mathbf{x} \\ \text{při dodržení} \quad & \mathbf{A} \cdot \mathbf{x} = \mathbf{b} \\ & \mathbf{x} \geq \mathbf{0} \end{aligned} \tag{3.2}$$

Při řešení se většinou používá tabulka podobná této:

$$\begin{pmatrix} Z & -c_{1,1} & \cdots & -c_{1,n} & 0 & \cdots & 0 \\ b_{1,1} & a_{1,1} & \cdots & a_{1,n} & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ b_{m,1} & a_{m,1} & \cdots & a_{m,n} & 0 & \cdots & 1 \end{pmatrix} \tag{3.3}$$

Z představuje aktuální hodnotu kriteriální funkce. Čísla prvním řádku $c_1, \ldots c_n$ začínají například na hodnotách opačných ke koeficientům kriteriální funkce a v průběhu výpočtu udávají takzvané *relativní ceny* (*relative costs*) – ty říkají, jak strmě se budeme blížit optimu, pokud se začneme pohybovat po příslušné hraně. První, nejlevější sloupec $(b_1, \ldots b_n)$ udává hodnoty pravých stran jednotlivých omezení a $a_{r,1}, \ldots a_{r,n}$ jsou koeficienty jednotlivých proměnných v r-tém omezení. Jednotková matice na pravé straně obsahuje pomocné proměnné.

Jeden krok řešení pak probíhá takto:

1. Test aktuálního řešení na optimalitu (případně neomezenost) – v obou případech algoritmus končí
2. Určení proměnné, která do báze vstupuje (sloupec tabulky)
3. Určení proměnné, která vystupuje z báze (řádek tabulky)
4. Modifikace tabulky podle vybraného pivota (prvku v určeném sloupci a řádku)

Jednotlivé varianty metody se v přesném způsobu provádění těchto kroků liší, nicméně struktura zůstává stejná či velmi podobná.

„Naivní“ simplexová metoda

Nejjednodušším řešením, vhodným například pro počítání na papíře, je tato metoda, založená na Gauss-Jordanově eliminaci:

1. Hledáme zápornou hodnotu v prvním řádku (mimo Z). Pokud jsou všechny prvky kladné, dosáhli jsme optimálního řešení.
2. Vybereme sloupec takový, že má na prvním místě záporné číslo (označme jej j).
3. Pro určení řádku postupujeme tak, že hledáme takové i, pro které je výraz $b_i/a_{i,j}$ minimální, a zároveň $a_{i,j} > 0$, tedy:

$$i = \underset{i=1,\ldots,m|a_{i,j}>0}{\arg\min} \frac{b_i}{a_{i,j}} \tag{3.4}$$

   Pokud takové i neexistuje, našli jsme neomezené řešení.
4. Vybraným pivotem je pak prvek $a_{i,j}$. Provedeme Gauss-Jordanovu eliminaci.

Tuto metodu není možné použít vždy. Většinou se používá na LP problémy v určitém omezeném tvaru, například standardní maximalizační problém.

Standardní maximalizační problém Jde o LP problém, pro který platí, že všechna omezení jsou ve tvaru:

$$A_i \cdot x^T \leq b_i, \tag{3.5}$$

kde A_i značí i-tý řádek matice A, a zároveň platí

$$\forall i \in \{1, \ldots, n\} : b_i \geq 0. \tag{3.6}$$

Výhodou tohoto tvaru je, že máme okamžitě k dispozici přípustné řešení, se kterým můžeme začít, a sice vektor 0^n. Nevýhodou je, že v tomto tvaru lze zapsat jen omezenou podmnožinu LP problémů.

Dvoufázová simplexová metoda (*two-phase simplex method*)

Jednou z věcí, kterou naivní metoda neřešila, je nalezení počátečního přípustného řešení. Tento nedostatek lze odstranit pomocí určitého předzpracování problému. Existují různé takovéto postupy, například:

- Metoda velkého M (*Big-M method*) – zavádí takzvané umělé proměnné (*artificial variables*), kterým přiřazuje „velkou“ hodnotu kriteriální funkce tak, aby zajistila, že při dosažení optima budou tyto proměnné nulové.
- Dvoufázová metoda (*two-phase method*) – také využívá umělé proměnné. Zadanou kriteriální funkci v první fázi nahradí takovou funkcí, aby našla řešení, u kterého bude hodnota umělých proměnných rovna nule. Až jej najde, použije původní kriteriálni funkci a s ní dospěje k optimálnímu řešení.
- Kolizní metody (*crash methods*) – heuristické metody, které se rovněž snaží rychle přiblížit k optimu [4].

Popišme si detailněji dvoufázovou metodu (dle [2]). Předpokládejme bez újmy na obecnosti, že

$$\forall i \in \{1, \ldots, m\} : b_i \geq 0. \tag{3.7}$$

Pokud by toto nebylo splněno, můžeme příslušné omezení vynásobit -1.

Úlohu nyní převedeme do tvaru, kdy v ní nebudou nerovnosti. Docílíme toho přidáním doplňkových (přídatných, *slack/surplus variables*) proměnných. Například problém

$$\begin{aligned} \text{Maximalizujte} \quad & c_1 \cdot x_1 + \cdots + c_n \cdot x_n \\ \text{při dodržení} \quad & a_{1,1} \cdot x_1 + \cdots + a_{1,n} \cdot x_n \leq b_1 \\ & a_{2,1} \cdot x_1 + \cdots + a_{2,n} \cdot x_n \geq b_2 \\ & a_{3,1} \cdot x_1 + \cdots + a_{3,n} \cdot x_n = b_3 \end{aligned} \tag{3.8}$$

přepíšeme do tohoto tvaru:

$$\begin{aligned} \text{Maximalizujte} \quad & c_1 \cdot x_1 + \cdots + c_n \cdot x_n \\ \text{při dodržení} \quad & a_{1,1} \cdot x_1 + \cdots + a_{1,n} \cdot x_n + s_1 = b_1 \\ & a_{2,1} \cdot x_1 + \cdots + a_{2,n} \cdot x_n - s_2 = b_2 \\ & a_{3,1} \cdot x_1 + \cdots + a_{3,n} \cdot x_n = b_3 \end{aligned} \tag{3.9}$$

Následně přidáme umělou proměnnou pro každé omezení, které nebylo omezením na menší nebo rovno:

$$\begin{aligned} \text{Maximalizujte} \quad & c_1 \cdot x_1 + \cdots + c_n \cdot x_n \\ \text{při dodržení} \quad & a_{1,1} \cdot x_1 + \cdots + a_{1,n} \cdot x_n + s_1 = b_1 \\ & a_{2,1} \cdot x_1 + \cdots + a_{2,n} \cdot x_n - s_2 + a_1 = b_2 \\ & a_{3,1} \cdot x_1 + \cdots + a_{3,n} \cdot x_n + a_2 = b_3 \end{aligned} \tag{3.10}$$

Originální kriteriální funkci změníme tak, aby pouze minimalizovala umělé proměnné:

$$\begin{aligned} \text{Maximalizujte} \quad & -a_1 - a_2 \\ \text{při dodržení} \quad & a_{1,1} \cdot x_1 + \cdots + a_{1,n} \cdot x_n + s_1 = b_1 \\ & a_{2,1} \cdot x_1 + \cdots + a_{2,n} \cdot x_n - s_2 + a_1 = b_2 \\ & a_{3,1} \cdot x_1 + \cdots + a_{3,n} \cdot x_n + a_2 = b_3 \end{aligned} \tag{3.11}$$

Pokud se nám tento problém podaří vyřešit tak, aby zároveň hodnota kriteriální funkce byla rovna nule, našli jsme přípustné řešení původního problému. Můžeme nyní použít původní kriteriální funkci a dospět k optimálnímu řešení. Pokud by kriteriální funkce nebyla rovna nule, pak problém nemá přípustné řešení.

Modifikovaná simplexová metoda (*revised simplex method*)

Naivní simplexová metoda má ještě jednu nevýhodu, a tou je nutnost uchovávat v každé iteraci celou simplexovou tabulku. To ale není nutné a v praktických implementacích se proto používá modifikovaná simplexová metoda.

Udržují se pouze koeficienty nebázových proměnných (Bázové proměnné jsou ty, které ve svém sloupci tabulky mají jednu jedničku a jinak samé nuly. Ostatní proměnné jsou nebázové.) v aktuálním prvním řádku tabulky, sloupec tabulky odpovídající vybranému sloupci (příslušející proměnné vstupující do báze) a vektor pravých stran.

Aby bylo možné potřebné informace spočítat, udržujeme si takzvanou inverzní matici báze (B^{-1}). Ta se musí po každém kroku aktualizovat. Pro další detaily týkající se tohoto procesu doporučujeme například [12] nebo [2].

Degenerace a zacyklení

Degenerované řešení je takové, u něž mají některé bázové proměnné nulovou hodnotu. Takováto řešení pro nás nejsou příjemná z toho důvodu, že výstupu takovéto proměnné z báze se nemění hodnota kriteriální funkce (při geometrickém pojetí vlastně zůstáváme v jednom bodě, nedochází k posunu).

Nepříjemnost ani tak nespočívá v tom, že se v daném kroku nepřiblížíme k optimu, ale hlavně v možnosti zacyklení. Může se totiž klidně stát například toto:

1. Do báze vstupuje x_1, vystupuje x_2.
2. Do báze vstupuje x_3, vystupuje x_1.
3. Do báze vstupuje x_2, vystupuje x_3.
4. Do báze vstupuje x_1, vystupuje x_2 (znovu).

Reálné LP problémy bohužel ukazují, že degenerovaná řešení jsou spíše pravidlem nežli výjimkou [10]. Ačkoliv degenerovaná řešení nutně nevedou k zacyklení, byly vyvinuty techniky, jak mu zabránit. Uveďme si alespoň jednu z nich.

Blandovo pravidlo (Bland's rule)

1. Vstupující proměnnou nechť je proměnná odpovídající nejlevějšímu sloupci se zápornou relativní cenou.

2. Vystupující proměnnou nechť je proměnná odpovídající řádku splňujícímu podmínku (3.4). Pokud takových proměnných existuje více, vystupující se stává proměnná s nejnižším indexem.

Je dokázáno, že při použití tohoto pravidla simplexová metoda nemůže cyklit (pro důkaz viz například [10]).

Techniky používané v praxi

V praxi se používají některé další techniky (například kvůli zlepšení výkonu nebo numerické stability), o kterých se zmíníme v této části. Týkají se hlavně inverzní matice báze (B^{-1}).

Reinverze báze (*reinversion*) Jak jsme si již zmínili, u modifikované simplexové metody si udržujeme inverzní matici báze, kterou v každém kroku aktualizujeme. Tyto časté aktualizace s sebou ovšem přinášejí numerické chyby (vlivem zaokrouhlování). Proto se v praxi při řešení čas od času aktuální inverzní matice báze zahodí a vypočítá znova. Ačkoliv je inverze výpočetně náročná operace, lze efektivitu zvýšit použitím vhodných datových struktur a speciálních algoritmů [2].

Uložení inverzní matice báze ve formě součinu elementárních matic Většinou se také inverzní matice báze neukládá explicitně, ale ve formě určitého vhodného součinu. Při aktualizaci totiž dochází k tomu, že je B^{-1} (zleva) násobena velmi řídkou maticí (nazvěme ji Q). Matici B^{-1} po více krocích získáme součinem $Q_n \cdot \cdots \cdot Q_1 \cdot B^{-1}$ [2]. Dokonce se někdy používá i technika, kdy si matici Q rozložíme na součin takzvaných elementárních matic, které se liší od jednotkové matice jen v jednom prvku [22].

LU dekompozice inverzní matice báze Jinou používanou technikou je LU dekompozice, která B^{-1} ukládá jako součin $L \cdot U$, kde L, respektive U je dolní, respektive horní trojúhelníková matice. Různé varianty tohoto přístupu jsou popsány například v [17].

Dostupné nástroje pro řešení LP problémů

Lineární programování má tak široké použití v praxi, že existuje mnoho nástrojů pro řešení LP problémů. Pokud se zaměříme na volně dostupné implementace, asi nejznámějšími zástupci jsou knihovna GLPK (součást projektu GNU) a CLP (součást projektu COIN-OR) [11].

Tyto knihovny nabízejí implementaci základních metod pro řešení LP problémů zároveň se zdrojovým kódem. Bohužel při jejich návrhu nebylo počítáno s možností paralelizace výpočtů.

Existuje rovněž množství komerčního software. U něj je situace s paralelizací lepší, zvládají ji například produkty OSL nebo CPLEX. Další výhodou komerčních solverů je větší rychlost oproti volně dostupným implementacím.

3.2 G-matice – struktura pro zobrazení a analýzu víceroz-měrných her

V článku [14] autor představuje takzvanou G-matici, zajímavou strukturu z hlediska pohledu na hru a redukce striktně dominovaných strategií.

