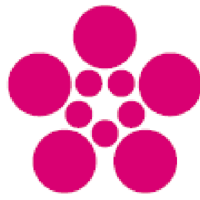


Ekonomická fakulta
Faculty of Economics

Jihočeská univerzita v Českých Budějovicích
University of South Bohemia in České Budějovice

Jihočeská univerzita v Českých Budějovicích

Ekonomická fakulta

Katedra matematiky a informatiky

Bakalářská práce

Vývoj aplikace pro řešení úloh lineárního programování pomocí nástroje Microsoft Solver Foundation

Vypracoval: Pavel Vysušil

Vedoucí práce: Mgr. Radim Remeš

České Budějovice 2016

JIHOČESKÁ UNIVERZITA V ČESKÝCH BUDĚJOVICÍCH
Fakulta ekonomická
Akademický rok: **2014/2015**

ZADÁNÍ BAKALÁŘSKÉ PRÁCE
(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Pavel VYSUŠIL**
Osobní číslo:	**E13558**
Studijní program:	**B6209 Systémové inženýrství a informatika**
Studijní obor:	**Ekonomická informatika**
Název tématu:	**Vývoj aplikace pro řešení úloh lineárního programování pomocí nástroje Microsoft Solver Foundation**
Zadávající katedra:	**Katedra aplikované matematiky a informatiky**

Zásady pro vypracování:

Microsoft Solver Foundation je sada vývojářských nástrojů pro matematické simulace, optimalizaci a modelování. Cílem bakalářské práce je vytvořit aplikaci pro vybraný typ úloh lineárního programování pomocí nástroje Microsoft Solver Foundation.

Metodický postup:

1. Studium odborné literatury.
2. Popis problematiky lineárního programování, typů úloh a metod počítačového řešení.
3. Konstrukce matematického modelu.
4. Implementace modelu ve výsledné aplikaci.

Rozsah grafických prací:	**dle potřeby**
Rozsah pracovní zprávy:	**40 - 50 stran**
Forma zpracování bakalářské práce:	**tištěná**

Seznam odborné literatury:

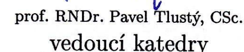
1. **HILLIER, Frederick S. a Gerald J. LIEBERMAN.** *Introduction to Operations Research.* **9. či 10. vydání, New York: McGraw-Hill, 2010, 2015. ISBN 978-007-132483-0, 978-1-259-25318-8.**
2. **JABLONSKÝ, Josef.** *Operační výzkum: kvantitativní metody pro ekonomické rozhodování.* **3. vyd. Praha: Professional Publishing, 2007, ISBN 978-80-86946-44-3.**
3. **Microsoft Solver Foundation 3.1. MICROSOFT.** *Microsoft Developer Network: MSDN Library* **[online]. 2015, 2015.03.20 [cit. 2015-03-27]. Dostupné z: https://msdn.microsoft.com/en-us/library/ff524509.**

Vedoucí bakalářské práce:	**Mgr. Radim Remeš** Katedra aplikované matematiky a informatiky
Datum zadání bakalářské práce:	**9. ledna 2015**
Termín odevzdání bakalářské práce:	**15. dubna 2016**

JIHOČESKÁ UNIVERZITA
V ČESKÝCH BUDĚJOVICÍCH
EKONOMICKÁ FAKULTA
Studentská 13 (26)
370 05 České Budějovice

L.S.

doc. Ing. Ladislav Rolínek, Ph.D.
děkan



prof. RNDr. Pavel Tlustý, CSc.
vedoucí katedry

V Českých Budějovicích dne 27. března 2015

Prohlášení

Prohlašuji, že svoji bakalářskou/diplomovou práci jsem vypracoval/a samostatně pouze s použitím pramenů a literatury uvedených v seznamu citované literatury. Prohlašuji, že v souladu s § 47 zákona č. 111/1998 Sb. v platném znění souhlasím se zveřejněním své bakalářské/diplomové práce, a to v nezkrácené podobě - v úpravě vzniklé vypuštěním vyznačených částí archivovaných Ekonomickou fakultou elektronickou cestou ve veřejně přístupné části databáze STAG provozované Jihočeskou univerzitou v Českých Budějovicích na jejích internetových stránkách, a to se zachováním mého autorského práva k odevzdanému textu této kvalifikační práce. Souhlasím dále s tím, aby toutéž elektronickou cestou byly v souladu s uvedeným ustanovením zákona č. 111/1998 Sb. zveřejněny posudky školitele a oponentů práce i záznam o průběhu a výsledku obhajoby kvalifikační práce. Rovněž souhlasím s porovnáním textu mé kvalifikační práce s databází kvalifikačních prací Theses.cz provozovanou Národním registrem vysokoškolských kvalifikačních prací a systémem na odhalování plagiátů.

Datum 14.4.2017 Podpis studenta

Poděkování

Touto cestou bych rád poděkoval vedoucímu mé bakalářské práce, panu Mgr. Radimu Remešovi, za vřelý přístup, podnětné rady a poskytnutí všech potřebných informací v rámci konzultací k této bakalářské práci. Dále bych chtěl poděkovat svým rodičům za podporu, kterou mi vždy ochotně poskytli. Díky patří i mému zaměstnavateli za to, že mi po celou dobu studia vycházel maximálně vstříc.

Obsah

Obsah 1

1 Úvod 3

2 Lineární programování 5

2.1 Historie a úvod . . . 5
2.2 Praktické řešení úlohy LP . . . 6
2.2.1 Rozpoznání problému a jeho definice . . . 6
2.2.2 Formulace ekonomického modelu . . . 6
2.2.3 Formulace matematického modelu . . . 7
2.2.4 Vlastní řešení matematického modelu . . . 7
2.2.5 Interpretace výsledků a jejich následná verifikace . . . 7
2.2.6 Implementace řešení . . . 7
2.3 Matematický model úlohy LP . . . 8
2.4 Celočíselné programování . . . 9
2.5 Metoda Simplex . . . 10
2.5.1 I. fáze . . . 10
2.5.2 II. fáze . . . 12
2.5.3 Možnosti zakončení výpočtu . . . 14

3 Problém splňování podmínek 15

4 Sudoku 17

4.1 Historie . . . 17
4.2 Pravidla . . . 19
4.3 Obtížnost . . . 20
4.4 Matematické modely . . . 21
4.4.1 Sudoku definované jako úloha binárního LP . . . 21
4.4.2 Řešitelnost Sudoku daného binárním lineárním modelem . . . 22
4.4.3 Sudoku definované jako úloha CSP . . . 22

5 Použité technologie a frameworky 23

5.1 Framework .NET a programovací jazyk C# . . . 23
5.2 Microsoft Solver Foundation . . . 23
5.2.1 Instalace MSF . . . 24

5.2.2 Integrace knihovny MSF v projektu Visual Studia 25
5.3 Windows Presentation Foundation 26
5.3.1 Hlavní výhody WPF 26
5.3.2 XAML 27
5.3.3 Data Binding 27

6 Implementace programu 28
6.1 Koncept aplikace a její funkcionality 28
6.1.1 Editace mřížky 28
6.1.2 Generování zadání zvolené obtížnosti 28
6.1.3 Řešitel Sudoku 29
6.1.4 Nahrání dat ze souboru 29
6.1.5 Uložení dat do souboru 30
6.1.6 Obnovení mřížky 30
6.1.7 Návrat do počátečního stavu 30
6.1.8 Kontrola základních pravidel Sudoku 30
6.2 Objektový model aplikace 31
6.3 Třídy realizující View 31
6.3.1 App 31
6.3.2 InfoWindow 31
6.3.3 MainWindow 32
6.3.4 SudokuGridView 33
6.4 Třídy realizující ModelView 34
6.4.1 GridItemModelView 35
6.4.2 SudokuModelView 35
6.5 Třídy realizující Model 37
6.5.1 FileIO 38
6.5.2 SudokuGenerator 38
6.5.3 SudokuSolver 39
6.5.4 SudokuTransform 43
6.6 Pomocné třídy 44
6.6.1 Třída Constants 44

7 Závěr 45

Summary 46

Použitá literatura 47

Seznam obrázků 49

Seznam zdrojových kódů 50

Seznam tabulek 51

1. Úvod

Lineární programování je jedním z nejpoužívanějších nástrojů operačního výzkumu, jehož metody se používají pro nalezení optimálních vstupních parametrů četného množství úloh, zejména ekonomického charakteru. Nejčastěji jsou tyto metody využívány při optimalizaci výrobních procesů, v plánování, ale i v dalších typech úloh a s jejich aplikací se dnes setkáme velmi často.

Cílem této práce je vytvořit programovou aplikaci, která by demonstrovala, jakým způsobem může vývojář při implementaci softwarového řešitele úlohy lineárního programování použít volně dostupnou knihovnu pro matematické modelování a optimalizaci - framework Microsoft Solver Foundation (dále jen MSF) a zhodnotit jeho použitelnost v praxi. Typické příklady úloh ekonomického charakteru, jejichž správná řešení se dají verifikovat (neboť jsou včetně výpočetních postupů popsána v dostupné literatuře), jsou rozsahem příliš malé na to, aby bylo možné objektivně posoudit přínos použití nástroje MSF a jeho případná omezení. Hlavním důvodem je zejména malý počet strukturních proměnných a omezujících podmínek matematického modelu. Zde se proto nabízí známý hlavolam Sudoku, který lze rovněž formulovat jako úlohu lineárního programování a který velikostí svého matematického modelu (pokud jde o již uvedený počet strukturních proměnných a omezujících podmínek) řádově převyšuje teoretické příklady z literatury.

Kromě rozsahu je Sudoku vhodnou úlohou i z dalších důvodů. Tím prvním je fakt, že správnost nalezeného řešení lze i přes velikost matematického modelu snadno verifikovat. Problém kompletního vyplnění mřížky Sudoku je totiž primárně určen člověku a nalezená kombinace čísel lze proto snadno ověřit pouhým „pohledem“, aniž by bylo nutné její správnost nějak složitě početně dokazovat. Zde je však nutné podotknout, že to se nevztahuje na posouzení, zda se jedná o validní zadání problému (takové, ke kterému existuje pouze jedno řešení). Na potvrzení této vlastnosti mřížky už pouhý „pohled“ nestačí. Druhým důvodem je možnost aplikaci dobře vyzkoušet a ověřit tak její obecnou použitelnost, neboť lze na internetu snadno volně získat spoustu zadání u kterých je často známá i obtížnost, k dispozici jsou i celé databáze zadání.

V teoretické části práce najdeme úvod do lineárního programování, postup pro řešení úlohy lineárního programování, konstrukci obecného matematického modelu, seznámení s metodou Simplex, problematiku splňování podmínek. Dále v práci najdeme informace o samotné hře Sudoku. Ta je formulována dvojím způsobem, v podobě dvou odlišných matematických modelů. První matematický model odpovídá úloze celočíselného programování s bivalentními proměnnými a jeho programová implementace je realizována řešitelem třídy *SimplexSolver* (nástroje MSF), druhý matematický model je popsán jako problém splňo-

vání podmínek (CSP[1]) a implementován řešitelem třídy *ConstraintSystem* (nástroje MSF).

Praktická část práce je věnována vývoji samotné aplikace, použitým technologiím a popisuje způsob, jak výsledná aplikace funguje. Softwarová aplikace, která je součástí této práce, byla vytvořena ve vývojovém prostředí MS Visual Studio, napsána v jazyce C# a kromě frameworku .NET využívá technologie Windows Presentation Foundation (dále WPF). Hlavní důvody, proč byl pro implementaci grafického uživatelského prostředí zvolen framework WPF jsou potom zmíněn v kapitole 5.3.1.

V závěru práce je zhodnocen přínos nástroje MSF a posouzena jeho využitelnost při řešení úlohy lineárního programování.

[1]Constraint Satisfaction Problem

2. Lineární programování

2.1 Historie a úvod

Lineární programování je jedním z nejvýznamnějších nástrojů operačního výzkumu, díky němuž došlo doslova k revolučnímu vývoji vědy a to hned na poli několika vědních oborů. Metody lineárního programování přinesly možnost stanovit obecné cíle a vytvořit postup, jak těchto cílů nejlépe dosáhnout ve chvíli, kdy řešíme velmi složité praktické problémy. „*Nástroji, kterými toho můžeme dosáhnout, jsou způsoby formulování problémů reálného světa pomocí detailních matematických výrazů (matematické modely), techniky pro řešení modelů (algoritmy) a stroje pro vykonání jednotlivých kroků algoritmů (počítače a software).*“ Tyto možnosti se naskytly krátce po druhé světové válce, když v roce 1947 George B. Dantzig představil světu metodu *Simplex*. Dalšími průkopníky té doby byli von Neumann, Kantorovič, Leontief a Koopmans (Dantzig & Thapa, 1997).

Krátce nato došlo v oblasti lineárního programování k prudkému vývoji, jehož dopad byl výjimečný. Nejčastější využití metod lineárního programování nalezneme při řešení obecného problému, jak rozdělit omezené zdroje mezi vícero činností tím nejlepším způsobem. Většinou se jedná o nastavení intenzit činností, soupeřících mezi sebou o omezené zdroje, které jsou potřebné pro jejich vykonání. Jinými slovy, je tedy třeba určit, jak alokovat zdroje jednotlivým činnostem a to nastavením jejich intenzity. K popisu problému se v lineárním programování používá matematický model. Slovo *lineární* znamená, že všechny matematické funkce popsané v tomto modelu musí být *lineárními funkcemi.* Naproti tomu ale slovo *programování* zde nemá význam počítačového programování, je užito spíše jako synonymum pro plánování. Lineární programování tak vlastně označuje *plánování činností*, jejichž účelem je dosažení optimálního výsledku, to znamená takového, který mezi všemi přípustnými alternativami nejlépe splňuje specifikované cíle (podle zadání matematického modelu). Ačkoli je přiřazení zdrojů k jednotlivým činnostem nejčastějším způsobem aplikace lineárního programování, nalezneme i spoustu dalších důležitých využití. Jakýkoli problém, jehož matematický model odpovídá nejobecnějšímu popisu modelu lineárního programování, je ve skutečnosti úlohou lineárního programování a jako takový lze tudíž vyřešit pomocí velmi účinného postupu, který se nazývá *Simplexová metoda.* Tímto postupem lze nalézt i řešení úloh velmi velkého rozsahu (Hillier & Lieberman, 2001).

2.2 Praktické řešení úlohy LP

Pro praktické řešení reálného rozhodovacího problému je nutné úlohu rozčlenit na několik na sebe navazujících fází. Jak uvádí (Jablonský, 1999), je vhodné učinit následující kroky:

2.2.1 Rozpoznání problému a jeho definice

Nejprve musíme daný problém pochopit a následně jej můžeme slovně popsat a to jak z hlediska struktury a souvisejících procesů, tak i z hlediska prostředků, jimiž lze řešit.

2.2.2 Formulace ekonomického modelu

Reálný problém bývá zpravidla velmi složitý na to, aby bylo možné snadno postihnout všechny jeho zákonitosti. V praxi to často není nutné a někdy ani žádoucí. Je třeba se soustředit pouze na nejpodstatnější prvky systému a na vazby mezi nimi. V ekonomickém modelu by měly být uvedeny především:

- *Cíl analýzy*
  Zde se jedná o jednoznačné určení cílového stavu, v němž se má zkoumaný systém ocitnout po vyřešení problému. Jedná se o nalezení extrému účelové funkce za současného splnění všech omezujících podmínek. Zpravidla jde o hledání maxima účelové funkce (cílem je maximalizace zisku) nebo minima účelové funkce (cílem je minimalizace nákladů).

- *Popis procesů*
  Jde o popis všech aktivit, které v reálném systému probíhají a které mají podstatný vliv na cíl analýzy. Ve výrobním programu může být procesem například výroba nějakého výrobku, intenzita tohoto procesu potom určuje objem vyrobené produkce. Ten má potom vliv na cíl analýzy, jímž může být maximalizace zisku, případně minimalizace nákladů.

- *Popis činitelů*
  Všechny procesy nemohou být prováděny s libovolnou intenzitou. Na jejich realizaci má vliv spousta činitelů, které je třeba vzít v úvahu, neboť s sebou přináší určitá omezení. Nejčastějším činitelem je omezení zdrojů (při výrobě obvykle ve formě surovin, času, pracovní síly, energie apod.), mohou jím být požadavky na minimální či maximální objem výroby a velmi často také relace mezi vstupními objemy jednotlivých zdrojů, případně relace mezi výstupními objemy finální produkce.

- *Popis vzájemného vztahu mezi procesy, činiteli a cílem analýzy*
  Vytvoření jednotky konkrétního produktu je svázáno s určitou spotřebou zdrojů, kterou je třeba definovat. A dále příspěvek, který bude vytvoření jednotky finálního produktu přinášet (tedy jaký bude její dopad na hodnotu účelové funkce).

2.2.3 Formulace matematického modelu

Ekonomický model se podobá slovní úloze v matematice. Je vyjádřen slovním a numerickým popisem problému. Matematický model je v podstatě jeho formální přepis do jazyka matematických výrazů a symbolů. Jednotlivé položky ekonomického modelu můžeme vyjádřit následujícím způsobem:

- *Cíl analýzy*
  V úlohách LP je vyjádřen lineární funkcí n proměnných (jedná se o již zmíněnou účelovou funkci), dále je specifikováno, jaký extrém této funkce hledáme.

- *Procesy*
  Jsou vyjádřeny proměnnými (které nazýváme strukturními proměnnými), intenzity provádění jednotlivých procesů jsou potom hodnoty těchto proměnných.

- *Činitelé*
  Jejich vyjádření je rozmanité a závisí na povaze dané úlohy a metodě jejího řešení, nejběžnější je vyjádření ve formě soustavy lineárních nerovnic (případně lineárních rovnic).

- *Vazby mezi procesy, činiteli a cílem analýzy*
  Tyto vztahy jsou popsány pomocí sady parametrů, které se nedají měnit. Slouží pouze k zachycení těchto vazeb, které vycházejí z vnitřní podstaty problému.

