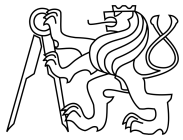
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Heuristiky pro propagaci intervalů
Student:	Jakub Kottnauer
Vedoucí:	doc. Ing. Stefan Ratschan
Studijní program:	Informatika
Studijní obor:	Teoretická informatika
Katedra:	Katedra teoretické informatiky
Platnost zadání:	Do konce letního semestru 2016/17

Pokyny pro vypracování

Cíl práce: Experimentální porovnání účinnosti heuristik pro řešení omezujících podmínek pomocí propagací intervalů.
Pokyny:
1) Implementujte algoritmus pro řešení omezujících podmínek metodami "branch and prune" a "hull consistency".
2) Identifikujte podstatné parametry algoritmu ovlivňující jeho účinnost.
3) Identifikujte charakteristiky problémových instancí ovlivňující chování algoritmu.
4) Studujte vybrané články o heuristikách.
5) Implementujte vybrané heuristiky z literatury nebo vlastní nápady.
6) Udělejte výpočetní experimenty s implementovanými heuristikami za účelem porovnání výpočetní účinnosti.
7) Interpretujte výsledky.

Seznam odborné literatury

- J. G. Cleary: Logical Arithmetic, Future Computing Systems, 1987, number 2, pp. 125--149.
- Frederic Benhamou and W. J. Older: Applying Interval Arithmetic to Real, Integer and Boolean Constraints, Journal of Logic Programming, 1997, vol. 32, number 1., pp. 1-24.
- Christophe Lecoutre: Constraint Networks: Techniques and Algorithms, Wiley-IST, 2009.
- Yahia Lebbah and Olivier Lhomme. Accelerating filtering techniques for numeric CSPs. Artificial Intelligence, 139(1):109–132, 2002.
- Goualard, Frédéric and Jermann, Christophe: A Reinforcement Learning Approach to Interval Constraint Propagation, Constraints, 13(1-2):206-226, 2008.

L.S.

doc. Ing. Jan Janoušek, Ph.D.
vedoucí katedry

prof. Ing. Pavel Tvrdík, CSc.
děkan

V Praze dne 26. listopadu 2015

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra teoretické informatiky

Bakalářská práce

Heuristiky pro propagaci intervalů

Jakub Kottnauer

Vedoucí práce: doc. Ing. Stefan Ratschan

9. května 2016

Poděkování

Děkuji doc. Ing. Stefanu Ratschanovi za vedení mé bakalářské práce a za ochotu poradit s jakýmkoliv problémem při její tvorbě.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o etické 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. § 46 odst. 6 tohoto zákona 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, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 9. května 2016 .....

Odkaz na tuto práci

Kottnauer, Jakub. *Heuristiky pro propagaci intervalů*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2016.

Abstrakt

Omezující podmínka je relace mezi proměnnými omezující množiny hodnot, kterých mohou proměnné nabývat. Problém splnitelnosti omezujících podmínek (CSP) je problém, jehož řešení spočívá v nalezení hodnot proměnných tak, aby byly splněny všechny zadané omezující podmínky. Hlavním cílem práce je otestování vlivu heuristik na efektivitu řešení numerických CSP pomocí propagací intervalů. Součástí práce byl napsán řešič v jazyce F# implementující algoritmy HC3 a branch-and-prune s podporou pro podmínky s operacemi sčítání, odčítání a násobení. Byly nalezeny podstatné parametry algoritmu ovlivňující jeho účinnost a následně byly s třinácti heuristikami provedeny výpočetní experimenty a jejich výsledky porovnány. Výstup práce bude možno využít při rozvažování, kterou heuristiku použít při řešení soustav omezujících podmínek převeditelných na podmínky s výše uvedenými operacemi.

Klíčová slova propagace intervalů, algoritmus HC3, branch and prune, problém splnitelnosti, omezující podmínky, heuristiky, konzistenční techniky, funkcionální programování, FSharp

Abstract

A constraint is a relation between variables which reduces the set of values that can be assigned to a variable. A constraint safisfaction problem (CSP) is the problem of finding values for the variables in a given constraint that satisfy the constraint. The main goal of this thesis is to test the influence of various heuristics on the efficiency of solving numerical CSP problems by interval propagation. A solver written in the F# language utilizing the HC3 algorithm and a branch-and-prune algorithm was created, important aspects of the HC3 algorithm influencing its efficiency were found and then experiments with thirteen different heuristics were performed. The results found in the thesis can be used when deciding which heuristic to use for solving constraint satisfaction problems.

