Zadání bakalářské práce

Název:	Lokální a systematické algoritmy pro řešení zobecněné varianty Sudoku
Student:	Marek Nevole
Vedoucí:	doc. RNDr. Pavel Surynek, Ph.D.
Studijní program:	Informatika
Obor / specializace:	Znalostní inženýrství
Katedra:	Katedra aplikované matematiky
Platnost zadání:	do konce letního semestru 2021/2022

Pokyny pro vypracování

Cílem výzkumu bude porovnat lokální a systematické algoritmy pro řešení zobecněné hry Sudoku na větším hracím poli. Sudoku chápeme jako jeden z hlavních testovacích problémů pro paradigma splňování omezení (CSP – constraint satisfaction problem), pro jehož řešení můžeme vybírat z rychlých neúplných lokálních algoritmů, nebo z úplných systematických. Otevřenou otázkou je, který z těchto směrů bude pro hru Sudoku modelovanou jako CSP výhodnější. Úkoly pro uchazeče budou následující:

1. Prozkoumejte existující algoritmy lokální a systematické algoritmy pro CSP, jako kandidáti se nabízejí metody založené na stoupání do kopce nebo pokročilé metody prohledávání, jako je backtracking s udržováním konzistence.

2. Vybrané algoritmy implementujte formou softwarového prototypu a navrhněte koncepci jejich srovnání na CSP modelu hry Sudoku.

3. Proveďte experimentální porovnání implementovaných algoritmů a výsledky zhodnoťte.

–

[1] Rina Dechter: Constraint processing. Elsevier Morgan Kaufmann 2003, ISBN 978-1-55860-890-0, pp. I-XX, 1-481
[2] Ian Howell, Robert J. Woodward, Berthe Y. Choueiry, Christian Bessiere: Solving Sudoku with Consistency: A Visual and Interactive Approach. IJCAI 2018: 5829-5831
[3] Jean-Charles Régin: A Filtering Algorithm for Constraints of Difference in CSPs. AAAI 1994: 362-367

Elektronicky schválil/a Ing. Karel Klouda, Ph.D. dne 13. listopadu 2020 v Praze.

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Lokální a systematické algoritmy pro řešení zobecněné varianty Sudoku

Marek Nevole

Katedra aplikované matematiky
Vedoucí práce: Doc. RNDr. Pavel Surynek, Ph.D.

11. května 2021

Poděkování

Především bych chtěl poděkovat vedoucímu práce doc. RNDr. Pavlovi Surynkovi, Ph.D., za cenné rady, vědomosti a čas věnovaný konzultacím, bez kterých by tato práce nemohla vzniknout. V neposlední řadě bych rád poděkoval rodině za podporu při studiu.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisů, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, avšak pouze k nevýdělečným účelům. Toto oprávnění je časově, teritoriálně i množstevně neomezené.

V Praze dne 11. května 2021 . . . .

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2021 Marek Nevole. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí a nad rámec oprávnění uvedených v Prohlášení na předchozí straně, je nezbytný souhlas autora.

Odkaz na tuto práci

Nevole, Marek. *Lokální a systematické algoritmy pro řešení zobecněné varianty Sudoku.* Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2021.

Abstrakt

Cílem této práce bylo aplikovat současné algoritmy systematického a lokálního prohledávání na zobecněné variantě hry Sudoku modelované jako problém splňování omezení. Otázkou bylo, který přístup bude dosahovat lepších výsledků. V první části představujeme formulaci problémů jako problémy splňování omezení, a také algoritmy řešící tyto problémy. V druhé části aplikujeme obecná a teoretická východiska na hru Sudoku. Poslední část obsahuje experimenty jednotlivých algoritmů a vzájemné srovnání společně s diskuzí výsledků. Experimenty ukázaly, že systematické algoritmy dokážou lepé řešit herní plochy, jejichž stavový prostor je menší nebo ho lze redukovat. Lokální algoritmy získávají výhodu v opačném případě, ale obecně vyžadují větší časový limit.

Klíčová slova Sudoku, Problém splňování omezení, Prohledávání s návraty, Hranová konzistence, Simulované žíhání, Tabu prohledávání, Min conflicts

Abstract

The aim of this thesis was to apply algorithms of systematic and local search on a generalized variant of the game Sudoku modeled as a constraint satisfaction problem. The question was which approach would achieve better results. In the first part we present the formulation of problems as constraint satisfaction problems, as well as algorithms used to solve these problems. In the second part, we apply general and theoretical background to the game of Sudoku. The last part contains experiments of individual algorithms and mutual comparison together with a discussion of the results. Experiments have shown that systematic algorithms can better solve game boards whose state space is smaller or can be reduced. Local algorithms gain an advantage in the opposite case, but generally require a larger time limit.

Keywords Sudoku, Constraint satisfaction problem, Backtracking, Arc consistency, Simulated annealing, Tabu search, Min conflicts

Obsah

Úvod **1**

1 Teoretická východiska **3**

1.1 Zobecněná varianta hry Sudoku 3
1.2 Problém splňování omezení 3
1.3 Řešení CSP 4
1.4 Sudoku jako problém splňování omezení 5
1.5 Prohledávání s návraty 6
1.5.1 Vynucování konzistence a filtrační algoritmy 8
1.5.2 Filtrace během hledání řešení 16
1.5.3 Heuristiky 17
1.6 Lokální prohledávání 19
1.6.1 Simulované žíhání 19
1.6.2 Využití náhodné procházky 21
1.6.3 Tabu prohledávání 22
1.6.4 Min-conflicts 24
1.6.5 Kombinace metaheuristik 24

2 Aplikace na Sudoku **27**

2.1 Systematické prohledávání 27
2.2 Lokální prohledávání 27
2.3 Implementace a tvorba experimentů 30

3 Experimentální část **33**

3.1 Generování herních ploch 33
3.2 Testovací data 34
3.3 Systematické prohledávání 34
3.4 Lokální prohledávání 36
3.5 Srovnání 41
3.6 Diskuze 42

Závěr **43**

Literatura **45**

A Seznam použitých zkratek **49**

B Obsah přiloženého CD **51**

Seznam obrázků

1.1 Stavový prostor problému n dam, $n = 4$. 5
1.2 Sudoku 2-řádu jako CSP. 6
1.3 Průchod prohledávání s návraty stavovým prostorem Sudoku a příklad *Thrashing*-u. 8
1.4 Graf omezení pro Sudoku 2-řádu. 9
1.5 Graf hodnot pro 4. řádek Sudoku 2-řádu z obrázku 1.2. 13
1.6 Obrázek 1.5 před a po filtraci pomocí *GAC*. 16
1.7 ukázka *Hidden single*. 18
1.8 *Cooling schedule* - Parametr t po 50 iteracích, $t_0 = 1$, $\alpha = 0.9$. . . . 21
1.9 Vychýlení od počátku po 500 krocích s $p = 0.5$. 22

2.1 Krok algoritmu a změna účelové funkce. 29

3.1 Výsledky testu heuristik systematického prohledávání. 35
3.2 Výsledky testu GAC vs. AC3. 36
3.3 Výsledky ladění parametrů Simulovaného žíhaní. 38
3.4 Výsledky ladění parametru Tabu prohledávání. 39
3.5 Výsledky ladění parametru Tabu prohledávání se stochaistickým prvkem WalkSAT. 39
3.6 Výsledky testu porovnání metod lokálního prohledávání. 40
3.7 Výsledky testu porovnání metod systematického a lokálního prohledávání. 41

Seznam tabulek

2.1 Možné konfigurace algoritmů pomocí parametrů tříd 31

3.1 Průměrný počet prohledaných stavů v herních plochách Sudoku 3-řádu 35

Úvod

Sudoku je jednou z mnoha logických her založených na vyplňování políček čísly podle určitých pravidel. Nejstarší zmínky o těchto hrách vedou do starověké Číny, kde je zmíněna hra Magic squares [1]. Tato hra sestává z herní plochy o rozměrech 3×3, do které jsou čísla doplňována tak, aby v řádcích, sloupcích a v hlavních diagonálách byl součet roven 15. Dobově bližší zmínkou jsou Latinské čtverce, jejichž název byl inspirován pracemi matematika Leonharda Eulera, který zde používal znaky latinské abecedy. Na Latinské čtverce se pohlíží jako na kombinatorický problém, ve kterém je pole o velikosti $n \times n$ vyplněno n různými symboly takovým způsobem, že symboly ve sloupích a řádcích se neopakují. Tyto podmínky již připomínají pravidla hry, kterou dnešní svět zná pod názvem Sudoku.

Sudoku, tak jak ho dnes známe, je připisováno americkému architektu Howardu Garnsovi, který vytvářel a společně s Dell Magazines publikoval tuto hru pod názvem Number place. Toto jméno se ovšem neuchytilo, a poté co byla hra představena v Japonsku jako *Sūji wa dokushin ni kagiru* (volně přeloženo jako *Číslice musí být samostatné* nebo *Číslice jsou omezeny na jeden výskyt*) se této hře začalo zkráceně říkat Sudoku. [2]

V počítačové vědě je problém řešení zobecněné varianty Sudoku NP-Úplný problém [3], tudíž neexistuje algoritmus, který by dokázal každou herní plochu vyřešit v polynomiálním čase pomocí deterministického Turingova stroje. Existující algoritmy dokážou efektivně vyřešit Sudoku pro nízká n, ale s rostoucím n nastává kombinatorická exploze a rapidně narůstá doba řešení.

Postupů, jak strojově vyřešit Sudoku, je několik. Sudoku jako NP-Úplný problém lze převést na jiné NP-Úplné problémy. Jedny z nejrychlejších 9×9 řešičů transformují Sudoku na problém splnitelnosti booleovských formulí (SAT), a poté aplikují exponenciální DPLL algoritmus k vyřešení. Další možností je na Sudoku nahlížet jako na problém přesného pokrytí, který lze efektivně řešit pomocí algoritmu X od Donalda Knutha, jenž využívá techniku zvanou *Dancing Links* [4] od stejného autora.

Konkrétně v této práci se budeme zabývat řešením Sudoku jako řešení problému splňování omezení. Problémy splňování omezení lze řešit dvěma způsoby. Pomocí systematických, úplných algoritmů nebo využitím matematické optimalizace a lokálního prohledávání. Úvodní hypotézou je, že algoritmy systematického prohledávání budou vykazovat lepší výsledky, jak časové, tak úspěšnostní, oproti metodám lokálního prohledávání navzdory složitější implementace.

Cíl práce

Cílem této práce je prozkoumat, implementovat a porovnat existující lokální a systematické algoritmy používané k řešení problému splňování omezení. Přínosem této práce je odpověď na otázku, který přístup bude pro zobecněnou variantu hry Sudoku jako problém splňování omezení výhodnější.

Struktura práce

V první části zavádíme použité značení této práce, definujeme použité pojmy a představujeme současné algoritmy, které jsou použity v experimentální části práce.

Ve druhé kapitole ukazujeme, jak aplikovat teoretická východiska na hru Sudoku jako problém splňování omezení.

Třetí, poslední část věnujeme praktickým experimentům. Nejprve se zabýváme efektivním generováním herních ploch, na kterých provádíme experimenty. Poté porovnáváme implementované algoritmy a hledáme parametry, které zajistí optimální výkon algoritmů na menších datech. Z těchto algoritmů jsou poté vybrány ty, které vykazují nejlepší výsledky a jsou znovu testovány na všech datech. Kapitola je zakončena porovnáním a diskuzí nad výsledky.

Prováděné experimenty v podobě Jupyter notebooků využívají námi implementovaný softwarový prototyp Sudoku solveru, který využívá algoritmy z první části. Sudoku solver je implementovaný v programovacím jazyce Python.

Kapitola **1**

Teoretická východiska

V této kapitole čtenáře obeznámíme s principem modelování problémů jako problémy splňování omezení, s algoritmy, kterými lze efektivně tyto problémy řešit, s různými vylepšeními a v neposlední řadě s použitým značením této práce.

1.1 Zobecněná varianta hry Sudoku

Nejběžnější variantou hry je varianta, kde velikost herního pole je 9×9 a úkolem je doplnit čísla od 1 do 9. V této práci se ovšem zaměříme na zobecněnou variantu, kde Sudoku n-řádu je pole o velikosti $n^2 \times n^2$ složené z $n \times n$ bloků a čísla k doplnění jsou $\{1, 2, \ldots, n^2\}$.

Aby herní pole bylo správně vyplněno, číslice musí být doplněny podle následujících pravidel:

- Každé číslo se smí nacházet v každém řádku pouze jednou.
- Každé číslo se smí nacházet v každém sloupci pouze jednou.
- Každé číslo se smí nacházet v každém n×n čtverci pouze jednou.

1.2 Problém splňování omezení

Problém splňování omezení (angl. Constraint satisfaction problem, zkráceně CSP) je způsob, kterým lze modelovat určité matematické problémy. Tento přístup zakládá na faktu, že většinou není výhodné pro každý nový problém vytvářet nový unikátní způsob řešení, ale zjistit, zda nejde problém přeformulovat na nějaký známý problém, který již dokážeme efektivně řešit pomocí existujících algoritmů.

„CSP je trojice $P = (X, D, C)$, kde $X = \{x_1, \ldots, x_n\}$ je konečná množina proměnných, $D = \{D_1, \ldots, D_n\}$ je konečná množina domén, $D_i = \{v_1, \ldots, v_k\}$

je doména (obor hodnot) proměnné $x_i \in X$, která obsahuje hodnoty, které lze dosadit do proměnné, a $C = \{C_1, \ldots, C_i\}$ je konečná množina omezení. Omezení C_j je relací R_j, definovanou nad podmnožinou proměnných $S_j, S_j \subseteq X$. Relace R_j představuje legální současné přiřazení do svých proměnných. Pokud $S_j = \{x_{i1}, \ldots, x_{ir}\}$, pak R_j je podmnožinou kártézského součinu $D_{i1} \times \cdots \times D_{ir}$. Poté lze označit omezení C_j jako dvojici $\langle S_j, R_j \rangle$." [5]

Jedním z nejčastěji uváděných problémů, na kterých je CSP ilustrováno, je barvení map, které spočívá v obarvení území jednotlivých států tak, že státy, jenž mezi sebou sdílí hranici, nemohou být obarveny stejnou barvou. Individuální státy zde představují dílčí proměnné, domény obsahují všechny barvy, které lze využít, a omezení se vyskytují mezi sousedícími státy a zakazují stejné obarvení. Dalšími ilustračními příklady jsou problém N dam, kryptoaritmetika, logické hry jako Kakuro a další. Mezi problémy, se kterými je možné se setkat v praxi, lze zařadit sestavení rozvrhu, automatické plánování ve firmě, plánování/rozvrhování dopravy, přidělování zdrojů.