2.2.4 Vlastní řešení matematického modelu

Řešení matematického modelu je potom už pouze technickou záležitostí. Existují specifické postupy, jakými lze konkrétní typy matematických modelů řešit. Zároveň existuje celá řada počítačových aplikací, které lze pro výpočty použít. Ty mají svá technická (případně licenční) omezení a mezi sebou se liší i ve spoustě dalších parametrů. Jedním z těch podstatných je rozsah úlohy, na kterou lze daný program aplikovat. Další z možností je pro výpočet vyvinout vlastní programovou aplikaci a algoritmus výpočtu implementovat vlastními metodami nebo využít nějakou dostupnou matematickou knihovnu.

2.2.5 Interpretace výsledků a jejich následná verifikace

Výsledky získané řešením ekonomického modelu je nutné ověřit a zjistit, zda byl ekonomický model sestaven správně. Zda nebyly opomenuty některé stránky systému a zda získané optimální řešení má i své opodstatnění v praxi (zda je vůbec použitelné).

2.2.6 Implementace řešení

Pokud byla fáze verifikace úspěšná, lze přistoupit k implementaci řešení v rámci analyzovaného reálného problému. To by mělo přispět ke zlepšení fungování daného systému za současného naplnění sledovaného cíle.

2.3 Matematický model úlohy LP

Matematický model úlohy LP v obecné podobě lze zapsat následujícím způsobem:
maximalizovat (minimalizovat) účelovou funkci

$$z = c_1x_1 + c_2x_2 + \ldots + c_nx_n \tag{1}$$

při splnění podmínek

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n &\leq b_1 \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n &\leq b_2 \\ &\vdots \\ a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n &\leq b_m \end{aligned} \tag{2}$$

$$x_j \geq 0,\ j = 1, 2, \ldots, n \tag{3}$$

To lze zapsat i formou maticového zápisu:
Maximalizovat (minimalizovat) účelovou funkci

$$z = \mathbf{c}^{\mathrm{T}}\mathbf{x} \tag{4}$$

při splnění podmínek

$$\mathbf{Ax} \leq \mathbf{b} \tag{5}$$

$$\mathbf{x} \geq \mathbf{0} \tag{6}$$

kde $\mathbf{A}$ je matice strukturních koeficientů o rozměru $m \times n$,
$\mathbf{c}^{\mathrm{T}} = (\mathrm{c_1, c_2, \ldots, c_n})$ je n - složkový řádkový vektor cenových koeficientů,
$\mathbf{x} = (\mathrm{x_1, x_2, \ldots, x_n})^{\mathrm{T}}$ je n - složkový sloupcový vektor strukturních proměnných,
$\mathbf{b} = (\mathrm{b_1, b_2, \ldots, b_m})^{\mathrm{T}}$ je m - složkový sloupcový vektor hodnot pravé strany,
$\mathbf{0} = (0, 0, \ldots, 0)^{\mathrm{T}}$ je n - složkový sloupcový nulový vektor.

Matematický model úlohy lze zapsat i pomocí sumací (ten budeme dále používat):
maximalizovat (minimalizovat)

$$z = \sum_{j=1}^{n} c_jx_j \tag{7}$$

při splnění vlastních omezení a podmínek nezápornosti

$$\sum_{j=1}^{n} a_{ij}x_j \leq b_i,\ i = 1, 2, \ldots, m \tag{8}$$

$$x_j \geq 0,\ j = 1, 2, \ldots, n \tag{9}$$

kde n je počet strukturních proměnných modelu, m je počet vlastních omezení,
$c_j,\ j = 1, 2, \ldots, n$ je cenový koeficient příslušející j-té proměnné,
$b_i,\ i = 1, 2, \ldots, m$ je hodnota pravé strany příslušející i-tému vlatnímu omezení a
$a_{ij},\ i = 1, 2, \ldots, m,\ j = 1, 2, \ldots, n$ je strukturní koeficient vyjadřující vztah mezi i-tým činitelem a j-tým procesem.

Přípustné řešení je takové řešení, které vyhovuje všem vlastním omezením i podmínkám nezápornosti. *Optimální řešení* je přípustné řešení s nejlepší hodnotou účelové funkce. Vlastní omezení a podmínky nezápornosti jsou dány ve formě soustavy lineárních nerovnic. Abychom mohli získat základní řešení úlohy, musíme tuto soustavu nejprve transformovat na soustavu ekvivalentních rovnic. Tu získáme tak, že původní soustavu n proměnných rozšíříme o maximálně m proměnných, označovaných jako *přídatné proměnné*. Rovnici získáme z nerovnice typu „$\leq$“ přičtením, u nerovnice typu „$\geq$“ odečtením proměnné na levé straně původní nerovnice. *Základní řešení ekvivalentní soustavy rovnic* získáme tak, že z původní soustavy m rovnic a $m+n$ proměnných vybereme n tzv. *nezákladních proměnných*, ty položíme rovny 0 a vyřešíme vzniklou soustavu m rovnic o m neznámých, kterým říkáme *základní proměnné*. Pokud je některá z těchto m základních proměnných rovna 0, označuje se takové řešení za *degenerované základní řešení* (ekvivalentní soustavy rovnic). Vybrat m základních proměnných z celkového počtu $(m+n)$ lze $\binom{m+n}{m}$ způsoby a proto se může stát, že vypočtené základní řešení soustavy ekvivalentních rovnic bude obsahovat proměnné se zápornými hodnotami a poruší tak některou z podmínek nezápornosti (nebude přípustné). Základní řešení úlohy lineárního programování je přípustné základní řešení ekvivalentní soustavy rovnic této úlohy (Jablonský, 1999).

2.4 Celočíselné programování

Mezi úlohy lineárního programování můžeme zahrnout i úlohy celočíselného programování. Jsou to v podstatě úlohy lineárního programování rozšířené o tzv. *podmínky celočíselnosti*, které zajišťují, aby měly všechny proměnné nebo jen jejich část celočíselné hodnoty. Podle toho, zda se podmínky celočíselnosti vztahují na všechny proměnné v modelu nebo pouze na jejich podmnožinu, rozdělujeme úlohy na *ryze celočíselné* a *smíšeně celočíselné*[1]. Pokud navíc mohou proměnné v modelu nabývat pouze hodnot 0 a 1, potom hovoříme o *binárních* nebo *bivalentních proměnných*. Oba požadavky na vlastnosti proměnných většinou vyplývají přímo z charakteru dané úlohy a jejího modelu. Celočíselné programování se zabývá celou řadou problémů, nejčastějšími jsou úlohy výrobního plánování, úlohy o dělení materiálu, nutriční úlohy, přiřazovací problém, dopravní problém a okružní dopravní problém (Jablonský, 1999).

[1] V anglické literatuře se pro celočíselné programování používá zkratka ILP (Integer Linear Programming), pro smíšené celočíselné programování MILP (Mixed-Integer Linear Programming).

2.5 Metoda Simplex

Pro nalezení optimálního řešení úlohy lineárního programování se často používá iterační algoritmus, který se nazývá Simplexová metoda. Popis algoritmu je poměrně rozsáhlý, proto uvádím stručnou verzi, včetně ukázkových příkladů, které lze nalézt v (Jablonský, 1999). Primárním cílem je nalezení výchozího základního řešení úlohy lineárního programování. Pokud je nalezeno, vypočítají se postupně v jednotlivých krocích další základní řešení s lepší nebo alespoň stejnou hodnotou účelové funkce. Po konečném počtu opakování musí tento postup vést k nalezení řešení s nejlepší hodnotou účelové funkce nebo ke zjištění, že takové řešení neexistuje. Postup algoritmu se dělí na dvě fáze:

- *I. fáze* - nalezení výchozího základního řešení
- *II. fáze* - iterační postup vedoucí k optimalizaci účelové funkce z

2.5.1 I. fáze

Pokud matice strukturních koeficientů obsahuje soustavu m jednotkových vektorů, ze kterých lze vytvořit jednotkovou matici, potom říkáme, že je soustava m rovnic o $(m+n)$ proměnných v *kanonickém tvaru.* V tomto tvaru má m základních proměnných hodnotu pravé strany rovnice a n nezákladních proměnných hodnotu 0. V případě, že jsou všechny hodnoty pravé strany kladné, jedná se o základní řešení úlohy lineárního programování a za těchto podmínek lze přímo přistoupit k optimalizaci účelové funkce (fáze II.). To nastane u úlohy, která má všechna vlastní omezení zadána jako nerovnice typu „$\leq$“. Pokud vlastní omezení obsahují rovnost nebo nerovnost typu „$\geq$“ , musí se ekvivalentní soustava rovnic rozšířit o *pomocné proměnné*, protože zadaná soustava není v kanonickém tvaru. Tyto pomocné proměnné není třeba přidávat u nerovnic typu „$\leq$“ z toho důvodu, že při převodu nerovnice na rovnici už byla vytvořena přídatná proměnná, která sama o sobě zajišťuje získání jednotkového vektoru v matici strukturních koeficientů. U rovnic a nerovnic typu „$\geq$“ je naopak třeba pomocnou proměnnou přičíst. Rozdíl mezi přídatnými a pomocnými proměnnými spočívá v tom, že přídatné proměnné slouží k převodu soustavy omezujících podmínek na soustavu rovnic, pomocné proměnné potom zabezpečují její kanonický tvar.

Mějme dán následující matematický model úlohy s účelovou funkcí z, kterou chceme maximalizovat:

$$\begin{array}{rcrcl} 2x_1 & + & x_2 & = & z \\ 3x_1 & - & x_2 & \leq & 12 \\ x_1 & + & x_2 & \geq & 6 \\ -x_1 & + & 2x_2 & = & 9 \\ & & x_1, x_2 & \geq & 0 \end{array} \tag{10}$$

Soustavu omezujících podmínek převedeme na ekvivalentní soustavu rovnic zavedením přídatných proměnných x_3 a x_4:

$$\begin{array}{rcrcrcrcl} 3x_1 & - & x_2 & + & x_3 & & & = & 12 \\ x_1 & + & x_2 & & & - & x_4 & = & 6 \\ -x_1 & + & 2x_2 & & & & & = & 9 \end{array} \tag{11}$$

Po rozšíření soustavy o přídatné proměnné však soustava není v kanonickém tvaru a nelze z ní tak získat základní řešení. Kanonického tvaru rozšířené soustavy dosáhneme přidáním pomocných proměnných y_1 a y_2:

$$\begin{array}{rcrcrcrcrcrcl} 3x_1 & - & x_2 & + & x_3 & & & & & & & = & 12 \\ x_1 & + & x_2 & & & - & x_4 & + & y_1 & & & = & 6 \\ -x_1 & + & 2x_2 & & & & & & & + & y_2 & = & 9 \end{array} \tag{12}$$

Nyní máme soustavu v kanonickém tvaru. Řešením soustavy jsou vektory $\boldsymbol{x} = (0, 0, 12, 0)$ a $\boldsymbol{y} = (6, 9)$. Je zřejmé, že dosadíme-li nulové hodnoty x_1 a x_2 do soustavy (10), nebude splněna druhá podmínka. Získané řešení proto není přípustným řešením úlohy. Abychom jej nalezli, musíme najít takové řešení soustavy (12), ve kterém budou všechny pomocné proměnné rovny 0. Toho docílíme zavedením *pomocné účelové funkce*, nebo použitím *prohibitních cenových koeficientů*. V praxi se první varianta příliš nepoužívá, je vhodnější spíše pro manuální výpočty. Ukážeme si druhý postup. Abychom vyloučili z řešení pomocné proměnné, přidáme je do účelové funkce a přiřadíme jim maximálně nevýhodné koeficienty v podobě konstanty, např. $+M$ pro minimalizaci účelové funkce nebo $-M$ pro maximalizaci. V našem případě by účelová funkce vypadala následovně:

$$z = 2x_1 + x_2 - My_1 - My_2 \tag{13}$$

V řádku účelové funkce z_j v simplexové tabulce (1) vzniklé z této soustavy, vyloučíme Gaussovou eliminací koeficienty M u pomocných proměnných, čímž získáme upravenou účelovou funkci z_j^*. Tu potom používáme v následných výpočtech. Postup dalšího řešení už je analogický s fází II.

zákl.prom	x_1	x_2	x_3	x_4	y_1	y_2	β_i	$t = \frac{\beta_i}{a_{ik}}$
x_3	3	-1	1	0	0	0	12	-
y_1	1	1	0	-1	1	0	9	6
y_2	-1	2	0	0	0	1	6	$\frac{9}{2}$
z_j	-2	-1	0	0	M	M	0	
z_j^*	-2	-3M-1	0	$+M$	0	0	-15M	

Tabulka 1: Prohibitivní koeficienty v simplexové tabulce

2.5.2 II. fáze

V případě, že úspěšně proběhla první fáze nebo již bylo zadání úlohy v kanonickém tvaru, probíhá druhá fáze - optimalizace účelové funkce. Je dán následující matematický model úlohy s účelovou funkcí z, kterou chceme maximalizovat:

$$\begin{array}{rcrcl} 20x_1 & + & 14x_2 & = & z \\ 0.5x_1 & + & 0.25x_2 & \leq & 40 \\ 0.5x_1 & + & 0.5x_2 & \leq & 60 \\ & & 0.25x_2 & \leq & 25 \\ & & x_1, x_2 & \geq & 0 \end{array} \tag{14}$$

Po doplnění přídatných proměnných je soustava v kanonickém tvaru:

$$\begin{array}{rcrcccccrcl} 0.5x_1 & + & 0.25x_2 & + & x_3 & & & & & = & 40 \\ 0.5x_1 & + & 0.5x_2 & & & + & x_4 & & & = & 60 \\ & & 0.25x_2 & & & & & + & x_5 & = & 25 \end{array} \tag{15}$$

Rovnici účelové funkce z prvního řádku soustavy (14) vyjádříme v tzv. *anulovaném tvaru* odečtením členů její levé strany:

$$z - 20x_1 - 14x_2 = 0 \tag{16}$$

Soustavu rovnic (15) společně s rovnicí (16) a hodnotami t vyjádřenými jako $t = \frac{\beta_i}{a_{ik}}$ zapíšeme do schématu, který nazýváme *simplexová tabulka* (2). Řádek z_j v simplexové tabulce obsahuje tzv. *redukované ceny.*

zákl.prom	x_1	x_2	x_3	x_4	x_5	β_i	$t = \frac{\beta_i}{a_{ik}}$
x_3	$\frac{1}{2}$	$\frac{1}{4}$	1	0	0	40	80
x_4	$\frac{1}{2}$	$\frac{1}{2}$	0	1	0	60	120
x_5	0	$\frac{1}{4}$	0	0	1	25	-
z_j	-20	-14	0	0	0		

Tabulka 2: Výchozí simplexová tabulka

Nyní můžeme přistoupit k *testu optimality.* Řešení je optimální, jestliže jsou

- *při maximalizaci účelové funkce všechny redukované ceny nezáporné*: $z_j \geq 0,\ j \in N$
- *při minimalizaci účelové funkce všechny redukované ceny nekladné*: $z_j \leq 0,\ j \in N$

(kde N je množina indexů nezákladních proměnných).

Vidíme, že redukované ceny nezákladních proměnných jsou záporné, nejedná se proto o optimální řešení a je možné nalézt nové základní řešení s lepší hodnotou účelové funkce. Toho lze dosáhnout ve třech krocích:

1. zvolíme *vstupující proměnnou* (klíčový sloupec)
2. zvolíme *vystupující proměnnou* (klíčový řádek)
3. přepočteme simplexovou tabulku tak, že se vstupující proměnná stane základní proměnnou a vystupující proměnná nezákladní proměnnou

Vstupující proměnnou vybereme následovně:

- *při max. účelové funkce s nejnižší redukovanou cenou*: $z_k = min\, z_j, z_j < 0,\ j \in N$
- *při min. účelové funkce s nejvyšší redukovanou cenou*: $z_k = max\, z_j, z_j > 0,\ j \in N$

(kde N je množina indexů nezákladních proměnných).

Vystupující proměnná se potom nachází v řádku s nejnižším koeficientem $t = \frac{\beta_i}{a_{ik}}$.

V naší simplexové tabulce mají obě nezákladní proměnné zápornou redukovanou cenu, řešení tedy není optimální. Klíčovým sloupcem bude sloupec odpovídající proměnné x_1, protože obsahuje redukovanou cenu s nejnižším koeficientem. Klíčovým řádkem bude řádek proměnné x_3, jehož koeficient t je nejnižší. *Klíčový prvek* leží na průsečíku klíčového řádku a klíčového sloupce. Simplexová tabulka nyní vypadá následovně (3):

zákl.prom	x_1	x_2	x_3	x_4	x_5	β_i	$t = \frac{\beta_i}{a_{ik}}$
x_3	$\frac{1}{2}$	$\frac{1}{4}$	1	0	0	40	80
x_4	$\frac{1}{2}$	$\frac{1}{2}$	0	1	0	60	120
x_5	0	$\frac{1}{4}$	0	0	1	25	-
z_j	-20	-14	0	0	0		

Tabulka 3: Simplexová tabulka - výběr klíčového prvku

Pomocí *Gaussovy eliminační metody* přepočteme simplexovou tabulku tak, aby klíčový prvek měl hodnotu 1. Simplexovou tabulku po tomto přepočtu zachycuje tabulka (4).

zákl.prom	x_1	x_2	x_3	x_4	x_5	β_i	$t = \frac{\beta_i}{a_{ik}}$
x_1	1	$\frac{1}{2}$	2	0	0	80	160
x_4	0	$\frac{1}{4}$	-1	1	0	20	80
x_5	0	$\frac{1}{4}$	0	0	1	25	100
z_j	0	-4	40	0	0	1600	

Tabulka 4: Simplexová tabulka - druhý krok

Hodnota účelové funkce se zlepšila, ale jedna nezákladní proměnná má zápornou redukovanou cenu, takže řešení stále není optimální. Znovu vybereme vstupující proměnnou, kterou bude x_2, protože obsahuje redukovanou cenu s nejnižším koeficientem. Vystupující proměnnou je x_4, jejíž koeficient t je nejmenší, viz tabulka (4).

Tabulku opět přepočteme a získáme optimální řešení (5):

zákl.prom	x_1	x_2	x_3	x_4	x_5	β_i
x_1	1	0	4	-2	0	40
x_2	0	1	-4	4	0	80
x_5	0	0	1	-1	1	5
z_j	0	0	24	16	0	1920

Tabulka 5: Simplexová tabulka - optimální řešení

Nyní již žádná redukovaná cena u nezákladní proměnné nemá zápornou hodnotu, našli jsme optimální řešení úlohy: $x_1 = 40$, $x_2 = 80$.