Keywords interval propagation, HC3 algorithm, branch and prune, constraint satisfaction problem, constraints, heuristics, consistency, functional programming, FSharp

Obsah

Seznam obrázků

Seznam tabulek

Seznam algoritmů

Úvod

V mnoha oblastech lidské činnosti, ať už je to věda, podnikání, či sport, existují problémy, které kladou na svá řešení nějaká omezení. Takovým problémům se obecně říká problémy splnitelnosti omezujících podmínek (*constraint satisfaction problems*). Omezující podmínky zde zužují množiny hodnot, kterých mohou nabývat proměnné vyskytující se v problému.

Tato bakalářská práce se zabývá speciálním typem CSP problémů, nazývaných numerické CSP problémy. Numerické CSP jsou takové problémy, které se dají popsat soustavou rovnic a nerovnic a jejichž proměnné nabývají hodnot z reálných intervalů. Jedna z možností, jak tyto problémy řešit, se nazývá *propagace intervalů*. Tyto metody postupně zmenšují intervaly jednotlivých proměnných, až je dosaženo dostatečné přesnosti řešení. K rozhodnutí, v jakém pořadí zmenšovat intervaly a zpracovávat omezující podmínky, pomáhají heuristiky.

Hlavním cílem této práce je otestovat vliv různých heuristik na efektivitu hledání řešení numerických CSP problémů, přičemž jednotlivé heuristiky budou porovnány na základě několika indikátorů naměřených v průběhu experimentů.

Práce je rozdělena do čtyř hlavních kapitol - v první kapitole je vysvětlena teorie nutná k implementaci řešiče, jenž byl vytvořen v rámci této práce. Druhá kapitola přehledně shrnuje cíle práce, kterých je dosaženo v posledních dvou kapitolách. Návrh řešiče, který dostal jméno HullSolver, je včetně použitých algoritmů popsán ve třetí kapitole. Poslední částí textu je popis experimentů provedených s programem a shrnutí výsledků.

Součástí dokumentu jsou také tři přílohy. V příloze A jsou vysvětleny zkratky použité v bakalářské práci. V příloze B je uveden manuál k programu HullSolver a na závěr příloha C shrnuje obsah přiloženého CD.

Kapitola 1

Popis problematiky

První část práce představuje úvod do teorie řešení problémů s omezujícími podmínkami. Nejprve jsou zde popsány základní typy těchto problémů a způsoby, jak je řešit. Poté následuje popis několika algoritmů a na závěr kapitoly jsou uvedeny heuristiky, které mohou pomoci zvýšit efektivitu řešení.

1.1 Constraint programming

Constraint programming (či *programování s omezujícími podmínkami*) je jedním z odvětví umělé inteligence pro řešení optimalizačních úloh, konkrétně problémů splnitelnosti omezujících podmínek (CSP problémy). Asociace ACM v článku [1] označila toto téma jako jedno z klíčových oblastí pro budoucí výzkum, neboť se problémy s omezujícími podmínkami přirozeně vyskytují v každodenním životě. Předností constraint programming je navíc to, že uživatel popíše problém k vyřešení pouze deklarativně - nedá tedy počítači žádný postup k řešení, jen zadá aktuální stav problému a specializovaný program (řešič) najde řešení (pokud existuje).

Typickým příkladem problému s omezujícími podmínkami jsou různé hry, například sudoku. Jak ukazuje definice 1, CSP se skládá z proměnných, domén a omezujících podmínek. V sudoku jsou proměnnými volná políčka na hracím poli, z nichž každá má svoji *doménu* (množinu hodnot, kterých může teoreticky nabývat, v tomto případě čísla 1 až 9). Omezujícími podmínkami jsou samotná pravidla hry - požadavek na unikátnost číslice v řádku, sloupci, resp. ve čtverci. Pojmy *doména* a *omezující podmínka* přesněji vysvětlují definice č. 2 a 3 níže.

Definice 1 ([2, s. 14])**.** *CSP* (*problém splnitelnosti omezujících podmínek*) je trojice (V, D, C), kde

- V je uspořádaná posloupnost proměnných,
- D je uspořádaná posloupnost domén (viz definice č. 2) náležejících k proměnným,

- C je množina omezujících podmínek (viz definice č. 3).