Definice 6 *G-matice M n-hráčové hry G je částečná funkce*

$$M : (Q \times S_p \times S_p) \times S \to \mathbb{R} \tag{3.12}$$

taková, že platí:

$$M_{(p,s^f,s^t),(s_1,\ldots,s_n)} = \begin{cases} u_p(s) - u_p(s^t, s_{-p}) & \text{pro} \quad p \in Q, s^f = s_p \\ \underline{\text{nedef.}} & \text{jinak} \end{cases} \tag{3.13}$$

Sloupce této G-matice jsou označeny všemi strategickými profily hry a řádky všemi různými dvojicemi strategií jednotlivých hráčů (obecně trojicí $(\text{hr}, \text{strategie1}, \text{strategie2})$). Prvky matice jsou rozdíly zisků (užitků) při změně strategie (odpovídající označení řádku) při hraní daného strategického profilu (odpovídajícího označení sloupce):

$$G_{(p,sfrom,sto),s} = u_p(sfrom, s_{-p}) - u_p(sto, s_{-p}), \tag{3.14}$$

kde $(p, sfrom, sto)$ je identifikace řádku, s je strategický profil sloupce a u_p je užitková funkce hráče p.

Intuitivně lze říci, že řádek $(p, sfrom, sto)$ znázorňuje, jak výhodné je pro hráče p při různých strategických profilech hraní své strategie sfrom oproti hraní strategie *sto*.

Poznámka Prvky G-matice, pro které první strategie není hrána ve strategickém profilu sloupce, nemá smysl uvažovat. Při grafickém znázornění necháváme takovéto buňky prázdné a při výpočtu jejich obsah neuvažujeme.

Mějme například hru, na které jsme si ukazovali eliminaci striktně dominovaných strategií (tabulka 3.1). Příslušná G-matice pak bude vypadat, jak je uvedeno v tabulce 3.2.

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

Tabulka 3.1: Hra převáděná na G-matici 3.2

Uvědomme si, že pro větší hry je G-matice opravdu obrovská. Počet sloupců odpovídá počtu prvků množiny strategických profilů:

$$|S| = \prod_{p \in Q} |S_p|, \tag{3.15}$$

kde S_p je množina strategií hráče i, a počet řádků odpovídá součtu variací druhé třídy z S_p prvků bez opakování pro každého hráče p, tedy

$$\sum_{p \in Q} V_2(|S_p|) = \sum_{p \in Q} 2 \cdot C_2(|S_p|) = \sum_{p \in Q} 2 \cdot \binom{|S_p|}{2} = \sum_{p \in Q} \frac{|S_p|!}{(|S_p| - 2)!}. \tag{3.16}$$

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

Tabulka 3.2: G-matice pro hru 3.1

G-matice není užitečná jen proto, že dokáže zobrazit vícerozměrný prostor užitků jako dvourozměrnou tabulku. Pokud se blíže zamyslíme nad jejím obsahem, můžeme si uvědomit, že G-matice je jako stvořená pro analýzu striktní dominance strategií. Jestliže bude řádek $(p, sfrom, sto)$ obsahovat pouze záporná čísla, pak je strategie $sfrom$ striktně dominována strategií sto.

Redukce G-matice

Ještě před samotným převodem na problém lineárního programování lze s výhodou G-matici redukovat tak, že eliminujeme strategie příslušející řádkům, které obsahují pouze negativní (< 0) hodnoty (prázdné buňky zanedbáváme) – pak můžeme z G-matice odstranit veškeré řádky i sloupce obsahující eliminovanou strategii.

Ukažme si to na příkladu. Vidíme, že řádek **A→B** obsahuje pouze jedničky, můžeme jej tedy eliminovat, stejně jako i sloupce, obsahující strategii **A**. Výslednou G-matici vidíme v tabulce 3.3.

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

Tabulka 3.3: G-matice po prvním kroku eliminace

Nyní vidíme, že bychom mohli eliminovat stategii **E** (díky řádku **E→D**) a tento postup by nás stejnou cestou dovedl ke stejnému výsledku, jako eliminace dominovaných strategií – zbyl by pouze profil **CD**.

3.3 Převod G-matice na LP problém

V článku [14] autor navrhuje elegantní způsob, jak se dostat od zadané hry k problému lineárního programování právě s využitím *G-matice*.

Mějme G-matici o rozměrech $m \times n$. Způsob spočívá ve vytvoření LP problému, který má n proměnných (tedy tolik, kolik je ve (zredukované) matici strategických profilů) a $m+1$ omezení (čili jedno pro každý řádek plus jedno navíc). Řádek G-matice převedeme na omezení použitím obsahu každé neprázdné buňky v tomto řádku jako koeficientu příslušné

proměnné; takto vzniklý výraz položíme větší nebo roven nule. Například řádek **B**→**C** se transformuje na omezení

$$-p_{BD} + p_{BE} \geq 0. \tag{3.17}$$

Posledním omezením je požadavek, aby součet hodnot proměnných byl roven jedné. To je logické vzhledem k interpretaci proměnných jako pravděpodobností hraní daných strategických profilů, tento součet z principu nemůže být větší ani menší než jedna.

3.3.1 Interpretace omezení

Omezení (3.17) lze jednoduše přepsat jako

$$p_{BE} \geq p_{BD}; \tag{3.18}$$

pravděpodobnost hraní profilu **BD** musí být větší nebo rovna pravděpodobnosti hraní profilu **BE**. To je nutná podmínka pro to, aby bylo dané rozdělení pravděpodobnosti korelovaným ekvilibriem. Pokud by

$$p_{BE} < p_{BD}, \tag{3.19}$$

pak v případě, kdyby nezávislá strana vybrala podle daných pravděpodobností strategický profil obsahující strategii **B** (a tento hráči sdělila), by si hráč vedl lépe, kdyby neposlechl a namísto toho hrál strategii **C**.

3.3.2 Stanovení kriteriální funkce

Když nyní víme, jak transformovat řádky G-matice na příslušná omezení, chybí nám k úplnému LP problému již pouze vytvořit kriteriální funkci.

Pokud jde o typ hledaného optima, logicky zvolíme maximum, jelikož se snažíme o to, aby užitky všech hráčů byly co nejvyšší. Problémem ovšem zůstávají koeficienty kriteriální funkce.

Uvážíme-li interpretaci korelovaného ekvilibria, docházíme k tomu, že každé rozdělení, které splní vytvořená omezení, bude nutně korelovaným ekvilibriem. V reálném světě ovšem mají hráči tendenci rozhodovat se nejen na základě toho, aby dosáhli vyváženého stavu, ve kterém si nikdo nebude moci polepšit, ale chtějí rovněž globálně maximalizovat svůj zisk. Můžeme to ilustrovat například tak, že ekvilibrium, které každému hráči zaručuje zisk 100, je jakýmsi způsobem „lepší“ či atraktivnější než jiné, jež přinese každému užitek o velikosti 1.

Vhodně zvolená kriteriální funkce nás může pomoci nasměrovat na takováto „lepší“ ekvilibria. Obecně ale neexistuje mechanismus, jak její koeficienty nejvhodněji volit [14], už jen z toho důvodu, že je velmi složité a někdy až prakticky nemožné určit, co je nejlepší pro všechny. Celý problém hledání globálně optimálního ekvilibria svou výpočetní složitostí mnohokrát překračuje námi používaný jednoduchý způsob hledání „jakéhokoliv“ ekvilibria [19].

Existují dva logické a přiměřeně složité způsoby, jak tyto koeficienty zvolit:

1. Nejjednodušší cestou je vzít všechny koeficienty kriteriální funkce rovny jedné. Tím v podstatě rezignujeme na jakoukoliv možnost posuzovat „lákavost“ ekvilibrií v kontextu celé hry a soustřeďujeme se pouze na dodržení definice ekvilibria.

2. Pokud koeficienty určíme jako

$$c_s = \sum_{p \in Q} u_p(s), \tag{3.20}$$

kde s je strategický profil, Q je množina hráčů a u_p je užitková funkce p-tého hráče, zohledníme při rozhodování součet užitků všech hráčů při hraní daného strategického profilu. To nám může pomoci dát prioritu profilům, které jsou celkově výhodnější, ale v jistých situacích tento přístup nemusí být ideální. Pokud bychom měli například dva profily užitků, $(1000, 1, 1, 1, 1)$ a $(50, 50, 50, 50, 50)$, cítíme, že druhý je globálně jaksi „více žádoucí“, nicméně zvolený způsob dá přednost prvnímu.

Příklad transformace G-matici v tabulce 3.3 můžeme transformovat na tento LP problém (pomiňme nyní možnost, že bychom ji mohli dále redukovat):

$$\begin{aligned} \text{Maximalizujte} \quad & 4p_{BD} + 7p_{CD} + 4p_{BE} + 5p_{CE} \\ \text{při dodržení} \quad & -p_{BD} + p_{BE} \geq 0 \\ & p_{CD} - p_{CE} \geq 0 \\ & p_{BD} + p_{CD} \geq 0 \\ & -p_{BE} - p_{CE} \geq 0 \end{aligned} \tag{3.21}$$

3.4 Systém OpenMP

Jedním z požadavků zadání byla možnost paralelizace výpočtu. S ohledem na přenositelnost jsme zvolili systém OpenMP.

Jedná se o soubor direktiv překladače, knihovních funkcí a proměnnných prostředí ovlivňujících paralelní běh programu. Lze jej využít v programovacích jazycích Fortran a C/C++ a je přenositelný na mnoho různých platforem.

Použití OpenMP způsobí to, že jsou paralelizované části kódu namapovány na příslušný mechanismus cílové platformy. V případě GNU/Linuxu jsou tedy namapována na POSIXová vlákna, v MS Windows zase na nativní WinAPI vlákna.

Jednou z nejdůležitějších vlastností OpenMP je jednoduchost použití. Pokud tedy máme například cyklus `for`, u něhož chceme, aby běžel paralelně, můžeme jednoduše použít jedinou direktivu překladače (viz obrázek 3.1).

```
int n = array.size();
#pragma omp parallel for
for (int i=0; i<n; i++) {
  array[i] /= i+1;
}
```

Obrázek 3.1: Příklad použití OpenMP k paralelizaci cyklu `for` (jazyk C++)

Zároveň OpenMP poskytuje důležité mechanismy synchronizace přístupu ke sdíleným zdrojům, například kritickou sekci nebo semafory. Pro další detaily ohledně použití a výkonnosti OpenMP odkazujeme případné zájemce na knihu [3].

3.5 Způsob zadání hry

Nejjednodušším způsobem zadání hry v normální formě je pravděpodobně obyčejný soubor se serializovanou tabulkou, která reprezentuje hru v normální formě.

Tento přístup však může narazit v situaci, kdy chceme řešit tak velké hry, že se do paměti nevejdou, nebo celou tabulku aktuálně nemáme k dispozici.

Proto výsledná knihovna umožňuje specifikovat vstup alternativním způsobem. Je inspirován způsobem popsaným v článku [14], kde je navržen tzv. *cellModel*. Jde zjednodušeně o funkci, která ze vstupu, jímž je strategický profil, spočítá hodnotu užitkové funkce pro všechny hráče v případě, že je hrán právě tento profil.

Tento přístup může být výhodný například v situaci, kdy se nám povede zredukovat G-matici po načtení několika prvních řádků. V tom případě nebudeme ztrácet čas ani paměť opětovným počítáním strategických profilů, které se v korelovaném ekvilibriu nemohou vyskytnout.

Kapitola 4

Implementace knihovny pro hledání korelovaného ekvilibria

V této kapitole se zaměříme především na zevrubný popis nástroje CE solver jako celku, jakož i popis algoritmů a datových struktur v něm použitých. Kratší části (4.8, 4.7) věnujeme „vedlejším produktům“, jednodušším programům souvisejích s CE solverem, které jsme vytvořili.

Hlavní myšlenky efektivního hledání korelovaného ekvilibria byly nastíněny v článku [14], jenž se jako jeden z mála zabývá praktickou implementací. Tento článek byl naší hlavní inspirací, ačkoliv jsme navrhovaný přístup modifikovali.

Existují i další zajímavé metody, za všechny jmenujme algoritmus FDDS (*Fast Detection of Dominant Strategies*), který hry přímo neřeší, ale spíše procházením strategického prostoru objevuje jisté zajímavé strategie. Procházení začíná z několika víceméně náhodně zvolených strategických profilů, čímž se podobná například metodě Monte-Carlo (viz článek [13]).

Časovou složitostí některých problémů ve speciální tvaru se pak zabývá například článek [21].

CE-solver byl vytvořen s ohledem na co největší efektivitu výpočtu, a tedy schopnost řešení co nejrozsáhlejších her. Neméně podstatným cílem však bylo, aby výsledný produkt poskytoval co největší volnost ve způsobu zadání řešených her.

4.1 Hlavní myšlenky

CE-solver funguje na základech nastíněných v kapitole 3, tedy minimalizace G-matice a následné řešení LP problému. V této části popíšeme některé důležité vlastnosti nástroje, jakož i odchylky od návrhu řešení v článku [14] a zlepšení oproti exitující implementaci [15].

4.1.1 Vstup nástroje

S ohledem na co největší jednoduchost použití zároveň s možností analyzovat co nejširší škálu různých vstupů jsme zvolili jednoduchý a flexibilní způsob zadání hry: Uživatel vytvoří objekt, který je potomkem abstraktní třídy `Game` (pro podrobnosti viz 4.2), se kterým pak CE-solver pracuje.

Tato cesta potenciálně umožňuje zadání her, jejichž stavový prostor je větší než paměť počítače. Může to být užitečné například v situaci, kdy máme obrovskou hru a potřebné užitky dopočítáváme, až jsou potřeba (*on demand, lazy evaluation*).