2.5.3 Možnosti zakončení výpočtu

Výpočet úlohy může být zakončen několika způsoby, úzce s tím souvisí počet řešení úlohy:

- *Úloha má jedno řešení.*
  Jsou-li v simplexové tabulce všechny redukované cenové koeficienty u nezákladních proměnných při maximalizaci účelové funkce kladné (při minimalizace účelové funkce záporné), potom má úloha jediné optimální řešení.

- *Úloha má nekonečně mnoho řešení.*
  Jsou-li všechny redukované ceny při maximalizaci účelové funkce nezáporné (při minimalizaci účelové funkce nekladné) a zároveň je alespoň jeden koeficient účelové funkce u nezákladní proměnné roven 0, potom má úloha alternativní optimální řešení.

- *Úloha nemá omezenou hodnotu účelové funkce.*
  Pokud jsou všechny koeficienty zvoleného klíčového sloupce nekladné, potom nemá úloha omezenou hodnotu účelové funkce. Někdy říkáme, že má úloha optimální řešení v nekonečnu.

- *Úloha nemá řešení.*
  Je-li minimum pomocné účelové funkce větší než 0 (nepodařilo se v první fázi výpočtu eliminovat všechny pomocné proměnné), potom úloha nemá přípustné řešení.

3. Problém splňování podmínek

Spousta úloh z oblasti operačního výzkumu, umělé inteligence, teorie grafů i lineárního programování může být definována jako problém splňování podmínek (anglicky CSP[1]). Jsou jimi zejména kombinatorické úlohy, přiřazovací problémy, úlohy z teorie her a další. Charakteristickým rysem těchto úloh je fakt, že definiční obor proměnných v modelu se skládá pouze z konečné množiny prvků, stejně tak jako množina neznámých. V programování s podmínkami tedy jde o přiřazení hodnot k proměnným za současného splnění omezujících podmínek. Tím má úloha definovaná jako constraint satisfaction problem velice blízko k úloze celočíselného lineárního programování. Celou řadu úloh celočíselného lineárního programování lze definovat jako úlohu CSP a obráceně. CSP úlohy mají většinou složitost NP-complete, algoritmy lineárního programování obvykle polynomiální, takže bývá někdy výhodné převést úlohu CSP na úlohu celočíselného lineárního programování, ale jsou i případy, kdy je tomu naopak (Rivin & Zabih, 1989).

Problém splňování podmínek můžeme definovat následovně (Hoeve, 2017):
Nechť x je proměnná. Její doména (definiční obor), označme ji jako $D(x)$, je množina hodnot, kterých může proměnná x nabývat. Doménu můžeme vyjádřit následovně:

$D(x) = \{d_1, d_2, \ldots, d_m\}$, zkráceně $x \in \{d_1, d_2, \ldots, d_m\}$, přičemž uvažujeme proměnné pouze z konečné množiny prvků.

Nechť $Y = \{y_1, y_2, \ldots, y_k\}$ je konečná řada proměnných, kde $k > 0$. *Omezení* C na Y je definováno jako podmnožina kartézského součinu domén proměnných z Y:

$C \subseteq D(y_1) \times D(y_2) \times \cdots \times D(y_k)$. To zapisujeme jako $C(Y)$ nebo $C(y_1, y_2, \ldots, y_k)$.

Omezení nazýváme *binárním omezením*, pokud je definováno na dvou proměnných. *Obecné omezení* je potom takové omezení, které je definováno na více než dvou proměnných.

Úloha splňování podmínek je definována jako konečná řada proměnných $X = \{x_1, x_2, \ldots, x_n\}$ s odpovídajícími doménami $D = D(x_1), D(x_2), \ldots, D(x_n)$, každá na podmnožině X.

CSP velmi často zkráceně zapisujeme jako $P = (X, D, C)$.

[1]Constraint Satisfaction Problem, také Constraint Satisfaction Programming

Nechť $P = (X, D, C)$ je problém splňování podmínek,
kde $X = \{x_1, x_2, \ldots, x_n\}$ a $D = D(x_1), D(x_2), \ldots, D(x_n)$.

N-tice $(d_1, d_2, \ldots, d_m) \in D(x_1) \times D(x_2) \times \cdots \times D(x_n)$ *splňuje* omezení $C \in C$
na proměnných $x_{i1}, x_{i2}, \ldots, x_{im}$ pokud $(d_{i1}, d_{i2}, \ldots, d_{im}) \in C$.

Pokud taková n-tice nesplňuje C, říkáme, že C je nekonzistentní.

N-tice $(d_1, d_2, \ldots, d_m) \in D(x_1) \times D(x_2) \times \cdots \times D(x_n)$ je *řešením* CSP, pokud splňuje všechna omezení $C \in C$.

V oblasti úloh splňování podmínek se často setkáváme s omezením *Alldifferent*, jeho definice je následující:

Nechť x_1, x_2, …, x_n jsou proměnné, jejichž doménami jsou $D(x_1)$, $D(x_2)$, …, $D(x_n)$.
Potom platí:

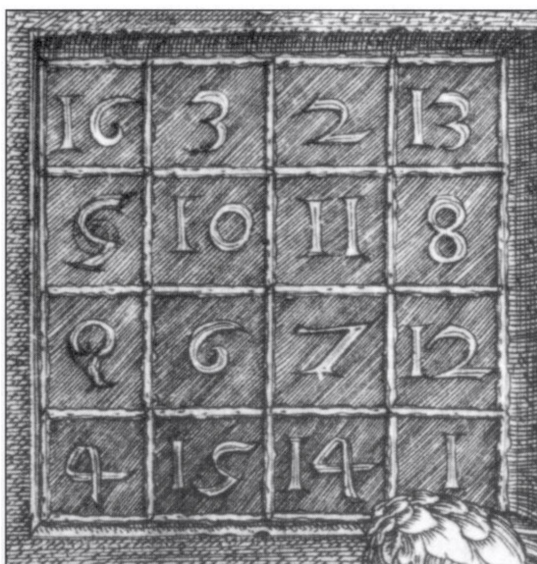
$$Alldifferent(x_1,\, x_2,\, \ldots,\, x_n) = \{d_1,\, d_2,\, \ldots,\, d_n\} \mid d_i \in D(x_i),\, d_i \neq d_j \,\forall\, i \neq j \tag{17}$$

(Všechny prvky množiny x_1, x_2, …, x_n jsou po dvou disjunktní:
$\forall(\, x_i, x_j \in x_1,\, x_2,\, \ldots,\, x_n) \,:\, x_i \cap x_j \neq \emptyset \Rightarrow x_i = x_j \,)$.

4. Sudoku

4.1 Historie

V současnosti nejpopulárnější hlavolam světa se nezrodil náhle. Existovalo množství předchůdců, kteří se objevili již ve starověku. Jedním z nich je magický čtverec. Jedná se o čtvercovou tabulku s čísly, v jejímž každém řádku i sloupci se vyskytuje určité číslo právě jednou. Magické čtverce byly předmětem zájmu již ve starověké Číně a dosahovaly velmi vysoké úrovně vývoje. To potvrzuje i dílo Xugu Zhaiqi Suanfa (v anglickém překladu Continuation of Ancient Mathematical Methods for Elucidating the Strange [Properties of Numbers]), které v roce 1274 napsal Yang Hui a v němž bylo téma magických čtverců velmi často zmiňováno a objevily se v něm i magické čtverce desáté úrovně. Dílo však neobsahuje žádný návod, jak tyto čtverce vytvářet, nejednalo se proto zřejmě o výsledky autorova aktuálního bádání, ale šlo tak spíše o soupis dosavadních znalostí dostupných v tehdejší Číně. V arabském světě byly kolem roku 990 známy magické čtverce úrovně 6 a zmiňovány byly i čtverce úrovně 9. Později popsal arabský učenec al-Buni pravidla na jejich sestrojení. Z arabského světa se dostaly do Evropy kolem roku 1315 prostřednictvím díla byzantského vzdělance Manuela Moschopoulose. Magické čtverce se poté staly zdrojem fascinace i v Evropě, magický čtverec velikosti 4×4 zakomponoval v roce 1514 do svého díla Melancholie I. známý umělec Albrecht Dürer. Magické čtverce se hojně vyskytovaly i v indické, čínské a japonské matematice bok po boku s více propracovanými magickými kruhy či hexagony. Latinský čtverec je potom odnoží magického čtverce, je to pole o rozměru $n \times n$, jehož stavebními prvky jsou písmena, která se objeví v každém řádku i sloupci pouze jednou. Takový čtverec je velmi snadné vytvořit tak, že se první slovo napíše do první řádky a v následujících řádcích je zapsána cyklická permutace zvoleného slova. Aby to nebylo tak jednoduché, matematici, počínaje Eulerem v roce 1781 hledali ortogonální latinské čtverce. To znamená dva latinské čtverce, které se překrývaly tak, aby se každý z n^2 možných seřazených párů písmen mohl vyskytnout pouze jednou (Cooke, 2005). Tématem latinských čtverců z dob Eulera byl inspirován vznik proslulého novodobého hlavolamu, který dnes známe pod názvem Sudoku a jehož zrod je spjat s koncem druhé poloviny 20. století. Za jeho autora je považován americký architekt a tvůrce hlavolamů na volné noze Howard Garns a prvně jej v podobě, kterou známe dnes, pu-



Obr. 1: Melancholie I. (Dürer, 1514).

blikoval v roce 1979 v New Yorku. Rébus se tehdy objevil pod názvem *Number Place* v magazínech Dell Pencil Puzzles a Word Games a příliš nevzbudil pozornost. O pár let později, v dubnu roku 1984 uvedl poprvé rébus v Japonsku Kaji Maki, prezident firmy Nikoli, v té době nejznámějšího japonského magazínu křížovek a rébusů. Hra byla uvedena pod názvem „*Suuji wa dokushin ni kagiru*“, což by se volně dalo přeložit jako „každé číslo se může vyskytnou jen jednou“. Později byl název zkrácen na *Sudoku* (*su* znamená číslo, *doku* znamená jediné). Hra se v Japonsku okamžitě stala velmi populární, zejména v roce 1986, kdy magazín Nikoli zavedl dvě inovace, které se týkaly předvyplněných polí. Počet těchto polí byl omezen, nově publikovaná Sudoku měla předvyplněno maximálně 32 čísel. Druhá změna spočívala v tom, že předvyplněná čísla tvořila v mřížce symetrické vzory. V Japonsku je firma Nikoli stále držitelem ochranné známky názvu Sudoku. Z tohoto důvodu bylo Sudoku schováno i za spoustou dalších anglických názvů, jako např. DigitHunt, Single Number, Squared Away, Nine Numbers, ... (Carter, 2006) Ovšem za celosvětovým boomem Sudoku stojí Wayne Gould, bývalý soudce z Hong Kongu, který v roce 1997 v Tokyu hlavolam objevil náhodou v místním obchodu s knihami. Zmocnila se jej myšlenka, že by měl vytvořit digitalizovanou verzi hry a tak od roku 1997 do roku 2003 vyvíjel počítačový program, který by dokázal vygenerovat zdrojové Sudoku. V roce 2004 nabídl neznámý hlavolam nazvaný Su Doku britskému deníku The Times. Výsledek byl ohromující, během několika málo dní začaly i ostatní deníky tisknout svou verzi hry. Sudoku se velmi brzy stalo populární i v Austrálii a na Novém Zélandu a ještě v roce 2005 se stalo hlavolamem s nejrychleji vzrůstající oblibou. V dubnu téhož roku, čtvrtstoletí po svém původním uvedení, se hra dostala znovu do New Yorku a rychle se rozšířila po celých Spojených státech, zejména prostřednictvím největších federálních deníků jako jsou USA Today a Daily News, které začaly nahrazovat své tradiční křížovky hrou Sudoku. Úspěch Sudoku spočívá především v tom, že se jedná o číselný rébus a nepotřebuje proto žádná písmena specifického jazyka. Proto pro něj neexistují žádné bariéry, zejména jazykové, které by bránily jeho rozšíření. Dnes existuje obrovské množství rozmanitých internetových stránek, věnujících se klasickému Sudoku (rozměr 9×9 polí) i jeho existujícím variantám. Podobná je situace i na poli mobilních platforem. Sudoku zdomácněla i v mnoha současných denících či magazínech. K masovému rozšíření i popularitě Sudoku přispělo zejména to, že existují spousty aplikací a webů, kde lze hrát Sudoku zdarma. Hlavolam je oblíbený všemi věkovými skupinami, fanoušky najdeme i mezi seniory a existují i velmi jednoduché varianty určené dětem. V současnoti se jedná o nejpopulárnější hlavolam, který se jako virus rozšířil po celém světě a dnes téměř neexistuje nikdo, kdo by nevěděl, co je to Sudoku (Osterberg, 2015).

4.2 Pravidla

Jedna z možností, jak formulovat pravidla Sudoku zní např. takto: je dána čtvercová mřížka skládající se z 9×9 polí, částečně vyplněná čísly od 1 do 9 (někdy také označovanými jako vstupy). Cílem hry je doplnit zbývající prázdná pole mřížky hodnotami od 1 do 9 tak, aby se v každé z devíti řad, v každém z devíti sloupců a v každém z devíti (disjunktních) bloků buněk o velikosti 3×3 polí vyskytovalo každé z těchto čísel právě jednou. Pravidla související s řadami a sloupci jsou zřejmá. V dalším textu budeme uvažovat pořadí bloků v mřížce tak, jak ukazuje obrázek 2, na kterém jsou bílými poli s číslem bloku označeny počátky bloků. Na obrázku 3 vidíme pořadí polí v bloku č. 4. Pokud bychom uvažovali souřadnice celého bloku v rámci celé mřížky, pak tento blok má souřadnice [2,1].

Obr. 2: Počátky bloků (autor, 2017)

Obr. 3: Blok v mřížce (autor, 2017)

Obrázek 4 ukazuje úlohu Sudoku (nekompletní mřížku čísel), obrázek 5 je kompletně vyplněná mřížka, nazývaná řešením Sudoku.

Obr. 4: Úloha Sudoku (autor, 2017)

Obr. 5: Řešení Sudoku (autor, 2017)

Řady, sloupce a bloky mají podobnou roli, pokud jde o omezení, která se na ně vztahují. Jednotkou nazvěme řadu, sloupec nebo blok buněk. Dvě různé buňky sdílejí jednotku, pokud jsou ve stejné řadě, sloupci nebo bloku. Tato symetrická relace mezi dvěma různými buňkami nezáleží na jejich obsahu, pouze na jejich umístění v mřížce (Berthier, 2012).

4.3 Obtížnost

Stanovení obtížnosti konkrétní úlohy Sudoku je nesnadný úkol, neboť velmi závisí na jejím subjektivním vnímání řešitelem. Každý člověk má svůj individuální přístup, v různých situacích může použít odlišné techniky řešení. Pro některého hráče je zásadní samotný proces luštění Sudoku, při kterém se snaží používat pouze logické sledy kroků a odhaluje nové způsoby, jak se v obtížných úlohách posunout kupředu. Jinému přináší uspokojení ze hry rychlé dosažení cíle, bez ohledu na postupy, které při řešení použil. Proto v obtížných fázích hry raději použije hrubou sílu (zkoušení čísel metodou pokus - omyl) a po překlenutí překážky použije opět logicky zdůvodněné postupy. Někdy tento přístup skutečně může vést k rychlejšímu dokončení úlohy. První metrikou, která se pro hodnocení obtížnosti zadání Sudoku nabízí, je počet předvyplněných polí. Tato metrika však příliš neodpovídá lidskému vnímání obtížnosti. Každý kdo někdy zkusil řešit složitější Sudoku ví, že vyplnění několika klíčových čísel může být obtížnější a časově náročnější, než vyplnění všech ostatních zbývajících polí dohromady. Proto vznikly metriky založené na ohodnocení jednotlivých úkonů, které vedou k vyřešení úlohy. Těch je celá řada, (Pelánek, 2011) uvádí např. tyto:

- *Kombinovaný rating logických technik*
  Jeho princip spočívá ve spuštění modelu lidského řešitele a zjištění četností výskytu jednotlivých logických technik užitých v průběhu řešení úlohy. Ty jsou následně za pomoci jednoduchých statistických funkcí použity k vytvoření celkového ratingu. Tento přístup používá většina generátorů Sudoku.

- *Serate metrika*
  Je základní metrikou nástroje Sudoku Explainer, jejímž autorem je N. Juillerat. Pracuje s více než 20 technikami, metrikou je nejvyšší obtížnost aplikovaných logických technik.

- *Serate LM metrika*
  Lineární model nad technikami použitými v nástroji Sudoku Explainer. Počítá se v ní, kolikrát byla daná technika v jednotlivé úloze použita. Polovina z testované množiny úloh slouží pro výpočet parametru lineárního modelu, samotná metrika je spočtena na zbylých úlohách z testovací množiny.

- *Fowlerova metrika*
  Je základní metrikou používanou v nástroji G. Fowlera. Metrika je vyjádřena poměrně složitým vzorcem, do něhož vstupuje počet výskytů jednotlivých logických technik.

- *Metrika „počtu zamítnutí“*
  Tato metrika za níž stojí citovaný autor předpokládá, že člověk používá pouze dvě základní techniky (hidden single a naked single) a složitost vnímá v závislosti na celkovém počtu kroků, které musí provést, aby zamítl všechna nesprávná čísla, pokud k dalšímu postupu nemůže použít jednu ze jmenovaných technik.