Známý SAT problém[1] se také dá formulovat jako problém s omezujícími podmínkami.

1.1.1 Doména

Definice 2 ([3, s. 10])**.** *Doména* (nebo také *definiční obor*) proměnné je množina všech hodnot, kterých může proměnná nabývat.

Doménou, jak ukazuje definice 2, může být jakákoliv množina. Nejjednodušší jsou diskrétní konečné domény (podmnožiny přirozených čísel, $\{red, green, blue\}$ při obarvování grafu, $\{true, false\}$ u SAT problém, apod.). Diskrétní domény mohou být také nekonečné - v takovém případě se může stát, že řešení nelze popsat jednoduše výčtem hodnot z domén a je nutné jej popsat vztahem mezi proměnnými [4, s. 139].

V reálném světě se často vyskytují problémy se spojitými doménami, které mohou být reprezentovány pomocí intervalů (lineární programování je příkladem takového problému).

1.1.2 Omezující podmínka

Definice 3 ([3, s. 11])**.** *Omezující podmínka* c na konečné posloupnosti proměnných $(x_1, \ldots, x_n)$ s doménami $(D_1, \ldots, D_n)$, $n \in \boldsymbol{N}$, je podmnožina kartézského součinu $D_1 \times \cdots \times D_n$.

Omezující podmínka dle definice 3 je tedy relace mezi proměnnými a lze ji zapsat jako $c(x_1, \ldots, x_n)$. Vzhledem k tomu, že se jedná o relaci, lze omezující podmínky rozdělit podle arity na relace unární ($x \leq 1$), binární ($x = y$) a více-ární ($x + y > z$) [4, s. 139].

V praktické části této práce budou omezující podmínky reprezentovány rovnicemi (jednoduchá rovnice $x = 0, x \in \boldsymbol{N}$ zjevně omezuje možné hodnoty proměnné x, jedná se tedy o omezující podmínku).

Omezující podmínka je *redundantní* (nadbytečná) [3, s. 20], nemá-li její odebrání z CSP vliv na řešení problému (v problému $P = (V, D, C)$, kde $C = c_1 : x > y, c_2 : x > y, c_3 : y < x$ jsou libovolné dvě podmínky redundantní).

Pro práci je důležité ještě jedno dělení omezujících podmínek, a to na *primitivní* a *komplexní* podmínky (více viz [5]). Omezující podmínka reprezentovaná rovnicí v primitivním tvaru obsahuje maximálně jednu aritmetickou operaci (maximální arita podmínky je tedy rovna třem). Příkladem mohou být podmínky $x = y^2$ či $a = \sin b$). Toto rozdělení je důležité, protože algoritmus využitý v praktické části umí pracovat pouze s primitivními podmínkami

[1] Boolean satisfiability problem

a program je očekává na vstupu. Před spuštěním programu je tedy nutné provést manuální rozklad podmínek a označit, které z proměnných se vyskytovaly v původních podmínkách. Takovým proměnným se říká *dominantní*, ostatní jsou *nedominantní*, nebo také *pomocné* proměnné.

Rozdíl mezi ekvivalentními jednoduchými a komplexními podmínkami ukazují příklady č. 1.1 a 1.2. V tomto příkladu jsou dominantními proměnnými x, y a z.

$$x + 2y + z = 1 \tag{1.1}$$

$$v_1 = 2y \quad \wedge \quad v_2 = z - 1 \quad \wedge \quad v_3 = x + v_1 \quad \wedge \quad v_3 + v_2 = 0 \tag{1.2}$$

1.2 Numerical constraint satisfaction problem

Tato práce se zabývá řešením *numerických CSP* (*NCSP*, *numerical CSP*), což je podmnožina CSP problémů, jejichž domény jsou spojité (nejčastěji jsou to intervaly) a které se dají reprezentovat soustavami rovnic či nerovnic. Rovnice nebo nerovnice vlastně kladou na proměnné nějaká omezení a zmenšují tak množiny hodnot, kterých mohou nabývat [6].