1.3 Řešení CSP

Přiřazení

Dosazením do proměnné z její domény vzniká přiřazení. Přiřazení podmnožiny proměnných $(x_{i1}, \ldots, x_{ik})$ je zapsáno jako k-tice seřazených dvojic $(\langle x_{i1}, a_{i1} \rangle, \ldots, \langle x_{ik}, a_{ik} \rangle)$, kde každá dvojice $\langle x, a \rangle$ zastupuje přiřazení hodnoty a do proměnné x. Pro zjednodušení se zápis $(\langle x_1, a_1 \rangle, \ldots, \langle x_i, a_i \rangle)$ zkráceně zapisuje jako $\bar{a} = (a_1, \ldots, a_i)$. Přiřazení je kompletní, pokud je přes všechny proměnné z X. [5]

Splnění omezení

Omezení $C_i = \langle S_i, R_i \rangle$ je splněno právě tehdy, když přiřazení $\bar{a}$ obsahuje všechny proměnné z množiny S_i a zobrazení z $\bar{a}$ na S_i se nachazí v R_i. Nechť máme omezení $C_{xy} = \langle S_{xy} = \{x, y\}, R_{xy} = \{(1,1), (1,0)\} \rangle$, poté přiřazení $\bar{a} = (\langle x, 1 \rangle, \langle y, 1 \rangle, \langle z, 0 \rangle)$, pak je omezení C_{xy} považováno za splněné, protože zobrazení z $\bar{a}$ do S_{xy} je rovno $(1,1)$, které je součástí R_{xy}. [5]

Konzistentní částečné přiřazení

Částečné přiřazení je nazváno konzistentní, pokud jsou splněna všechna omezení, jejichž všechny proměnné z S_i jsou přiřazené. Zobrazení z přiřazení $\bar{a}$ na množinu proměnných S_i je značeno jako $\bar{a}[S_i]$. [5]

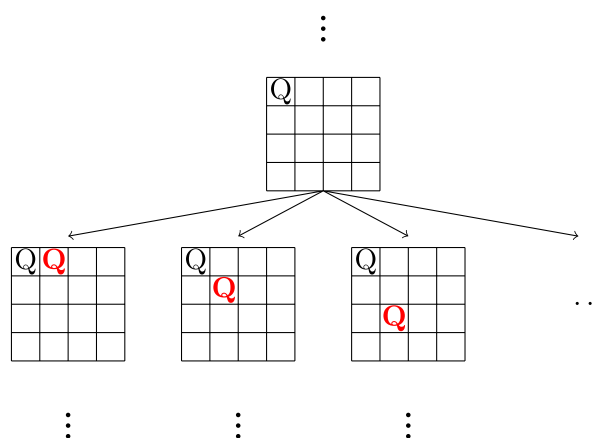
Řešení [5]

CSP $P = (X, D, C)$ je považováno za vyřešené, pokud přiřazení je kompletní a konzistentní přes všechna omezení. Řešení je značeno jako

$$sol(P) = \{\bar{a} = (a_1, \ldots, a_n) | a_i \in D_i, \forall S_i \in C, \bar{a}[S_i] \in R_i\} \tag{1.1}$$

Stavový prostor CSP

Pro následující systematické algoritmy je hledání řešení CSP modelováno jako stavový prostor. Samotné hledání řešení je poté úlohou prohledávání stavového prostoru. Jednotlivé stavy odpovídají přiřazením. Akce, které vedou mezi stavy, reprezentují změny v přiřazení jedné nebo více proměnných. Stavy nemusí obsahovat konzistentní přiřazení. Na obrázku 1.1 lze pozorovat příklad stavového prostoru pro hledání řešení problému 4 dam. Červeně zvýrazněné symboly představují jednotlivé akce, provedené mezi stavy. Problém n dam je formulován v CSP tak, že jednotlivé sloupce jsou proměnné a čísla řádků představují hodnoty domén. Omezení jsou binární, vždy mezi dvěma sloupci, a výsledná relace nad omezením obsahuje dvojice řádků, ve kterých mohou být dámy, aby se vzájemně neohrožovaly.



Obrázek 1.1: Stavový prostor problému n dam, $n = 4$.

1.4 Sudoku jako problém splňování omezení

Sudoku modelované jako CSP (X, D, C) bude značeno následovně:

- Jednotlivá políčka herní plochy reprezentují jednotlivé proměnné z množiny X. Proměnné budou značeny jako $x_{i,j}$, kde i, j je celočíselná dvojice označující řádek a sloupec na herní ploše.

- Množina $D_{i,j}$ je doménou proměnné $x_{i,j}$ a obsahuje hodnoty, které lze do této proměnné dosadit.
- Omezení u Sudoku je nejčastěji formulováno jedním z následujících způsobů (v této práci budou využity oba způsoby):
  - Binární omezení, kde pro každou dvojici proměnných $x_{i,j}, x_{u,v}$, které se nachází ve stejném řádku, sloupci nebo n×n bloku je vytvořeno omezení $x_{i,j} \neq x_{u,v}$. Toto omezení může být také značeno jako $\mathit{Diff}(x_{ij}, x_{uv})$. Pro řádek o 9 proměnných toto znamená 36 nových omezení. Počet omezení na řádek podle řádu Sudoku lze vyjádřit jako $\binom{n^2}{2}$, pro $n > 1$.
  - n-ární omezení, kde jsou jednotlivá omezení celé řádky, sloupce nebo $n \times n$ bloky. Toto omezení se často nazývá *allDifferent* nebo také *Constraint of difference* a je značeno jako $\mathit{allDifferent}(x_{i,j}, \ldots, x_{u,v})$, kde tato k-tice, $k = n^2$ obsahuje omezované proměnné. [6]
- Počátečně vyplněné proměnné přidávají unární omezení ve tvaru $x_{i,j} = v$, kde $x_{i,j}$ je vyplněná proměnná hodnotou $v \in \{1, \ldots, n^2\}$.

i \ j	1	2	3	4
1	**1**	**2**		
2	**4**		**3**	
3	**2**			**3**
4	**3**	**1**		

Obrázek 1.2: Sudoku 2-řádu jako CSP.

1.5 Prohledávání s návraty

Základním algoritmem pro řešení CSP je prohledávání s návraty (BT). BT funguje na principu prohledávání do hloubky. Algoritmus postupně vybírá proměnné, a dosazováním do proměnných vytváří částečné přiřazení. Po každém přiřazení je zkontrolováno, zda je konzistentní. Pokud je konzistentní, algoritmus pokračuje, v opačném případě se algoritmus vrací a zkouší dosud

nevyzkoušené hodnoty. Řešení je vráceno v případě, kdy přiřazení je přes všechny proměnné a je konzistentní.

Algoritmus 1: Backtracking(Assignment, csp)

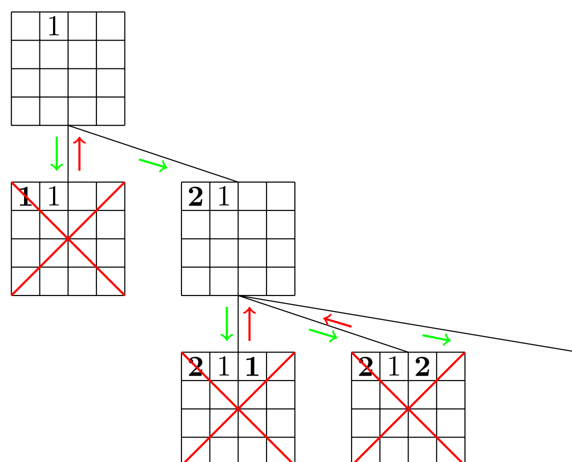
```
1  begin
2      if Assignment is Complete and Satisfies All Constraints then
3          return Solution
4      end
5      Var ← Next Variable
6      foreach Value ∈ D_Var do
7          Assignment ← Assignment ∪ {Var ← Value}
8          if Assignment is Valid then
9              Result ← Backtracking(Assignment, csp)
10             if Result ≠ False then
11                 return Solution
12             end
13         end
14         Assignment ← Assignment \ {Var ← Value}
15     end
16     return False
17 end
```

Algoritmus je systematický a úplný. Nikdy nezkouší dvakrát stejné částečné přiřazení a zaručeně vrátí správné řešení, pokud existuje. V nejhorším případě prohledá celý stavový prostor. Tento přístup představuje hned několik problémů. Velikost stavového prostoru s rostoucím n ve většině případů roste exponenciálně či rychleji, a s tím i samotná doba běhu. Algoritmus také často naráží na podobné slepé konce. Nejjednodušším příkladem tohoto problému je obrázek 1.3, na kterém lze pozorovat průběh BT na Sudoku 2-řádu, které má již předvyplněnou proměnnou $x_{1,2}$. Podle omezení již tuto hodnotu nemůžeme dosadit do stejného sloupce, řádku či 2×2 bloku. Algoritmus ovšem tento fakt nijak nepromítá do domén ostatních proměnných, tudíž často naráží na tyto slepé konce. Tento problém BT se označuje jako *thrashing* [5, 7].

Alan K. Mackworth ve své práci [7], dle mého nejlepšího vědomí, poprvé zobecnil, sjednotil a definoval značení těchto problémů. Ve své práci také ukázal několik algoritmů, a tím vytvořil podklad pro budoucí výzkum a vznik třídy filtračních algoritmů.

Mackworth popisuje 3 hlavní zdroje neefektivity následovně:

1. První zdroj neefektivity se týká unárních omezení. Nechť máme proměnnou x, nad kterou je omezení $c = \langle \{x\}, R \rangle$, a pokud existuje hodnota a z domény D_x, která není součástí R, tudíž nesplňuje omezení c, pak každé přiřazení $\langle x, a \rangle$ bude důvodem okamžitého vracení. Příkladem je zmiňovaný obrázek 1.3.



Obrázek 1.3: Průchod prohledávání s návraty stavovým prostorem Sudoku a příklad *Thrashing*-u.

2. Druhý zdroj neefektivity nastává v následující situaci. Nechť máme částečné přiřazení $\bar{a} = (a_1, \dots, a_n)$ a pro přiřazení $\langle x_i, a_i \rangle$ nelze splnit omezení $c = \langle \{x_i, x_j\}, R_{ij} \rangle$ $(i < j)$, pak BT vyzkouší všechny hodnoty z domény D_j pro proměnnou x_j, neuspěje a vrací se k proměnné x_{j-1}, pro kterou vyzkouší všechny nevyzkoušené hodnoty společně se všemi hodnotami z x_j. Toto může pokračovat pro všechny kombinace hodnot proměnných $x_{i+1}, \dots, x_{j-1}, x_j$, dokud se BT nevrátí k proměnné x_i a nevyřadí hodnotu a_i.

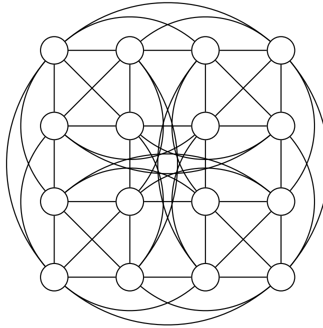
3. Třetí velký zdroj neefektivity nastává v případě, kdy máme přiřazení $\langle x_i, a_i \rangle$ a $\langle x_j, a_j \rangle$, která splňují omezení $\langle \{x_i\}, R_i \rangle$, $\langle \{x_j\}, R_j \rangle$ a $\langle \{x_i, x_j\}, R_{ij} \rangle$, ale neexistuje hodnota a_k, která by splňovala všechna omezení $\langle \{x_k\}, R_k \rangle$, $\langle \{x_i, x_k\}, R_{ik} \rangle$ a $\langle \{x_k, x_j\}, R_{kj} \rangle$ najednou. Stejně jako u problému 2 toto může být drahé k objevení, ale také se to může objevit několikrát během hledání řešení.

Přepsáno do značení bližšího této práci z [7].

1.5.1 Vynucování konzistence a filtrační algoritmy

Pro usnadnění pochopení je vhodné zde za definovat graf omezení. Problém splňování omezení lze reprezentovat grafem omezení. Vrcholy tohoto grafu jsou jednotlivé proměnné $x \in X$. Hrana vede mezi dvěma vrcholy x_i, x_j, pokud jsou oba součástí podmnožiny S nějakého omezení $c \in C$. Tedy pokud nevede hrana mezi vrcholy, pak mezi těmito proměnnými neexistuje žádné omezení. [5] Obrázek 1.4 je příkladem pro Sudoku.

Problémy popsané v předešlé kapitole lze pojmenovat postupně jako vrcholová nekonzistence, hranová nekonzistence a k-nekonzistence (*Path incon-*



Obrázek 1.4: Graf omezení pro Sudoku 2-řádu.

sistency). Opakem k těmto problémům jsou vlastnosti, které jim předcházejí a zabraňují vzniku *thrashing*-u. Těmito vlastnostmi jsou vrcholová konzistence, hranová konzistence a k-konzistence (*Path consistency*). Tyto vlastnosti lze vynucovat na doménách proměnných a výrazně tím zmenšit stavový prostor. O CSP se poté dá říct, že je vrcholově, hranově nebo k-konzistentí, pokud všechny vrcholy, hrany nebo cesty délky $k-1$ jsou konzistentní v grafu omezení.

Vrcholová konzistence

Vrchol x je vrcholově konzistentní právě tehdy, když pro všechna $v \in D_x$ jsou splněna všechna unární omezení nad proměnnou x, $\langle\{x\}, R\rangle$.

Vrcholová konzistence je nejslabší a nejjednodušší technikou. K dosažení této vlastnosti stačí pouze odstranit hodnoty z domén proměnných, které porušují nějaké unární omezení. Toto zvládne jedním průchodem přes CSP algoritmus *NC-1*. *NC-1* dosáhne vrcholové konzistence v čase $O(|X||D|)$ [8], kde $|X|$ je počet proměnných a $|D|$ je kardinalita největší domény.