4.4 Matematické modely

4.4.1 Sudoku definované jako úloha binárního LP

Pro vytvoření binárního lineárního matematického modelu Sudoku poslouží jako základ obecný model planárního 3-D přiřazovacího problému uvedený v (Burkard & Çela, 1999), v němž položíme $n = 9$. Tento model popsaný pomocí vzorců (18) - (22) definuje úlohu Latinských čtverců, která má svá čtyři omezení shodná se Sudoku. Aby se jednalo o matematický model Sudoku, stačí už jen doplnit omezení hodnot v rámci bloku (23) a přidat do modelu fixní hodnoty v mřížce (24). Takto definovaný binární lineární model Sudoku o velikosti 9×9 polí je popsán množinou 9^3 (tj. 729) celočíselných proměnných $x_{i,j,k} \in \{0,1\}$, kde i označuje řádkovou souřadnici proměnné x, její sloupcovou souřadnici vyjadřuje index j a k je index odpovídající hodnotě pole. V tomto modelu hodnota proměnné $x_{i,j,k} = 1$ znamená, že pole se souřadnicemi $[i,j]$ má hodnotu k.

$$min \sum_{i=1}^{9}\sum_{j=1}^{9}\sum_{k=1}^{9} c_{i,j,k}x_{i,j,k} \tag{18}$$

$$x_{i,j,k} \in \{0,1\}1 \ \forall (i,j,k) \in \{1,\ldots,9\} \tag{19}$$

$$\sum_{i=1}^{9} x_{i,j,k} = 1 \ \forall j,k \in \{1,\ldots,9\} \tag{20}$$

$$\sum_{j=1}^{9} x_{i,j,k} = 1 \ \forall i,k \in \{1,\ldots,9\} \tag{21}$$

$$\sum_{k=1}^{9} x_{i,j,k} = 1 \ \forall i,j \in \{1,\ldots,9\} \tag{22}$$

$$\sum_{j=3l-2}^{3l}\ \sum_{i=3m-2}^{3m} x_{i,j,k} = 1 \ \forall k \in \{1,\ldots,9\},\ \forall (l,m) \in \{1,\ldots,3\} \tag{23}$$

$$x_{i,j,k} = 1 \ \forall (i,j,k) \in G \tag{24}$$

Podmínka (20) omezuje hodnoty polí v jednotlivých sloupcích mřížky (v každém z devíti sloupců může být každé z čísel 1 až 9 uloženo právě jednou). Podmínka (21) je analogickým omezením hodnot polí v jednotlivých řádcích. Podmínka (22) je specifická pro binární matematický model. Vzhledem k tomu, že každé z 81 polí je popsáno 9 proměnnými (potenciálními hodnotami pole), chceme, aby hodnoty 1 nabývala pouze jedna z těchto 9 proměnných. Tyto tři podmínky tak popisují pravidla úlohy Latinských čtverců. Podmínka (23) omezuje hodnoty prvků v každém z devíti bloků 3×3 polí. Poslední podmínka (24) je omezením definujícím fixně zadaná pole z množiny vstupů G. Každá ze čtyř podmínek (20) - (23) je popsána soustavou 81 rovnic, celkem je v modelu $\mid 4 \times 81 \mid + \mid G \mid$ rovnic. Např. model úlohy s 26 zadanými hodnotami je popsán soustavou 350 rovnic.

4.4.2 Řešitelnost Sudoku daného binárním lineárním modelem

Za *korektní zadání* Sudoku je považováno pouze to, které má jedno (unikátní) řešení. Nejmenší počet předvyplněných polí, ke kterému existuje jedinečné řešení je 17, jak dokazují ve své práci (McGuire, Tugemann & Civario, 2012). Pokud tedy máme menší počet polí, s jistotou můžeme tvrdit, že bude mít úloha vícenásobné řešení. Taková úloha je degenerovaná. Čím menší je počet vstupních hodnot mřížky, tím více se přibližujeme úloze Latinských čtverců a taková úloha má obtížnost NP-Hard. Běžné algoritmy lineárního programování při jejím řešení selhávají a je nutné je nějakým způsobem upravit, v této souvislosti je zmiňována kromě jiných i Maďarská metoda, která je vhodná pro řešení přiřazovacích problémů (Burkard & Çela, 1999). Problémy řídké matice modelu Sudoku definovaného jako úloha binárního lineárního programování popisují ve své práci i (Tang, Wu & Zhu, 2015), kteří navrhli úpravu algoritmu LP, která dosahuje významně vyšší úspěšnosti při řešení složitých Sudoku.

4.4.3 Sudoku definované jako úloha CSP

(Crawford et al., 2009) popisuje model úlohy Sudoku jako úlohu CSP následovně:

$$x_{i,j} \in \{1, \dots, 9\} \ \forall i, j \in \{1, \dots, 9\} \tag{25}$$

$$\forall i \in \{1, \dots, 9\} \ Alldifferent\{x_{i1}, x_{i2}, \dots, x_{i9}\} \tag{26}$$

$$\forall j \in \{1, \dots, 9\} \ Alldifferent\{x_{1j}, x_{2j}, \dots, x_{9j}\} \tag{27}$$

$$\begin{aligned} \forall i, j \ Alldifferent\{&x_{ij}, x_{i(j+1)}, x_{i(j+2)}, x_{(i+1)j}, x_{(i+1)(j+1)}, \\ &x_{(i+1)(j+2)}, x_{(i+2)j}, x_{(i+2)(j+1)}, x_{(i+2)(j+2)}\} \quad (28) \\ &i = k * 3 + 1, \ j = l * 3 + 1 \ \forall k, l \in \{0, 1, 2\} \quad (29) \end{aligned}$$

kde vzorec (25) udává množinu proměnných modelu s určením domény jednotlivých prvků, (26) je omezením hodnot řádků mřížky (každé číslo z množiny 1 až 9 se smí vyskytovat v každém řádku pouze jednou), omezení (27) je analogickým omezením hodnot ve sloupcích mřížky a (28) je omezením hodnot v jednotlivých blocích buněk mřížky. Rovnice (29) generují souřadnice prvních buněk v bloku, které jsou vstupem pro omezení (28). Tyto souřadnice odpovídají buňkám označeným na obrázku (2).

5. Použité technologie a frameworky

5.1 Framework .NET a programovací jazyk C#

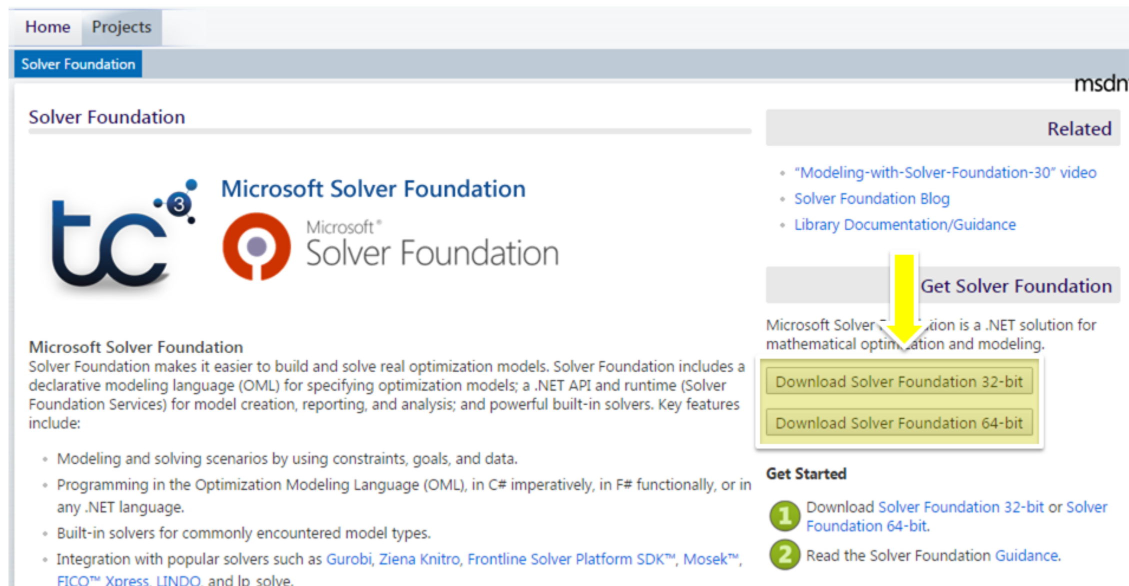
Microsoft .NET Framework je platforma určená pro operační systémy Windows, která obsahuje základní knihovny tříd a rozhraní potřebné k běhu aplikací napsaných v podporovaných jazycích. Počínaje Windows ve verzi Vista je framework přímou součástí operačního systému. Programový kód napsaný v některém z podporovaných programovacích jazyků je přeložen do mezijazyka CIL (Common Intermediate Language) a v prostředí Windows následně spuštěn prostřednictvím CLR (Common Language Runtime). CLR je společné běhové prostředí, zajišťující potřebnou funkcionalitu pro běh programů přeložených do jazyka CIL. Výhodou CLR je, že vzniklý kód je řízený a může tak využívat automatickou správu paměti či řízení výjimek. Jedním z podporovaných programovacích jazyků je C#, což je moderní, objektově orientovaný a typově bezpečný jazyk založený na základech jazyka C. C# byl představen firmou Microsoft v roce 2000 a od roku 2002 se stal součástí vývojového prostředí Visual Studio .NET (Roudenský & Khorshid, 2017).

5.2 Microsoft Solver Foundation

Microsoft Solver Foudation (MSF) je sada vývojářských nástrojů, jejichž hlavní uplatnění najdeme v matematických simulacích, optimalizacích či modelování, využijeme je především při vytváření aplikací běžících v prostředí automatické správy paměti komponenty .NET nebo na bázi platformy CLR. Použít lze v libovolném programovacím jazyku CLR, nejvhodnějším jazykem je Visual C#. Podporovány jsou ale i jazyky Visual Basic, Visual C++, Visual F# a IronPython, tj. programovací jazyky běžící v prostředí operačních systémů Windows na bázi frameworku .NET. Knihovna je použitelná při tvorbě matematických aplikací společně s dalšími technologiemi jako jsou ASP.NET, Silverlight nebo WPF. Microsoft Solver Foundation je univerzálním prostředím pro programování aplikací (API), které je možné spustit i vzdáleně jako službu informačního systému. Nejčastějšími oblastmi využití MSF jsou simulace matematických úloh, případně modelování komplexních systémů, v nichž hledáme takové hodnoty strukturních proměnných, které realizují hledaný extrém účelové funkce za současného splnění omezujících podmínek. Řešit lze tak nejen modely lineárního a nelineárního programování, MSF nabízí podporu i pro řešení úloh kvadratického či smíšeného programování, úloh s omezujícími podmínkami (CSP) a dalších. V prostředí Microsoft Visual Studio lze MSF použít jako knihovna tříd. Ta obsahuje i ukázkové příklady pro vytvoření modelu a nahrání modelu ve formátu MSP. Integrovat lze rovněž s Microsoft Office jako doplněk aplikace Microsoft Excel (Microsoft, 2017a).

5.2.1 Instalace MSF

Knihovna MSF existuje v 32-bitové nebo 64-bitové variantě. Tu zvolíme podle verze nainstalovaného operačního systému kliknutím na jeden z odkazů na webové stránce: `https://msdn.microsoft.com/en-us/devlabs/hh145003`, jak ukazuje obrázek:



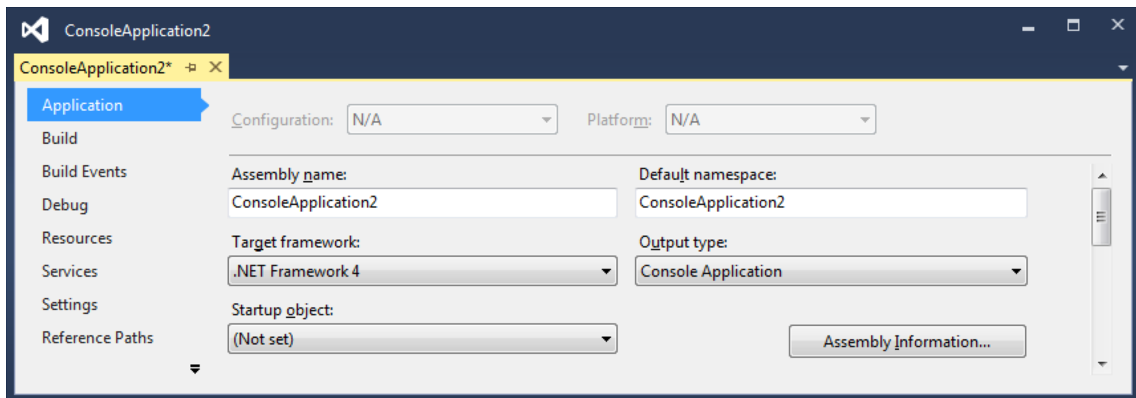
Obr. 6: Stažení knihovny MSF

Vybereme většinou 64-bitovou verzi. Po kliknutí na tlačítko *Download Solver Foundation 64-bit* se do počítače stáhne instalační balíček MicrosoftSolverFoundation64.msi. Následně spustíme vlastní instalaci. Instalátor nás vyzve k souhlasu s licenčními podmínkami, dále pokračujeme výběrem umístění, kam se má knihovna nainstalovat (většinou vybereme implicitní umístění nabídnuté instalátorem) a dokončíme instalaci. Instalátor v jejím průběhu přidá potřebné soubory do počítače. Pro nás je zásadní jeden soubor, a to vlastní .dll knihovna funkcí - Microsoft.Solver.Foundation.dll. Po instalaci ji najdeme ve složce, kde se nacházejí soubory rozhraní .NET Framework verze 4.0 (nejčastěji `\Program Files (x86)\Reference Assemblies\Microsoft\Framework\.NETFramework\v4.0\`). Na knihovnu se budeme odkazovat v projektu Visual Studia (viz dále) a také ji bude třeba zkopírovat do složky ke zkompilovanému .exe souboru. To je nutné proto, aby šlo v aplikaci použít funkci řešitele i v případě, že bude spuštěna v počítači, na kterém není knihovna MSF nainstalována.

5.2.2 Integrace knihovny MSF v projektu Visual Studia

Aby bylo možné využít knihovny MSF v projektu MS Visual Studia, je potřeba učinit následující kroky:

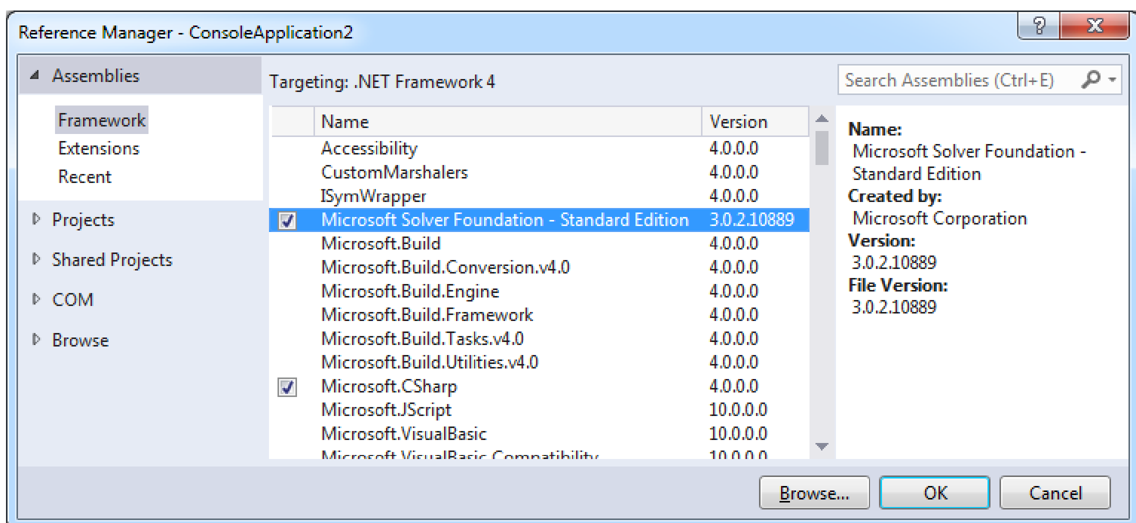
1. V nově vytvořeném projektu vybereme v menu *Project* položku *Properties*. V dialogovém okně, které se otevře (obr.7) nastavíme *Target Framework* na **.NET Framework 4.0**.



Obr. 7: Výběr cílového frameworku (autor)

2. Potom v okně *Solution Explorer* našeho projektu klikneme pravým tlačítkem myši na volbu *References* a v následně zobrazeném menu vybereme položku **Add Reference**.

3. Po otevření dialogového okna *Reference Manager* (obr. 8) zaškrtneme referenci **Microsoft Solver Foundation**. Následně už stačí pouze v programovém kódu připojit odpovídající jmenné prostory přes direktivu **using**.



Obr. 8: Přidání reference na knihovnu MSF (autor)

5.3 Windows Presentation Foundation

Windows Presentation Foundation (zkráceně WPF) je technologie vyvinutá firmou Microsoft, určená k vývoji širokého spektra desktopových aplikací pro operační systémy Windows. Oproti dříve vzniklým technologiím, jako např. Windows Forms[1] a GDI[2], je vývoj počítačové aplikace ve frameworku WPF o poznání jednodušší a rychlejší. Lze v něm totiž použít specifické programátorské techniky, kterých by se jinými technologiemi dalo jen stěží dosáhnout. Ve srovnání s předchozími technologiemi je WPF poněkud odlišné pokud jde o jeho filozofii, programovací model i základní terminologii.

5.3.1 Hlavní výhody WPF

- **Široká integrace**
  WPF obsahuje vestavěnou podporu nejen pro 2D a 3D grafiku, ale i pro zvuk, video a další prvky, které vývojář potřebuje ve své aplikaci použít. Všechny tyto oblasti pokrývá WPF ve svém programovacím modelu konzistentním způsobem, stejně tak jako způsob skládání jednotlivých typů médií a jejich renderování.

- **Nezávislost na rozlišení**
  Primárně je framework WPF zaměřen na použití vektorové grafiky, což s sebou přináší tu výhodu, že velikost jednotlivých grafických elementů nezáleží na použitém rozlišení ani nastavení DPI[3]. Pozitivním důsledkem je např. to, že přechod na vyšší rozlišení nebude spojen se zmenšením velikosti prvku a obráceně - že změna velikosti prvku není spojena s nutností měnit rozlišení obrazovky.

- **Hardwarová akcelerace**
  WPF je založeno na technologii Direct3D, která pro zobrazení grafických objektů využívá harwarovou akceleraci. Při renderování objektů tedy nespotřebovává výpočetní čas CPU, ale místo toho výpočet svěří procesoru grafického adaptéru. Rovněž je implementována i softwarová podpora renderingu pro případ, že zařízení neobsahuje high-end grafický adaptér nebo dojde k překročení dostupných prostředků GPU, jako je např. paměť grafické karty.