Samozřejmě bychom si nemohli dovolit opomenout vstup ze souboru. Ten je realizován pomocí objektů `FileGame` a `FileSolver`, které načítají soubor na napojují ho na rozhraní CE-solveru.

4.1.2 Uložení G-matice

Článek [14] navrhuje přímočarý způsob reprezentace G-matice, a to uchování jakéhosi „dvojseznamu“ profilů pro každý řádek. V prvním seznamu jsou uchovávány strategické profily odpovídající pozitivním položkám G-matice v dotyčném řádku, ve druhém zase negativní.

Při eliminaci některé strategie pak dochází k průchodu všemi řádky a odebrání strategických profilů, které tuto strategii obsahují. Strategii lze eliminovat, pokud se u některého řádku vyprázdní pozitivní část seznamu.

Naproti tomu my jsme zvolili způsob, kdy si pro každý řádek ukládáme pouze profil, u kterého muselo skončit procházení. Řádek se prochází, dokud jsou buňky negativní, přičemž se přeskakují profily, které obsahují zakázané strategie. Tímto způsobem zmizela nutnost explicitně ukládat rozsáhlou G-matici, proto má CE-solver výrazně nižší paměťové nároky.

4.1.3 Zbytečné opakované počítání užitků

Může se stát, a v reálných modelech se často stává, že počítání užitků hráčů je výpočetně náročná operace, která trvá nezanedbatelnou dobu. Užitky v jednotlivých strategických profilech jsou přitom obecně potřeba vícekrát a nebylo by rozumné je počítat opakovaně, pokud se tomu můžeme vyhnout.

To lze zařídit tak, že spočítané užitky uložíme pomocí do speciální datové struktury, čímž zabráníme zbytečnému zpomalování. Tuto strukturu budeme v dalším textu nazývat „cache vypočtených užitků“ nebo zkráceně jen „cache“. Samozřejmě se může stát, získání potřebných užitků z modelu hry je velmi rychlé; potom můžeme cache „vypnout“ a vystříhat se tak zpomalení způsobeného zbytečným ukládáním.

4.1.4 Paralelizace

CE-solver jsme tvořili s důrazem na zrychlení výpočtu pomocí paralelizace. Tu využíváme při eliminaci řádků G-matice tak, že jednotlivá vlákna zpracovávají každé svůj řádek a ověřují, zda může být odpovídající strategie eliminována.

Pro paralelizaci jsme použili systém OpenMP, který důkladně popíšeme v části 4.4.

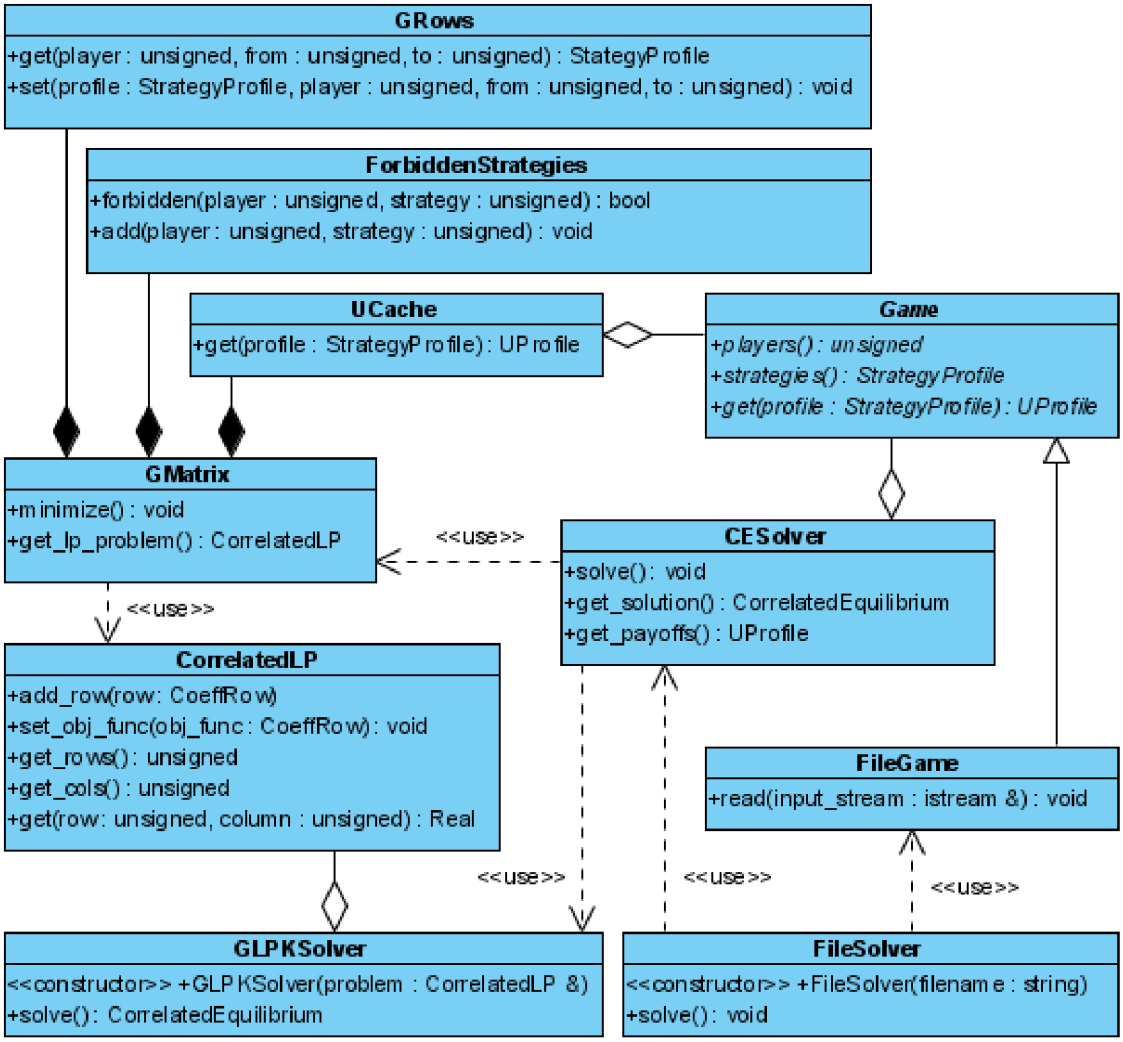
4.2 Datové struktury CE-solveru

V této části naznačíme, jaké třídy CE-solver používá a jak jsou spolu propojeny.

Schéma aplikace je zřejmé ze zjednodušeného diagramu tříd na obrázku 4.1. Ten by měl poskytnout základní představu o struktuře a zároveň usnadnit orientaci v popisu jednotlivých tříd.

Strategický profil – `StrategyProfile`

Tato jednoduchá struktura uchovává záznam o strategickém profilu. Z důvodu efektivity je implementována jako statické pole. Přístup k položkám je řešen operátorem `[]` (hranaté závorky), zároveň je naimplementován i operátor `<`, takže lze strategickými profily indexovat asociativní pole standardní knihovny STL `std::map`.



Obrázek 4.1: Diagram tříd CE-solveru

Profil užitků `UProfile` Profil užitků funguje na takřka totožném principu, pouze se místo indexů strategií ukládají užitky jednotlivých hráčů při hraní příslušného strategického profilu.

Například pro hru v tabulce 2.4 strategickému profilu CD, tedy `[2,0]` odpovídá profil užitků `[3,4]` (strategie hráčů indexovány od nuly, tedy pro řádkového hráče A=0, B=1 a C=2, pro sloupcového D=0 a E=1, indexy strategií i užitků v profilu v pořadí řádkový hráč, sloupcový hráč).

Model hry – `Game`

Tento objekt slouží jako abstraktní bázová třída pro jakoukoliv hru, kterou má nástroj řešit. Obsahuje tři čistě virtuální metody, jež je potřeba pro konkrétní hru naimplementovat, a sice

1. zjištění počtu hráčů,

2. určení, kolik má daný hráč strategií, a

3. spočítání hodnot užitků při hraní daného strategického profilu

Model hry vlastně tvoří abstraktní reprezentaci jakékoliv hry v normální formě. Byl navržen s ohledem na co největší jednoduchost, aby co nejméně omezoval potenciálního modeláře, který by chtěl knihovnu použít.

Cache spočítaných užitků – `UCache`

Tuto cache *můžeme* využít v případě, že počítání užitků trvá nezanedbatelnou dobu, a bylo by tedy velmi nevýhodné je počítat opakovaně.

Je implementována jako kontejner `std::map<StrategyProfile, UProfile>`, tedy asociativní pole profilů užitků indexované strategickými profily.

Pokud chceme cache použít, uděláme to předáním hodnoty *true* jako booleovského argumentu šablony objektu `CESolver`, jinak funguje jako falešná cache, která pokaždé zjišťuje užitky přímo v modelu hry.

Pozice v G-matici – `GRows`

Tato třída slouží pro uložení prvků G-matice, na kterých muselo skončit procházení příslušného řádku (pro podrobnosti viz 4.3). Jednoduše můžeme říci, že si sem ukládáme strategické profily, u nichž už dále nemohlo pokračovat procházení příslušného řádku (z důvodu nalezení buňky s pozitivním obsahem).

Řádky G-matice jsou identifikovány jednoduchou strukturou `RowID`, která obsahuje identifikaci hráče a jeho dvou strategií. Zároveň má definován operátor `<` (menší než), díky čemuž je možné použít standardní kontejner STL `std::map<RowID, StrategyProfile>`.

Zakázané strategie – `ForbiddenStrategies`

Při redukci G-matice postupně zjišťujeme, že racionální hráč nikdy nebude hrát určité své strategie. Informace o tom, které jsme takto vyloučili, si uchováváme právě v této třídě, jejímž základem je pole booleovských příznaků.

LP problém – `CorrelatedLP`

V této třídě ukládáme nenulové prvky výsledné redukované G-matice a koeficienty funkce užitku. Slouží jako vstup LP solveru.

Rozhraní pro GLPK – `GLPKSolver`

Tato jednoduchá třída zajišťuje řešení LP problému ve formě objektu `CorrelatedLP` pomocí knihovny GLPK. Je vlastně schopna vytvořit objekt v interní reprezentaci knihovny GLPK, spustit její metodu řešení a vrátit výsledek ve formě, používané ve zbytku nástroje.

G-matice – `GMatrix`

G-matice slouží jako jakési výpočetní jádro celé knihovny. Děje se zde nejdůležitější a nejsložitější operace, kterou je minimalizace hry, proto jde o třídu poměrně rozsáhlou a komplikovanou.

Kromě samotné minimalizace (prováděné s použitím pomocných objektů typu `GRows`, `ForbiddenStrategies` a `UCache`) také převádí (minimalizovanou) hru na objekt typu `CorrelatedLP`, který je pak vstupem LP solveru.

V podstatě všechny důležité metody této třídy popisujeme v části 4.3, která detailněji rozebírá nejdůležitější použité algoritmy.

Jádro – `CESolver`

Vlastní objekt `CESolver` využívá všechny dříve popsané objekty k tomu, aby splnil požadovaný úkol, tedy našel korelované ekvilibrium zadané hry. Není nijak složitý, jednoduše hru zminimalizuje pomocí G-matice a vyřeší vzniklý LP problém pomocí objektu `GLPKSolver`.

4.3 Algoritmus minimalizace G-matice

Zjednodušeně bychom mohli říct, že algoritmus prochází jednotlivé řádky G-matice, dokud existuje možnost, že je strategie danému řádku odpovídající dominovaná. Pokud zjistíme, že dominovanou v tuto chvíli být nemůže, poznačíme si, kde jsme skončili, a přecházíme k následujícímu řádku.

Hledání eliminovatelných strategií spočívá v postupném procházení řádků G-matice a hledáním takových, ve kterých jsou (mimo nedefinovaných hodnot) pouze záporná čísla. Strategii $sfrom$ takového řádku můžeme eliminovat.

4.3.1 Průchod G-maticí po řádcích

Hlavní výpočet redukované G-matice probíhá v metodě `iterate()` objektu `G-matrix`. Postupně procházíme všechny dvojice různých strategií jednotlivých hráčů, jež identifikují řádky G-matice, a v rámci těchto řádků zkoušíme, zda by je (i jim příslušnou strategií) nebylo možné eliminovat.

Přesnější strukturu tohoto průchodu lze spatřit na obrázku 4.2.

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

Obrázek 4.2: Algoritmus průchodu řádky G-matice

4.3.2 Průchod řádkem G-matice

V předchozí části jsme naznačili schéma průchodu G-maticí na úrovni řádků. Nyní si ukážeme, jak procházíme řádkem G-matice při zjišťování, zda můžeme ze hry odstranit jemu příslušející strategii.

Nejdříve zjistíme, u kterého strategického profilu naposledy skončilo procházení tímto řádkem (z objektu `rows` třídy `GRows`). Navíc se ale mohlo stát, že tento strategický profil obsahuje neplatnou strategii (pokud byla tato strategie eliminována po předchozím průchodu tímto řádkem), což ošetříme funkcí `valid_profile`, která v případě, kdy je opravdu některá ze strategií v tomto profilu zakázaná, postoupí řádkem na nejbližší platný strategický profil. Pokud vrátí hodnotu *false*, znamená to, že jsme dospěli na konec řádku, a tím pádem můžeme strategii eliminovat (přidat do zakázaných strategií).