Algoritmus 2: NC-1(X,D,C)

```
1 begin
2   foreach var ∈ X do
3     foreach value ∈ D_var do
4       if value is not consistent with any unary constraint then
5         D_var ← D_var \ {value}
6       end
7     end
8   end
9 end
```

Hranová konzistence

Hrana (x_i, x_j) *je hranově konzistentní právě tehdy, když pro všechny hodnoty* $v_i \in D_i$ *existuje hodnota* $v_j \in D_j$ *taková, že je splněno omezení* $\langle\{x_i, x_j\}, R_{ij}\rangle$.

Základní operací algoritmů, které vynucují hranovou konzistenci, je funkce *Revise*, která pro hranu (x_i, x_j) jednoduše odstraní z domény D_i všechny hodnoty, pro které neexistuje hodnota z domény D_j, se kterou by bylo možné splnit binární omezení $\langle\{x_i, x_j\}, R_{ij}\rangle$.

Algoritmus 3: Revise(x_i, x_j)

```
1  begin
2      change ← False
3      foreach value ∈ D_i do
4          if there are no values ∈ D_j that would satisfy the constraints
            then
5              D_i ← D_i \ {value}
6              change ← True
7          end
8      end
9      return change
10 end
```

Funkce *Revise* tedy na hraně (x_i, x_j) ihned po aplikaci vynutí hranovou konzistenci, ale zároveň nemusí platit, že hrana (x_j, x_i) je také hranově konzistentní nebo že po aplikaci *Revise* na nějakou další hranu (x_j, x_k) bude hrana (x_i, x_j) stále konzistentní. Následkem tohoto faktu je, že nestačí jeden průchod této funkce přes všechny hrany.

Prvním algoritmem, který toto řeší je algoritmus *AC-1*, který volá *Revise* na všechny hrany. Toto opakuje dokud je při průchodu všemi hranami zredukována alespoň jedna doména. Jedno volání *Revise* provede maximálně $|D|^2$ kontrol splnění binárního omezení. V nejhorším případě jeden průchod redukuje pouze jednu doménu o jednu hodnotu. Poté proběhne maximálně $|X||D|$ průchodů. Tudíž časová složitost *AC-1* je $O(|D|^3|X||E|)$, kde $|E|$ je počet hran. [5, 8]

Hlavním a zjevným zdrojem neefektivity je opětovný průchod přes všechny hrany, pokud byla redukována pouze jedna doména. Algoritmus *AC-3* toto řeší tak, že z počátku přidá všechny hrany do fronty, ze které je postupně vybírá. Na vybrané hrany je aplikováno *Revise*. Pokud při aplikaci na hranu (x_i, x_j) proběhla změna v doméně D_i, pak algoritmus přidá do fronty pouze ty hrany, u kterých mohla redukcí D_i vzniknout nekonzistence a zároveň nejsou ve frontě. Toto jsou hrany ve tvaru (x_k, x_i) takové, že mezi x_k a x_i vede hrana v grafu omezení, $k \neq i$ a $k \neq j$, protože hrana (x_i, x_j) nemůže na přímo změnit konzistenci hrany (x_j, x_i). [7] Algoritmus pokračuje dokud není fronta

prázdná.

Algoritmus 4: AC-3(X,D,C)

```
1  begin
2      NC-1(X,D,C)
3      Queue ← Arcs
4      while Queue ≠ ∅ do
5          Arc(x_i, x_j) ← delete any Arc ∈ Queue
6          if Revise(Arc(x_i, x_j)) then
7              if |D_i|= ∅ then
8                  return False
9              else
10                 Queue ← Queue ∪ {Arc(x_k, x_i)|(x_k, x_i)∈ Arcs,k ≠ i,
                    k ≠ j }
11             end
12         end
13     end
14     return True
15 end
```

Stejně jako u *AC-1*, *AC-3* přidá z počátku všechny hrany do fronty a časová složitost operace *Revise* je stále $|D|^2$. V nejlepším případě je CSP již hranově konzistentní, a tedy na každou hranu bude aplikována funkce *Revise* pouze jednou. Z toho plyne, že dolní mez algoritmu *AC-3* je $\Omega(|D|^2|E|)$. Naopak v nejhorším případě funkce *Revise* odstraní pouze jednu hodnotu z domény a navíc ve frontě nebude ani jedna hrana, která má být přidána. Toto může nastat pro každou proměnnou až $|D|$ krát. Tudíž v nejhorším případě je časová složitost *AC-3* $O(|D|^3|E|)$ [5, 8].

Algoritmů, které vynutí hranovou konzistenci, je mnoho. Za zmínku stojí také *AC-4*, který jako první dosahuje optimální časové složitosti v nejhorším případě $O(|D|^2|E|)$ [5, 9]. *AC-4* nevyužívá operaci *Revise*, ale zavádí hned několik datových struktur. První z nich je pole *counter*(x_i, a_i, x_j), které ukládá počet hodnot a_j z domény D_j, které s hodnotou a_i z domény D_i splňují binární omezení na hraně (x_i, x_j). Dále je pro každou dvojici $\langle x_j, a_j\rangle, x_j \in X, a_j \in D_j$ vytvořena množina $S_{(x_j,a_j)}$, ve které jsou uloženy všechny dvojice $\langle x_i, a_i\rangle$, se kterými je splněno omezení na hraně (x_i, x_j). Kdyby byla odstraněna hodnota a_j, tak *counter*(x_i, a_i, x_j) bude dekrementován o 1. Poslední datovou strukturou je *List*, který obsahuje dvojice $\langle x, a\rangle$, se kterými nelze splnit omezení na žádné hraně, na které se nacházejí. Algoritmus je rozdělen do dvou fází. V první jsou inicializovány a připraveny zmíněné datové strukutury. Ve druhé jsou postupně odebírány dvojice $\langle x, a\rangle$ z *Listu*, následně dochází k odstranění hodnoty a z domény proměnné x, pomocí množiny $S_{(x,a)}$ jsou dekrementovány ostatní countery, pokud *counter*(x_i, a_i, x_j) je roven 0, pak je přidána dvojice $\langle x_i, a_i\rangle$ do *Listu*. Nevýhodou *AC-4* je paměťová složitost, která je také rovna

$O(|D|^2|E|)$ [5]. Druhou nevýhodou je výkon v první, inicializační fázi. Richard J. Wallace ukazuje ve své práci [10], že ve většině případů je *AC-3* rychlejší algoritmus než *AC-4* i přes teoretickou neoptimální časovou složitost.

Na nevýhody *AC-4* reagoval Christian Bessiére, který představil algoritmus *AC-6* [11], který zachovává optimální časovou složitost *AC-4*, ale vylepšuje paměťovou složitost na $O(|D||E|)$ společně s lepším výkonem v průměrných případech.

k-konzistence

Množina k proměnných je k-konzistentní právě tehdy, když pro částečné konzistentní přiřazení $k-1$ proměnných existuje hodnota pro nepřiřazenou proměnnou taková, že všechna omezení mezi těmito k proměnnými jsou splněna.

Zobecněním konceptu lokální konzistence je k–konzistence. Vrcholová, resp. hranová, konzistence je k–konzistence pro $k = 1$, resp. $k = 2$. CSP je silně k–konzistentní právě tehdy, když je také i–konzistentní pro všechna i $(i < k)$. Pokud je CSP silně n-konzistentní, kde $n = |X|$, pak CSP nabývá vlastnosti globální konzistence.

Vlastností globálně konzistentního CSP je možnost nalezení řešení bez využití prohledávácích algoritmů. Každá kombinace zbylých hodnot v doménách proměnných tvoří konzistentní řešení. [5] I přes tuto vlastnost se tyto algoritmy používají pouze zřídka, kvůli časové složitosti, která je většinou exponenciální vzhledem ke k. Vyjímkou bývají algoritmy, které vynucují 3–konzistenci. Tyto algoritmy se většinou nazývají *path consistency*. *PC-1*, *PC–2* [7] a *PC-3* [9] fungují na velice podobném principu jako *AC-1*, *AC-3* a *AC-4*. Nicméně časová složitost těchto algoritmů je i tak příliš vysoká. Konkrétně *PC-3*, který používá stejné datové struktury jako *AC-4*, zavede 3-konzistenci v nejhorším případě v čase $O(|D|^3|E|^3)$.

Zobecněná hranová konzistence

Předchozí algoritmy byly založeny na předpokladu, že omezení v CSP jsou unární nebo binární. V praxi na čistě binární CSP téměř nenarazíme. V reakci Mohr a Masini představili algoritmus *GAC4* [12], který je zobecněnou verzí *AC-4*. *GAC4* dokáže filtrovat pro n-ární omezení. Časová složitost v nejhorším případě je identická s *PC-3* a počet vyřazených hodnot je stejný nebo větší než u *AC-4* [12].

Samotná pravidla Sudoku lze reprezentovat pomocí speciálního n-árního omezení *AllDifferent*. *AllDifferent* je omezením, kde všechny proměnné pokryté tímto omezením musí mít přiřazenou rozdílnou hodnotu. Toto omezení lze filtrovat jako běžné n-ární omezení, pomocí algoritmu *GAC4*. Ovšem Jean-Charles Régin představil nový přístup [6], který dokáže pro omezení *AllDifferent* vynutit zobecněnou hranovou konzistenci značně efektivněji. Tímto algoritmem se zabývá tato kapitola.

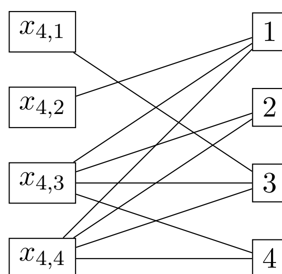
Mohr a Massini [12] definují zobecněnou hranou konzistenci pro n-ární omezení následovně:

Definice 1 *CSP* (X, D, C) *je* ***hranově konzistentní*** *právě tehdy, když pro* $\forall x_i \in X, \forall a_i \in D_i, \forall c = \langle S, R\rangle \in C$ *omezující proměnnou* x_i, $\forall x_j, \ldots, x_k \in S, \exists a_j, \ldots, a_k$ *takové, že omezení* c *je pro* $(a_j, \ldots, a_i, \ldots, a_k)$ *splněno.*

Réginův GAC algoritmus je založený z velké části na teorii grafů, kterou propojuje pomocí dokázaných vět s definicemi hranové konzistence.

Definice 2 *Nechť máme omezení AllDifferent* $c = \langle S, R\rangle$, *pak bipartitní graf* $GV(c) = (S, \bigcup_{i\in S} D_i, E)$, *kde* $(x_i, a) \in E$ *pokud* $a \in D_i$, *se nazývá* ***grafem hodnot*** *z* c.

Obrázek 1.5 je příkladem bipartitního grafu hodnot na Sudoku 2-řádu konkrétně na řádku $i = 4$ z 1.2.



Obrázek 1.5: Graf hodnot pro 4. řádek Sudoku 2-řádu z obrázku 1.2.

Definice 3 *Podmnožina hran* M *v grafu* G *je nazvána* ***párování grafu****, pokud žádné 2 hrany neobsahují společný vrchol.* ***Maximální párování grafu*** *je párování, které obsahuje největší možný počet hran.* ***Perfektní párování grafu*** *pokrývá všechny vrcholy v* G*. Pokud je párování perfektní, pak je i maximální.*

Věta 1 *Nechť máme CSP* (X, D, C)*. CSP je hranově konzistentní právě tehdy, když pro každé AllDifferent omezení* $c \in C$ *každá hrana z* $GV(c) = (S, \bigcup_{i\in S} D_i, E)$ *je součástí nějakého párování, které pokrývá* S.

Důkaz věty 1 je součástí [6]. Předešlá věta propojuje zavedené definice z teorie grafů a hranové konzistence. Z této věty vyplývá, že hrany, které nejsou součástí žádného párování, jenž pokrývá celou množinu proměnných S,

je potřeba odstranit. Na základě tohoto je možné sestavit první část algoritmu, která redukuje domény pro jedno omezení *AllDifferent*.

Algoritmus 5: GAC

```
1 begin
2     G ← Value Graph(c)
3     M ← Find Maximum Matching(G)
4     if |M| < |S| then
5         return False
6     end
7     Remove Edges From(M,G)
8     return True
9 end
```

Graf hodnot na řádku 2 lze vytvořit v čase $O(|S| + |\bigcup_{i \in S} D_i| + d|S|)$, kde d je maximální kardinalita domén. Řádek 3 využívá algoritmus, který vyhledá maximální párování. Vhodným algoritmem je například Hopcraft–Karpův algoritmus, jenž vrátí maximální párování v bipartitním grafu v čase $O(d|S|\sqrt{|V|})$ [13], kde $|V|$ je počet vrcholů, tedy $|S| + |\bigcup_{i \in S} D_i|$. Podmínka na řádcích 4-6 vrátí *False*, pokud párování M nepokrývá celou množinu S, tudíž není možné splnit omezení c pro žádnou kombinaci hodnot z domén. V následujících odstavcích ukážeme, že řádek 7 má lineární časovou složitost. Tedy pro jedno omezení *AllDifferent* algoritmus vyfiltruje za $O(d|S|\sqrt{|V|})$.

K vysvětlení, jak najít hrany, které nepatří k žádnému párování, jenž pokrývá celou množinu S, je potřeba zavést několik následujících definic.

Definice 4 *Nechť M je párování grafu. Jestliže hrana leží v M, pak je označena jako **párovací**, v opačném případě **nepárovací**. Vrchol, který je součástí jedné hrany z M se nazývá **párovací**, jinak **volný**. Střídající cesta/cyklus je obyčejná cesta/cyklus skládající se z hran, jenž jsou střídavě párovací a nepárovací. Délka střídající cesty/cyklu je počet hran, které obsahují.*

Vlastnost 1 (C. Berge, Hypergraphs [14]) *Hrana leží v nějakém maximálním párování, jestliže tato hrana leží na střídavé cestě sudé délky začínající ve volném vrcholu nebo na střídavé kružnici vzhledem k libovolnému maximálnímu párování.*

Pro zjednodušení zápisu následující věty bude bipartitní graf G značen jako $G = (X, Y, E)$.