- **Deklarativní programování**
  Deklarativní programování není spojeno pouze s WPF, to se používá už déle než 25 let, nicméně WPF ho posunuje na vyšší úroveň zavedením jazyka XAML[4], což je značkovací jazyk, který vychází z jazyka XML. WPF používá XAML pro deklaraci uživatelských prvků aplikace a jejich vlastností.

- **Bohaté možnosti skládání a přizpůsobení elementů**
  Ovládací prvky aplikace mohou být skládány a přízpůsobeny požadavkům programátora dříve nevídaným způsobem aniž by vznikala potřeba psát obrovské množství

[1] knihovna tříd frameworku .NET určená pro tvorbu grafického rozhraní aplikace
[2] Graphical Device Interface - jedna z hlavních součástí operačního systému Windows
[3] Dots Per Inch - udává, kolik pixelů zobrazovacího zařízení se vejde do délky jednoho palce
[4] zkratka pocházející ze slov **EX**tensible **A**pplication **M**arkup **L**anguage

kódu. Ve WPF lze snadno měnit vzhled aplikace výrazným způsobem pouze použitím „skinů", stylů, šablon nebo témat.

5.3.2 XAML

XAML je deklarativní programovací jazyk použitelný pro sestavení a inicializaci objektů. Jde o jazyk XML rozšířený o sadu pravidel, vztahujících se k jednotlivým elementům a jejich atributům. Obsahuje pravidla mapování elementů a atributů na objekty, vlastnosti objektů a jejich hodnoty. XAML obsahuje navíc i množinu pravidel a klíčových slov, popisujících způsob zpracování XML souboru parserem či kompilátorem. Ke každému XAML souboru je při jeho vložení do projektu automaticky vytvořen i odpovídající zdrojový kód (nazývaný *code-behind*). Tento kód obsahuje definici částečné třídy popsané v XAML souboru. Jejím úkolem je inicializovat elementy definované v XAML. Programátor potom může v případě potřeby doplnit další (podpůrné) metody, vztahující se k těmto elementům.

5.3.3 Data Binding

Termín *data* se ve WPF obecně používá pro popis libovolného .NET objektu. Jednotkou dat může být objekt kolekce, XML soubor, webová služba, databázová tabulka, uživatelský objekt a další. V data bindingu jde o spojení .NET objektů za účelem poskytnutí vizuální reprezentace prvků v XML souboru, databázi nebo kolekci (např. prostřednictvím ListBoxu nebo DataGridu). Propojené vlastnosti těchto objektů je třeba nějak synchronizovat (změna propojené vlastnosti zdrojového objektu musí být notifikována cílovému objektu). Nejjedodušší způsob, kterým toho lze docílit je ten, že zdrojový objekt bude implementovat rozhraní *INotifyPropertyChanged*, obsahující event *PropertyChanged* (Nathan, 2014).

6. Implementace programu

6.1 Koncept aplikace a její funkcionality

V současnosti existuje nepřeberné množství verzí hry Sudoku, od primitivních až po ty velmi sofistikované. SudokuMSF je aplikace, která je určena především uživatelům mírně pokročilým, jimž nabízí neomezený zdroj zadání úloh a příležitost si je přímo v aplikaci vyřešit. Kromě generátoru zadání je možné si předvyplněnou mřížku nahrát i z textového souboru, nebo ji rozpracovanou do souboru uložit. Má však další možnosti, které ocení i pokročilejší uživatelé, například okamžitou kontrolu mřížky z hlediska základních pravidel Sudoku nebo výstupní report řešitele, v němž lze najít další varianty řešení, existují-li. Následuje detailní popis funkcionalit programu.

6.1.1 Editace mřížky

Těsně po spuštění aplikace se v pravé části okna zobrazí mřížka 9×9 polí modré barvy. Ta lze editovat myší nebo vstupem z numerické klávesnice. Stisknutí levého tlačítka myši zvyšuje hodnotu buňky, pravé tlačítko má funkci opačnou. Kromě tlačítek lze použít i kolečko myši (mousewheel), přičemž pootočení kolečkem vzhůru má stejnou funkci jako stisk levého tlačítka myši, pohyb kolečkem směrem dolů je ekvivalentní stisku pravého tlačítka. Výchozí hodnotou je prázdné pole, následují hodnoty 1, 2, ..., 9. Po devítce je v pořadí opět prázdné pole. Zadat hodnotu vybraného pole lze kromě myši i z numerické klávesnice, klávesami 1-9. Mezerník a klávesa Delete vloží prázdné pole. Kromě těchto polí se v mřížce mohou objevit i pole bílá (vstupy), to jsou pole, která nelze ve standardním režimu měnit. Takto jsou označena předvyplněná pole, vytvořená generátorem nebo neprázdná pole načtená z externího souboru (givens). Volbu editovat vstupní pole lze aktivovat zaškrtnutím checkboxu *Edit Givens*. Aplikaci v tomto stavu ukazuje příloha A.

6.1.2 Generování zadání zvolené obtížnosti

Program dokáže generovat úlohy různých obtížností, rozděleny jsou do dvou základních typů. První typ je daný subjektivní složitostí vnímanou člověkem (tzn. obtížností postupů, které je potřeba aplikovat, aby uživatel dokázal úlohu zdárně dokončit). Tato zadání jsou rozdělena do tří kategorií - jednoduchá (easy), středně obtížná (moderate) a obtížná (hard). Nastavit je lze zaškrtnutím odpovídajícího checkboxu v sekci *Program Settings - Level based on difficulty*. Druhý typ je definován počtem předvyplněných polí. V tomto režimu si uživatel může nechat vygenerovat mřížky obsahující 25, 27, 30, 35, 40 nebo 45 předvyplněných polí. Pro výběr jsou k dispozici checkboxy v sekci *Program Settings - Level based on # of givens*. Zvolené alternativy jsou vzájemně výlučné a je tak možné vybrat pouze jeden typ obtížnosti z obou kategorií. Po stisknutí tlačítka aplikace *Generate Sudoku* je vygenerováno odpovídající zadání, jehož neprázdná pole (givens) mají bílou barvu

a nelze je ve standardním nastavení programu měnit. Ostatní pole zůstávají modrá a jsou přístupná pro vstup uživatele.

6.1.3 Řešitel Sudoku

S pomocí vestavěného řešitele (solveru) je možné nalézt řešení úlohy. Spustit solver lze v jakékoli fázi běhu programu stisknutím tlačítka aplikace *Solve Sudoku.*Výstupem je u validního zadání Sudoku (existuje-li alespoň jedno řešení) kompletně vyplněná vstupní mřížka a dále informace o počtu řešení. U nevalidních zadání (existuje-li více možných řešení) nabídne řešitel první nalezené řešení a informaci o jejich počtu (zjišťuje se však maximálně 10 řešení). Varianty lze potom zjistit z reportu (viz dále). Solver tak lze použít i pro manuální vytvoření vlastního validního zadání. A to například tak, že se zapnutou volbou *Check Rules* vloží uživatel několik vstupních hodnot a ty uloží do souboru. Následně spustí řešitele (který kompletně vyplní mřížku, pokud existuje alespoň jedno řešení) a zjistí i jejich počet. Pokud jich existuje více, nechá kromě svých vyplněných polí (bílé barvy) i několik vybraných polí vyplněných řešitelem, ostatní smaže a takto mřížku uloží do nového souboru. Takto lze pokračovat, dokud nenalezne zadání, jehož řešení je unikátní.

6.1.4 Nahrání dat ze souboru

Počáteční množinu zadaných čísel lze získat i načtením hodnot z textového souboru, přičemž není pevně dán formát souboru, ale záleží především na jeho obsahu. Podstatný je přitom výskyt numerických znaků 0 až 9 a jejich pořadí v souboru. Předpokládá se zadání ve formě sekvence čísel, která reprezentují obsazení mřížky postupně z levého horního rohu mřížky po řádcích až k poslednímu prvku v pravém dolním rohu mřížky. Nula zde reprezentuje prázdné pole. Tento postup byl zvolen především proto, aby bylo možné do aplikace nahrát různé formáty zadání, které lze najít na internetu (případně vstupní soubor získat zkopírováním mřížky do textového souboru a ten potom buď vůbec nebo jen pouze s minimálním úsilím editovat). Na internetu se často setkáme s případy, kdy bývají prázdná pole reprezentována i jinými znaky než 0. Aplikace proto při načítání analyzuje i jiné než numerické znaky a pokud najde libovolný zástupný znak (podle četnosti výskytu), který by se zbývajícími numerickými znaky 1 až 9 v součtu představoval 81 vstupních hodnot, vyhodnotí jej jako náhradu za prázdné pole. Pokud by takových znaků existovalo více, bude za něj považován první nalezený z nich. Za relevantní je považováno pouze prvních 400 bytů vstupního souboru. Pokud mezi nimi není nalezeno zadání, považuje se vstupní soubor za nekorektní a uživatel je o tom informován prostřednictvím informačního okna. Korektní vstup je potom přímo načten do mřížky a hodnota Givens udává počet neprázdných buněk. Různé varianty zadání, které program dokáže načíst jsou dodány spolu s aplikací. Za základ je považován formát csv, který lze snadno vytvořit v libovolném textovém editoru nebo v libovolném tabulkovém procesoru. Výběr vstupního souboru probíhá přes standartní formulář operačního systému stisknutím tlačítka aplikace *Load Sudoku.* Počáteční cesta je nastavena na adresář aplikace.

6.1.5 Uložení dat do souboru

Obsah mřížky lze uložit kdykoli do textového souboru (včetně kompletně vyplněné mřížky, bez ohledu na korektnost zadání). Data jsou uložena ve formátu csv (oddělovačem znaků je středník). Proces výběru výstupního souboru probíhá opět přes standardní formulář operačního systému, který umožňuje vytvořit nový soubor, případně přepsat existující soubor. Počáteční cesta je nastavena do adresáře aplikace. K vyvolání této funkcionality slouží tlačítko aplikace *Save Sudoku*.

6.1.6 Obnovení mřížky

Stisknutím aplikačního tlačítka *Reinitialize Input* lze mřížku kdykoli vrátit do stavu, ve kterém se nacházela těsně po vygenerování aktuálního zadání nebo po načtení dat ze souboru. Tato funkcionalita má význam především tehdy, když uživatel za použití rozšířených funkcí přepíše v mřížce vstupní hodnoty získané generátorem a potřebuje provedené úpravy odvolat (vrátí se zpět ale všechny změny, nejen ty v bílých polích). Hodit se může i v případě, že bude potřebovat čisté vygenerované zadání bez následně provedených úprav uložit do souboru (neboť všechna vyplněná čísla v mřížce, ať už se jedná o předvyplněná pole nebo uživatelem vložené hodnoty, se po uložení do souboru a jeho znovunačtení stávají vstupem - bílými poli).

6.1.7 Návrat do počátečního stavu

Tato funkcionalita slouží pro návrat programu do stavu, v jakém se aplikace nacházela po svém spuštění (tzn. prázdná mřížka připravená k editaci a předvolená varianta obtížnosti generátoru s hodnotou Easy). Dosáhnout tohoto stavu aplikace je možné pomocí tlačítka *Clear All*.

6.1.8 Kontrola základních pravidel Sudoku

Tuto rozšířenou funkcionalitu lze použít ke kontrole stavu mřížky z hlediska základních pravidel Sudoku a aktivovat ji může uživatel zaškrtnutím checkboxu *Check rules*. Kontrola se vztahuje na všechny vyplněné buňky v mřížce. Čísla polí, která jsou z pohledu pravidel Sudoku ve vzájemném konfliktu získají červenou barvu, pozadí takových polí zůstává u vstupních hodnot bílé, u ostatních získá žlutou barvu. Počet porušení pravidel se potom zobrazí v dolní části aplikace jako hodnota *Rules conflicts*. Kontrola proběhne ihned po jakékoli změně libovolného pole mřížky. Vzhledem k tomu, že uživatel nejčastěji pro vstup hodnot zvolí nejpohodlnější způsob - zadávání myší, při němž se hodnota polí nastavuje sekvenčně na bázi počtu kliků tlačítek myši nebo kroků pootočení kolečkem, je tato funkcionalita standardně vypnutá, protože by uživatele, který si chce úlohu vyřešit bez jakékoli nápovědy, rušila. Aplikaci v tomto režimu ukazuje příloha C.

6.2 Objektový model aplikace

Programový kód aplikace se skládá celkem z 11 tříd (viz příloha D), které lze rozdělit do 4 následujících skupin podle metodiky MVVM dostupné z (Microsoft, 2017b):

1. Třídy realizující View
2. Třídy realizující ModelView
3. Třídy realizující Model
4. Ostatní (pomocné) třídy

6.3 Třídy realizující View

Jedná se o třídy, jejichž účelem je zobrazení aplikace jako takové, včetně všech jejích ovládacích prvků, informačních oken, systémových dialogů a taktéž vyvolání konkrétních funkcionalit programu v závislosti na interakci s uživatelem. Tyto třídy tudíž neobsahují žádnou business logiku a nemají proto přímou vazbu na třídy typu model, jež realizují vlastní výkonné funkce aplikace. Obsahují však pomocnou logiku. Ta se váže k ovládacím prvkům aplikace, které jsou v rámci těchto tříd definovány a zajišťují vlastní interakci s uživatelem, včetně nastavení vstupních zařízení, zajišťujících ovládání aplikace (klávesnice, myš). Charakteristickým znakem těchto tříd je to, že se jedná o částečně definované třídy, tzn. že část každé takové třídy je popsána programovým kódem jazyka C# (deklarace třídy obsahuje klíčové slovo „partial“) a zbytek třídy, tzn. definice vlastních prvků uživatelského rozhraní a jejich vzhled, je potom uložen v separátním souboru ve formátu XAML. Zde je nutno poznamenat, že vzhled celé aplikace i jednotlivých grafických prvků velmi záleží na individuálním nastavení grafického rozhraní v cílovém počítači, jehož některé parametry nelze ve WPF (v aktuální verzi) přímo měnit. Pro ideální zobrazení aplikace doporučuji nastavit v ovládacích panelech Windows *Základní motiv* zobrazení nebo některý z motivů prostředí *Aero* a v nastavení systému upřesnit možnosti výkonu a ve vizuálních efektech vypnout volbu „*Animovat prvky a ovládací prvky oken*“ (v anglické verzi Windows „*Animate controls and elements inside windows*“), které v aplikaci působí rušivě. View v aplikaci SudokuMSF realizují následující třídy:

6.3.1 App

Třída App je hlavní třída zapouzdřující aplikaci WPF. Poskytuje statické metody, vlastnosti a události potřebné pro řízení běhu aplikace. Je vytvořena v projektu automaticky a většinou není třeba ji nijak modifikovat.

6.3.2 InfoWindow

Tato částečná třída, odvozená z třídy `System.Windows.Window` zajišťuje zobrazování pomocných oken informačního charakteru. V aplikaci se vyskytují dva odlišné typy informačních oken. Jeden pro zobrazení stavových informací, jako jsou informace o aktuálně

běžícím výpočtu řešení a úspěšném či neúspěšném otevření nebo uložení souboru. Druhý typ okna je uzpůsobený k zobrazení výsledného reportu z průběhu řešení úlohy.

Zdrojový kód 1 Deklarace konstruktoru a metod třídy InfoWindow

```
public InfoWindow(int type, string text, int time)
public void CloseInfoWindow()
private void TimeElapsed(object sender, ElapsedEventArgs e)
private void WindowLoaded(object sender, RoutedEventArgs e)
```

Konstruktor třídy má tři parametry, **type** určuje typ okna, **text** je vlastní obsah okna, **time** je čas v milisekundách, po jehož uplynutí se okno samo zavře. Text je v okně zobrazován jako obsah objektu **textBox** (instance třídy **System.Windows.Controls.TextBox**) definovaného v příslušném XAML souboru. Veřejná metoda **CloseInfoWindow()** otevřené informační okno uzavře. Vyvolat ji může jiný objekt z okolí nebo zevnitř třídy neveřejná metoda **TimeElapsed()**. Ta je v konstruktoru třídy nastavena jako ovladač události **Elapsed** neveřejné proměnné **_timer** (instance třídy **System.Timers.Timer**) a je automaticky volána po uplynutí časového intervalu nastaveného parametrem **time**. Metoda **WindowLoaded()** je volána automaticky při otevření okna jako obsluha události **Loaded**. Jejím úkolem je u oken zobrazujících stavové informace z okna odstranit ovládací prvky pro minimalizaci, maximalizaci a zavření okna. Reportovací okno se nezavírá automaticky, proto je mu ponecháno tlačítko pro uzavření okna.

6.3.3 MainWindow

MainWindow je ústřední třídou grafického uživatelského rozhraní, zobrazuje celou aplikaci včetně všech ovládacích prvků. Jedná se opět o částečnou třídu, jejíž hlavní obsah, jednotlivé grafické a ovládací elementy, jsou definovány v poměrně složité struktuře XAML souboru. Aplikace obsahuje přes 30 zobrazovaných grafických prvků (kromě nich i několik nezobrazovaných, jako jsou objekty třídy **Grid**). Většina z nich jsou tlačítka pro spouštění konkrétních akcí aplikace (objekty třídy **Button**), zaškrtávátka pro nastavení parametrů aplikace (objekty třídy **CheckBox**) a popisky pro zobrazení všech textů a některých výstupních informací aplikace (objekty třídy **Label**). Najdeme zde ale i rozbalovací text (objekt třídy **Expander**), umožňující skrýt checkboxy, které obsahuje, ikonu reportu (objekt třídy **Image**) a vlastní mřížku Sudoku, což je uživatelský ovládací prvek definovaný samostatně jako objekt třídy **SudokuGridView**. Třída **MainWindow** obsahuje neveřejnou proměnnou **_sudokuModelView** typu **SudokuModelView**, jejíž instanci vytváří v průběhu inicializace v konstruktoru a rovněž na ni nastaví svůj **DataContext**. **SudokuModelView** poskytuje sadu veřejných vlastností, které jsou formou databindingu spojeny s vlastnostmi odpovídajících prvků třídy **MainWindow**. Příklad spojení vlastností **Content** a **Foreground** objektu třídy **Label** s vlastnostmi **Status** a **StatusColor** objektu třídy **SudokuModelView** ukazuje následující příklad (2).