Domény proměnných v NCSP problémech jsou reálné intervaly, a cílem řešiče je co nejvíce tyto intervaly zúžit (zúžení domény podle omezující podmínky se říká *propagace omezující podmínky*, v případě NCSP se tento proces někdy nazývá *propagace intervalů*), při zachování všech řešení soustavy rovnic tvořené omezujícími podmínkami. To ukazuje následující jednoduchý příklad 1.3.

$$x^2 = 1 \qquad x \in \boldsymbol{R} \tag{1.3}$$

Nechť rovnice 1.3 představuje omezující podmínku a pro doménu D_x proměnné x platí $D_x = (-5; 5)$. Všechna řešení této rovnice určitě leží v intervalu D_x, ale také například v $\langle -1; 2)$, nikoliv však v $(0; 2)$ (tento interval obsahuje jen jedno ze dvou řešení). Metody řešení numerických CSP dokáží problém vyřešit nalezením minimálního intervalu obsahujícího všechna řešení, tj. $D'_x = \langle -1; 1\rangle$. Nalezený interval vlastně tvoří jakési jednodimenzionální ohraničení všech možných řešení. V případě více než jedné proměnné je toto ohraničení tvořeno kartézských součinem intervalů a anglicky se nazývá *box*.

Výhodou těchto problémů je fakt, že se dají (alespoň do počtu třech dominantních proměnných) snadno graficky vizualizovat. Domény proměnných představují jednotlivé osy, omezujícími podmínkami jsou funkce a boxy obalující řešení jsou skutečnými 2D či 3D „krabicemi“ okolo řešení.

Nebude-li řečeno jinak, bude se zbývající text právě věnovat především problematice numerických CSP.

1.3 Intervalová aritmetika

Intervalová aritmetika je rozšířením aritmetiky reálných čísel na intervaly. Následující definice jsou převzaty [7], s výjimkou, že pro značení otevřených a uzavřených intervalů je využita notace používaná v české literatuře:

- $\langle a; b\rangle \equiv \{x \in R | a \leq x \leq b\}$ - uzavřený interval
- $(a; b) \equiv \{x \in R | a < x < b\}$ - otevřený interval

Pro sčítání, odčítání, násobení a dělení intervalů platí následující vztahy ($x = \langle a; b\rangle$ a $y = \langle c; d\rangle$ jsou intervaly a odpovídající vztahy platí i pro otevřené intervaly):

$$x + y = \langle a + c; b + d\rangle \tag{1.4a}$$
$$x - y = \langle a - c; b - d\rangle \tag{1.4b}$$
$$x \cdot y = \langle min(ab, ad, bc, bd); \; max(ab, ad, bc, bd)\rangle \tag{1.4c}$$
$$\frac{x}{y} = x \cdot \frac{1}{y} \text{ pokud } 0 \notin y, \text{ kde } \frac{1}{y} \equiv \langle \frac{1}{b}; \frac{1}{a}\rangle \tag{1.4d}$$

Odčítání lze převést na sčítání pomocí tohoto vztahu:

$$-x = -\langle a; b\rangle = \langle -b; -a\rangle \tag{1.5a}$$

Dalšími potřebnými operacemi jsou průnik a sjednocení:

$$x \cap y = \{i \in R | i \in x \wedge i \in y\} \tag{1.6a}$$
$$x \cup y = \langle min(a, c); max(b, d)\rangle \tag{1.6b}$$

Jako *intervalový obal* (*interval hull*, značeno H) se označuje nejmenší interval „obalující“ podmnožinu reálných čísel. Například platí, že $H(\{2, 5, 8\}) = \langle 2; 8\rangle$, $H((1; 3) \cup (5; 7)) = (1; 7)$.

V souvislosti s intervaly je třeba zmínit ještě tzv. *dependency problem* (problém závislosti). To je situace, kdy se jeden interval vyskytuje na více místech výrazu a výsledný interval může být zbytečně široký. Dependency problem se projeví i na velmi jednoduché rovnici $x - x = 0, x \in \langle 0; 10\rangle$. Na první pohled je správným řešením interval $\langle 0; 0\rangle$, podle pravidel pro součet v intervalové aritmetice bude ale výsledek $\langle 0; 10\rangle - \langle 0; 10\rangle = \langle -10; 10\rangle$.