Věta 2 *Nechť máme bipartitní graf $G = (X, Y, E)$, jeho párování M, které pokrývá množinu X, a graf $G_O = (X, Y, Succ)$, získaný orientací hran v grafu G následovně:*

$$\forall x \in X : Succ(x) = \{y \in Y | (x, y) \in M\}$$
$$\forall y \in Y : Succ(y) = \{x \in X | (x, y) \in (E/M)\}$$

, pak platí tyto vlastnosti:

1. *Každá orientovaná kružnice v G_O odpovídá střídavé orientované kružnici v G a naopak.*
2. *Každá orientovaná cesta sudé délky v G_O, která začíná ve volném vrcholu, odpovídá střídavé sudé cestě v G, která začíná ve volném vrcholu, a obráceně.*

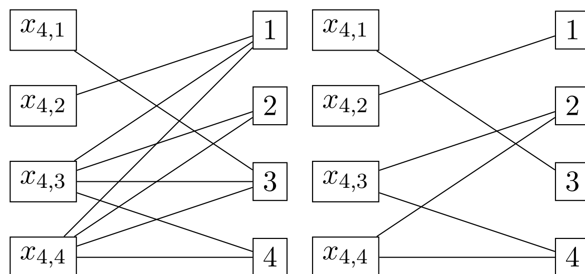
Důkaz věty 2 je opět součástí [6]. Z této věty je již možné vytvořit proceduru, která tyto hrany odstraní.

Algoritmus 6: GAC - Remove Edges From(M,G) [6]

```
1  begin
2  |  Obtain oriented graph G_O from G
3  |  Mark all edges in G_O as Unused
4  |  Look for all directed edges that belong to a directed simple path
   |    which begins at a free vertex by a breadth-first search starting
   |    from free vertices, and mark them as "used"
5  |  Compute strongly connected components and mark all edges
   |    connecting two vertices in the same component as Used
6  |  foreach Unused edge e do
7  |  |  if e ∉ M then
8  |  |  |  Remove e from G
9  |  |  end
10 |  end
11 end
```

Řádek 4 odpovídá vlastnosti 2 z věty 2. Využívá BFS, které spouští z volných vrcholů, a hledá orientované hrany, jenž leží na orientované cestě sudé délky. Tyto hrany označí za použité. K vyhledání orientovaných kružnic na řádku 5 jsou spočítané silně souvislé komponenty grafu G_O. Za použité jsou označeny hrany, které spojují 2 vrcholy v jedné silně souvislé komponentě. Pro nalezení silně souvislých komponent je využit Tarjanův algoritmus, který je najde v čase $O(n+m)$ [15], kde m je počet hran a n je počet vrcholů v grafu. Hrany, které nejsou označeny za použité nebo neleží v párování M, jsou poté odstraněny.

Obdobně jako u algoritmů vynucujících hranovou konzistenci nestačí pouze spustit algoritmus na každé *AllDifferent* omezení jednou. Lze využít stejné myšlenky jako u algoritmu *AC-3*, tedy udělat frontu všech omezení, které budou náhodně nebo postupně vybírány a redukovány pomocí *GAC*. Při odstranění hodnot z domény proměnné x jsou do fronty vložena omezení, která omezují proměnnou x a nerovnají se současnému omezení, které je redukováno.



Obrázek 1.6: Obrázek 1.5 před a po filtraci pomocí *GAC*.

1.5.2 Filtrace během hledání řešení

Techniky představené v předcházejících kapitolách jsou nejčastěji používané před samotným hledáním řešení, a tím zmenšují velikost stavového prostoru. Spolehlivě dokážou redukovat domény proměnných, a v ojedinělých případech najdou řešení nebo fakt, že řešení neexistuje před samotným využitím prohledávacího algoritmu.

Algoritmy fungující na prohledávání do hloubky lze dále vylepšit tím, že zavedeme pracovní domény, ze kterých budeme po přiřazení hodnot do proměnných odstraňovat hodnoty, jenž není možné dosadit a vytvořit konzistentní přiřazení.

Dopředná kontrola

Nejjednodušší a zároveň nejslabší technikou filtrování je dopředná kontrola (FC). Po přiřazení $\langle x_i, a_i \rangle$ kontroluje všechna omezení $c \in C$, ve kterých se proměnná x_i nachází. Omezení $c = \langle S, R \rangle$ je kontrolováno tak, že pro všechny nepřiřazené proměnné $x \in S$ jsou vyzkoušeny všechny hodnoty z jejich pracovních domén $v \in D_x$, hodnota v je vyřazena z pracovní domény D_x, pokud není součástí R se současným částečným přiřazením $\bar{a}'$. Pro Sudoku je tato technika ještě jednodušší, v tom že po přiřazení $\langle x_i, a_i \rangle$ pouze stačí vyřadit hodnotu a_i z pracovních domén proměnných, se kterými je x_i v omezení.

Udržování hranové konzistence

Přímým opakem je udržování hranové konzistence (MAC). Po každém přiřazení $\langle x_i, a_i \rangle$ je zavolán algoritmus, který vyfiltruje domény tak, aby byly hranově konzistentní (např. AC3). Pokud je některá z domén vyprázdněna, pak dosazenou hodnotu a_i zamítáme a odstraníme z domény dosazené proměnné D_i, poté opět spouštíme algoritmus hranové konzistence, ale tentokrát na celou doménu $D_i \setminus a_i$. Jestliže je znovu libovolná doména prázdná, pak současný stav nemá řešení a algoritmus se ve stavovém prostoru vrací. [5] Vynucování hranové konzistence po každém přiřazení a v každém stavu může být v praxi výpočetně drahé.

1.5.3 Heuristiky

Pracovní domény a filtrace během hledání řešení nám dovolují použít další techniky, které mohou výrazně vylepšit efektivitu algoritmu BT. Tyto heuristiky fungují jako strategie při výběru další proměnné, která má být přiřazena, nebo v jakém pořadí do proměnné dosazovat hodnoty z její domény. Heuristiky v algoritmu BT usměrňují průchod stavovým prostorem tak, že slepé konce nebo konzistentní přiřazení jsou objevena dříve. Následující heuristiky jsou obecné pro CSP.

Pro výběr proměnné

Nejběžnější heuristika se často nazývá *Fail first* [5] nebo také *Minimum remaining values (MRV)*. Proces výběru proměnné probíhá tak, že jako další je vybrána proměnná, která nejvíce omezí stavový prostor, což je proměnná, která má ve své současné pracovní doméně nejmenší počet hodnot. Společně s filtrací během hledání řešení je možné, že heuristika vrátí proměnnou s kardinalitou domény rovné 0, a v tom případě heuristika narazila na slepý konec a prohledávání se vrací. Tato heuristika vytváří dynamické řazení proměnných (DVO), tudíž pořadí, v jakém jsou proměnné, se může měnit.

Druhou často používanou heuristikou je *Degree Heuristic*. Oproti MRV tato heuristika vytváří statické řazení proměnných (SVO). Proměnné jsou seřazené sestupně, podle stupně, který má vrchol v grafu omezení.

Při používání MRV se často může stát, že existuje několik domén se stejnou minimální kardinalitou. Jako efektivní řešení se používá způsob, který Daniel Frost a Rina Dechter použili ve své práci [16], kde kombinují tyto heuristiky tak, že primárně je využita MRV, a v případě že existuje několik stejně velkých minimálních domén, se vybere proměnná s větším stupněm.

V tomto je Sudoku speciálním případem, že všechny proměnné v grafu omezení mají stejný stupeň a nelze tedy využít této kombinace.

Pro řazení hodnot

Hlavním cílem v práci [16] nebyly heuristiky pro výběr proměnných, ale heuristiky pro řazení hodnot. Frost a Dechter zde představují algoritmus *look-ahead value ordering (LVO)*, který tyto heuristiky využívá. Ve své práci používají 4 heuristiky. 2 z nich následně představíme a na 2. navážeme heuristikou, kterou využíváme.

Nejlepší výsledky přinesla heuristika *Min-conflicts (MC)*, která řadí hodnoty v z domény D_{cur} současně přiřazované proměnné X_{cur} podle počtu hodnot z domén D' dosud nepřiřazených proměnných, které nemohou splnit omezení s hodnotou v. Tuto heuristiku používáme jako opravnou heuristiku při hledání kandidátních přiřazení v lokálním prohledávání, kterým se zabývá další kapitola.

Druhou heuristikou, která vykazovala dobré výsledky je *Max-domain-size (MD)*. Hodnoty v jsou řazené sestupně podle velikosti nejmenší domény, která po přiřazení a filtraci vznikne. Velice podobnou heuristikou, která je implementována v Sudoku Solveru, je *Least constraining value (LCV)*. Preferované hodnoty jsou ty, které nejméně omezí dosud nepřiřazené proměnné. Cílem těchto heuristik je nechat volnější výběr pro budoucí přiřazení, a tím se vyhnout slepým koncům.

U Sudoku tyto heuristiky často objevují tzv. *Hidden singles*. Na obrázku 1.7 lze pozorovat *hidden single* v červeném kruhu. Tuto hodnotu nelze dosadit do jiné proměnné. Přiřazení této hodnoty by nijak neomezilo domény ostatních nepřiřazených proměnných.

1 3 (4)	2 3	2 3	1
1 3	2 3	2 3	4
3	1	4	2
2	4	1	3

Obrázek 1.7: ukázka *Hidden single*.

1.6 Lokální prohledávání

Opakem systematických, úplných algoritmů jsou algoritmy lokální. Tyto algoritmy jsou založené na principu *Stoupání do kopce* (angl. *Hill climbing*). Nejčastěji pracují již s celým náhodným přiřazením a přechází od jednoho celého přiřazení k dalšímu, pomocí lokálních oprav (např. změna hodnoty proměnné, výměna hodnot dvou proměnných). Podle typu lokálních oprav, které provádíme, je vytvářen stavový prostor, a také kandidátní přiřazení, mezi kterými bude zvoleno to nejlepší. Které je nejlepší, je určeno zvolenou účelovou funkcí, která ohodnotí jednotlivá přiřazení. Tuto funkci se poté algoritmus snaží minimalizovat/maximalizovat tak, že jako další přiřazení vybírá to s lepším ohodnocením. Tento přístup má jeden problém. Není zaručeno, že bude vráceno optimální řešení, protože zde nastává možnost uváznutí v lokálním extrému účelové funkce. Uváznutí v lokálním extrému zde znamená, že neexistuje takové další přiřazení, vytvořené lokální opravou, které by bylo ohodnoceno lépe než to stávající. Z lokálního extrému se lze dostat několika způsoby. V následujících podkapitolách budou představeny existující metaheuristiky, které tento problém řeší. V poslední podkapitole bude představen algoritmus pro řešení Sudoku, který je využit v experimentální části, pomocí kombinace těchto technik. Z části představený v [17].

Algoritmus 7: Hill Climbing(csp)

```
1  begin
2      i ← 0
3      Current ← Random Initialization(csp)
4      while i < Max_Iter ∧ Current ≠ Optimal Solution do
5          Next ← Candidate Solution(Current)
6          # V tomto případě účelovou funkci minimalizujeme
7          if f(Current) > f(Next) then
8              Current ← Next
9          end
10         i ← i+1
11     end
12     return Current
13 end
```

V případě Sudoku budeme využívat účelové funkce, které budeme chtít minimalizovat. Proto v následujících podkapitolách bereme přiřazení, které minimalizuje účelovou funkci jako lepší.

1.6.1 Simulované žíhání

Jak název napovídá, tato metaheuristika simuluje žíhání oceli, což je proces, při kterém dochází k opakovanému zahřívání a chlazení kovů za účelem snížení tvrdosti, zvýšení tažnosti a napomáhající eliminaci vnitřních pnutí.

Simulované žíhání oproti běžnému stoupání do kopce představuje dvě vylepšení. Prvním je možnost přijmutí horšího přiřazení za určitých okolností. Při výběru kandidátního přiřazení je vypočítána změna účelové funkce značena δ,

$$\delta = f(Y) - f(X) \tag{1.2}$$

, kde X je současné přiřazení a Y je kandidátní. Pokud je přiřazení lepší nebo stejné z hlediska účelové funkce, tak provedeme změnu. V případě, že je horší, je změna provedena s pravděpodobností

$$P(\text{přijmutí } Y) = \begin{cases} \exp(-\delta/t) & \delta > 0, \\ 1 & \delta \leq 0 \end{cases} \tag{1.3}$$

[18, 19]. Parametr t se nazývá teplota. Teplota je druhým vylepšením této metody. V triviálním případě je hodnota teploty konstantní. Účinnějším způsobem, jak zacházet s teplotou, je v průběhu vyhledávání teplotu snižovat (chladit). Výsledkem je zpočátku volnější prohledávání stavového prostoru, které dovoluje více horších přiřazení, postupně s nižší teplotou se stává hladovější a konverguje blíže k optimálnímu řešení. [18]

Algoritmus 8: Simulated Annealing(csp)

```
1  begin
2      t ← t_0
3      Current ← Random Initialization(csp)
4      while ¬ Stop Criterion do
5          Next ← Candidate Solution(Current)
6          δ ← f(Next) − f(Current)
7          if δ ≤ 0 then
8              Current ← Next
9          else if exp(−δ/t) > Random(0,1) then
10             Current ← Next
11         t ← Change t according to Cooling Schedule
12     end
13     return Current
14 end
```

Důležitým krokem k dosažení optimálního výkonu algoritmu je volba počáteční teploty t_0. Pokud je teplota příliš vysoká, algoritmus bude přijímat téměř všechny kroky a chováním bude připomínat více náhodnou procházku, která je představena v další podkapitole. Nízká počáteční teplota naopak znemožní přijímání zhoršujících kroků, což povede k uváznutí v lokálním minimu stejně jako u stoupání do kopce.