Zdrojový kód 2 Příklad databindingu

```
<Label x:Name="LabelStatusValue" Content="{Binding Status}" Grid.Row="3" Width="215" ...
  ... Foreground="{Binding StatusColor}"/>
```

V „code behind“ potom najdeme tyto metody (3):

Zdrojový kód 3 Deklarace konstruktoru a metod třídy MainWindow

```
public MainWindow()
private void ButtonClickMethod(object sender, RoutedEventArgs e)
private void ButtonMouseEnter(object sender, MouseEventArgs e)
private void ButtonMouseLeave(object sender, MouseEventArgs e)
private void ImageMouseLeftButtonDown(object sender, MouseButtonEventArgs e)
private void LoadFile(object sender, RoutedEventArgs e)
private void SaveFile(object sender, RoutedEventArgs e)
```

`ButtonMouseEnter()` a `ButtonMouseLeave()` nastavují barvu textu tlačítka v závislosti na přítomnosti kurzoru myši nad ním. Metoda `ButtonClickMethod()` po kliknutí na některá tlačítka (`ButtonGenerate`, `ButtonSolve`, `ButtonReinitialize`, `ButtonClearAll`) volá odpovídající metody třídy `SudokuModelView`. Metoda `ImageMouseLeftButtonDown()` po kliknutí myši na ikoně reportu volá metodu `RunReport()` třídy `SudokuModelView`. Metody `LoadFile()` a `SaveFile()` nejprve vyvolají standardní systémové dialogy pro otevření a uložení souboru a následně volají metody `LoadSudokuFile()` a `SaveSudokuFile()` třídy `SudokuModelView`, kterým v parametrech předávají jméno souboru a jméno souboru obsahující celou cestu k souboru. V dialogu otevření souboru není definována žádná maska typu souboru (lze otevřít libovolný soubor), pro uložení souboru je nastavena maska *.csv* odpovídající textovému formátu, v němž aplikace vytváří soubory[1].

6.3.4 SudokuGridView

Tato po částech definovaná třída odvozená z třídy `UserControl` má na starosti zobrazení samotné mřížky Sudoku a změnu hodnot jejích prvků v závislosti na vstupu od uživatele. Mřížka je definována v XAML souboru prostřednictvím objektu třídy `Uniformgrid`, což je obecně uniformní mřížka prvků, v našem případě o velikosti 9×9 polí, sestavená z objektů třídy `Button`. Vlastnosti jednotlivých tlačítek jako jsou souřadnice jeho umístění v mřížce, barva textu a pozadí, samotný text tlačítka a další jsou databindingem napojeny na vlastnosti odpovídajících prvků seznamu `SudokuItemsBoard` třídy `SudokuModelView`, což je seznam objektů třídy `GridItemModelView`. Nejdůležitější řádky kódu XAML souboru uvádí příklad (5). Třída dále obsahuje metody, zajišťující změnu hodnoty prvku mřížky v návaznosti na uživatelský vstup z myši, případně klávesnice (4).

Zdrojový kód 4 Metody třídy SudokuGridView

```
public SudokuGridView()
private void AddItem(int value)
private void AddItem(string value, bool increase)
private void ButtonPreviewKeyDown(object sender, KeyEventArgs e)
private void ButtonPreviewMouseLeftButtonDown(object sender, MouseButtonEventArgs e)
private void ButtonPreviewMouseRightButtonDown(object sender, MouseButtonEventArgs e)
private void ButtonPreviewMouseWheel(object sender, MouseWheelEventArgs e)
```

Metoda `ButtonPreviewMouseLeftButtonDown()` nejprve z parametru `sender` zjistí, které tlačítko událost vyvolalo, přetypuje jeho `Datacontext` na objekt typu `GridItemModelView` a uloží jej do privátní proměnné `_item`. Pokud je tlačítko editovatelné (jeho vlastnost

[1] CSV je zkratka pro Comma-Separated Values

Zdrojový kód 5 Databinding vlastností jednotlivých tlačítek v mřížce

```
<UserControl.Resources>
  <Style x:Key="ButtonStyle" TargetType="{x:Type Button}">
    <Setter Property="Grid.Row" Value="{Binding Row}"/>
    <Setter Property="Grid.Column" Value="{Binding Column}"/>
    <Setter Property="Button.Content" Value="{Binding Value}"/>
    <Setter Property="Button.Background" Value="{Binding Background}"/>
    <Setter Property="Button.Foreground" Value="{Binding Foreground}"/>
    <Setter Property="Button.IsEnabled" Value="{Binding Editable}"/>
  </Style>
</UserControl.Resources>
...
<ItemsControl ItemsSource="{Binding SudokuItemsBoard}">
  <ItemsControl.ItemsPanel>
    <ItemsPanelTemplate>
      <UniformGrid Rows="9" Columns="9"/>
    </ItemsPanelTemplate>
  </ItemsControl.ItemsPanel>
  <ItemsControl.ItemTemplate>
    <DataTemplate>
      <Button Style="{StaticResource ButtonStyle}" ... />
    </DataTemplate>
  </ItemsControl.ItemTemplate>
</ItemsControl>
```

`Editable` je rovna `true`), zavolá se dvouparametrová verze přetížené metody `AddItem()`, která o jedničku zvýší jeho aktuální hodnotu. Velmi podobná je i funkcionalita metody `ButtonPreviewMouseRightButtonDown()`, jen hodnotu prvku o jedničku sníží. Seznam hodnot je cyklický v rozmezí 0 až 9, to znamená že po čísle 9 následuje opět 0 a před 0 je číslo 9. Místo 0 se do názvu tlačítka ukládá prázdný řetězec. Stisknutí levého tlačítka myši tedy zvyšuje hodnotu prvku, stisk pravého tlačítka myši hodnotu pole o jedničku sníží. Metoda `ButtonPreviewMouseWheel()` funguje na stejném principu, pohyb kolečkem vzhůru je analogický stisknutí levého tlačítka myši a obráceně. Metoda `ButtonPreviewKeyDown()` je automaticky vyvolána po stisknutí klávesy. Pokud se jedná o klávesu numerické klávesnice s hodnotou 1 až 9, je odpovídající hodnota do pole vložena přímo prostřednictvím jednoparametrové verze metody `AddItem()`. Klávesy *Delete* a *mezerník* hodnotu pole smažou.

6.4 Třídy realizující ModelView

Tyto třídy realizují styčnou plochu mezi jednotlivými view, zajišťujícími vizuální reprezentaci dat, a modely, které tato data zpracovávají a mění. Pokud view některá data změní, musí být prostřednictvím modelview tato změna promítnuta i do odpovídajícícho modelu. Způsob, jakým toho lze dosáhnout je již zmíněný databinding. Aby se z modelview do view dostala informace o změně klíčového vizuálního prvku, musí být také view nějak upozorněno. Prostředkem k této notifikaci je vyvolání události `PropertyChanged` s informací, která vlastnost v modelview byla změněna. Jednotlivá modelview proto implementují rozhraní `INotifyPropertyChanged`.

6.4.1 GridItemModelView

Je modelview jednoho konkrétního pole mřížky Sudoku. Třída obsahuje veřejné vlastnosti **Value**, **Row**, **Column**, **Foreground**, **Background**, **Editable**, **IsGiven** a **Collides**, které v tomto pořadí nesou hodnotu o hodnotě pole, řádkové a sloupcové souřadnici, barvě textu a pozadí pole. **Editable** je příznak, zda smí uživatel hodnotu pole měnit, **IsGiven** je příznak, zda je pole předvolenou hodnotou (vstupem), **Collides** je příznak, zda je pole podle pravidel Sudoku v konfliktu s jiným polem. Třída implementuje pouze ovladač události **NotifyPropertyChange()** a neobsahuje žádné další metody.

6.4.2 SudokuModelView

Třída **SudokuModelView** představuje hlavní modelview aplikace. Na základě interakce uživatele s view **MainWindow** vznikají požadavky na události v modelview, které podle vnitřní logiky požadavek zpracuje a dále deleguje na jednotlivé modely. Výstupní data z modelů opět modelview zpracuje a předává prostřednictvím notifikací **NotifyPropertyChange()** zpět do view. Tato třída má několik veřejných vlastností, jejichž hodnoty jsou databindingem napojeny na ovládací prvky ve view. Jejich výčet uvádí (6).

Zdrojový kód 6 Vlastnosti třídy SudokuModelView

```
public bool AllowEditGivens
public string Collisions
public string CollisionsColor
public bool Difficulty25
public bool Difficulty27
public bool Difficulty30
public bool Difficulty35
public bool Difficulty40
public bool Difficulty45
public bool DifficultyEasy
public bool DifficultyHard
public bool DifficultyModerate
public string Givens
public bool ImmediateCheck
public string Level
public string LevelColor
public Visibility ReportIconVisibility
public string Status
public string StatusColor
public List<GridItemModelView> SudokuItemsBoard
```

Princip spojení vlastnosti prvku náležejícího view (objekt **MainWindow)** s veřejnou vlastností objektu **SudokuViewModel** byl již ukázán v příkladu (2). Element **LabelStatusValue**, jehož vlastnost **Content** je svázána s vlastností **Status** objektu **SudokuModelView** čeká, až dostane pokyn ke změně hodnoty. Jak k tomu dojde, ukazuje příklad (7). Stejným způsobem pracují i ostatní vlastnosti třídy. V uvedeném příkladu programového kódu dojde v setteru i ke změně hodnoty vlastnosti **StatusColor**. Ta si při změně své hodnoty vyvolá svou metodu **NotifyPropertyChange()**. Vlastnosti nesoucí hodnoty typu **bool** jsou napojeny na odpovídající checkboxy v levé části aplikace, vlastnosti typu **string** potom na barevné labely nacházející se v dolní části aplikace pod Sudoku mřížkou. Vlastnost **ReportIconVisibility** je spojena s vlastností **Visibility** objektu třídy **Image** (ikona

reportu). Nejzajímavější vlastností třídy je lineární seznam **SudokuItemsBoard** složený z objektů třídy **GridModelView**. Modifikací hodnot vlastností prvků seznamu tak dosáhne **SudokuModelView** vizuální změny hodnot Sudoku mřížky ve view. Pro své potřeby však stav mřížky udržuje ve své soukromé datové struktuře - dvourozměrném poli typu **int[,]** uloženém v proměnné **_matrix**, kterou pak předává jednotlivým modelům ke zpracování v parametrech metod těchto modelů.

Zdrojový kód 7 Vlastnost Status - notifikace změny hodnoty

```
public string Status
{
  get { return _status; }
  set
  {
    _status = value;
    switch (value)
    {
      case StatusNotAvailable:   StatusColor = ColorStatusNotAvailable; break;
      case StatusNoSolution:     StatusColor = ColorStatusNoSolution; break;
      case StatusUniqueSolution: StatusColor = ColorStatusUniqueSolution; break;
      default:                   StatusColor = ColorStatusMultipleSolutions; break;
    }
    NotifyPropertyChange("Status");
  }
}
```

Pojďme se podívat na metody třídy, které jsou shrnuty v přehledu (8)

Zdrojový kód 8 Metody třídy SudokuModelView

```
public SudokuModelView(MainWindow mainWindow)
public void NotifyPropertyChange(string propertyName)
public void AddItem(int row, int column, int value, bool isEditable, bool isGiven, bool
doCheck)
public void CheckSolutions()
public void ClearAll()
public void LoadSudokuFile(string wholePath, string fileName)
public void Reinitialize()
public void RunGenerator()
public void RunReport()
public void RunSolver()
public void SaveSudokuFile(string wholePath, string fileName)
public void SetGivens()
private void RunTransformation()
private void SetDifficulty(int levelType, int difficulty, bool labelLevelChange)
private void StoreActualMatrix()
private void UpdateBoard()
private void UpdateDifficultySettings(int levelType)
private void UpdateGrid()
private int CheckConflicts()
```

V konstruktoru se inicializují všechny soukromé proměnné, zejména pak výše uvedené vlastnosti, po jeho provedení je aplikace připravena k použití. **NotifyPropertyChange()** posílá do view zprávy o změně hodnot vlastností třídy. Metoda **AddItem()** je tou metodou, která nastaví poli se souřadnicemi **[row,column]** hodnotu **value**, dále příznak, zda je možné měnit hodnotu pole (**isEditable**) a příznak, zda se jedná o předvyplněné pole (**isGiven**). Změny provede jak v seznamu **SudokuItemsBoard** (ty se promítnou hned do view) tak v soukromé proměnné **_matrix**. Podle hodnoty parametru **doCheck** se po uložení

hodnot ještě provede test na konflikty v mřížce, který může vyvolat další změnu ve view (změnu barvy textu a pozadí konfliktních buněk, viz příloha C), nikoli však v poli `_matrix`. Metoda `CheckSolutions()` zjistí voláním metody solveru `CheckMoreSolutions()` počet řešení úlohy a upraví podle toho status zobrazený ve view. Metoda `ClearAll()` nastaví mřížku do stavu, v jakém se nacházela po inicializaci třídy v konstruktoru (hodnoty v checkboxech zůstávají nezměněny). Metoda `LoadSudokuFile()` se pokusí načíst (voláním metody `LoadFile()` třídy `FileIO`) soubor zadaný parametrem `wholePath`, pokud se načtení podaří, je matice hodnot ze souboru uložena do pole `_matrix`. Parametr `fileName` se použije pro úpravu titulku okna aplikace. Analogicky funguje i metoda `SaveSudokuFile()` se stejnými parametry. Metoda `Reinitialize()` vloží do mřížky hodnoty, které se v ní nacházely po posledním úspěšném nahrání dat ze souboru nebo po spuštění generátoru Sudoku. Ty získá ze soukromého pole `_loadedMatrix`. Metoda `RunGenerator()` nejprve nechá voláním metody `Generate()` třídy `SudokuGenerator` vygenerovat zadání složitosti odpovídající zvolenému checkboxu ve view. To uloží do pole `_matrix`, spustí metodu `RunTransformation()` třídy `SudokuTransform`, která pole `_matrix` změní a následně jej zkopíruje do pole `_loadedMatrix`. Metoda `RunReport()` vyvolává reportovací okno, viz příloha B. Metoda `RunSolver()` spustí výpočet, zobrazí informační okno o průběhu výpočtu. Po ukončení výpočtu okno zavře a zpřístupní ikonu reportu. Metoda `SetDifficulty()` podle stavu checkboxů nastaví hodnoty svých vlastností `Level` a `LevelColor` (čímž dojde k nastavení těchto hodnot ve view). Metoda `SetGivens()` nastaví hodnotu své vlastnosti `Givens` podle počtu nenulových prvků v matici `_matrix`, tím se tato změna okamžitě promítne do view. Metoda `RunTransformation()` provede změnu obsahu matice `_matrix` voláním metody `Transformation()` třídy `SudokuTransform`. Metoda `StoreActualMatrix()` ukládá aktuální stav pole `_matrix` do pole `_loadedMatrix`. Metoda `UpdateBoard()` nastaví barvu textu a pozadí všech buněk podle hodnot matice `_matrix` a hodnoty vlastnosti `ImmediateCheck`. Metoda `UpdateDifficultySettings()` podle parametru `levelType` nastaví jednu ze svých vlastností s prefixem `Difficulty` na hodnotu `true`, ostatní na `false`. Tím dojde k zaškrtnutí odpovídajícího checkboxu ve view. Metoda `UpdateGrid()` nastaví prázdným polím možnost editace a příznak, že se nejedná o vstupy (předvyplněné hodnoty zadání), polím obsahujícím hodnotu naopak. Metoda `CheckConflicts()` nastaví prvkům seznamu `SudokuItemsBoard` vlastnost `Collides` na `true` (tím je označí za prvky, které jsou podle pravidel Sudoku v konfliktu s jinými), spočte počet těchto kolizí a ten uloží do své vlastnosti `Collisions`.

6.5 Třídy realizující Model

Tyto třídy realizují vlastní výkonné funkce (business logiku) a jsou oddělené od view a nezávislé na vnitřní implementaci modelview i na sobě navzájem. Třída `SudokuModelView` je vůči těmto třídám v postavení klienta, který předává vstupní informace a dostává zpět odpovídající výstupy. Z pohledu modelu nezáleží na tom, zda je view konzolová aplikace nebo má grafické uživatelské prostředí.

6.5.1 FileIO

Třída **FileIO** implementuje dvě metody jejichž deklarace jsou uvedeny v příkladu (9).

Zdrojový kód 9 Deklarace metod třídy FileIO

```
public bool LoadFile(string name, int[,] matrix, ref string message)
public bool SaveFile(string name, int[,] matrix, ref string message)
```

Metoda **LoadFile()** slouží k načtení Sudoku mřížky z textového souboru. Parametr **name** je název souboru včetně cesty, v proměnné **matrix** je předáno dvourozměrné pole typu **int[,]**, kam se při úspěšném rozpoznání obsahu souboru načtou hodnoty mřížky. Metoda do vstupního bufferu nejprve načte (maximálně) 400 znaků a soubor zavře. Pokud skončí pokus o načtení souboru nezdarem, je obsloužena výjimka **IOException** a její text je předán do parametru **message** a vrácena hodnota **false** jako příznak neúspěchu. Při úspěšném načtení alespoň 81 znaků ze vstupního souboru máme k dispozici minimální množství znaků, kterým lze definovat obsah celé mřížky a má proto smysl se pokusit získaná data interpretovat. Metoda spočte četnosti všech znaků, které se v souboru vyskytují. Dohromady počítá četnosti numerických znaků '0'..'9'. Pokud je těchto znaků méně než 81, je možné, že za prázdné pole vystupuje v datech ještě jiný zástupný znak než '0' a v takovém případě hledá metoda ve vstupním bufferu první znak, jehož četnost výskytu by se v součtu s ostatními numerickými znaky rovnala 81. Pokud takový znak existuje, je obsah bufferu sekvenčně načten do pole **matrix** a tento zástupný znak je interpretován jako hodnota 0, ostatní nenumerické hodnoty jsou ignorovány. Metoda potom předá do parametru **message** zprávu o úspěšném načtení souboru a vrátí hodnotu **true** jako příznak úspěchu. V opačném případě došlo k neúspěšnému rozpoznání obsahu souboru, odpovídající zpráva je předána do parametru **message** a metoda vrací hodnotu **false**. Metoda **SaveFile()** ukládá obsah pole hodnot **int** předaných parametrem **matrix** do souboru, jehož jméno získá z parametru **name**. Formát souboru je *.csv* (Sudoku mřížka je uložena v textovém souboru jako dvourozměrná tabulka, v níž jsou jednotlivé numerické znaky oddělené středníkem a každá řada devíti hodnot začíná na novém řádku). Pokud soubor **name** existuje, je přepsán, pokud neexistuje, je vytvořen nový soubor a zpráva o uložení dat je předána do parametru **message**, metoda vrací v obou případech **true**. Pokud při zápisu dojde k chybě, je obsloužena výjimka **IOException** a její text je předán do parametru **message** a vrácena hodnota **false** jako příznak neúspěchu.