1.4 Řešení a splnitelnost CSP

Přiřazení konkrétní hodnoty z domény proměnné se nazývá *ohodnocení* (*label*). Přiřazuje-li ohodnocení hodnotu všem proměnným problému, jedná se o *úplné ohodnocení* (*compound label*). Tyto pojmy stačí k definici řešení a splnitelnosti problému s omezujícími podmínkami (definice 4, resp. 5).

Definice 4 ([2, s. 14])**.** *Řešení* (*solution*) CSP je takové úplné ohodnocení, pro které platí všechny omezující podmínky.

Definice 5 ([2, s. 14])**.** CSP problém je *splnitelný* (*satisfiable*), existuje-li pro něj řešení.

Podle charakteru problému někdy stačí nalézt jen jedno řešení, jindy je třeba nalézt všechna řešení, jindy je nutné najít optimální řešení.

Jak ale vůbec řešení problému probíhá? Nejprve je nutné zavést pojmy *ekvivalence problémů* a *redukce problému*. Dva CSP problémy jsou ekvivalentní, jsou-li nadefinované nad stejnou posloupností proměnných a stejnou množinou omezujících podmínek a mají stejné množiny řešení [3, s. 18]. Například problémy

$$P_1 = ((x, y), (\{-1, 0, 1, 2, 3\}, \{1, 2, 3\}), \{x - y = 0\})$$
$$P_2 = ((x, y), (\{1, 2, 3\}, \{1, 2, 3\}), \{x - y = 0\})$$

jsou ekvivalentní a splnitelné. Na stejném příkladu lze demonstrovat i redukci problému. Redukce problému je transformace na ekvivalentní a jednodušší CSP [3, s. 23]. Problém P_2 tak mohl vzniknout z problému P_1 redukcí domény proměnné x, přičemž byly z domény dané proměnné odebrány tzv. *nekonzistentní* hodnoty (hodnoty, pro které neexistuje řešení). Jak redukce může probíhat vysvětluje kapitola 1.4.2. V tuto chvíli ji stačí považovat za černou skříňku, která na vstupu přijme CSP a vrátí ekvivalentní zredukovaný problém.

1.4.1 Branch-and-prune algoritmus

Obecný postup pro řešení CSP, který je popsán algoritmem 1 (převzatý z [3, s. 22]) využívá k nalezení řešení právě redukování problému.

Algoritmus `Solve` má tři důležité části - volání `Happy` funkce, ověření *atomicity* a rozdělení zredukovaného problému (*splitting*). Spuštění algoritmu navíc často předchází předzpracování zadaného problému, jako například detekce a odstranění redundantních podmínek, nebo rozklad komplexních podmínek na primitivní.

1. Funkce `Happy` ověřuje, jestli je program „spokojený“ s aktuálním výsledkem a jestli může skončit. To může znamenat například to, jestli

Algoritmus 1 Algoritmus Solve

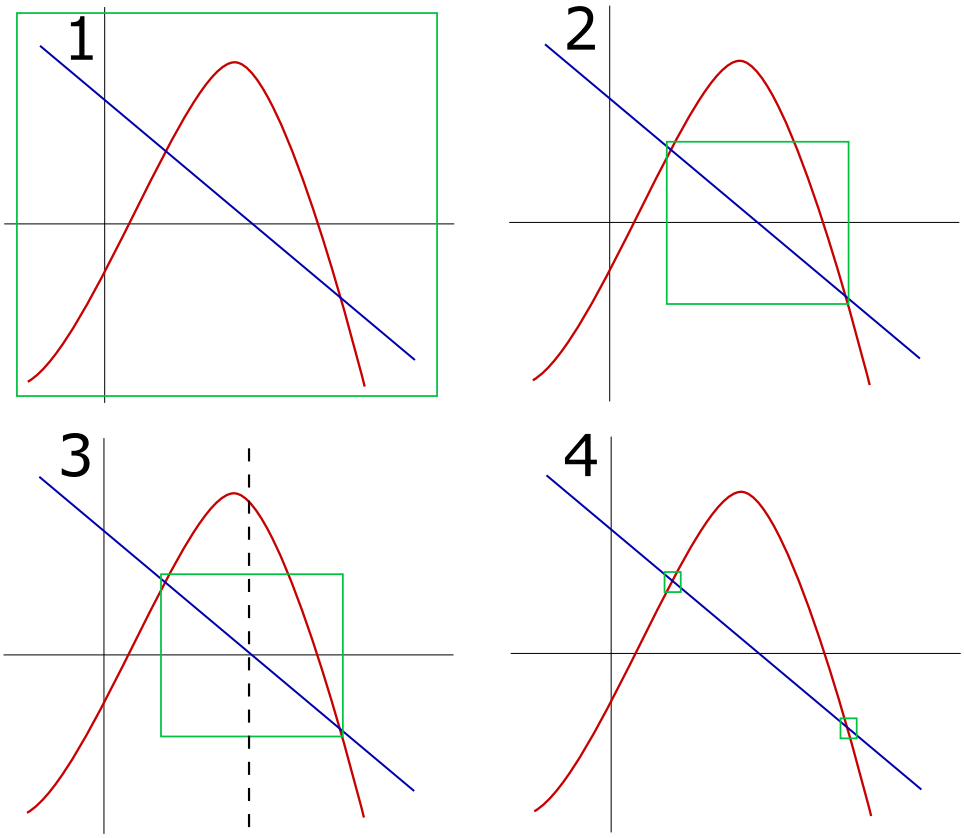
In: CSP problém $P = (V, D, C)$ k vyřešení.
Out: Zredukovaný problém P.

```
1:  procedure Solve(P)
2:      continue ← true
3:      while continue do
4:          Zredukuj problém P
5:          if not Happy then
6:              if atomic then
7:                  continue ← false
8:              else
9:                  P1, P2 ←Split - rozděl problém a opakuj pro jednotlivé části
10:                 Solve(P1)
11:                 Solve(P2)
12:             end if
13:         else
14:             return P
15:         end if
16:     end while
17:     return P
18: end procedure
```