Neexistuje žádný způsob, jak přesně vypočítat ideální počáteční teplotu. Vhodným způsobem, který vrátí dostatečně dobrý odhad, je najít maximální možný rozdíl $\delta_{\max}$ účelové funkce mezi dvěma soudícími přiřazeními. Hodnotu

t_0 poté lze vypočítat z

$$p = \exp(-\delta_{\max}/t_0) \tag{1.4}$$

, kde p je námi zvolená pravděpodobnost přijetí zhoršujícího kroku, jenž zhorší hodnotu účelové funkce o $\delta_{\max}$. [20]

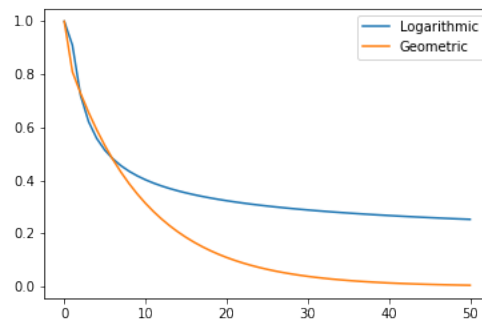
Proces snižování teploty se nazývá *Cooling Schedule*, a také má velký vliv na výkon algoritmu. Základní strategií je logaritmické chlazení. Nevýhodou je pomalá konvergence, která zapříčiňuje delší běh.

$$t_i = \frac{\alpha t_0}{\ln(1+i)}. \tag{1.5}$$

Rychlejší strategií chlazení je geometrická, kde výpočet probíhá následovně

$$t_i = t_0 \alpha^i. \tag{1.6}$$

Parametr α je konstanta v rozmezí 0 až 1, nejčasteji používaná hodnota je v rozmezí 0.8 až 0.99. [21]



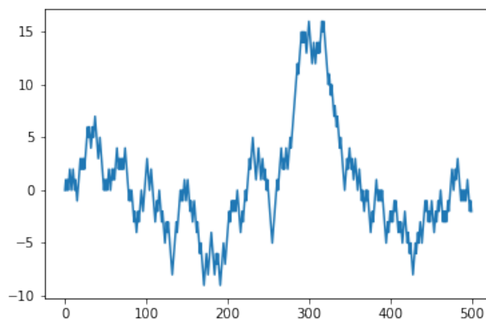
Obrázek 1.8: *Cooling schedule* - Parametr t po 50 iteracích, $t_0 = 1$, $\alpha = 0.9$.

1.6.2 Využití náhodné procházky

Pro následující metodu je vhodné si nastínit matematický princip náhodné procházky. Náhodná procházka je název pro proces, popisující cestu tvořenou určitým počtem náhodných kroků stejné délky. Náhodnou procházku v jednorozměrném prostoru si lze představit jako číselnou osu celých čísel, po které se pohybujeme buď +1 s pravděpodobností p nebo -1 s pravděpodobností $1 - p$.

WalkSAT

WalkSAT je metodou lokálního prohledávání, která využívá myšlenky náhodné procházky. Ačkoliv z názvu vyplývá, že metoda je určena k řešení splnitelnosti booleovských formulí, Rina Dechter popisuje variantu, jak WalkSAT využít pro obecná omezení.



Obrázek 1.9: Vychýlení od počátku po 500 krocích s $p = 0.5$.

Metoda se skládá ze dvou kroků. V prvním kroku je náhodně vybráno omezení, které není splněno současným plným přiřazením. Ve druhém kroku může nastat jedna ze dvou situací. Náhodně změníme hodnoty proměnných s pravděpodobností p nebo s pravděpodobností $1-p$ hladově změníme hodnoty tak, že minimalizujeme počet nově vzniklých nesplněných omezení. Tento stochaistický prvek náhodné procházky je využit k uniknutí z lokálních extrémů. [5]

1.6.3 Tabu prohledávání

„*Tabu prohledávání je založeno na předpokladu, že řešení problémů, aby se kvalifikovalo jako inteligentní, musí zahrnovat adaptivní paměť a responzivní prohledávání.*“ [22]

Pojmy *Tabu Search* a *Metaheuristics* byly poprvé představeny v práci Freda Glovera [23]. Tabu prohledávání (TS) je další metaheuristikou využívanou k řešení kombinatorických problémů pomocí lokálního prohledávání. Narozdíl od ostatních přístupů TS zavádí adaptivní využití paměti. Stavový prostor kandidátních přiřazení prochází systematicky a dynamicky mění okolí dalších přípustných přiřazení. Hlavní myšlenkou je zakazování určitých tahů k předejití cyklení v prostoru a opakování stejných tahů. Tyto zakázané tahy se nazývají *tabu*.

V okolí současného přiřazení je vybráno vždy nejlepší možné kandidátní přiřazení, které nejvíce minimalizuje účelovou funkci. Výběr stojí na domněnce, že vybírání špatných tahů, úmyslně či náhodně, nepřidá žádnou hodnotu, kromě vyšplhání z lokálních optim. [23] TS se nesnaží vyhýbat lokálním optimům, ba naopak k nim směřuje, ale za využití paměti, se z nich dokáže dostat.

Nejčastěji je paměť využita k udržování tzv. krátkodobé paměti, která je implementována např. pomocí kruhového spojového seznamu (tabu list). Z počátku je do tabu listu vloženo n umělých dummy tahů, které jsou během

prohledávání stavového prostoru přepisovány tak, že nejstarší tah je nahrazen nově provedeným. Tahy jsou zde poté uloženy tak, aby nebylo možné se vrátit. Pro ilustraci na problému n-dam, pokud změníme pozici dámy z $x_{1,1}$ na $x_{1,3}$, pak je do tabu listu uložena skutečnost, která zakazuje vrátit dámu z $x_{1,3}$ zpět na $x_{1,1}$.

Možností, jak přijmout tahy označené jako tabu a flexibilně a dynamicky prohledávat okolní stavy, je zavedení aspiračního kritéria. Toto kritérium v nejjednodušší formě povoluje zvolení tabu tahů, pokud jejich přiřazení má za důsledek nejlepší hodnotu účelové funkce, dosud získanou. Složitější aspirační kritérium využívá aspirační list $A(z)$, který je inicializován v rozumném rozsahu možných hodnot využité účelové funkce tak, že $A(z) = z - 1$. Tabu tah t je poté přijmut, pokud zlepšuje hodnotu účelové funkce ze z na hodnotu menší než $A(z)$. Aspirační list je následně aktualizován na $A(z) = f(t)$.

Algoritmus 9: Tabu Search(csp)

```
1  begin
2      i ← 0
3      Tabu list ← ∅
4      Current ← Random Initialization(csp)
5      Current_Score ← Best_Score ← f(Current)
6      while i < Max_iter ∧ Current ≠ Optimal Solution do
7          Next ← Best Candidate Solution Using Aspiration Criterion
             And Tabu List(Current)
8          Move ← (Next, Current)
9          Current ← Next
10         Current_Score ← f(Next)
11         if Current_Score < Best_Score then
12             Best_Score ← Current_Score
13         Update Tabu list(Move)
14         i ← i+1
15     end
16     return Current
17 end
```

Rysy a vlastnosti [22]

- Práce s adaptivní pamětí
  - Selektivita (strategické zapomínání)
  - Abstrakce a dekompozice
  - Časování
    - Nedávnost událostí
    - Frekvence událostí

    * Rozpoznání mezi krátkodobou a dlouhodobou pamětí
  - Kvalita a dopad
    * Relativní atraktivita alternativních možností
    * Velikost změn ve struktuře nebo v omezujících vztazích
  - Kontext
    * Regionální, strukturální, sekvenční vzájemná závislost
- Dynamické prohledávání prostoru
  - Strategicky zavedené omezení a pobídky (tabu podmínky a aspirační kritéria)
  - Zaměření na výhodné okolí a dobré příznaky řešení (intenzifikace)
  - Prohledávání slibných nových okolí (diverzifikace)
  - Nemonotónní prohledávající vzory (strategické oscilování)
  - Integrace a rozšiřování řešení (path relinking)

TS nabízí velké množství způsobů, jak využívat paměť a dynamicky prohledávat stavový prostor. Je tedy možné flexibilně přizpůsobit implementaci pro řešení konkrétních problémů. Součástí těchto vlastností je i velká volnost parametrizace, jenž je důležitou součástí optimalizace implementace.

1.6.4 Min-conflicts

Min-conflicts je heuristika, vytvořená přímo pro řešení problémů splňování omezení. Při lokálním prohledávání se využívá jako opravná heuristika, podle které se volí kandidátní přiřazení, které minimalizuje počet porušených omezení. Proces hledání lze rozdělit do dvou částí. Nejprve je náhodně vybrána jedna z proměnných, která vytváří konflikt, poté se hledá proměnná, se kterou se prohodí.

1.6.5 Kombinace metaheuristik

N. Musliu a F. Winter ve svém článku [17] představili hybridní algoritmus, který kombinuje techniky systematického a lokálního prohledávání pro řešení Sudoku. Jako metaheuristiku zvolili tabu prohledávání, které využívá tabu list jako krátkodobou paměť a ukládá své poslední kroky. K této metaheuristice zakomponovali prvek WalkSATu a náhodné procházky, kdy kandidátní přiřazení, jenž zhorší hodnotu účelové funkce, je přijato jen s určitou pravděpodobností. Toto dělá tabu prohledávání hladovějším a snaží se zamezit okamžitému úniku z lokálního minima.

Algoritmus 10: Tabu_Search_with_WalkSAT

```
1  begin
2      i ← 0
3      Tabu list ← ∅
4      Current ← Initial Assignment
5      Current Score ← Best Score ← f(Current)
6      while i <Max Iter ∧ Current ≠ Optimal Solution do
7          Next ← Best Candidate Solution Using Aspiration Criterion
             And Tabu List(Current)
8          if Aspiration criterion is met then
9              Current ← Next
10             Current Score ← f(Next)
11         else
12             if f(Next) <Current Score ∨ random ≤
                 acceptanceProbability then
13                 Current ← Next
14                 Current Score ← f(Next)
15             end
16         end
17         udpate changes to Tabu list
18         if Current Score <Best Score then
19             Best Score ← Current Score
20         end
21         i ← i + 1
22     end
23 end
```

Kapitola 2

Aplikace na Sudoku

V této kapitole je ukázáno, jak aplikovat obecné algoritmy z teoretické části na Sudoku jako problém splňování omezení.

2.1 Systematické prohledávání

Aplikace systematických algoritmů je přímočará. Výhodou je možnost jednoduché implementace obecných řešičů CSP, kde není potřeba se specifikovat na konkrétní problém.

Sudoku je oproti jiným hrám a reálným problémům velice fixní. Fixní ve smyslu konfigurace problému. Jediné co se mezi zadáními liší je velikost a samotné předvyplněné hodnoty. Počet hodnot v doménách je z počátku stejný a úměrný velikosti, každá proměnná je omezena vždy jen 3 pravidly.

GAC

Implementaci algoritmu GAC lze pro Sudoku zjednodušit. Počet proměnných jednoho *AllDifferent* omezení je stejný jako počet hodnot, které do těchto proměnných budou dosazeny. Tedy po maximálním párování z algoritmu 5, které musí být přes všechny proměnné *AllDifferent* omezení, aby omezení mohlo být splněno, bude graf hodnot obsahovat pouze vrcholy označené jako párovací. Důsledkem tohoto je, že řádek 4 z algoritmu 6 může být v implementaci přeskočen.

2.2 Lokální prohledávání

Metody lokálního prohledávání narozdíl od systematických jsou většinou implementované pro konkrétní problém. Částmi, které jsou potřeba implementovat, jsou inicializace prvotního přiřazení, účelová funkce, která bude řízena metaheuristikou k optimální hodnotě, operátor kandidátního přiřazení, krok

(iterace) algoritmu a restartování/ukončení. V našem Sudoku solveru implementujeme metaheuristiky simulované žíhání, tabu prohledávání a tabu prohledávání s prvkem WalkSAT. Inicializace, účelová funkce a restart/ukončení jsou pro tyto metaheuristiky totožné. Iterace algoritmu a operátor kandidátního přiřazení jsou rozdílné a popsané pro implementované metody.

Inicializace

První kandidátní přiřazení je zcela náhodné až na výjimku toho, že v $n \times n$ blocích se vyskytují hodnoty 1 až n^2 pouze jednou. Tedy bloková omezení jsou po inicializaci vždy splněna. [18]

Operátor kandidátního přiřazení

Hledání kandidátního přiřazení využívá heuristiku *Min-conflicts* tak, že nejprve je náhodně vybrána počátečně nevyplněná proměnná, která vytváří konflikt, tedy políčko, jehož hodnota se v řádku nebo sloupci opakuje, a poté je ve stejném $n \times n$ bloku vyhledána proměnná, se kterou si prohodí hodnoty, a tím minimalizují počet konfliktů nebo-li účelovou funkci. U simulovaného žíhání je konfliktní proměnná vyměněna s náhodnou nepředvyplněnou proměnnou. Tabu prohledávání zkouší konfliktní proměnnou vyměnit za všechny nefixované proměnné v $n \times n$ bloku a rozhoduje se podle aspiračních kritérií a obsahu tabu listu, více k rozdílům je v iteracích algoritmů. [17]

Účelová funkce

Důsledkem prvotní inicializace a způsobu, jakým je hledáno kandidátní přiřazení, je ušetření výpočetního výkonu na kontrolování blokových omezení a výpočtu účelové funkce. Tedy účelová funkce, kterou algoritmus bude chtít minimalizovat, je definována pro kandidátní přiřazení $\bar{a}$ následně [18]:

$$f(\bar{a}) = \sum_{i=1}^{n^2} r(i) + \sum_{j=1}^{n^2} c(j) \tag{2.1}$$

, kde $r(i)$, resp. $c(j)$ jsou funkce, jejichž výsledkem je počet konfliktů na řádku i, resp. ve sloupci j.

Jelikož při každém kroku algoritmu jsou vyměněny pouze 2 proměnné, tak se změní hodnota účelové funkce maximálně jen pro 2 řádky a 2 sloupce. Proto by bylo nevýhodné počítat účelovou funkci pro celou herní plochu a je zde využita delta evaluace, která funguje tak, že má uložené počty konfliktů pro řádky a sloupce, a k výslednému skóre přičítá rozdíl ovlivněných řádků nebo sloupců. Obrázek 2.1 je příkladem změny účelové funkce při výměně hodnot 2 proměnných.