Poté následuje hlavní cyklus, který prochází řádkem, dokud jsou hodnoty jeho prvků záporné. Tyto hodnoty počítá z modelu hry s použitím cache funkce *difference* a posun zajišťuje funkce `next_profile`, která se z daného *platného* strategického profilu přesune na další platný strategický profil v daném řádku G-matice.

Za zmínku stojí fronta `skipped`, která obsahuje profily, které v okamžiku, kdy byly potřeba, počítalo jiné vlákno aplikace. V této situaci jsme se v řádku normálně posunuli, jako bychom narazili na zápornou hodnotu, nicméně jsme si tuto skutečnost poznamenali do fronty `skipped`.

V případě, že se nám povedlo posunout v řádku, nebo dokonce dojít na jeho konec, musíme ověřit položky z fronty `skipped`, jež jsme přeskočili. Procházíme jí a počítáme příslušné hodnoty, dokud buď není prázdná, nebo nenarazíme na nezáporný prvek. Pokud jsme na něj narazili, nastavíme proměnnou `end` na *false* (protože jsme ve skutečnosti na konec řádku nedošli).

Nakonec v případě, že jsme došli na konec řádku, vložíme příslušnou strategii do zakázaných a uchováme dosaženou pozici v tomto řádku. Pokud jsme alespoň provedli posun, uložíme aktuální profil do objektu `rows` třídy `GRows`.

Proměnná `change` Jedná se o booleovskou proměnnou, která udává, zda v G-matici nastala změna. Pokud je po skončení iterace nastavena na *true*, pak má smysl provádět (a provádí se) iterace znovu.

4.4 Paralelizace výpočtu

CE-solver byl vyvíjen s ohledem na možnost paralelizace a tím zrychlení výpočtu. Za tímto účelem jsme použili systém OpenMP.

Jedinou částí, u které má paralelizace smysl, je samotný výpočet v metodě `iterate()` (popsán v části 4.3). Vzhledem ke zvolenému přístupu není vhodné paralelizovat v rámci jednoho řádku G-matice, proto je paralelizace realizována „po řádcích“, tedy tak, že jednotlivá vlákna mají v jednom okamžiku každé přidělen unikátní řádek G-matice, se kterým pracují.

4.4.1 Synchronizace přístupu ke sdíleným zdrojům

Stinnou stránkou zavedení paralelismu je nutnost zvýšené obezřetnosti u dat sdílených mezi vlákny. Za tímto účelem používáme direktivu překladače poskytovanou OpenMP, `critical`.

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

Obrázek 4.3: Algoritmus průchodu řádkem G-matice – objekt `rows` (třídy `GRows`) obsahuje místa, u kterých skončilo procházení daných řádků (viz 4.2), proměnná `change` udává, zda došlo ke změně v zakázaných strategiích (viz 4.3.2).

Ta dokáže zajistit, že blok kódu jí označený bude v jeden okamžik provádět maximálně jedno vlákno.

Data, kterých se toto týká, jsou:

- proměnná `change`
- pozice v řádcích G-matice `GRows`
- zakázané strategie `ForbiddenStrategies`
- cache spočítaných užitků `UCache`

U (booleovské) proměnné `change` je situace nejjednodušší. Můžeme s výhodou použít direktivu OpenMP `reduction(||:change)`, jež je pro tento případ jako stvořená. Sémantiku má takovou, že všechna vlákna mají svou kopii proměnné, jen při synchronizaci vláken na konci paralelního bloku OpenMP zařídí, že se nad těmito kopiemi provede specifikovaná operace (v tomto případě logické „nebo“).

Pro pozice v řádcích stačí zajistit, aby čtení a přidání prvku probíhalo v kritické sekci, což je triviální. Pro bližší detaily doporučujeme případně prozkoumat přímo zdrojové kódy.

U zakázaných strategií je situace velmi obdobná, v kritická sekce dovolí v jeden okamžik přístup k datům pouze jednomu vláknu (ať už kvůli čtení, nebo kvůli zápisu).

Synchoronizace cache vypočtených užitků

Nejzajímavější je situace u cache užitků. Zde si totiž uchováváme i informaci o tom, které užitky jsou počítány v tuto chvíli jinými vlákny. Zároveň se ovšem musíme důsledně vyvarovat toho, abychom užitek počítali v kritické sekci – tím bychom paralelizací nemohli zrychlit výpočetně nejnáročnější část výpočtu.

V případě, že se některé vlákno rozhodne, že potřebuje hodnoty užitků pro daný strategický profil, zavolá metodu `get()` objektu třídy `UCache`. Zde mohou nastat tři situace:

1. užitky již byly vypočteny
2. užitky zatím nikdo nepožadoval
3. užitky se právě počítají

První případ je nejjednodušší, stačí je pouze vyzvednout z cache.

Ve druhém případě je potřeba uložit informaci o tom, že požadované užitky jsou počítány, a začít je počítat (zavoláním příslušné metody modelu hry (třída `Game`)). Jak jsme již zmínili, tento výpočet *nesmí* probíhat v kritické sekci. Po vypočtení uložíme užitky do cache a odstraníme informaci o tom, že jsou právě počítány.

Třetí situace přináší drobné komplikace. Vlákno potřebuje informaci, kterou právě počítá někdo jiný, takže na jednu stranu tato informace není k dispozici, ale na druhou stranu by bylo zbytečné snažit se ji v danou chvíli počítat. Proto metoda `get()` sděluje, že užitky jsou právě počítány a algoritmus průchodu řádkem se s tím vyrovnává tak, jak je popsáno v části 4.3.2.

Schéma celého algoritmu je naznačeno na obrázku 4.4.

4.5 Použití CE-solveru pro řešení her

CE-solver lze obecně použít dvěma způsoby.

Pokud máme všechny užitky spočítané ve vhodném formátu souboru, není nic jednoduššího, než použít jednoduchý nástroj `cefs` (viz 4.5.1).

V případě, že nemůžeme použít předchozí způsob, nezbývá nám, než se vydat o něco složitější cestou a vytvořit model hry respektující určité rozhraní (viz 4.5.2).

4.5.1 Correlated Equilibrium File Solver

CE-solver obsahuje obsahuje jednoduchou konzolovou aplikaci `cefs` (*Correlated Equilibrium File Solver*), která je schopná načíst hru ze souboru, vytvořit z ní model hry a zavolat CE-solver s vhodnými parametry. Vnitřně používá objekty `FileGame` a `FileSolver`.

Podporovány jsou hry v jednoduchém formátu používaném v původním CE-solveru (viz [15]), což je vlastně serializovaná tabulka s jednoduchou hlavičkou, jež obsahuje informaci o počtu hráčů a jejich strategií. Navíc dokáže načíst i formát používaný knihovnou Gambit pro hry v normální formě (viz [16]).

```
bool get(StrategyProfile sp, out: PayoffProfile up):
    result = false, computed = false
    CRITICAL SECTION
    {
    if (no one else is computing sp):
        result = true
        if (sp already computed):
            up = data[sp]
            computed = true
        else:
            mark sp as currently computed
    }
    if (result == false):
        return false
    if (not computed):
        up = game[sp]
        CRITICAL SECTION
        {
        data[sp] = up
        unmark sp as currently computed
        }
    return true
```

Obrázek 4.4: Synchronizace přístupu ke cache užitků v metodě `get` třídy `UCache`: V první kritické sekci zjišťujeme, zda je požadovaný užitek právě počítán. Pokud ano, vracíme *false*, jinak zjišťujeme, zda již byl profil užitků vypočten. Byl-li, vracíme jej, nebyl-li, poznačíme profil jako právě počítaný a mimo kritickou sekci zavoláme model hry (`game`). Po skončení výpočtu vložíme spočítaný profil užitků do cache (`data`) a zrušíme informaci o tom, že je profil právě počítán (opět v kritické sekci).

Formát vstupního souboru

Definujme si formát vstupní hry v notaci vycházející z EBNF (*Extended Backus-Naur Form*):

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

Od EBNF se liší tím, že používá vyjádření specifického počtu opakování na základě dříve „načtené“ proměnné.

Tento formát připomíná serializovanou vícerozměrnou tabulku, uvozenou jednoduchou hlavičkou udávající její rozměry (pomocí počtu hráčů a jejich strategií).

Buňky tabulky jsou řazeny lexikograficky po strategických profilech, tedy například $(1,1)$, $(1,2)$, $(2,1)$, $(2,2)$ pro dvouhráčovou hru se dvěma strategiemi na hráče (dvojice (a,b) značí, že první hráč hraje svou a-tou a druhý svou b-tou strategii).

Příklad uložení nám známe hry 4.1 do souboru můžeme vidět na obrázku 4.5.

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

Tabulka 4.1: Jednoduchá hra, kterou chceme uložit do souboru (viz obrázek 4.5)

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

Obrázek 4.5: Hra z tabulky 4.1 uložená v souboru

Další detaily týkající se použití mohou případní zájemci najít v manuálu (viz přílohu A nebo soubor manual.pdf šířený s nástrojem).

4.5.2 Řešení her zadaných modelem

Při tomto způsobu použití vytváří modelář potomka třídy `Game`, který specifikuje řešenou hru. Jde o cestu velmi jednoduchou a přímočarou, v podstatě kopírující matematickou definici hry, nicméně složitost v sobě ponese samotná specifikace chování v případě rozsáhlých modelů.

Jak již bylo řečeno, teoreticky je takto možné vyřešit i hru, jejíž stavový prostor je větší než celková paměť počítače.

Když máme naimplementovaný takovýto model, předáme ho jako argument při tvorbě objektu samotného solveru, eventuálně i s dalšími argumenty (pro příklad viz obrázek 4.6).

```
//game - created model of the game
CESolver<true> ces(game); //create CESolver object
ces.solve(); //call its method solve()
CorrelatedEquilibrium ce = ces.get_solution();
```

Obrázek 4.6: Použití objektu `CESolver` k vyřešení vytvořené hry.

Kvůli použitelnosti nástroje nejen v českých zemích jsme vytvořili uživatelský manuál v anglickém jazyce, v němž je použití rozebráno detailněji. Tento manuál přikládáme jako přílohu A.

4.6 Experimentální zhodnocení CE-solveru

V této části se budeme zabývat experimentováním s vytvořeným CE-solverem, především z pohledu zjišťování jeho výkonnosti, ale i jiných charakteristik.

Veškeré testování probíhalo na serveru `ju.fit.vutbr.cz`. Jde o osmiprocesorový stroj (procesory Intel Xeon X5355, frekvence 2,66 GHz) se 16 GB operační paměti.

4.6.1 Testovací data

K testování jsme měli k dispozici sedm her, jež pocházejí z praktických problémů týkajících se chování hráčů na trhu s elektrickou energií. Jde o dva typy her, a sice

- osmihráčové, jež modelují chování výrobců elektrické energie v České republice, a
- šestnáctihráčové, které modelují chování osmi výrobců a osmi odběratelů elektřiny v rámci střední Evropy.

Rádi bychom vyzdvihli, že jde o reálná data, nikoli teoreticky vykonstruované abstraktní problémy. I když jde jen o podmnožinu problémů, které se řeší jako hledání korelovaného ekvilibria, můžeme si udělat lepší představu o tom, jak se bude CE-solver chovat v praxi. Nedochází k naprostému odtržení od reality, jež by bylo pro testování nástroje určeného do praxe krajně nešťastné.

Kvantitativní charakteristiky her shrnujeme v tabulce 4.2. Všechny tyto hry se nacházejí i na CD přiloženém k této práci pojmenované jako `1.g`, `2.g`, ... a `7.g`.

Hra	Hráčů	Velikost [kB]	$\lvert S\rvert$	Řádků G-matice
`1.g`	16	1 037	6 048	290 304
`2.g`	8	447	6 720	27 659 520
`3.g`	16	5 232	30 240	76 204 800
`4.g`	8	18 072	270 000	28 343 520 000
`5.g`	16	50 419	294 000	19 813 248 000
`6.g`	8	66 879	1 257 984	777 132 195 840
`7.g`	8	131 295	2 066 400	3 832 841 376 000

Tabulka 4.2: Kvantitativní charakteristika her použitých pro experimenty ($|S|$ značí počet strategických profilů)

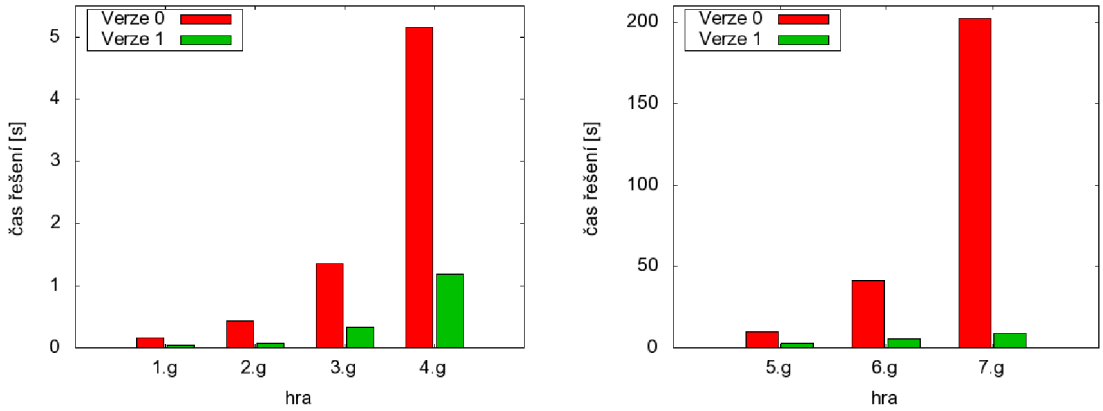
4.6.2 Porovnání s existující implementací

CE-solver jsme porovnali s existujícím nástrojem pro hledání korelovaného ekvilibria (viz [15]). Jde o první implementaci metody, popsané v článku [14], kterou vytvořil sám jeho autor. Napsán je v jazyce C++, využívá knihovnu GMP (*GNU Multiple Precision Arithmetic Library* – knihovna pro práci s čísly s libovolnou přesností) kvůli indexaci strategických profilů a ke zrychlení pomocí paralelního běhu používá systém OpenMP. Jelikož jde o první implementaci, spíše prototypovou, budeme o tomto nástroji dále hovořit jako o CE-solveru verze 0. CE-solver, který jsme vytvořili v rámci této práce, pak nazýváme CE-solverem verze 1.