byl nalezen dostatečný počet řešení, nebo jestli bylo dokázáno, že žádné řešení neexistuje, apod.

2. Ověření atomicity je kontrola, zda je zredukovaný problém již dostatečně malý, aby byl přijat jako řešení. Pro určení velikosti problému lze využít několik metrik, v případě NCSP se nejčastěji používá měření velikosti domén dominantních proměnných a to buďto absolutní, nebo relativní vzhledem k původní velikosti (například požadavek na zmenšení domény na tisícinu původní velikosti).

3. Nebyl-li problém dostatečně zmenšen, dojde k jeho rozdělení na více částí (resp. na několik menších CSP). To v praxi nejčastěji znamená, že dojde k rozdělení domén podle nějakého pravidla, kterých lze vymyslet poměrně velké množství: u CSP s diskrétními doménami je jedno z možných rozdělení takové, kdy se z domény nějaké proměnné vyjme jedna hodnota a ta se použije u problému P_1 a zbytek hodnot z domény se použije u problému P_2. U problému P_1 pak lze velmi rychle rozhodnout, zda patří do řešení, nebo ne. U numerických CSP problémů je nejjednodušší nějakým způsobem rozdělit jeden či více intervalů (domén). V programu vytvořeném pro tuto práci je při každém rozdělení rozpůlena doména jedné proměnné, která je vybrána metodou *round-*



Obrázek 1.1: Průběh algoritmu Solve s rozdělováním na menší problémy

robin. To znamená, že jsou voleny jedna po druhé, tak jak jdou za sebou v definici problému zadaného uživatelem.

Ohledně rozdělení problému na menší části se možná nabízí otázka, proč je to vůbec potřeba, když redukce problému (řádek č. 4 v pseudokódu) by měla být schopná odstranit nekonzistentní hodnoty z domén. V některých typech problému (a je to případ i NCSP problémů řešených zde) a v závislosti na použitém algoritmu se může stát, že se algoritmus zastaví na krajních hodnotách domény. Obrázek č. 1.1 znázorňuje průběh řešení NCSP problému s využitím splittingu.

Červená a modrá křivka jsou omezující podmínky (soustava dvou rovnic) a zelený rámeček okolo nich v bodu 1 je box tvořený doménami proměnných. Řešení problému se pochopitelně nalézá v průsečících obou křivek, algoritmus ale při prvním průchodu dokáže box zmenšit maximálně tak, jak je ukázáno v bodu 2. Následně dojde k rozpůlení jedné z domén a algoritmus se znovu spustí pro obě poloviny, které se již podaří maximálně zredukovat.