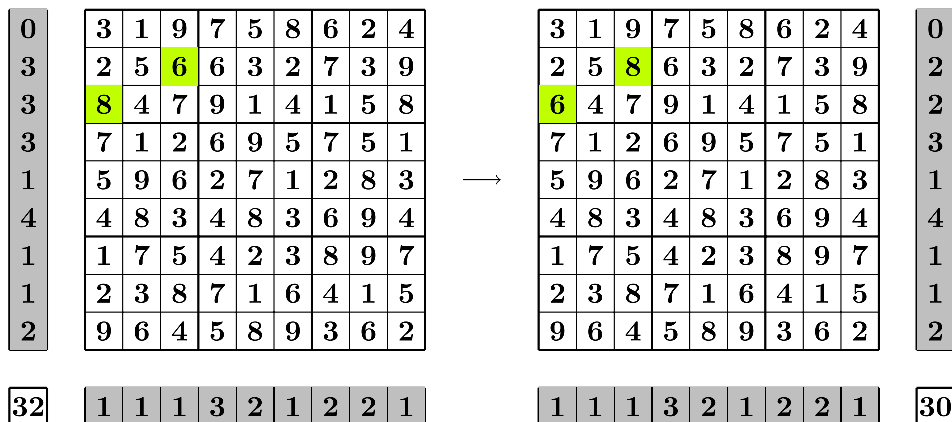
Iterace algoritmu

Simulované žíhání

Jedna iterace simulovaného žíhání začíná vždy využitím operátoru kandidátního přiřazení, kde jsou vybrány nefixované proměnné. Jedna, která tvoří konflikt a druhá náhodná. Tyto proměnné jsou zaměněny a je vypočítán rozdíl účelové funkce δ. Pokud záměna minimalizuje hodnotu účelové funkce, pak je krok uznán. Pokud se hodnota účelové funkce zvýšila, pak je krok uznán s pravděpobností $\exp(-\delta/t)$, v případě, že není uznán zhoršující krok, je záměna vrácena. Posledním krokem iterace je chlazení teploty.

Tabu prohledávání a TS s prvkem WalkSAT

Výběr ideálního kandidátního přiřazení probíhá v několika krocích. Operátor kandidátního přiřazení vybere nefixní konfliktní proměnnou, a s ní vyzkouší všechny možné záměny ve stejném bloku. Rozdíly hodnot účelové funkce si ukládá. Následně je vyzkoušeno základní aspirační kritérium na nejlepší záměně. Pokud je splněno je záměna provedena. Pokud není, pak algoritmus přechází k nejlepší záměně, která není součástí tabu listu a provádí tuto záměnu. Rozdíl pro TS s prvek WalkSAT je, pokud výměna zhoršuje hodnotu účelové funkce, pak je přijata pouze s pravděpodobností P. Posledním krokem iterace metaheuristiky je aktualizování tabu listu a zapsaní posledního kroku.



Obrázek 2.1: Krok algoritmu a změna účelové funkce.

Restart a ukončení

Jak bylo několikrát zmíněno, algoritmy iterovaného lokálního prohledávání nejsou úplné. Toto znamená, že není garantováno, zda se algoritmus zastaví a vrátí řešení. Iterační algoritmy se tedy často omezují počtem iterací, které mohou vykonat předtím než dojde k automatickému ukončením. Pokud po

dosažení limitu iterací není vráceno optimální řešení je algoritmus spouštěn znovu pro další náhodnou inicializaci. Toto je omezeno časovým limitem. Jestliže ani poté není vráceno optimální řešení je pokus označen za neúspěšný. Počet iterací je určen vztahem $itern^2$. Pokud je při prohledávání nalezena nová nejmenší hodnota účelové funkce, pak je počet iterací vynulován. Určení hodnoty konstanty *iter* je v experimentální části.

Parametry metod lokálního prohledávání

Simulované žíhání implementuje 2 měnitelné parametry: P *a* α. P určuje pravděpobnost, se kterou bude přijato nejhorší možné kandidátní přiřazení a je využito k výpočtu počáteční teploty t_0. α určuje rychlost chlazení teploty pomocí geometrického chlazení.

TS a TS s prvkem WalkSAT nabízí parametr c, který představuje konstantu ve vztahu $tabu_list_size = cn^2$. TS s prvkem WalkSAT má ještě o jeden parametr více. P určuje pravděpodobnost, se kterou bude přijato zhoršující kandidátní přiřazení.

Ladění hodnot parametrů je součástí experimentální části.

2.3 Implementace a tvorba experimentů

Veškerá implementace Sudoku Solveru byla provedena v jazyce Python 3.8[1]. Python nabízí velké množství standardních a externích knihoven, a díky své syntaxi nám dovoluje produkovat poměrně efektivní a čitelný kód. Python je tedy vhodný pro prototypování a dovoluje nám se soustředit na zkoumaný problém než na detaily jeho implementace. Jedinou externí knihovnou, která byla při tvorbě Sudoku Solveru využita, je NumPy[2]. NumPy je matematická knihovna, která se stala základem pro všemožné vědecké výpočty v Pythonu. Knihovna ulehčuje a zefektivňuje tvorbu vícedimenzionálních polí a objektů z nich vytvořených(např. matice). Součástí je také množství rychlých operací nad těmito objekty.

Experimenty byly zrealizovány pomocí Jupyter[3] notebooků, jenž dovolují zařadit vizualizaci k experimentům a je možné snadně experimenty monitorovat a replikovat. Kromě standardních Python knihoven a tříd ze Sudoku solveru je využita knihovna Matplotlib[4], jenž nám umožňuje vytvořit čitelnou vizualizaci dat z výsledků. Použité knihovny se nacházejí vždy ve vrchní části.

[1] https://www.python.org/
[2] https://numpy.org/
[3] https://jupyter.org/
[4] https://matplotlib.org/

Tabulka 2.1: Možné konfigurace algoritmů pomocí parametrů tříd.

Backtracking		Min_Conflicts	
ac	GAC AC3	method	TS_WalkSAT TS SA
var_h	MRV InOrder		
val_h	LCV AscendingValues		
bruteforce	boolean		

Kapitola 3

Experimentální část

Tato kapitola obsahuje výsledky a diskuzi k vykonaným experimentům. Část kapitoly je věnovaná generování testovacích dat. Poté jsou hledány nejlepší možné konfigurace algoritmů systematického prohledávání a ladění parametrů u metod lokálního prohledávání. Oba přístupy jsou následně testovány na testovacích datech a porovnávány. Kapitola končí diskuzí zaobírající se výsledky a možnými zlepšeními.

3.1 Generování herních ploch

K generování herních ploch používáme jednoduchý generátor popsaný v práci [18]. Tento generátor využívá algoritmu Root-Solution, který vytvoří zcela validní a úplnou herní plochu n-tého řádu. Z této plochy lze pomocí permutací:

- Permutací sloupců $n \times n$ bloků.
- Permutací řádků $n \times n$ bloků.
- Permutací sloupců políček v jednom sloupci $n \times n$ bloků.
- Permutací řádků políček v jednom řádku $n \times n$ bloků.

vytvořit další herní plochy, které budou stále splňovat všechna omezení. Pomocí těchto operací lze z tohoto základního řešení vygenerovat až $n!^{2(n+1)} - 1$ rozdílných ploch[18].

Kromě parametru n generátor přijímá parametr p, který určuje počet předvyplněných herních polí v rozsahu $\langle 0; 1 \rangle$ v procentech. Po provedení permutací je náhodně zvoleno $1 - p$ herních polí, které budou vyprázdněny. Nakonec je nová herní plocha uložena do souboru.

Generátor je schopný vytvořit a uložit kolem 800 herních ploch 3-řádu během 1 vteřiny. Test rychlosti je součástí Jupyter notebooku s generováním testovacích dat.

Algoritmus 11: Root-Solution(n)[18]

```
1  begin
2      x ← 0
3      for i ← 1 to n do
4          for j ← 1 to n do
5              for k ← 1 to n^2 do
6                  grid[n(i − 1) + j][k] ← x (mod n^2) + 1
7                  x ← x + 1
8              end
9              x ← x + n
10         end
11         x ← x + 1
12     end
13 end
```

3.2 Testovací data

Pro experimenty bylo vygenerováno celkem 1200 herních ploch. 20 pro každou hodnotu p od 0 do 0,95 s 0,05 inkrementy, tedy 400 herních ploch pro jeden řád Sudoku. Pro experimenty byly využity herní plochy 3, 4 a 5-řádu.

3.3 Systematické prohledávání

Použité algoritmy systematického prohledávání nejsou parametrizovatelné. Tedy cílem těchto testů bylo najít nejlepší možnou kombinaci algoritmů. Testy byly provedeny na herních plochách 3 a 4-řádu.

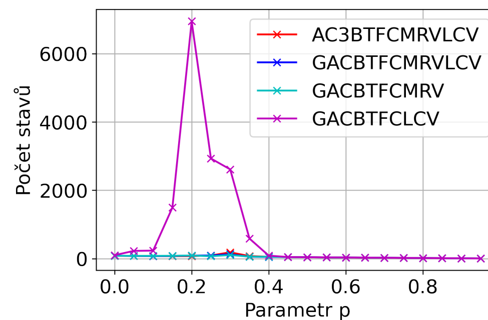
První test byl spuštěn na zlomku herních ploch 3-řádu. Test sloužil pouze jako demonstrace důležitosti algoritmů hranové konzistence a představených heuristik oproti čistému prohledávání s návraty, které lze přirovnat k brute-force algoritmům. Hlavní měřenou statistikou tohoto testu byl počet prohledaných stavů stavového prostoru před nalezením řešení.

Z tabulky 3.1 je patrné, že BT s dopřednou kontrolou a společně s heuristikami prohledá v průměru několika násobně méně stavů. S vynucením hranové konzistence je tento rozdíl řádový. Velmi špatných výsledků dosahuje heuristika MRV bez vynucení hranové konzistence. Jelikož před prohledáváním nejsou vyfiltrované domény, tak jsou všechny o velikosti n^2. Tedy heuristika MRV v prvním kroku vybírá proměnnou zcela náhodně, což může způsobit, že se BT vydá špatným směrem ve stavovém prostoru a objevení tohoto může být výpočetně drahé.

Tabulka 3.1: Průměrný počet prohledaných stavů v herních plochách Sudoku 3-řádu.

Algoritmus	Počet stavů
BruteforceBT	2356
BT+FC	841
BT+FC+MRV	481049
BT+FC+LCV	625
BT+FC+MRV+LCV	40971
AC3+BT+FC+MRV+LCV	44
GAC+BT+FC+MRV+LCV	43
GAC+BT+FC+MRV	44

Na základě prvního testu byl proveden test heuristik, zda není výhodnější některou z heuristik vynechat a tím ušetřit výpočetní výkon. Test byl uskutečněn na 400 herních plochách 3-řádu. Měřenou statistikou byl opět počet stavů tentokrát v závisloti na p.

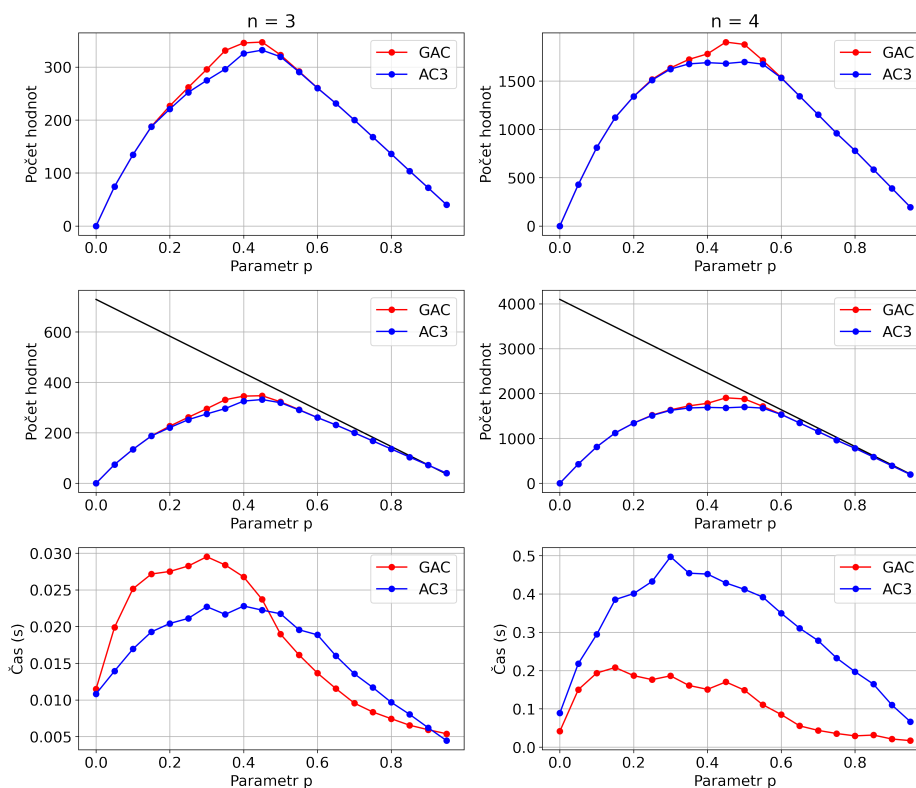


Obrázek 3.1: Výsledky testu heuristik systematického prohledávání.

Výsledkem 3.1 je fakt, který jsme předpokládali z teoretické části, že kombinace těchto heuristik je opravdu lepší než samostatné heuristiky.

GAC vs. AC3

Z předešlých výsledků není zcela zřejmý rozdíl mezi použitím GAC a AC3, jelikož obě metody vykazují podobné výsledky. Následující experiment byl opět proveden na 400 herních plochách 3-řádu, ale oproti předchozímu experimentu bylo přidáno 400 herních ploch 4-řádu. Měřenými veličinami byl počet vyfiltrovaných hodnot v závislosti na parametru p a potřebný čas k vyfiltrování těchto hodnot opět v závisloti na p.



Obrázek 3.2: Výsledky testu GAC vs. AC3.

Algoritmus GAC vždy v nejhorším případě odstraní stejný počet hodnot jako AC3, ale v rozsahu parametru p od 0,2 do 0,6 dosahuje až o 10–20% lepších výsledků. Prostřední řada grafů na obrázku 3.2 obsahuje úsečku, znázorňující počet hodnot v doménách před filtrováním. Algoritmy se k této úsečce přibližují a při hodnotě parametru $p > 0{,}5$ ji téměř kopírují. Což znamená, že dokáží vyfiltrovat téměř všechny hodnoty, které nebude možné dosadit, a tím výrazně zmenšit stavový prostor před samotným prohledáváním. Z poslední řady grafů je možné pozorovat, že GAC dosahuje lepších výsledků s rostoucím n, co se týče času stráveného filtrováním. Počet hran, se kterými je AC3 inicializován, je $4n^2\binom{n^2}{2}+n^4(n-1)^2$. Ve srovnání s GAC, které pracuje s $3n^2$ *AllDifferent* omezeními.