6.5.2 SudokuGenerator

Třída **SudokuGenerator** slouží k vygenerování nové úlohy Sudoku. Deklarace konstruktoru a metody **Generate()** je uvedena v příkladu (10).

Zdrojový kód 10 Deklarace konstruktoru a metody třídy SudokuGenerator

```
public SudokuGenerator()
public void Generate(int levelType, int difficulty, int[,] matrix)
```

V konstruktoru se nejprve vytvoří instance třídy **Random** a uloží se do neveřejné proměnné **_random**, ta slouží ke generování náhodného čísla určeného k výběru uloženého vstupního zadání z interní databáze. Databáze je realizována ve formě dvou dvourozměrných polí

typu **string[,]**, která jsou přímo v konstruktoru explicitně inicializována hodnotami uloženými v souboru se zdrojovým kódem. Tato varianta byla zvolena proto, aby se předešlo případným problémům při zpracování dat uložených v externím souboru. První pole obsahující 6×999 hodnot typu **string** o délce 81 znaků představuje databázi 999 zadání v každé ze šesti úrovní obtížnosti definovaných počtem předvyplněných polí a je uloženo v soukromé proměnné **_boardGivens**. Data byla získána z veřejně dostupného zdroje (printable-sudoku-puzzles.com 2017, s. 2017) Druhé pole obsahuje 3×24 hodnot typu **string** o délce 81 znaků a představuje databázi 24 zadání v každé ze tří úrovní obtížnosti definovaných složitostí postupů, kterých je třeba užít k vyřešení úlohy. Pole je uloženo v neveřejné proměnné **_boardDifficulty**. Data byla získána z (http://www.7sudoku.com/ 2017). Metoda **Generate()** podle hodnoty parametru **levelType** vybere jedno ze dvou výše uvedených polí, podle parametru **difficulty** vybere index v poli podle dané obtížnosti a z dostupného rozsahu uložených hodnot vybere náhodně jedno zadání (jeden záznam typu **string**). Tento záznam potom pomocí parseru převede na hodnoty **int** a uloží je na odpovídající pozice matice **matrix** předané v třetím parametru metody.

6.5.3 SudokuSolver

Třída **SudokuSolver** je jádrem celé aplikace, obsahuje dvě odlišné metody pro řešení úlohy Sudoku. Kromě samotného výpočtu ještě generuje report, z něhož lze kromě technických informací o jeho průběhu získat i případná další možná řešení u neunikátních zadání úlohy. V třídě je implementováno 7 metod, viz příklad (11). Konstruktor třídy nej-

Zdrojový kód 11 Konstruktor a metody třídy SudokuSolver

```
public SudokuSolver(int[,] array)
public int CheckMoreSolutions(out string report)
public void Solve(int[,] array)
private void ModifyReport()
private void CSPSolver(int[,] array)
private void SimplexSolver(int[,] array)
private void ReplaceReportDecisions(string input, ref string reportString, int
decisionsModified, bool CSP)
```

prve vytvoří pole rozhodnutí (neznámých) pro oba modely a spočte počet předvolených hodnot ve vstupním poli (parametr **array**). Pole rozhodnutí modelu Simplex solveru je uloženo jako neveřejná proměnná **_decisionsMatrixSimplex**, jedná se o jednorozměrné pole o velikosti 729 prvků složené z objektů třídy **Decision** (třída ze jmenného prostoru **Microsoft.SolverFoundation.Services**). Neveřejná proměnná **_decisionsMatrixCSP** představuje dvourozměrné pole 9×9 objektů třídy **Decision**, tam jsou potom uložena rozhodnutí pro model CSP solveru. Po dokončení výpočtu se potom v jednotlivých objektech obou polí nacházejí výsledné hodnoty modelů. Metoda **Solve()** nejprve smaže report z předchozího řešení (proměnná **_report**), vytvoří kontext řešeného problému, smaže předchozí modely, pokud úloha nějaké obsahuje, vytvoří nové modely a spustí řešení. Podle počtu předvolených hodnot ve vstupním poli nejprve řešení lineárního modelu (metoda **SimplexSolver()**) a následně řešení CSP modelu (metoda **CSPSolver()**). Simplex solver je volán ve chvíli, kdy má úloha alespoň 17 předvolených hodnot, CSP solver je volán vždy. Zdrojový kód metody je uveden v příkladu 12.

Zdrojový kód 12 Metoda Solve - inicializace modelů a spuštění výpočtu

```
public void Solve(int[,] array)
{
  _report = "";
  _problem = SolverContext.GetContext();
  if (_givens > 16)
  {
    _problem.ClearModel();
    _model = _problem.CreateModel();
    _model.Name = "Simplex Model";
    SimplexSolver(array);
  }
  _problem.ClearModel();
  _model = _problem.CreateModel();
  _model.Name = "CSP Model";
  CSPSolver(array);
}
```

Metoda **SimplexSolve()** se skládá ze sedmi kroků. Nejprve se do modelu vloží jednotlivá rozhodnutí (neznámé, jejichž hodnoty chceme zjistit). Tato rozhodnutí je nutné nejprve pojmenovat jednoznačným identifikátorem a následně je můžeme do modelu úlohy vložit metodou **AddDecission()** třídy **Model**. Náš model má 729 rozhodnutí, máme je uložená v poli a vkládáme je do modelu v cyklu. Doménou rozhodnutí jsou celá nezáporná čísla.

Zdrojový kód 13 SimplexSolver - vložení rozhodnutí do modelu

```
for (int i = 0; i < N3; i++)
{
  keyName = string.Format("Var{0}{1}{2}", i / N2, (i / N) % N, (i % N) + 1);
  _decisionsMatrixSimplex[i] = new Decision(Domain.IntegerNonnegative, keyName);
  _model.AddDecision(_decisionsMatrixSimplex[i]);
}
```

V modelu vzniknou rozhodnutí pojmenovaná „*Var001*“, „*Var002*“, „*Var003*“, „*Var004*“, „*Var005*“, „*Var006*“, „*Var007*“, „*Var008*“, „*Var009*“, „*Var011*“, „*Var012*“, atd. Potom se do modelu přidají jednotlivé podmínky matematického modelu, např. následující úryvek programového kódu (14) implementuje podmínky popsané rovnicí (22). Pro-

Zdrojový kód 14 SimplexSolver - přidání podmínek do modelu

```
for (int i = 0; i < N2; i++)
{
    keyName = string.Format("Constraint_Cell_{0}{1}", i / N, i % N);
    constraintExpr = "";
    for (int j = 0; j < N3; j++)
        if ((j >= i * N) && (j < (i + 1) * N))
            constraintExpr += _decisionsMatrixSimplex[j].Name + "+";
    constraintExpr = constraintExpr.TrimEnd('+') + "==1";
    _model.AddConstraint(keyName, constraintExpr);
}
```

měnná **keyName** nese unikátní název podmínky, podle kterého ji lze jednoznačně identifikovat. Ten může být jakýkoli (lze použít alfanumerické znaky a podtržítko), nicméně vhodně zvolený název může velmi usnadnit práci při ladění programu. Řetězcová proměnná **constraintExpr** je vyjádřením každé jednotlivé podmínky. Např. pro buňku se souřadnicemi [3,2] v poli **array[,]** (ve view jde o pole ve čtvrtém řádku a třetím sloupci)

je do modelu vložena podmínka nesoucí název „*Constraint_Cell_32*“ a její obsah je:

„*Var321+Var322+Var323+Var324+Var325+Var326+Var327+Var328+Var329==1*“.

V následném kódu metody jsou potom vloženy do modelu další podmínky, které jsou vyjádřeny podobnými programovými konstrukcemi. Níže jsou uvedeny příklady dalších omezení, která tyto fragmenty kódu generující a v nichž figuruje stejné pole Sudoku mřížky:

Podmínka podle (21) omezující výskyt čísla 2 ve čtvrtém řádku Sudoku mřížky:
„*Var302+Var312+Var322+Var332+Var342+Var352+Var362+Var372+Var382==1*“

Podmínka podle (20) omezující výskyt čísla 5 ve třetím sloupci Sudoku mřížky:
„*Var025+Var125+Var225+Var325+Var425+Var525+Var625+Var725+Var825==1*“

Podmínka podle (23) omezující výskyt čísla 7 ve čtvrtém bloku Sudoku mřížky:
„*Var307+Var317+Var327+Var407+Var417+Var427+Var507+Var517+Var527==1*“

Podmínka podle (24) ve čtvrtém řádku a třetím sloupci Sudoku mřížky je uloženo číslo 3:
„*Var323==1*“

Po vložení všech podmínek do modelu, už stačí jen přidat účelovou funkci (viz zdrojový kód (15)). V našem případě jde o minimalizaci sumy hodnot všech proměnných v modelu. Parametry výpočtu lze ještě upravit nastavením vhodných hodnot direktivy **`simplex`**, což je objekt třídy **`SimplexDirective`**, který je předán metodě **`Solve()`** objektu **`_problem`** jako parametr. Výsledek výpočtu je potom vrácen jako objekt třídy **`Solution`**. Z tohoto objektu lze zavoláním metody **`GetReport()`** získat report třídy **`Report`**, v našem případě převedený na **`string`** metodou **`ToString()`**.

Zdrojový kód 15 SimplexSolver - definice účelové funkce a spuštění výpočtu

```
_model.AddGoal("Minimize", GoalKind.Minimize, Model.Sum(_decisionsMatrixSimplex));
SimplexDirective simplex = new SimplexDirective();
simplex.Arithmetic = Arithmetic.Double;
simplex.Pricing = SimplexPricing.SteepestEdge;
simplex.IterationLimit = 300;
_solution = _problem.Solve(simplex);
_report += _solution.GetReport(ReportVerbosity.All).ToString();
```

Metodu **`CSPSolver()`** lze implementovat s využitím podstatně menšího objemu programového kódu, proto si ji v příkladu (16) ve zhuštěném zápisu uvedeme celou.

Stejně jako v metodě **`SimplexSolver()`** je nejprve nutné do modelu vložit proměnné matematického modelu. Potom se přidají omezení, plynoucí z matematického modelu (pravidla Sudoku). V CSP úloze definujeme omezení ve formě nerovnic - říkáme, které prvky se sobě nesmí rovnat (všechny prvky musí být vzájemně odlišné - omezení *AllDifferent*). Podmínky vkládáme do modelu pomocí trojnásobného (vnořeného) cyklu. Proměnná i (cyklus první úrovně) iteruje přes všechny prvky v dané jednotce, tzn. určuje ve kterém řádku, sloupci

Zdrojový kód 16 Metoda CSPSolver

```
private void CSPSolver(int[,] array)
{
  for (int i = 0; i < N; i++)                                  // vložení rozhodnutí do modelu
    for (int j = 0; j < N; j++)
    {
      _decisionsMatrixCSP[i,j] = new Decision(Domain.IntegerRange(1,N), "Dcsn"+i+j);
      _model.AddDecisions(_decisionsMatrixCSP[i, j]);
    }
  for (int i = 0; i < N; i++)
  {
    var x = (i % 3) * 3;
    var y = (i / 3) * 3;
    for (int j = 0; j < N - 1; j++)
    {
      var px1 = x + (j % 3);
      var py1 = y + (j / 3);
      for (int k = j + 1; k < N; k++)          // vložení omezení do modelu - sloupce, řádky, bloky
      {
        var px2 = x + (k % 3);
        var py2 = y + (k / 3);
        _model.AddConstraint("Row"+i+j+k, _decisionsMatrixCSP[i,j] != _decisionsMatrixCSP[i,k]);
        _model.AddConstraint("Col"+i+j+k, _decisionsMatrixCSP[j,i] != _decisionsMatrixCSP[k,i]);
        _model.AddConstraint("Block"+px1+py1+px2+py2, _decisionsMatrixCSP[px1,py1] !=
                                                         _decisionsMatrixCSP[px2,py2]);
      }
    }
  }
  for (int i = 0; i < N; i++)                        // vložení omezení do modelu - předvyplněná pole
    for (int j = 0; j < N; j++)
      if (array[i,j] != 0)
        _model.AddConstraint("Givens"+i+j, _decisionsMatrixCSP[i,j] == array[i,j]);
  _solution = _problem.Solve();                                                 // výpočet řešení
  if (_solution.Quality != SolverQuality.Infeasible)             // uložení výsledku do pole array
  {
    for (int i = 0; i < N; i++)
      for (int j = 0; j < N; j++)
        array[i,j] = (int) _decisionsMatrixCSP[i,j].GetDouble();
    _solutionsCount = 1;
  }
}
```

nebo bloku se porovnávané prvky nacházejí. Proměnná *j* (cyklus druhé úrovně) iteruje přes jednotlivé prvky v dané jednotce, od prvního až k předposlednímu. Udává index prvku v jednotce, který bude porovnáván se zbylými prvky v jednotce (vystupuje v nerovnici na levé straně). Proměnná *k* (cyklus třetí úrovně) iteruje v rámci jednotky od prvku s indexem $j + 1$ (nejbližší pravý soused prvku *j* v jednotce) až k poslednímu (nejvzdálenější pravý soused prvku *j* v jednotce). Indexuje tedy v jednotce prvek, který se bude v nerovnici nacházet na pravé straně. Takto např. v rámci první řady porovnáme postupně prvky se souřadnicemi [0; 1], [0; 2], [0; 3], ..., [0; 8], [1; 2], [1; 3], ..., [1; 8], ..., [5; 6], [5; 7], [5; 8], [6; 7], [6; 8], [7; 8]. Tímto způsobem dojde k vyjádření nerovností mezi všemi prvky v jednotce (v jedné jednotce s 9 prvky je takových nerovností 36). Pro vyjádření podmínek vztahujících se ke sloupcům stačí pouze v souřadnicích porovnávaných prvků zaměnit pořadí indexů. Porovnávání prvků bloku funguje analogicky, jen je třeba vyjádřit absolutní souřadnice prvních prvků bloku a relativní souřadnice zbylých prvků v bloku. Absolutní souřadnice prvků bloku vyjadřují proměnné *x* a *y* (udávají tak v závislosti na hodnotě proměnné *i* prvky mřížky vyznačené na obrázku (3)). Proměnné *px1* a *py1* udávají souřadnice prvku v mřížce odpovídající *j-tému* prvku daného bloku (prvek, nacházející se na levé straně nerovnice), proměnné *px2* a *py2* udávají souřadnice prvku v mřížce odpovídající *k-tému* prvku daného bloku (prvek, nacházející se na pravé straně nerovnice). Po vložení všech nerovností plynoucích z omezujících podmínek modelu se v dalším cyklu vloží předvyplněné hodnoty mřížky ve formě dalších podmínek (ve formě rovnic). Následně je spuštěn výpočet. Pokud existuje řešení, jsou do pole **`array[,]`** vloženy hodnoty řešení, získané z matice rozhodnutí (objektu třídy **`Decision`**, zavoláním jeho metody **`GetDouble()`** a pře-

typováním vrácené hodnoty na typ **int**). Ve třídě **SudokuSolver** se ještě nachází metoda **ModifyReport()**. Ta s přispěním pomocné metody **ReplaceReportDecisions()** zamění v původním reportu získaném z objektu **_solution** výpis hodnot jednotlivých rozhodnutí za jejich maticovou reprezentaci (viz spodní část okna v příloze B). Veřejná metoda **CheckMoreSolutions()** zjistí u modelu CSP, zda existují další řešení voláním metody **GetNext()** objektu **_solution** a pokud ano, rozšíří report o informace o tomto řešení. Počet možných řešení je omezen na 10, další případná řešení už se dále nezjišťují.

6.5.4 SudokuTransform

Tato třída změní obsah částečně vyplněné vstupní mřížky Sudoku tím, že nad ním provede několik náhodně zvolených transformací. Tyto transformace ale zachovávají počet nenulových hodnot v mřížce, řešitelnost úlohy i počet řešení, některé z nich však mohou mít vliv na subjektivně vnímanou obtížnost takového zadání, neboť se jejich aplikací může měnit vzájemné rozložení nenulových hodnot. Výčet metod třídy **SudokuTransform** je uveden v příkladu (17).