CE-solver verze 0 jsme porovnávali se sekvenční verzí CE-solveru verze 1. Výsledky shrnuje tabulka 4.3 a grafy na obrázku 4.7.

	1.g	2.g	3.g	4.g	5.g	6.g	7.g
Verze 0	0,16	0,43	1,36	5,16	9,89	41,38	202,44
Verze 1	0,05	0,07	0,33	1,18	2,76	5,29	8,76
Podíl	31 %	16 %	24 %	23 %	28 %	13 %	4 %

Tabulka 4.3: Časy řešení jednotlivých her pro verzi 0 a verzi 1 (údaje v sekundách)



(a) Hry 1.g až 4.g

(b) Hry 5.g až 7.g

Obrázek 4.7: Porovnání CE-solveru verze 0 a verze 1

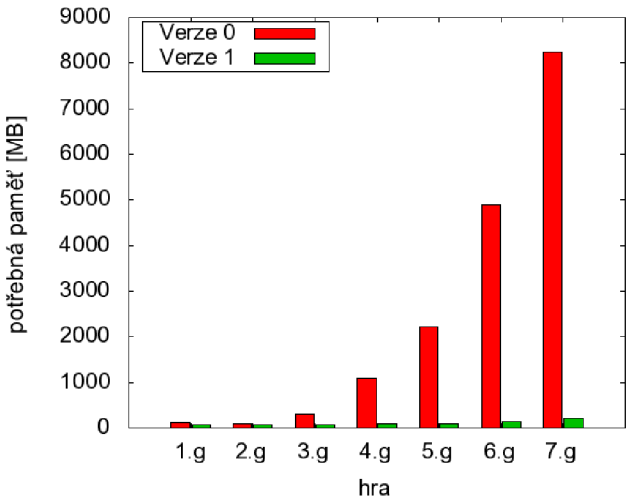
Můžeme konstatovat, že se nám povedlo *velmi razantně* zlepšit výkon, jak ukazují grafy na obrázku 4.3. Navíc musíme zmínit, že nás CE-solver je výrazně rychlejší i přes to, že původní využívá všechny procesory, zatímco náš pouze jeden.

Dále nás zajímalo, jak se verze liší s ohledem na spotřebovanou paměť. Naše očekávání, že verze 1 bude používat markantně *méně* paměti díky odlišnému přístupu uložení G-matice (jak bylo nastíněno v části 4.1), se stoprocentně vyplnilo.

Hra	Verze 0	Verze 1	Podíl
1.g	119	70	59 %
2.g	99	72	72 %
3.g	296	73	25 %
4.g	1081	88	8 %
5.g	2216	90	4 %
6.g	4879	150	3 %
7.g	8250	203	2 %

Tabulka 4.4: Porovnání spotřebované paměti CE-solveru u jednotlivých her (hodnoty v megabajtech)

Jak můžeme vidět v tabulce 4.4 a grafu 4.8, paměť použitá verzí 1 je u větších her doslova zlomkem té, jež je potřeba u verze 0. To je velmi příjemné zjištění, neboť na rychlost výpočtu může mít velmi výrazný vliv „swapování“ (odkládání dat z operační paměti na disk při její nedostatečné velikosti). Nám se tedy povedlo výrazně posunout hranici velikosti hry, v níž je schopen daný počítač hru vyřešit, aniž by musel „swapovat“, odhadem cirka na padesátinu, pokud budeme uvažovat nejskeptičtější odhad – všimněme si totiž, že podíl spotřebované



Obrázek 4.8: Graf spotřebované paměti CE-solveru u jednotlivých her

paměti s rostoucí velikostí hry má klesající tendenci.

Povedlo se nám tedy výrazně zlepšit čas hledání korelovaného ekvilibria. Výsledek je o to více potěšující, že náš solver spotřebovává méně paměti.

4.6.3 Vhodnost použití cache užitků

Jak již bylo řečeno, použití cache pro uchování spočítaných hodnot užitků má smysl v případě, že výpočet užitku trvá nezanedbatelnou dobu. To je poněkud vágní formulace, proto jsme se rozhodli otestovat, od jaké hranice se už používání vyplatí a do kdy je spíše na obtíž.

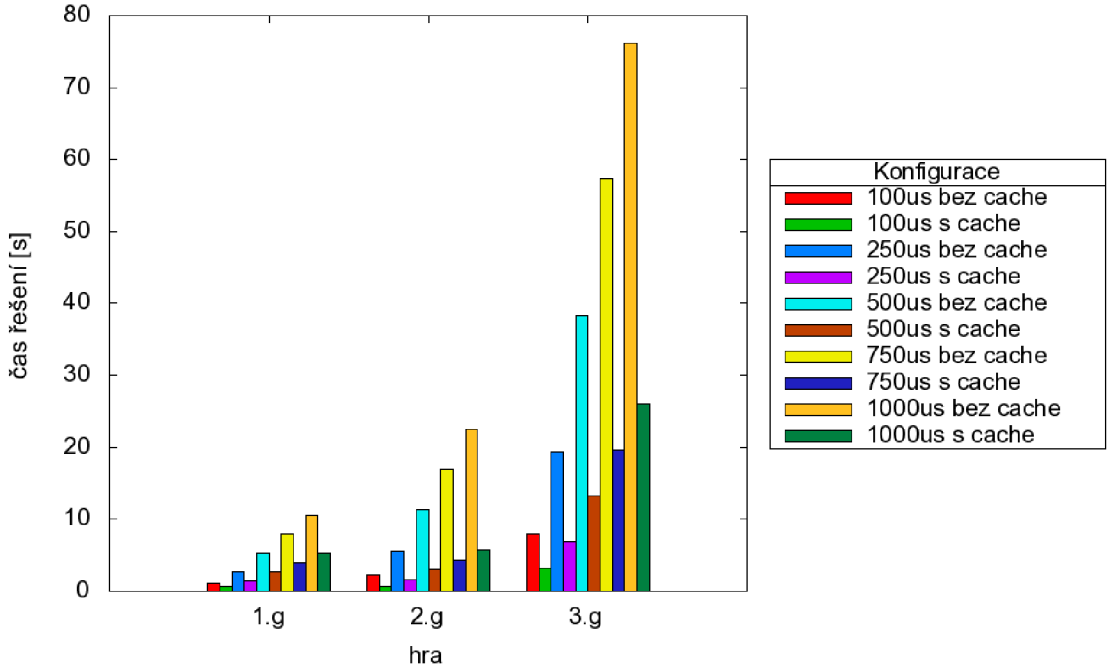
Jelikož nemáme k dispozici hru, která by skutečně počítala užitky delší dobu, nasimulovali jsme toto chování umělým vytvořením zpoždění při přístupu k užitkům načteným ze souboru. Nejdříve jsme chtěli využít systémovou funkci `usleep`, která dokáže pozastavit volající vlákno na určitý počet mikrosekund, ale nezaznamenali jsme žádnou změnu rychlosti ani při změně zpoždění ze 100 mikrosekund na 1 milisekundu.

Proto jsme jako zpoždění použili cyklus `for`, který přičítá k původně nulové proměnné jedničku až do dosažení určité hranice. Experimentálně jsme zjistili, že jedné milisekundě odpovídá zhruba hranice 220 000. Uvědomujeme si, že zde naměřené výsledky jsou závislé na použitém počítači, nicméně pro vytvoření rámcové představy bohatě postačují.

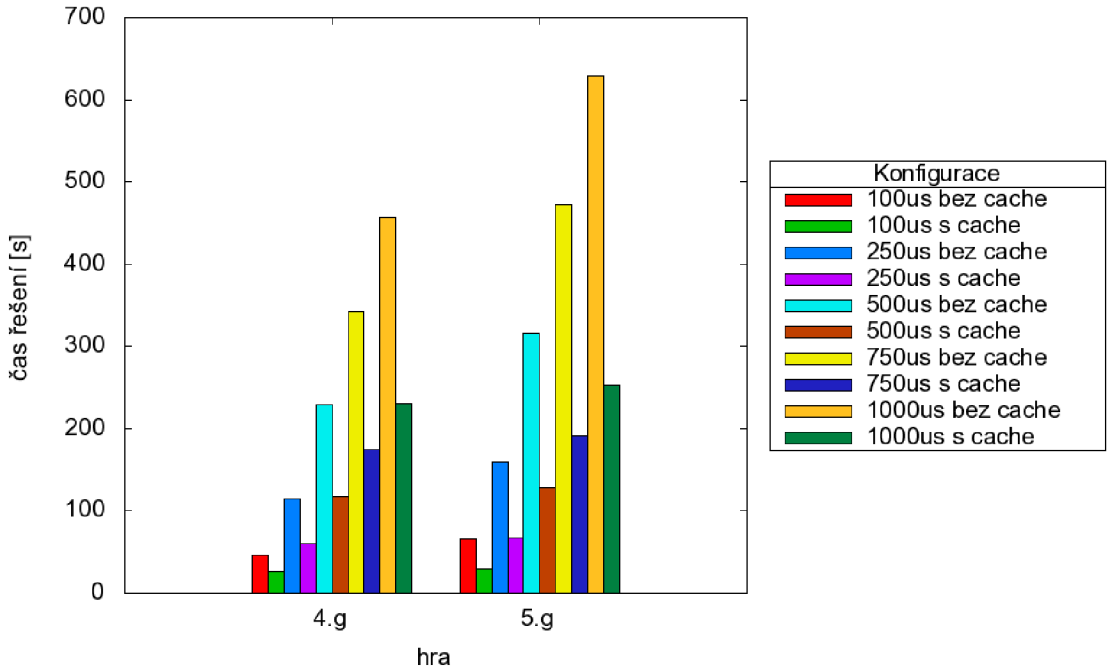
Naším prvotním odhadem bylo, že použití cache bude znamenat zrychlení v případě, že se doba výpočtu jednoho užitku bude pohybovat v řádu stovek mikrosekund. Proto jsme otestovali doby běhu pro různá zpoždění z intervalu 100 μs až 1 ms. Výsledky shrnují grafy 4.9 a 4.10.

Jaké bylo naše překvapení, když jsme zjistili, že cache zrychluje výpočet i při zpoždění pouhých 100 μs! Nezbylo nám než otestovat rozsah do 100 μs. Naměřené hodnoty jsme opět vynesli do grafů, které můžete vidět na obrázcích 4.11 a 4.12.

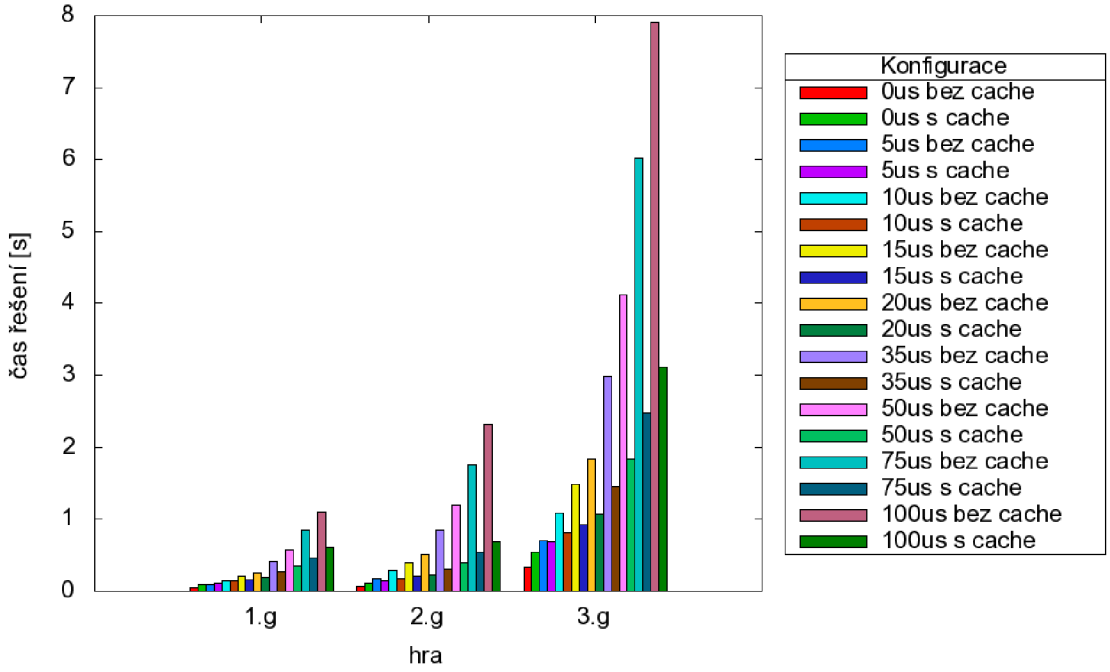
Můžeme usoudit, že použití cache vypočtených užitků se začíná (alespoň při naší hardwarové konfiguraci) vyplácet v případě, že spočtení užitku trvá cirka 5–10 μs. Tato experimentálně zjištěná hodnota není sice nijak obecně platná, nicméně nám dává poměrně cennou informaci v tom smyslu, že máme předtavu o řádu zpoždění, kdy se cache vyplatí.