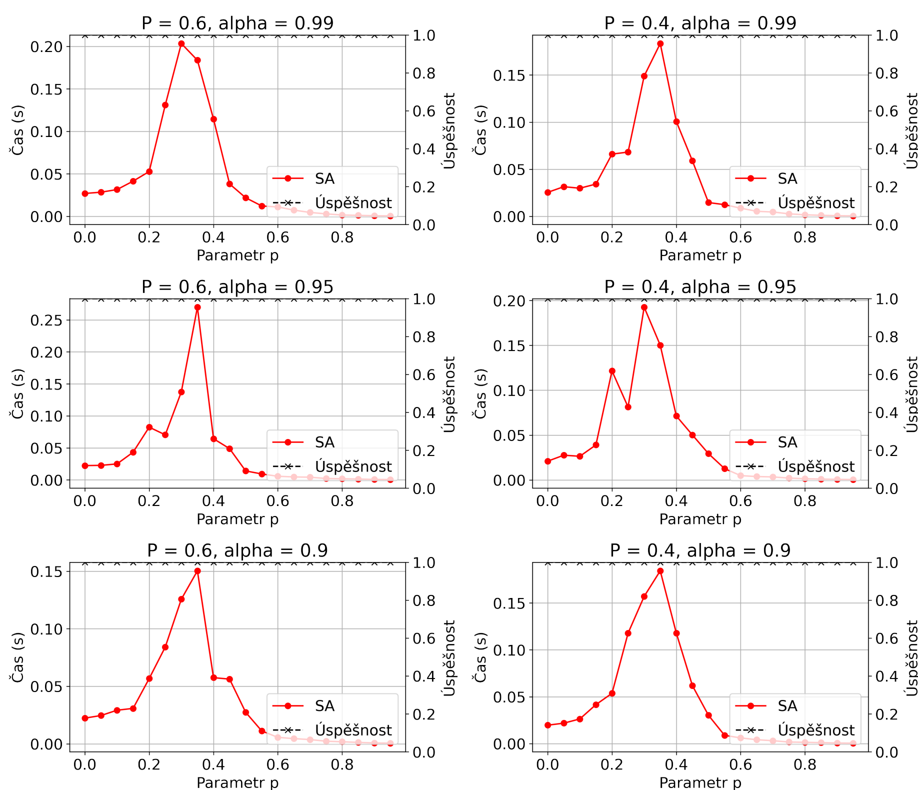
3.4 Lokální prohledávání

Hledání nejlepší metaheuristiky pro řešení Sudoku probíhalo laděním parametrů. Součástí Sudoku solveru jsou metaheuristiky simulované žíhání, tabu prohledávání a tabu prohledávání se stochaistickým prvkem WalkSAT. Metaheuristiky byly laděny na všech 400 herních plochách 3-řádu. Pro iterační

limit byla zvolena konstanta s hodnotou 70, tedy algoritmus byl zastaven, pokud hodnota účelové funkce neklesla pod dosud nejlepší dosaženou hodnotu za posledních $70n^2$ iterací. Pro tuto hodnotu metaheuristiky vykazovaly nejlepší výsledky. Pro menší hodnotu byly algoritmy restartovány, dříve než mohly narazit na lokální minimum a pro výrazně vyšší hodnoty naopak trávily v těchto minimech příliš mnoho času. Po překročení iteračního limitu byly algoritmy znovu spuštěny s novou počáteční inicializací, pokud celkový čas běhu nepřesáhl časový limit. Časový limit byl určen podle řádu herní plochy. 5 sekund pro $n = 3$, 30 sekund pro $n = 4$. Pokud nebylo nalezeno optimální řešení do časového limitu, pak byl pokus označen za neúspěšný. Časy pokusů, které byly neúspěšné, nebyly započítány do výsledných časů. V testech byly měřeny 2 veličiny. Čas a úspěšnost. Obě v závislosti na parametru p.

Simulované žíhání

Simulované žíhaní nabízí 2 parametry. Parametr P, jenž určuje pravděpodobnost, se kterou bude přijato nejhorší možné kandidatní přiřazení (Nejhorší možný tah zvětší hodnotu účelové funkce o 4 body.) po výpočtu prvotní teploty, a parametr α. α je využita při ochlazování teploty, čím menší α, tím rychlejší bude chlazení. Test byl spuštěn pro hodnoty $\alpha \in \{0.99, 0.95, 0.9\}$ a $P \in \{0.6, 0.4\}$.



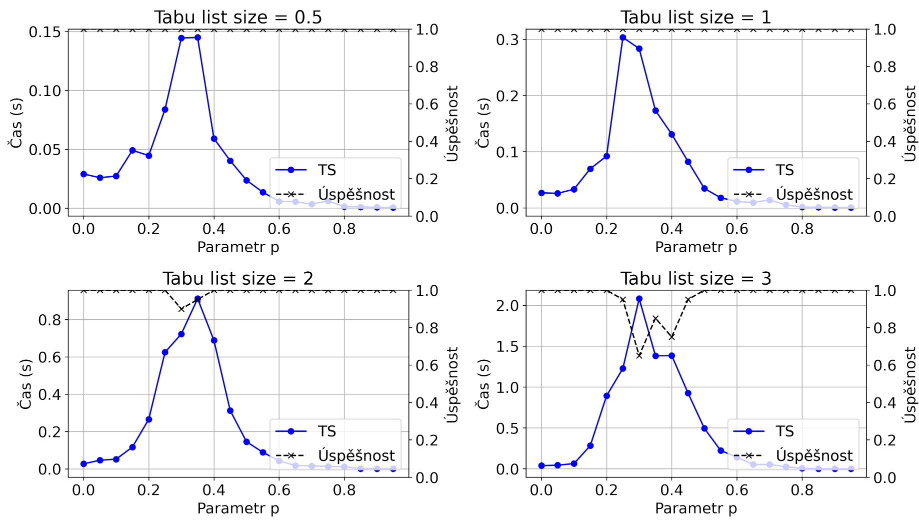
Obrázek 3.3: Výsledky ladění parametrů Simulovaného žíhaní.

Z výsledků 3.3 lze pozorovat, že úspěšnost byla pro všechny kombinace 100 %. Nejlepších výsledků dosáhla kombinace P =0,6 a α =0,9. Nedaleko za ní byly P =0,4, α =0,99 a P =0,4 a α =0,9. Z těchto výsledků lze vyvodit, že rychlejší konvergování k nižším teplotám a menší šance přijmutí zhoršujících kandidátů přináší pro Sudoku lepší výsledky. Poprvé zde můžeme pozorovat výrazně vyšší časy pro herní plochy s p od 0,2 do 0,5. Tento jev se nazývá *Phase Transition* a budeme se jím zabývat v diskuzi na konci této kapitoly.

Tabu prohledávání

Samotné tabu prohledávání nabízí pouze jeden parametr, a tím je konstanta c, jenž představuje velikost tabu listu. Výsledná velikost tabu listu je cn^2. Test byl spuštěn pro hodnoty $c \in \{0.5, 1, 2, 3\}$.

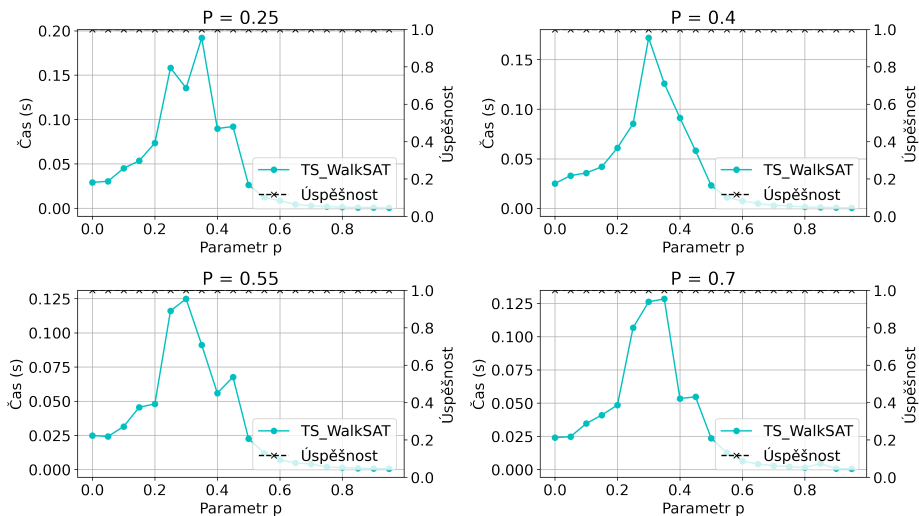
Na grafech 3.4 pozorujeme snížení úspěšnosti a velký časový nárůst pro hodnoty {2, 3}. Tabu list ideální velikosti by měl zabraňovat cyklickým změnám, ale neměl by jinak omezovat průchod prostorem. Nejlepších výsledků bylo dosaženo s hodnotou parametru rovné 0,5.



Obrázek 3.4: Výsledky ladění parametru Tabu prohledávání.

Tabu prohledávání se stochaistickým prvkem WalkSAT

Na základě výsledků předchozího experimentu jsme využili hodnotu parametru, určujícího velikost tabu listu, rovné 0,5. Díky tomu stačí tuto metaheuristiku ladit jen pro parametr P, jenž určuje s jakou pravděpodobností bude přijat kandidát, který zhoršuje hodnotu účelové funkce. Testované hodnoty byly {0.25,0.4,0.55,0.7}.

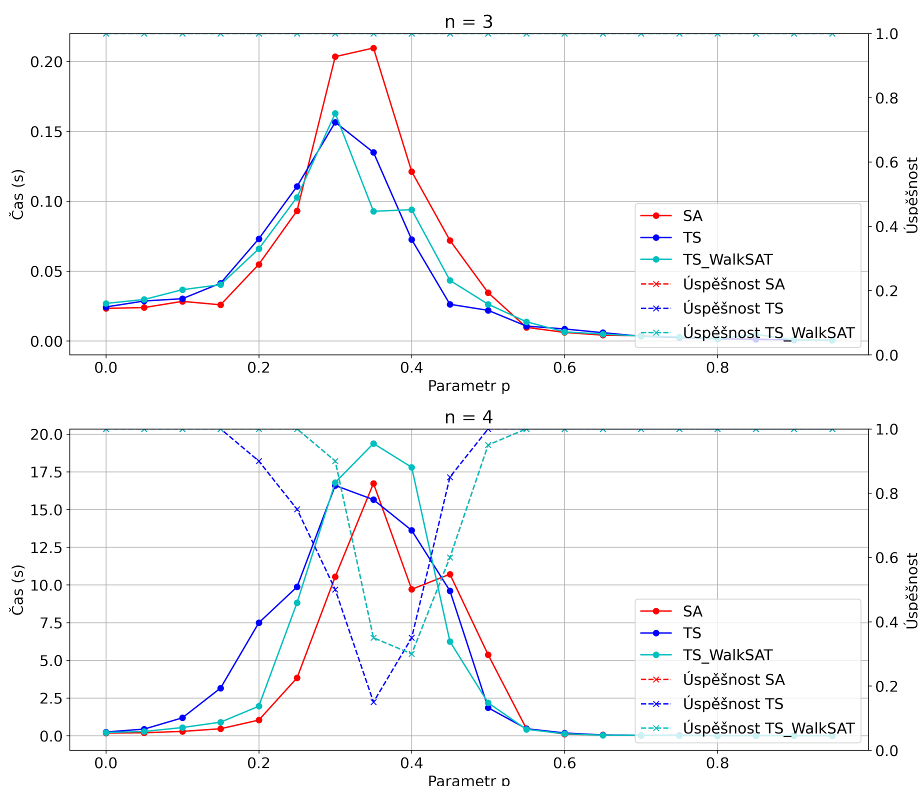


Obrázek 3.5: Výsledky ladění parametru Tabu prohledávání se stochaistickým prvkem WalkSAT.

Jako u simulovaného žíhání je úspěšnost 100 % pro všechny hodnoty P.

Časově nejrychlejší je P =0,55, jen o pár milisekund pomalejší je P =0,70. Čím větší hodnota P, tím více tato metaheuristika napodobuje samotné tabu prohledávání. Při $P = 1$ jsou totožné. V rámci testování jsme zvolili pro následné experimenty hodnotu P =0,55 jako lepší.

Závěrem této podkapitoly je test metaheuristik s nejlepšími dosaženými parametry mezi sebou. Pro rekapitulaci těmito hodnotami parametrů jsou: simulované žíhání P =0.6, α =0,9, tabu prohledávání c =0,5, tabu prohledávání se stochaistickým prvkem WalkSAT c =0,5, P =0,55. Testy byly provedeny na herních plochách 3 a 4-řádu.



Obrázek 3.6: Výsledky testu porovnání metod lokálního prohledávání.

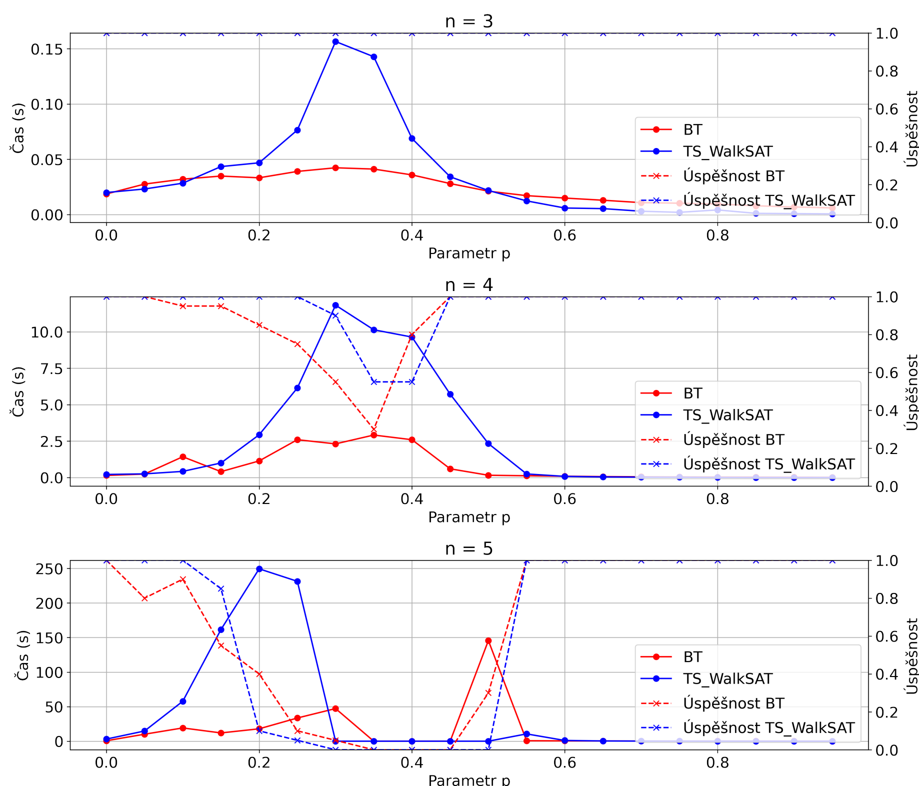
Z výsledků na 3.6 shledáváme podobné výsledky všech metaheuristik na Sudoku 3-řádu. Na herních plochách 4-řádu jsou výsledky již rozdílné. Velkým rozdílem je pokles úspěšnosti pro hodnoty p od 0,25 až 0,45. Dle úspěšnosti si vedly totožně simulované žíhaní a tabu prohledávání s prvkem WalkSAT. Časově nejlépe si vedlo simulované žíhaní. Naopak nejhůře v obou měřených statistikách dopadlo samotné tabu prohledávání.