Zdrojový kód 17 Deklarace konstruktoru a metod třídy SudokuTransform

```
public SudokuTransform()
public void Transformation(int[,] _inputMatrix)
private void FlipAntiDiagonally()
private void FlipDiagonally()
private void FlipHorizontally()
private void FlipVertically()
private void Permutation()
private void Rotation()
private void SwapBlocksHorizontally()
private void SwapBlocksVertically()
private void SwapColumnsInBlock()
private void SwapRowsInBlock()
private void TransformMatrix()
```

V konstruktoru se nejprve vytvoří instance třídy **Random** a uloží se do neveřejné proměnné **_randomGenerator**, sloužící ke generování náhodných čísel v několika metodách třídy. Následně je vytvořeno dvourozměrné pole typu **int[,]** a uloženo do proměnné **_transformMatrix**. Do tohoto pole se ukládá transformovaná vstupní matice. Metoda **Transformation()** nejprve nastaví neveřejnou proměnnou **_inputMatrix** na stejně pojmenovaný parametr metody (dvourozměrné pole typu **int[,]**, s jehož obsahem budou následně pracovat další metody třídy), vybere náhodný počet transformací, které se nad vstupní maticí provedou (0-11), potom vybere jednotlivé transformace. Ty jsou vybrány v náhodném pořadí bez opakování a následně jsou sekvenčně volány vybrané metody, které postupně jedna po druhé mění transformační matici. Po provedení každé jednotlivé transformace je volána metoda **TransformMatrix()**, která obsah transformované matice uloží zpět do pole **_inputMatrix**. Metody, jejichž transformace vstupní mřížky nemají vliv na její topologii, tedy ani na obtížnost úlohy (neboť nemění relativní souřadnicové vzdálenosti mezi jednotlivými nenulovými prvky), jsou **FlipAntiDiagonally()** (překlopení matice podle vedlejší diagonály), **FlipDiagonally()** (překlopení matice podle hlavní diagonály), **FlipHorizontally()** (překlopení matice podle osy y), **FlipVertically()** (překlopení

matice podle osy x), `Permutation()` (záměna hodnot nenulových prvků) a `Rotation()` (náhodně zvolený počet (1 až 3) rotací o 90 stupňů). Další 4 operace mohou mít vliv na změnu obtížnosti, neboť netransformují matici jako celek, ale mění jednotlivé její části. Metoda `SwapBlocksHorizontally()` (zamění dva náhodně vybrané sloupce bloků, tzn. prohodí 3 sloupce z jednoho bloku za 3 sloupce z jiného bloku, v rámci bloků zůstává pořadí sloupců zachováno), `SwapBlocksVertically()` (zamění dva náhodně vybrané řádky bloků, tzn. prohodí 3 řádky z jednoho bloku za 3 řádky z jiného bloku, v rámci bloků zůstává pořadí řádků zachováno), `SwapColumnsInBlock()` (zamění dva náhodně vybrané sloupce v rámci jednoho náhodně vybraného sloupce bloků) a `SwapRowsInBlock()` (zamění dva náhodně vybrané řádky v rámci jednoho náhodně vybraného řádku bloků).

6.6 Pomocné třídy

6.6.1 Třída Constants

Třída `Constants` je pomocná statická třída, obsahující definice veřejných konstant typů `int`, `bool`, `string`, výčtových typů a třídy `DifficultyType`, kterou využívá většina tříd projektu, zejména pak třída `SudokuModelView`. Jejím účelem je zpřehlednění zdrojového kódu eliminací „magických čísel“ zejména v předávaných parametrech metod a mezích cyklů a dále usnadnění modifikace programových konstrukcí, v nichž figurují nepojmenované konstanty, vyskytující se na více místech v kódu.

7. Závěr

Cílem této práce bylo vytvořit programovou aplikaci, pomocí níž by bylo možné vyřešit konkrétní úlohu, definovanou jako problém lineárního programování a použít k tomu veřejně dostupnou sadu vývojářských nástrojů pro řešení matematických a optimalizačních úloh, knihovnu Microsoft Solver Foundation.

Teoretická část práce byla proto stručným úvodem do oblasti lineárního programování, popsán byl způsob formulace obecné úlohy lineárního programování, konstrukce matematického modelu a dále kroky, které je třeba učinit při praktickém řešení úlohy lineárního programování. Čtenář byl také seznámen se stěžejní metodou lineárního programování, simplexovým algoritmem a dalším hojně využívaným nástrojem operační analýzy, CSP programováním. Zvolenou úlohou byla známá hra Sudoku, popsán byl její vznik, pravidla, obtížnost a dva matematické modely, které byly podkladem pro praktickou část práce. Rovněž byly zmíněny technologie, které byly potřebné pro vytváření konečné aplikace.

Praktická část byla věnována vývoji samotného počítačového programu, který měl předvést možnosti, jak použít funkcionalitu knihovny MSF v praxi a integrovat ji ve vlastní aplikaci. Výsledná aplikace je důkazem, že MSF je určitě použitelným nástrojem a programátorovi umožňuje soustředit se na implementaci matematického modelu, nemusí implementovat samotné algoritmy výpočetních metod. Problémem však někdy může být nedostatečná dokumentace knihovny, která velmi chybí ve chvíli, kdy něco nefunguje. Binární lineární model Sudoku je poměrně rozsáhlý a prokázal potíže, které má simplexový algoritmus se silně degenerovanými úlohami (kterými jsou Sudoku s malým počtem předvyplněných polí). Aplikace měla určitou zamýšlenou funkcionalitu (popsanou v práci, zejména šlo o zjištění počtu řešení úlohy), kterou jsem měl v úmyslu v aplikaci implementovat. Některá zadání Sudoku, případně prázdná či velmi řídká mřížka, by však nešly pomocí simplexového solveru vyřešit nebo by řešení trvalo neúměrně dlouho. Implementován byl proto i řešitel na bázi CSP programování, který vhodně doplňuje simplexovou verzi a dále tak prokazuje použitelnost nástroje MSF.

Summary

This bachelors thesis is dedicated to linear programming, one of the most widely used operations research tools. More specifically, the goal is to create a software application for solving selected problem of linear programming by using tools of Microsoft Solver Foundation library. This open framework involves high quality solvers (including Simplex and CSP solvers) and provides a set of development tools for mathematical simulation, optimization, modeling and many more. This software library is finally integrated into the target application in order to accomplish solving selected problem - a well-known Sudoku puzzle.

There are two mathematical models of Sudoku that are implemented. The first is defined as an Integer Linear Programming problem (ILP) which is solved using Simplex method, the second is described as a Constraint Satisfaction Problem (CSP), solved using tool of a CSP solver tool. The application is able to generate new Sudoku puzzle of a certain level set by the user, load a Sudoku grid from a file, save game to a file and finally solve given Sudoku (either created by the built-in generator or loaded from a file). Additionally, instant Sudoku rules check is also applicable.

The target program application was written in C# language using Microsoft Visual Studio 2015, Windows Presentation Foundation (WPF) framework and Microsoft Solver Foundation framework as well.

Key words: Linear Programming, Simplex method, Microsoft Solver Foundation, Sudoku, Constraint Satisfaction Problem, C#, Microsoft Visual Studio 2015, Windows Presentation Foundation.

Použitá literatura

7Sudoku (2017). (Accessed on 01/10/2017). URL: http://www.7sudoku.com/ (cit. na s. 39).

Berthier, Denis (2012). *Pattern-Based Constraint Satisfaction and Logic Puzzles*. Morrisville, NC: Lulu Press (cit. na s. 19).

Burkard, R. E. & Çela, E. (1999). ?Linear assignment problems and extensions? In: *Handbook of combinatorial optimization*. Sv. 186, s. 75–149. DOI: 10.1007/978-1-4757-3023-4_2 (cit. na s. 21, 22).

Carter, E. (2006). *The Sudoku Master: Advanced Strategies & Visual Techniques*. Morrisville, NC: Lulu Press (cit. na s. 18).

Cooke, R. (2005). *The History of Mathematics: A Brief Course*. 2. vyd. Hoboken, NJ: John Wiley & Sons (cit. na s. 17).

Crawford, Broderick et al. (2009). ?Solving Constraint Satisfaction Puzzles with Constraint Programming? In: s. 345. DOI: 10.1007/978-3-642-02298-2_52 (cit. na s. 22).

Dantzig, G. B. & Thapa, M. N. (1997). *Linear programming: 1: Introduction*. Ed. Glynn, P. New York, NY: Springer-Verlag (cit. na s. 5).

Hillier, F. S. & Lieberman, G. J. (2001). *Introduction to operations research*. Ed. Kane, K. Ed. Munson, E. M. Ed. Lorkovic, M. 7. vyd. New York, NY: McGraw-Hill (cit. na s. 5).

Hoeve, W. van (2017). ?The Alldifferent Constraint: A Survey? Angl. In: URL: https://www.andrew.cmu.edu/user/vanhoeve/papers/alldiff.pdf (cit. 17. 02. 2017) (cit. na s. 15).

Inkala, Arto (2017). *aisudoku.com/index_en.html*. (Accessed on 04/03/2017). URL: http://aisudoku.com/index_en.html.

Jablonský, J. (1999). *Operační výzkum*. 2. vyd. Praha, Czech Republic: Vysoká škola ekonomická v Praze (cit. na s. 6, 9, 10).

McGuire, Gary, Tugemann, Bastian & Civario, Gilles (2012). ?There is no 16-Clue Sudoku: Solving the Sudoku Minimum Number of Clues Problem? In: *CoRR* abs/1201.0749. URL: http://arxiv.org/abs/1201.0749 (cit. na s. 22).

Microsoft (2017a). *Microsoft Solver Foundation 3.1*. URL: https://msdn.microsoft.com/en-us/library/ff524509(v=vs.93).aspx (cit. 11. 02. 2017) (cit. na s. 23).

– (2017b). *The MVVM Pattern*. (Accessed on 03/06/2017) (cit. na s. 31).

Nathan, A. (2014). *WPF 4.5 Unleashed*. Ed. Wiegand, G. Indianapolis, IN: Sams (cit. na s. 27).

Osterberg, K. (2015). *Sudoku Game: What I Can Teach You About Sudoku*. Morrisville, NC: Lulu Press (cit. na s. 18).

Pelánek, Radek (2011). *Difficulty Rating of Sudoku Puzzles by a Computational Model.* URL: `http://aaai.org/ocs/index.php/FLAIRS/FLAIRS11/paper/view/2517` (cit. na s. 20).

Printable 9x9 Sudoku Puzzles: files (2017). (Accessed on 01/10/2017). URL: `http://printable-sudoku-puzzles.com/wfiles/` (cit. na s. 39).

Rivin, Igor & Zabih, Ramin (1989). ?An Algebraic Approach to Constraint Satisfaction Problems? In: *Proceedings of the 11th International Joint Conference on Artificial Intelligence - Volume 1.* IJCAI'89. Detroit, Michigan: Morgan Kaufmann Publishers Inc., s. 284–289. URL: `http://dl.acm.org/citation.cfm?id=1623755.1623799` (cit. na s. 15).

Roudenský, P. & Khorshid, M. M. (2017). *Programujeme hry v jazyce C#.* Brno, Czech Republic: Computer Press (cit. na s. 23).

Tang, Yuchao, Wu, Zhenggang & Zhu, Chuanxi (2015). ?An improved strategy for solving Sudoku by sparse optimization methods? In: *CoRR* abs/1507.05995 (cit. na s. 22).

Seznam obrázků

1 Melancholie I. (Dürer, 1514). 17
2 Blok v mřížce 19
3 Počátky bloků 19
4 Úloha Sudoku 19
5 Sudoku mřížka 19

6 Stažení knihovny MSF 24
7 Výběr cílového frameworku 25
8 Přidání reference na knihovnu MSF 25

Seznam zdrojových kódů

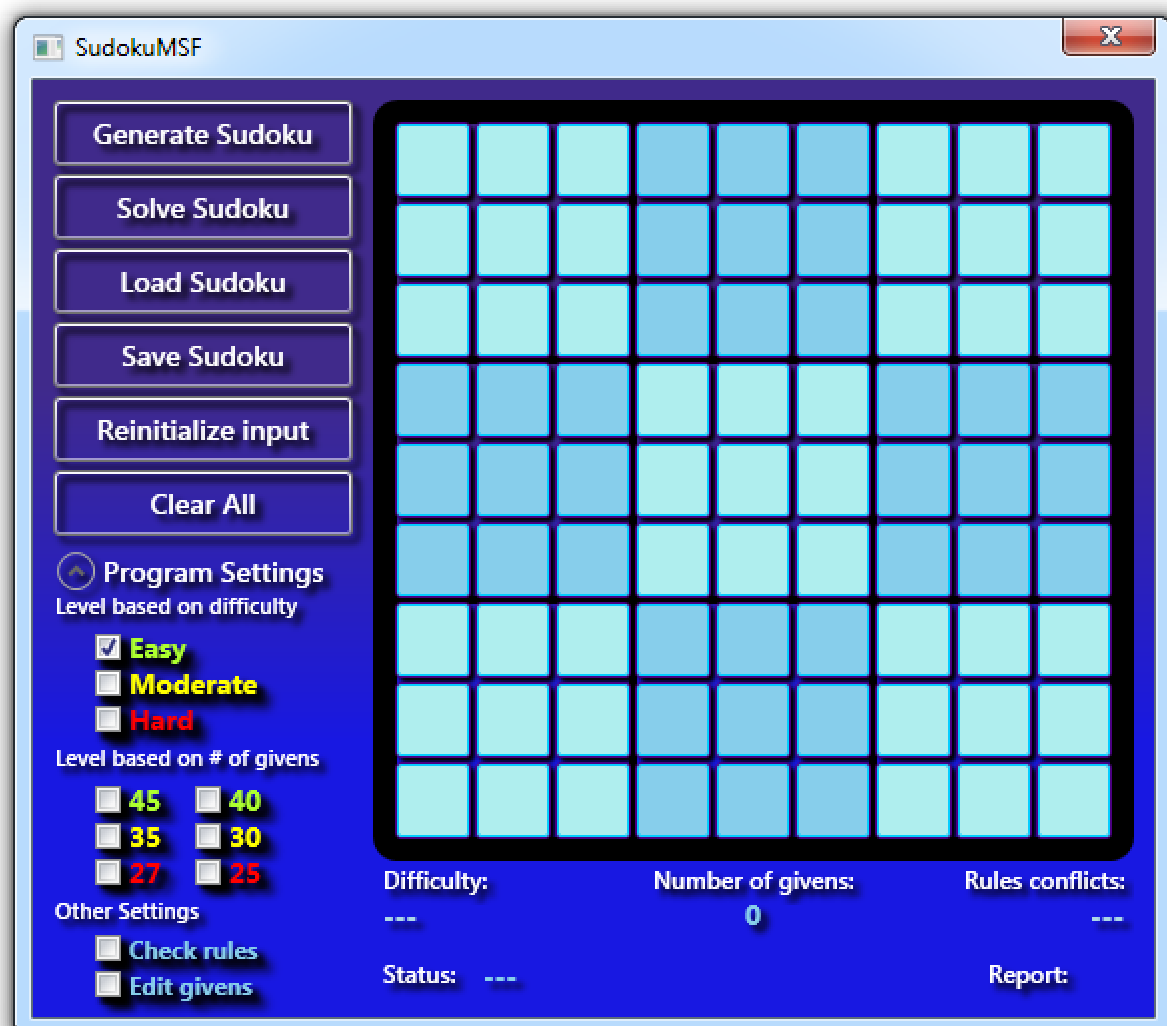
1 Deklarace konstruktoru a metod třídy InfoWindow 32
2 Příklad databindingu 32
3 Deklarace konstruktoru a metod třídy MainWindow 33
4 Metody třídy SudokuGridView 33
5 Databinding vlastností jednotlivých tlačítek v mřížce 34
6 Vlastnosti třídy SudokuModelView 35
7 Vlastnost Status - notifikace změny hodnoty 36
8 Metody třídy SudokuModelView 36
9 Deklarace metod třídy FileIO 38
10 Deklarace konstruktoru a metody třídy SudokuGenerator 38
11 Konstruktor a metody třídy SudokuSolver 39
12 Metoda Solve - inicializace modelů a spuštění výpočtu 40
13 SimplexSolver - vložení rozhodnutí do modelu 40
14 SimplexSolver - přidání podmínek do modelu 40
15 SimplexSolver - definice účelové funkce a spuštění výpočtu 41
16 Metoda CSPSolver 42
17 Deklarace konstruktoru a metod třídy SudokuTransform 43

Seznam tabulek

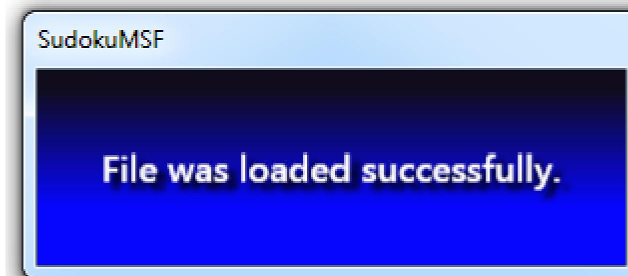
1 Prohibitivní koeficienty v simplexové tabulce 11
2 Výchozí simplexová tabulka 12
3 Simplexová tabulka - výběr klíčového prvku 13
4 Simplexová tabulka - druhý krok 13
5 Simplexová tabulka - optimální řešení 14

Přílohy

Příloha A - aplikace po jejím spuštění



Příloha B - informační a reportovací okno



SudokuMSF - Solver Report

```
===Solver Foundation Service Report===
Date: 12.4.2017 16:39:25
Version: Microsoft Solver Foundation 3.0.2.10889 Standard Edition
Model Name: Simplex Model
Capabilities Applied: MILP
Solve Time (ms): 150
Total Time (ms): 256
Solve Completion Status: Optimal
Solver Selected: Microsoft.SolverFoundation.Solvers.SimplexSolver
Directives:
Simplex(TimeLimit = -1, MaximumGoalCount = -1, Arithmetic = Double,
Pricing = SteepestEdge, IterationLimit = 300, Algorithm = Default,
Basis = Default, GetSensitivity = False)
Algorithm: Primal
Arithmetic: Double
Variables: 729 -> 1 + 0
Rows: 353 -> 0
Nonzeros: 3673
Eliminated Slack Variables: 0
Pricing (double): SteepestEdge
Basis: Slack
Pivot Count: 0
Phase 1 Pivots: 0 + 0
Phase 2 Pivots: 0 + 0
Factorings: 3 + 0
Degenerate Pivots: 0 (0.00 %)
Branches: 0

+-------+-------+-------+
| 5 7 3 | 6 2 1 | 4 9 8 |
| 6 8 9 | 5 4 7 | 2 1 3 |
| 2 1 4 | 3 8 9 | 5 7 6 |
+-------+-------+-------+
| 3 9 7 | 2 6 4 | 8 5 1 |
| 1 2 5 | 9 7 8 | 6 3 4 |
| 4 6 8 | 1 3 5 | 7 2 9 |
+-------+-------+-------+
| 7 4 1 | 8 9 2 | 3 6 5 |
| 9 3 2 | 4 5 6 | 1 8 7 |
| 8 5 6 | 7 1 3 | 9 4 2 |
+-------+-------+-------+
```

Příloha C - aplikace v režimu kontroly pravidel

Příloha D - objektový model aplikace