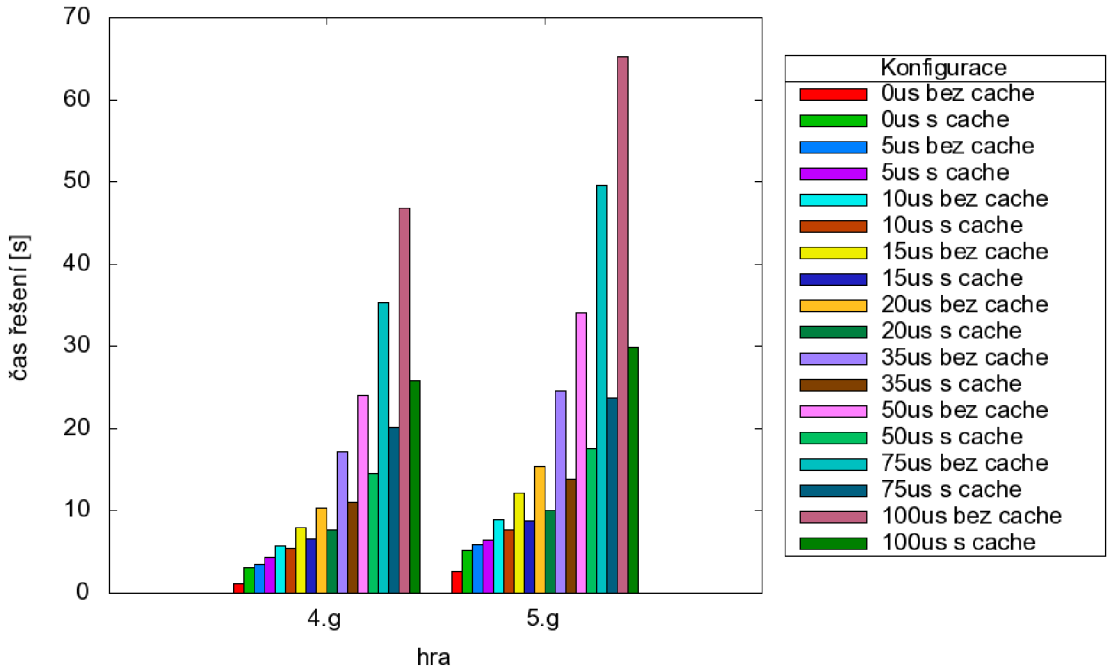
Obrázek 4.9: Porovnání rychlostí při různé náročnosti výpočtu užitku a vliv použití cache při zpožděních 100–1000 μs pro hry 1.g až 3.g



Obrázek 4.10: Porovnání rychlostí při různé náročnosti výpočtu užitku a vliv použití cache při zpožděních 100–1000 μs pro hry 4.g a 5.g



Obrázek 4.11: Porovnání rychlostí při různé náročnosti výpočtu užitku a vliv použití cache při zpožděních 0–100 μs pro hry 1.g až 3.g



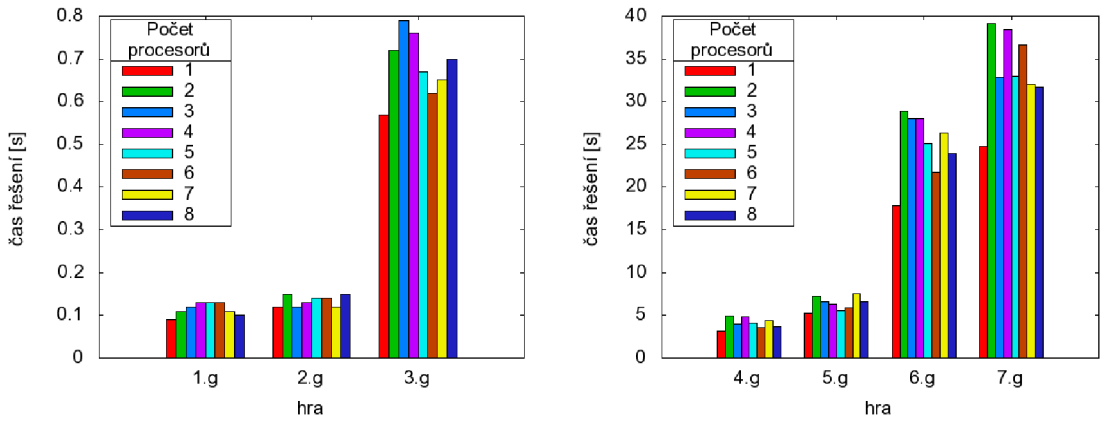
Obrázek 4.12: Porovnání rychlostí při různé náročnosti výpočtu užitku a vliv použití cache při zpožděních 0–100 μs pro hry 4.g a 5.g

Jelikož jde podle našeho názoru o relativně nízkou hodnotu, doporučili bychom potenciálním vývojářům v případě pochyb či nerozhodnosti cache použít.

4.6.4 Vliv paralelizace na rychlost výpočtu

Rovněž nesmíme opomenout otestovat zrychlení, kterého jsme dosáhli díky paralelizaci. Změřili jsme proto doby řešení pětice menších her z naší sady (zbývající dvě, `6.g` a `7.g`, jsou při vložení zpoždění příliš časově náročné) při vybraných nastaveních zpoždění. Zkoumali jsme závislost počtu použitých procesorů na zrychlení výpočtu. Vždy jsme použili cache užitků (z důvodů popsaných dříve).

Nejdříve se podívejme na situaci s nulovým zpožděním. V takovéto konfiguraci žádné zrychlení neočekáváme, neboť pouze potřebná synchronizace zabere čas větší, než paralelismem získáme. Výsledky shrnuje graf 4.13.



(a) Hry 1.g až 3.g

(b) Hry 4.g až 7.g

Obrázek 4.13: Zrychlení paralelizací bez zpoždění

Očekávání nás protentokrát nezklamalo a vidíme, že se čas výpočtu použitím více procesorů dokonce zvýšil. Vysvětlujeme si to tím, že samotný výpočet tvoří mizivou část celkové doby běhu, tím pádem režie tvoří časově nezanedbatelnou část běhu.

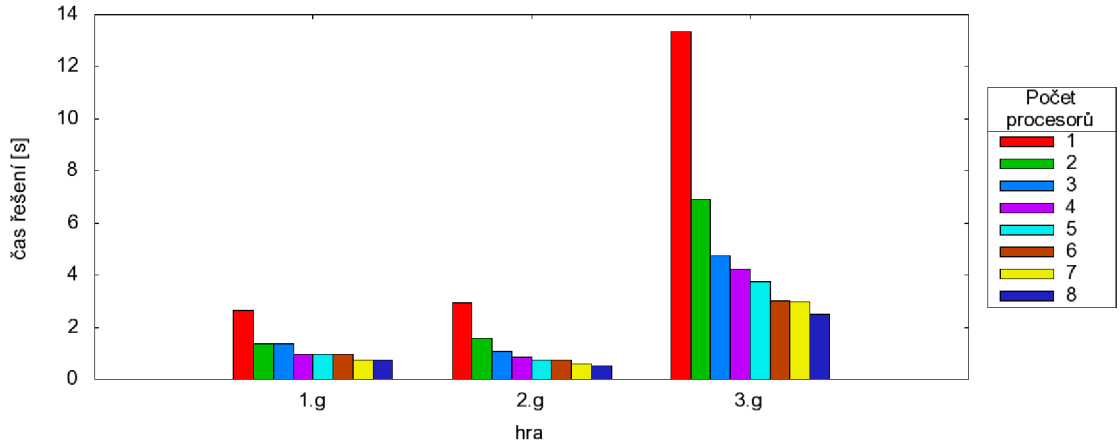
Dále jsme zkusili vložit zpoždění o velikosti půl milisekundy (500 μs). Tím by se nám měl výpočet užitků stát velmi časově náročnou částí řešení. Naměřené výsledky jsme vynesli do grafů 4.14, 4.15 a 4.16.

Můžeme si všimnout skutečnosti, že u her `4.g` a `5.g` dochází k lineárnímu zrychlení cirka do počtu tří procesorů. Při větším počtu někdy dochází k zajímavému jevu, kdy zvýšení počtu procesorů nepřinese vůbec žádné zrychlení. Domníváme se, že to zapřičiňuje struktura konkrétních her. Eliminace strategií jsou u nich zvláštním způsobem provázány tak, že je možné většinu času procházet méně řádků G-matice, než je počet procesorů.

Ostatní hry vykazují téměř lineární zrychlení.

Závěrem konstatujme, že efektivní zrychlení při zapojení více procesorů může být až lineární, pokud

1. většinu času zabírá počítání užitků a
2. hra nemá takové vlastnosti, které by zamezily zapojení všech procesorů po většinu času minimalizace.



Obrázek 4.14: Zrychlení paralelizací při zpoždění 500 μs (hry 1.g až 3.g)

4.7 Nástroj pro analýzu dominance strategií

V rámci práce jsme vytvořili jednoduchý nástroj, který analyzuje strategie v zadané hře z hlediska dominance. Nazvali jsme jej *Game Dominance Analyzer* a krátce si ho v této části popíšeme.

Funkčnost nástroje lze popsat velmi jednoduše. Veškeré podstatné věci se dějí v rámci objektu třídy `Analyzer`. Ten s pomocí objektu třídy `Game` (viz část 4.2) načte hru, do které pak (volitelně) přistupuje pomocí objektu třídy `UCache`. Nad touto hrou je schopen provádět jednoduché dotazy.

4.7.1 Dotazovací jazyk

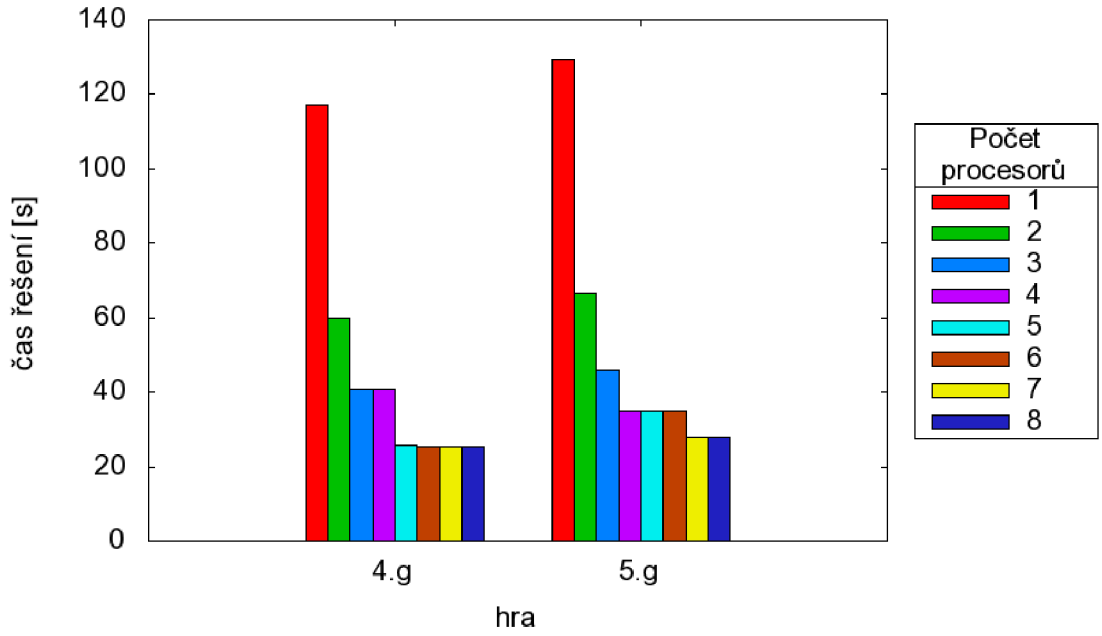
Pro účely tohoto nástroje jsme vytvořili jednoduchý dotazovací jazyk, který nám umožňuje zkoumat vztahy mezi strategiemi z hlediska dominance. Pojmenovali jsme jej *Game Query Language* (dále GQL).