Z experimentů na metodách lokálního prohledávání můžeme pozorovat trend, kterým se ubíraly výsledky jednotlivých metaheuristik, jenž ukazoval, že hladovější a rychlejší konvergování k lokálním optimům není pro problém jako

je Sudoku špatné ba naopak lepší. Na základě tohoto trendu jsme zvolili tabu prohledávání s prvkem WalkSAT jako lepší metaheuristiku než simulované žíhání pro následující testy. Z důvodu myšlenky této metaheuristiky, která se blíží tomuto trendu více.

3.5 Srovnání

Posledním a nejdůležitějším experimentem bylo porovnání obou přístupů. Za systematické prohledávání byla zvolena kombinace GAC, BT, FC, MRV, LCV. Metaheuristikou lokálního prohledávání bylo tabu prohledávání s prvkem WalkSAT s parametry c =0,5, P =0,55. Testy byly provedeny na všech 1200 herních plochách řádů 3,4 a 5. Měřenými statistikami byly opět čas a úspěšnost v závislosti na p. Časové limity pro $n = 3$ a $n = 4$ zůstaly stejné a to 5 a 30 sekund. Časový limit pro $n = 5$ byl nastaven na 300 sekund.



Obrázek 3.7: Výsledky testu porovnání metod systematického a lokálního prohledávání.

Sudoku 3-řádu netvoří pro algoritmy žádný problém. Oba přístupy dokázaly vyřešit herní plochy se 100% úspěšností v dobrém čase. Prohledávání s návraty je zde rychlejší. Pro $n = 4$ již můžeme pozorovat pokles úspěšnosti

obou přístupů. U BT od $p = 0{,}1$ do $p = 0{,}4$. TS s WalkSAT je na tom lépe. Pokles nastal v rozsahu $p = 0{,}3$ až 0,4. Na posledním grafu 3.7 jsou výsledky pro plochy 5-řádu. Zde jsou algoritmy v celku neúspěšné a vyžadují mnohem větší časový limit. Pokles pro upravené TS nastává opět déle než u BT, ale od $p = 0{,}3$ do $p = 0{,}5$ je úspěšnost nulová. BT je zcela neúspěšné pouze v rozsahu p 0,35 až 0,45. Oba algoritmy si znovu s přehledem dokážou poradit s problémy od $p = 0{,}55$.

3.6 Diskuze

Z dosažených výsledků nelze jednoznačně udělat závěr o tom, který přístup je lepší. Metoda lokálního prohledávání dosahovala lepších výsledků při nízkých hodnotách p. Významný rozdíl mezi metodami vyšel také v testu na plochách 4-řádu, kde TS bylo úspěšnější. Naopak při testu $n = 5$ TS selhávalo už při $p = 0{,}2$. Pro hodnoty $p > 0{,}6$ jsou výsledky téměř totožné.

Phase transitions

Implementované algoritmy s rostoucím n začaly ztrácet úspěšnost kolem hodnoty $p = 0{,}35$. Okolo této hodnoty se náchazejí nejtěžší problémy. Tento jev, kdy dramaticky klesne úspěšnost a problémy se stanou mnohem těžšími, se nazývá *phase transition*. Cheeseman, Kanefsky a Taylor [24] ve své práci uvádí, že každý NP-úplný problém má řadící parametr a nejtěžší problémy se nacházejí okolo kritických hodnot tohoto parametru. Pro Sudoku byl tímto parametrem, parametr p. Kritickými hodnotami(hranicemi, kde nástávala phase transition) v našich výsledcích byly přibližně $p = 0{,}2$ a $p = 0{,}5$.

Možná zlepšení

Zlepšení algoritmů by se mělo zabývat hlavně těžkými problémy mezi hranicemi phase transition. Tyto problémy obsahují největší množství lokálních optim a slepých konců pro systematické prohledávání. Systematické algoritmy je možné vylepšit vynucováním silnější konzistence, a tím dále zmenšit stavový prostor na úkor výpočetního výkonu. Z výsledků se také ukázalo, že dopředná kontrola (FC) je příliš slabá pro herní plochy vyššího řádu. Zajímavým testem by bylo porovnání s algoritmem MAC. Metody lokálního prohledávání lze zlepšit v několika bodech. Prvním je inicializace prvotního přiřazení. Čím lepší bude prvotní přiřazení, tím blíž budeme hned z počátku globálnímu optimu, tedy v průměru by měl být počet iterací k dosažení optima menší. Zde se nabízí využití algoritmů hranové konzistence, které by mohly přinášet lepší výsledky než náhodná inicializace, která vytváří prvotní přiřazení tak, aby nebyla porušena bloková omezení. Sudoku je fixní problém, který se s velikostí moc nemění, tedy by stálo za úvahu, zda nelze vymyslet heuristiku specializovanou přímo pro tento problém.

Závěr

Tato práce byla zaměřena na aplikaci několika state–of–the–art algoritmů systematického a lokálního prohledávání na zobecněnou variantu hry Sudoku modelovanou jako problém splňování omezení. Vytyčené cíle práce byly splněny.

V první části jsme představili základní algoritmus BT doplněný o metody vynucování konzistencí GAC, AC3, FC a heuristiky MRV, LCV, které vedly prohledávání skrze stavový prostor k optimálnímu řešení. Dále byly představeny metaheuristiky lokálního prohledávání založené na stoupání do kopce jako simulované žíhání a tabu prohledávání, které stavěly na heuristice Min Conflicts.

Ve druhé části jsme představené algoritmy aplikovali na zobecněnou variantu hry Sudoku a implementovali jsme prototyp řešiče Sudoku v jazyce Python.

Posledním cílem bylo porovnání implementovaných algoritmů. Běžné herní plochy 3-řádu netvořily pro algoritmy žádný problém. Algoritmy tyto plochy vyřešily během několika setin až desetin sekundy. Systematický přístup zde byl rychlejší. Na plochách 4-řádu úspěšnost algoritmů poklesla. Největší úspěšnost zde měla metaheuristika lokálního prohledávání TS se stochaistickým prvkem WalkSAT, která měla na nejtěžších plochách až o 30 % větší úspěšnost. Plochy 5-řádu tvoří exponenciálně větší stavový prostor a algoritmy zde narazily a pro plochy s 20–55 % předvyplněnými poli měly téměř nulovou úspěšnost. Úvodní hypotéza nemůže být potvrzena. Systematické algoritmy jsou efektivnější pro problémy, kde stavový prostor není obrovský nebo je možné ho redukovat. Metody lokálního prohledávání nejsou tolik znevýhodněné zvětšeným stavovým prostorem, ale potřebují vyšší časový limit.

Dle dosažených výsledků by bylo vhodné pro budoucí práce zvážit kombinaci obou přístupů. Popřípadě se zaměřit na nejtěžší herní plochy a hledat v nich vlastnosti, na kterých by mohly vzniknout nové heuristiky. Nejtěžší herní plochy zpravidla z počátku obsahují 25–40 % předvyplněných polí.

Literatura

[1] Aiden, H.: Anything but square: from magic squares to Sudoku. 3 2006, [online], [cit. 19-01-2021]. Dostupné z: https://plus.maths.org/content/anything-square-magic-squares-sudoku

[2] Team, G.: Sudoku – The History (All You Need to Know). [online], [cit. 20-01-2021]. Dostupné z: https://www.gamesver.com/sudoku-the-history-all-you-need-to-know/

[3] Yato, T.; Seta, T.: Complexity and Completeness of Finding Another Solution and Its Application to Puzzles. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, ročník E86-A, 05 2003.

[4] Knuth, D.: Dancing Links. In *Millennial Perspectives in Computer Science: Proceedings of the 1999 Oxford-Microsoft Symposium in Honour of Sir Tony Hoare*, Cornerstones of Computing, Macmillan Education UK, 2000, ISBN 9780333922309, s. 187–214.

[5] Dechter, R.: *Constraint Processing*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003, ISBN 9780080502953.

[6] Régin, J.-C.: A Filtering Algorithm for Constraints of Difference in CSPs. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (Vol. 1)*, AAAI '94, USA: American Association for Artificial Intelligence, 1994, ISBN 0262611023, str. 362–367.

[7] Mackworth, A. K.: Consistency in Networks of Relations. *Artif. Intell.*, ročník 8, 1977: s. 99–118.

[8] Mackworth, A. K.; Freuder, E. C.: The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial Intelligence*, ročník 25, č. 1, 1985: s. 65–74, ISSN 0004-3702,

doi:https://doi.org/10.1016/0004-3702(85)90041-4. Dostupné z: https://www.sciencedirect.com/science/article/pii/0004370285900414

[9] Mohr, R.; Henderson, T. C.: Arc and path consistency revisited. *Artificial Intelligence*, ročník 28, č. 2, 1986: s. 225–233, ISSN 0004-3702, doi:https://doi.org/10.1016/0004-3702(86)90083-4. Dostupné z: https://www.sciencedirect.com/science/article/pii/0004370286900834

[10] Wallace, R. J.: Why AC-3 is Almost Always Better than AC-4 for Establishing Arc Consistency in CSPs. In *Proceedings of the 13th International Joint Conference on Artifical Intelligence - Volume 1*, IJCAI'93, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993, str. 239–245.

[11] Bessière, C.: Arc-consistency and arc-consistency again. *Artificial Intelligence*, ročník 65, č. 1, 1994: s. 179–190, ISSN 0004-3702, doi:https://doi.org/10.1016/0004-3702(94)90041-8. Dostupné z: https://www.sciencedirect.com/science/article/pii/0004370294900418

[12] Mohr, R.; Masini, G.: Good Old Discrete Relaxation. In *Proceedings of the 8th European Conference on Artificial Intelligence*, ECAI'88, USA: Pitman Publishing, Inc., 1988, ISBN 0273087983, str. 651–656.

[13] Hopcroft, J. E.; Karp, R. M.: A N5/2 Algorithm for Maximum Matchings in Bipartite. In *Proceedings of the 12th Annual Symposium on Switching and Automata Theory (Swat 1971)*, SWAT '71, USA: IEEE Computer Society, 1971, str. 122–125, doi:10.1109/SWAT.1971.1. Dostupné z: https://doi.org/10.1109/SWAT.1971.1

[14] Berge, C.: *Graphes et hypergraphes.* Paris: Dunod, 1970, 502 s.

[15] Tarjan, R.: Depth-First Search and Linear Graph Algorithms. *SIAM JOURNAL ON COMPUTING*, ročník 1, č. 2, 1972: s. 146–160.

[16] Frost, D.; Dechter, R.: Look-Ahead Value Ordering for Constraint Satisfaction Problems. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI'95, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1995, ISBN 1558603638, str. 572–578.

[17] Musliu, N.; Winter, F.: A Hybrid Approach for the Sudoku Problem: Using Constraint Programming in Iterated Local Search. *IEEE Intell. Syst.*, ročník 32, č. 2, 2017: s. 52–62, doi:10.1109/MIS.2017.29. Dostupné z: https://doi.org/10.1109/MIS.2017.29

[18] Lewis, R.: Metaheuristics can solve sudoku puzzles. *Journal of Heuristics*, ročník 13, č. 4, 08 2007: s. 387–401. Dostupné z: http://rhydlewis.eu/papers/META_CAN_SOLVE_SUDOKU.pdf

[19] Henderson, D.; Jacobson, S.; Johnson, A.: *The Theory and Practice of Simulated Annealing*. 04 2006, s. 287–319, doi:10.1007/0-306-48056-5_10.

[20] Ben-Ameur, W.: Computing the Initial Temperature of Simulated Annealing. *Computational Optimization and Applications*, ročník 29, 12 2004: s. 369–385, doi:10.1023/B:COAP.0000044187.23143.bd.

[21] Mahdi, W.; Medjahed, S. A.; Ouali, M.: Performance Analysis of Simulated Annealing Cooling Schedules in the Context of Dense Image Matching. *Computación y Sistemas*, ročník 21, 10 2017: s. 493–501, doi:10.13053/CyS-21-3-2553.

[22] Glover, F.; Laguna, M.; Marti, R.: *Tabu Search*, ročník 16. Springer, Boston, MA, 07 2008, doi:10.1007/978-1-4615-6089-0.

[23] Glover, F.: Future paths for integer programming and links to artificial intelligence. *Computers & Operations Research*, ročník 13, č. 5, 1986: s. 533–549, ISSN 0305-0548, doi:https://doi.org/10.1016/0305-0548(86)90048-1, applications of Integer Programming. Dostupné z: `https://www.sciencedirect.com/science/article/pii/0305054886900481`

[24] Cheeseman, P.; Kanefsky, B.; Taylor, W. M.: Where the Really Hard Problems Are. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI'91, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1991, ISBN 1558601600, str. 331–337.

Příloha **A**

Seznam použitých zkratek

CSP Constraint satisfaction problem

BT Backtracking

SAT Boolean satisfiability problem

NC Node consistency

AC Arc consistency

PC Path consistency

GAC Generalized arc consistency

FC Forward checking

MAC Maintaining arc consistency

MRV Minimal remaining values

DVO Dynamic variable ordering

SVO Static variable ordering

LCV Least constraining values

TS Tabu search

SA Simulated annealing

Příloha **B**

Obsah přiloženého CD

```
README.txt ....................... popis obsahu CD
src
├── SudokuSolver ................. zdrojové kódy implementace
├── Experiments .................. testovací data, experimenty a výsledky
└── Thesis ....................... zdrojová forma práce ve formátu LaTeX
text
└── thesis.pdf ................... text práce ve formátu PDF
```