Dotaz v tomto jazyce lze definovat pomocí EBNF takto:

```
query = [mode], range, ":", range, ":", range;
mode = "a" | "s" | "w";
range = enumeration | interval | all;
enumeration = n, {",", n};
interval = n, "-", n;
all = "x";
n = digit - "0", {digit} | "0";
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";
```

Část mód (`mode`) v dotazu specifikuje tři různé módy činnosti:

1. `a` – analýza zadaných strategií, výpis statistik ()
2. `s` – posouzení striktní (**S**trict) dominance
3. `w` – posouzení slabé (**W**eak) dominance



Obrázek 4.15: Zrychlení paralelizací při zpoždění 500 μs (hry 4.g a 5.g)

Povinnou částí dotazu jsou tři rozsahy oddělené dvojtečkou. První udává hráče, druhý „dominantní“ a třetí „dominované“ strategie (v uvozovkách proto, že jimi nejsou, ale budou z tohoto pohledu posuzovány). Nástroj pak zpracovává všechny specifikované dvojice strategií pro všechny specifikované hráče.

4.7.2 Módy činnosti nástroje

Chování i výstup se liší podle módu, ve kterém pracujeme.

Analyzační mód

V tomto módu pro všechny zadané dvojice strategií počítáme statistiky, které nám mohou pomoci udělat si určitý obrázek o atraktivnosti strategií pro hráče. U všech dvojic zjistíme, v kolika strategických kontextech (s_{-p}) daného hráče p je užitek při hraní „dominantní“ vyšší, stejný a nižší než při hraní strategie „dominované“.

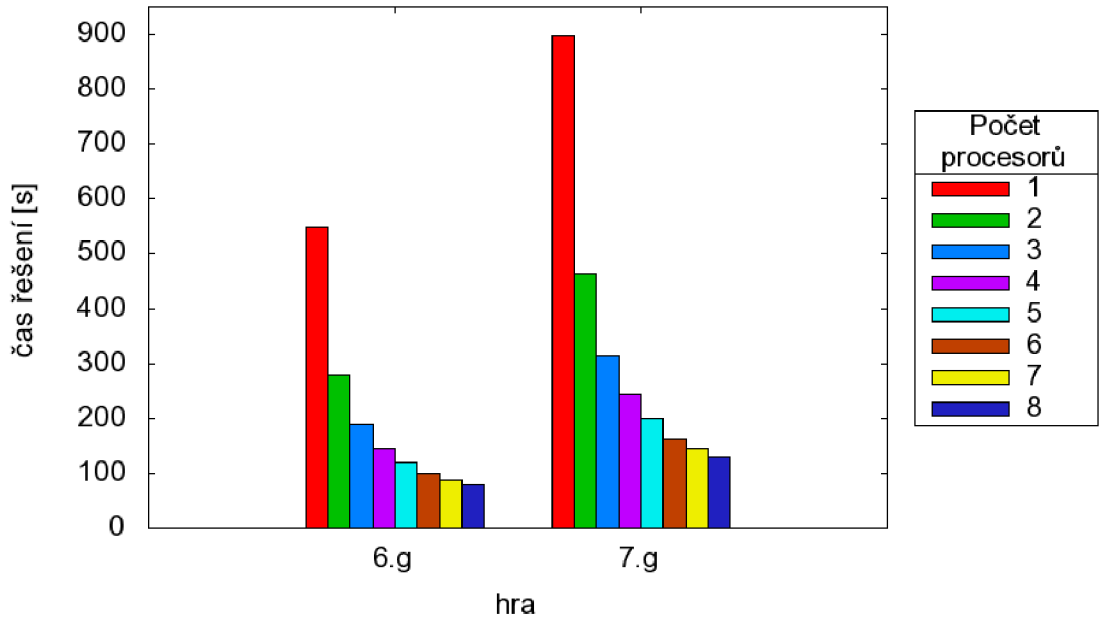
Výstup pak má takovouto formu:

```
p-s1,s2 >ng =ne <nl
```

Symbol `p` značí hráče, `s1` a `s2` jeho strategie, čísla `ng`, respektive `ne`, respektive `nl` pak počty strategických kontextů, ve kterých je pro hráče `p` výhodnější, respektive stejně výhodné, respektive méně výhodné hrát strategii `s1` oproti strategii `s2`.

Rozhodovací mód

V tomto módu pouze vypisujeme dvojice strategií, které vyhovují zadané podmínce dominance. Opět procházíme všechny dvojice a ve všech strategických kontextech porovnáváme užitek z hraní „dominantní“ strategie s užitkem z hraní „dominované“ strategie. Pokud je



Obrázek 4.16: Zrychlení paralelizací při zpoždění 500 μs (hry 6.g a 7.g)

ostře menší, pak můžeme tuto dvojici okamžitě zavrhnout. Když je větší a zjišťujeme slabou dominanci, poznamenáme si, že it může nastat. V případě rovnosti a rozhodování striktní dominance dvojici opět zavrhujeme.

Výsledky vypisujeme ve tvaru:

```
p-s1,s2
```

Vypisujeme pouze dvojice, u kterých strategie `s1` hráče `p` (striktně či slabě, podle konkrétního módu) dominuje jeho strategii `s2`.

4.7.3 Zhodnocení výsledků

Vytvořili jsme jednoduchý nástroj určený k analýze vztahů strategií ve hrách, především z pohledu dominance. Využili jsme při tom třídy, používané v CE-solveru, což dokládá možnost jejich dalšího použití i v jiných projektech. Hlavní přínos GDA vidíme v možnosti použití v situaci, kdy si chceme udělat obrázek o některé omezené podmnožině strategií dané hry, aniž bychom ji měli čas řešit celou.

4.8 RS-solver

V rámci výzkumu možností zlepšení způsobu řešení LP problémů jsme vytvořili tento experimentální nástroj, zaměřený na řešení řídkých (*sparse*) LP problémů modifikovanou simplexovou metodou (*revised simplex method*) – odtud název *Revised Sparse Solver*.

Na řídké problémy jsme se zaměřili proto, že samotná G-matice bývá velmi řídká (v závislosti na množství strategií jednotlivých hráčů).

4.8.1 Implementace RS-solveru

RS-solver jsme vytvořili v jazyce C++ za použití knihovny uBLAS z balíku knihoven Boost, která umožňuje snadné uložení řídkých matic a vektorů a zároveň implementuje základní operace lineární algebry.

Jak je částečně zřejmé z názvu, používáme modifikovanou dvoufázovou simplexovou metodu, tedy jsme schopni řešit i LP problémy, u nichž není k dispozici počáteční přípustné řešení. Zoptimalizovali jsme násobení, které je nutné pro aktualizace inverzní matice báze, s využitím speciální struktury matice, kterou jí násobíme (jednotková matice až na jeden sloupec).

Hojně využíváme `compressed_vector`, jeden z typů řídkých vektorů knihovny uBLAS. Jde o pole dvojic `(index, hodnota)` seřazené podle indexu, proto je velmi vhodný pro procházení nenulových položek – k tomu slouží příslušný iterátor.

Abychom se vyhnuli nebezpečí zacyklení z důvodu degenerovaných řešení, implementovali jsme dříve popsané Blandovo pravidlo (viz část 3.1.2). To se neuplatňuje vždy, ale pouze v situaci, kdy by se po provedení kroku nezměnila hodnota kriteriální funkce. Samozřejmě bychom k řešení dospěli, i kdyby se používalo stále, nicméně to není výhodné z hlediska efektivity. Pokud bychom vstupující proměnné vybírali pouze podle Blandova pravidla, většinou bychom postupovali pouze malinkými krůčky ke globálnímu extrému. Místo toho vybíráme vstupující proměnnou podle minimální relativní ceny, a pouze v případě, kdy dospějeme do degenerovaného řešení, aplikujeme Blandovo pravidlo.

4.8.2 Použití pro řešení LP problémů

RS solver lze použít dvěma způsoby, a sice pro řešení LP problémů uložených v souboru nebo pro řešení LP problémů zkonstruovaných za běhu programu. Vstupem je každopádně objekt třídy `Problem`.

LP problém uložený v souboru

RS solver dokáže řešit LP problémy v jednoduchém formátu, který si načrtneme. Opět použijeme syntaxi vycházející z EBNF, popsanou v části 4.5.1.

```
lp problem = "#", ws, variables, ws , constraints, ws,
        direction, ws, objective function, list of constraints;
variables = natural number; (* number of variables *)
constraints = natural number; (* number of constraints *)
direction = "0" | "1"; (* minimize or maximize *)
objective function = coefficients; (* coefficients of objective function *)
list of constraints = constraints * constraint;
constraint = type, ws, rhs, ws, lhs, ws; (* understood as: lhs type rhs *)
type = ">" | "=" | "<";
rhs = number;
lhs = coefficients;
coefficients = variables * (number, ws);
ws = ? one or more white space characters ?;
natural number = ? integer greater than zero, eg. 1, 2, ... ?;
number = ? number that can be understood as real number ?;
```

Oddělovači jsou bílé znaky. Hlavičku tvoří počet proměnných a počet omezení. Dále následuje směr a koeficienty kriteriální funkce a nejrozsáhlejší část tvoří seznam omezení.

Ilustrujme si zadání problému příkladem. LP problému

$$\begin{aligned} \text{Maximalizujte} \quad & 3x - 5y \\ \text{při dodržení} \quad & x + y \leq 5 \\ & -x + 2y \geq 12 \\ & 5x - 6y = 23 \end{aligned} \tag{4.1}$$

odpovídá soubor zobrazený na obrázku 4.17.

```
# 2 3
1       3  -5
< 5     1   1
> 12   -1   2
= 23    5  -6
```

Obrázek 4.17: LP problém (4.1) zadaný v souboru.

LP problém konstruovaný „ručně“

Samozřejmě je také možné vytvořit LP problém přímo v programu a RS solver použít jako knihovnu. Slouží k tomu příslušné metody třídy `Problem`, především metoda `add_constr`. Do dalších detailů ale nebudeme zbytečně zacházet.

Bližší informace týkající se obou způsobů zadání, stejně jako implementační detaily mohou případní zájemci nalézt v souboru `README`, případně přímo ve zdrojových kódech.

4.8.3 Zhodnocení výsledků

I přesto, že se přes naši veškerou snahu nepodařilo ve výpočetním výkonu vyrovnat knihovně GLPK, považujeme vzniklý nástroj za relativní úspěch, a to především z těchto důvodů:

1. Implementace poměrně blízce kopíruje matematickou definici algoritmu, proto může posloužit například zájemcům o studium simplexové metody.

2. RS solver má menší paměťové nároky než GLPK díky úspornému uložení dat v řídkých maticích a vektorech.

3. Od začátku byla snaha psát kód velmi čitelně s ohledem na možnost dalšího vývoje. Pokud bychom opět porovnávali s knihovnou GLPK, které pravděpodobně (s trochou nadsázky) rozumí pouze sám autor, můžeme s klidným srdcem říct, že zdrojové texty jsou velmi čitelné a připravené k dalšímu vývoji.

Kapitola 5

Závěr

V této práci jsme se zabývali především popisem teoretických východisek, návrhem a nakonec i implementací knihovny schopné najít korelované ekvilibrium v potenciálně obrovských nekooperativních hrách. Soustředili jsme se nejen na matematickou stránku definic a precizní popis problémů, ale zároveň jsme se vždy snažili prohlédnout skrz a vystihnout podstatu, případně se i více zamyslet nad probíranou problematikou, často i za cenu jisté neformálnosti či vágnosti.

V kapitole 2 jsme se seznámili s teorií her jako disciplínou aplikované matematiky, jež se používá k modelování rozhodovacích situací napříč rozmanitými obory lidského zkoumání; především jsme se zaměřili na nekooperativní hry v normální formě. Představili jsme si některé základní typy ekvilibrií (rovnovážných bodů), které se při analýze modelů využívají, a jejich vlastnosti. Hlouběji jsme se v tomto směru věnovali ekvilibriu korelovanému, jež umožňuje určitou míru synchronizace konání hráčů.

Jednou z nejdůležitějších technik pro praktické řešení her je eliminace dominovaných strategií. Vychází ze dvou důležitých předpokladů, a sice z racionality hráčů a z takzvané *common knowledge* („společné znalosti“). Ty nám umožňují často velmi výrazně redukovat strategický prostor hry, aniž by se změnila ekvilibria. Jelikož řešení reálných her je v plné velikosti ve většině případů nemožné, je provedení eliminace v podstatě jakousi „nutnou podmínkou“ pro nalezení korelovaného ekvilibria.

Technika eliminace je dále v mírně upravené formě prezentována v kapitole 3, a to jako redukce G-matice. Důkladně se předtím věnujeme popisu právě této G-matice, jež je strukturou velmi názorně reprezentující vícehráčové hry, která je zároveň velmi vhodná ke zmenšení stavového prostoru hry. Druhým pilířem této kapitoly je bezesporu popis jedné z nejznámějších oblastí operačního výzkumu, a sice lineárního programování. Podrobně jsme se jím zabývali z toho důvodu, že na něj lze převést problém hledání korelovaného ekvilibria. Konkrétní způsob rovněž rozvádíme právě v této kapitole. Dále jsme se věnovali popisu systému OpenMP, určenému k paralelizaci programů na architekturách se sdílenou pamětí.

Za klíčovou nezbývá než označit kapitolu 4. Především se v ní zaobíráme konkrétní strukturou a funkcionalitou CE-solveru, knihovny pro hledání korelovaného ekvilibria, kterou jsme vytvořili. Nejprve shrnujeme základní koncepci, jež jsme se v průběhu návrhu a implementace drželi. Poté následuje popis tříd, ze kterých je nástroj složen, a nejdůležitějších algoritmů, jež obstarávají minimalizaci hry pomocí technik inspirovaných zmíněnou G-maticí a její redukcí. Načrtli jsme základní myšlenky paralelizace výpočtu a s tím spojené úskalí. Samozřejmě nemůžeme opomenout experimentální zhodnocení výsledků implementace, které především ukázaly řádové zlepšení výkonu oproti prototypové implementaci.

Spíše okrajově se pak zabýváme dalšími dvěma vytvořenými programy. Prvním z nich je Game Dominance Analyzer, jednoduchý nástroj pro analýzu vlastností strategií hráčů v zadané hře. GDA hojně využívá tříd navržených a implementovaných původně pro CE-solver, což svědčí o jejich znovupoužitelnosti. Druhým výsledkem je knihovna či nástroj RS-solver, která si dala z úkol řešit především řídké problémy z oblasti lineárního programování. Výkonností se sice se specializovaným softwarem nemůže srovnat, ale přináší i určitá pozitiva, jako například čitelný kód, zřetelnou implementaci některých pokročilejších technik nebo menší paměťové nároky (v porovnání s knihovnou GLPK).

Za největší přínos této diplomové práce považujeme právě vytvoření CE-solveru. Ukázali jsme, že oproti prototypové implementaci si z hlediska výkonu stojí velmi dobře. Při testování na reálných hrách používaných k modelování trhu s elektřinou se ukázala řádově vyšší rychlost a (rovněž řádově) nižší spotřeba paměti. Navíc jde o koncepční řešení postavené na objektově orientovaném návrhu, což zajišťuje jednoduchost pochopení, vysokou čitelnost a nezanedbatelnou znovupoužitelnost.

Naskýtá se několik dalších možností vývoje, vycházejících z této práce. Ačkoliv má CE-solver obecně velmi dobré vlastnosti, jistě existuje prostor pro další zlepšení či rozšíření. Nabízí se například možnost dále vylepšovat použité datové struktury, jejichž výměna je ve většině případů díky objektově orientovanému návrhu a snaze o co nejjednodušší rozhraní velmi snadná.

S použitím vytvořených struktur by mohlo být zajímavé doplnit zatím v podstatě jednoúčelovou knihovnu na rozsáhlejší balík knihoven, zaměřených na efektivní analýzu a řešení matematických her. Horkým kandidátem na implementaci je dle našeho názoru metoda FDDS (*Fast Detection of Dominant Strategies*) – ta poskytuje poměrně přesnou představu o velmi rozsáhlých hrách, aniž by je celé řešila [13].

Velmi žádoucí by také bylo vytvořit komplexní model, který by opravdu počítal užitky nezanedbatelnou dobu, a na něm CE-solver vyzkoušet. Při dostatečné velikosti by pak byla příležitost zaměřit se například na algoritmy odebírání vypočtených hodnot z cache při případném přeplnění.

Literatura

[1] Aumann, R.: Subjectivity and Correlation in Randomized Strategies. In *Journal of Mathematical Economics 1*, Elsevier Science S.A., 1974, s. 67–96.

[2] Bertsimas, D.; Tsitsiklis, J. N.: *Introduction to Linear Optimization.* Athena Scientific, February 1997, ISBN 978-1-886529-19-9.

[3] Chapman, B.; Jost, G.; van der Pas, R.: *Using OpenMP: Portable Shared Memory Parallel Programming.* The MIT Press, October 2007, ISBN 978-0262533027.

[4] Chinneck, J. W.: Practical Optimization: A Gentle Introduction [online]. `http://www.sce.carleton.ca/faculty/chinneck/po.html`, 2007-11-20 [cit. 2009-05-07].

[5] Contributors: Cooperative game [online]. `http://en.wikipedia.org/w/index.php?title=Cooperative_game&oldid=258485514`, 2008-12-17 [cit. 2009-01-07].

[6] Contributors: Correlated equilibrium [online]. `http://en.wikipedia.org/w/index.php?title=Correlated_equilibrium&oldid=258487113`, 2008-12-17 [cit. 2009-05-04].

[7] Contributors: Game theory [online]. `http://en.wikipedia.org/w/index.php?title=Game_theory&oldid=262184164`, 2009-01-05 [cit. 2009-01-07].

[8] Contributors: Linear programming [online]. `http://en.wikipedia.org/w/index.php?title=Linear_programming&oldid=262073884`, 2009-01-05 [cit. 2009-01-07].

[9] Contributors: Chicken (game) [online]. `http://en.wikipedia.org/w/index.php?title=Chicken_(game)&oldid=286711107`, 2009-04-28 [cit. 2009-05-04].

[10] Dantzig, G. B.; Thapa, M. N.: *Linear Programming 2: Theory and Extensions.* Springer, July 2003, ISBN 978-0387986135, iSBN 978-0387986135.

[11] Fourer, R.: Linear Programming Frequently Asked Questions [online]. `http://www.faqs.org/faqs/linear-programming-faq/`, 2005-09-01 [cit. 2009-01-07].

[12] Hillier, F. S.; Lieberman, G. J.: *Introduction to Operations Research.* McGraw-Hill Science, May 2002, ISBN 978-0072535105.

[13] Hrubý, M.: Algorithmic Approaches to Game-theoretical Modeling and Simulation. *AUCO Czech Economic Review*, ročník 2, č. 3, 2008: s. 268–300, ISSN 1802-4696.

[14] Hrubý, M.; Čambala, P.: Efficient Computing of Correlated Equilibria in Multi-Player Games. In *Proceedings of 12th IASTED Conference on Artificial Intelligence and Soft Computing*, ACTA Press, 2008, s. 185–191.

[15] Ing. Martin Hrubý, P.: CE-Solver Swiki - A very efficient implementation of Aumann's Correlated Equilibrium [online]. `http://perchta.fit.vutbr.cz/CE-Solver/1`, 2008-11-23 [cit. 2009-05-17].

[16] McKelvey, R. D.; McLennan, A. M.; Turocy, T. L.: Gambit: Software Tools for Game Theory [online]. `http://www.gambit-project.org`, 2007-01-30 [cit. 2009-04-20].

[17] Morgan, S. S.: A Comparison of Simplex Method Algorithms [online]. `http://www.cise.ufl.edu/ davis/Morgan/index.htm`, 1997 [cit. 2009-05-08].

[18] von Neumann, J.; Morgenstern, O.: *Theory of Games and Economic Behavior*. Princeton University Press, May 2004, ISBN 0-691-11993-7.

[19] Nisan, N.; Roughgarden, T.; Tardos, E.; aj.: *Algorithmic Game Theory*. Cambridge University Press, September 2007, ISBN 978-0521872829.

[20] Osborne, M. J.: *An Introduction to Game Theory*. Oxford University Press, August 2003, ISBN 978-0195128956.

[21] Papadimitriou, C. H.: Computing correlated equilibria in multiplayer games. In *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*, ACM Press, 2005, s. 49–56.

[22] Papadimitriou, C. H.; Steiglitz, K.: *Combinatorial Optimization: Algorithms and Complexity*. Dover Publications, January 1998, ISBN 978-0486402581.

Příloha A

CE-Solver Manual

A.1 Introduction

CE-Solver is a tool for computing Correlated Equilibrium in possibly very large games. It useses slightly modified method proposed by dr. Hruby (see [14]), which is based on elimination of strictly dominated strategies using G-matrix.

A.2 Theory

A.2.1 Correlated equilibrium

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

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

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

A.2.2 G-matrix

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

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

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

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

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

A.2.3 Reduction

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

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

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

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

A.2.4 Example

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

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

Tabulka A.1: Example game

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

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

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

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

A.2.5 Computing CE as LP problem

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

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

Tabulka A.3: G-matrix after one step of elimination

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

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

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

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

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

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

A.3 Implementation

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

A.3.1 Main ideas

CE-solver input

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

Storage of G-matrix

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

Payoff cache

Computing payoffs is computationally hardest part of real-world models. Because of that, we created so called *payoff cache*, which is used to store computed payoffs. When they are needed again, which might happen several times during solving, we do not have to compute them and just read them from this cache (see A.3.2).

Parallelization

Parallelism is a natural choice to speed up computing. Because G-matrix reduction can be quite easily done in parallel, we designed the whole library with parallelism in mind. For this purpose, we chose *OpenMP* system because it is easy to use and provides good results on parallel architectures with shared memory.

A.3.2 Structure

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

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

Solving game

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

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

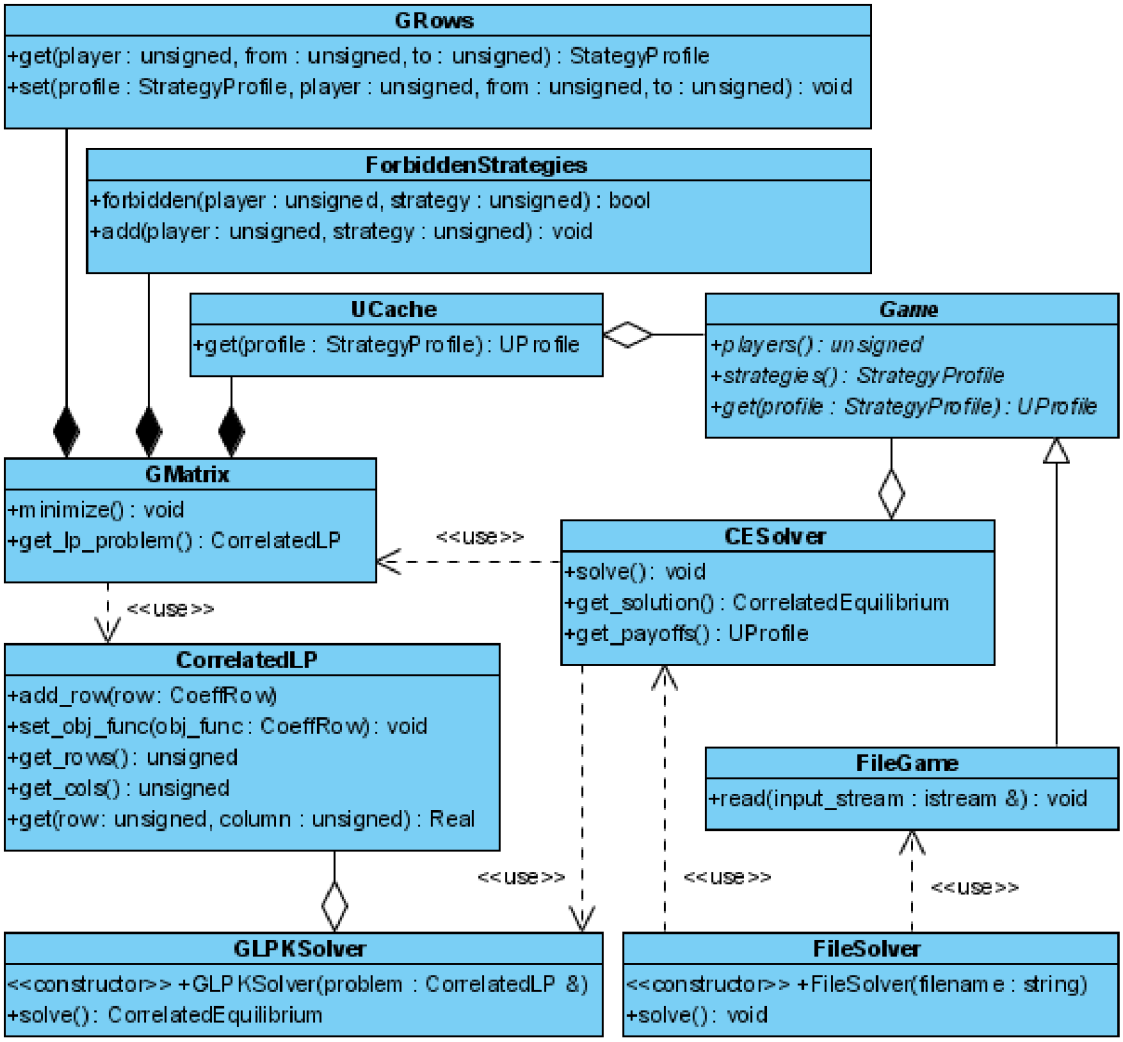
Because we use GLPK as a "black-box"component, the main algorithmic complexity lies in the method `minimize` of `GMatrix`, so we are going to describe this class more thoroughly.

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

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

G-matrix positions – `GRows`

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



Obrázek A.1: Class diagram of CE-solver

Eliminated strategies – `ForbiddenStrategies`

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

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

Cache of computed payoffs – `UCache`

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

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

Obrázek A.2: Iterating through rows of G-matrix

Correlated LP problem – `CorrelatedLP`

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

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

Interface to GLPK – `GLPKSolver`

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

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

A.4 Usage

A.4.1 Library interface

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

Game model – class `Game`

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

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

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

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

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

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

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

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

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

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

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

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

A.4.2 Correlated Equilibrium File Solver

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

File formats

There are two file formats accepted by CEFS.

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

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

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

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

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

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

Tabulka A.4: Game to be saved in file (see figure A.4)

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

Obrázek A.4: Game from table A.4 saved in file

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

Invocation

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

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

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

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

Help can by obtained using argument `-h`.

Mappings file

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

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

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

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

Obrázek A.5: Game from table A.4 saved in file

A.5 Conclusion

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

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