Algoritmům tohoto typu se říká *branch-and-prune* algoritmy, což je volně přeložitelné jako „větvení a odřezávání větví“ - řešení se při zavolání `Split`

Počáteční domény	$x \in \langle 0;2 \rangle$	$y \in \langle 1;3 \rangle$	$z \in \langle 4;6 \rangle$
$z = x + y$			$\langle 1;5 \rangle$
$y = z - x$		$\langle 2;6 \rangle$	
$x = z - y$	$\langle 1;5 \rangle$		
Nové domény	$x \in \langle 1;2 \rangle$	$y \in \langle 2;3 \rangle$	$z \in \langle 4;5 \rangle$

Tabulka 1.1: Příklad redukce domén proměnných v podmínce $z = x + y$

rozdělí na několik větví, z nichž ty, které nevedou k řešení, budou odřezány.

Jak bude vidět v implementační části, řešič vytvoření v této práci používá podobný algoritmus, viz algoritmus 5 na straně 21.

1.4.2 Propagace omezujících podmínek se sčítáním a násobením

Teoretický základ pro propagaci omezujících podmínek se sčítáním, násobením a několika dalšími operaci položil John G. Cleary v roce 1987 ve svém článku *Logical Arithmetic* [8]. Tabulka 1.1 ukazuje příklad (převzatý z [8]) redukce domény pro omezující podmínku ve tvaru $z = x + y$. Pro zúžení domény jedné proměnné stačí proměnnou vyjádřit z rovnice a provést průnik její domény s intervalem vzniklým z výrazu na druhé straně rovnice. Například pro zredukovanou doménu D'_z proměnné z platí $D'_z = D_z \cap (D_x + D_y)$.

Propagace podmínek s násobením lze provést podobně, s jednou komplikací, kterou odhalí následující příklad. Omezující podmínku $y = z \cdot x; x \in \langle -2;3 \rangle; y \in \langle -\infty;\infty \rangle; z \in \langle 1;1 \rangle$ lze chápat jako rovnici $y = \frac{1}{x}$ a je vidět, že žádný z intervalů není možné přímo zmenšit a to i přesto, že doména proměnné y obsahuje podmnožinu hodnot, kterých nemůže nabývat ($\langle -\frac{1}{2}; \frac{1}{3} \rangle$). Tento problém lze vyřešit pomocí funkce `Split` z algoritmu 1. O operacích, které trpí stejným problém se říká *intervalově konvexní*, zatímco operace jako sčítání jsou *intervalově konkávní* [8].

Samotný algoritmus pro propagaci omezující podmínky s násobením je poměrně dlouhý, a proto zde nebude uveden. Jeho délka je způsobena tím, že se rozvětvuje podle toho, které meze intervalů jsou větší či menší než nula. Rozvětvení je uvedeno v tabulce v článku [9].

V několika jednoduchých krocích (a to pouze s využitím pravidel intervalové aritmetiky) se tak povedlo zúžit domény všech proměnných a vztahy mezi proměnnými přitom stále platí. Existuje důkaz (viz [8]), že je zbytečné snažit se volat zužovací funkci na omezující podmínku vícekrát okamžitě po sobě, domény se vždy maximálně zmenší již při prvním spuštění. V případě, kdy se proměnná vyskytuje ve více než jedné omezující podmínce ale může nastat situace, kdy následkem zúžení domény podle jedné podmínky dojde k rozšíření domény vzhledem k jiné podmínce a tak může být potřeba znovu zúžit tuto doménu podle již použité podmínky.

1.5 Konzistenční techniky a algoritmy

Konzistenční techniky slouží k rychlému odebrání nekonzistentních (neplatných) hodnot z domén proměnných a jejich cílem je dostat CSP do *konzistentního stavu*, kdy domény neobsahují žádné nekonzistentní hodnoty. V této kapitole je popsáno několik základních technik využitelných především pro nenumerické CSP a je ukázán přechod od nich na techniku zvanou *hull consistency*, používanou pro NCSP, která je pro tuto práci nejdůležitější, protože právě ona je využita v praktické části.

1.5.1 Node consistency

Definice 6 ([2, s. 16])**.** Proměnná je *node konzistentní*, je-li každá hodnota z její domény konzistentní pro všechny unární omezující podmínky.

Nejjednodušší konzistenční technikou je *node consistency* [10] (viz algoritmus č. 2), pomocí které se dá velmi snadno (alespoň u diskrétních domén) dosáhnout konzistence tím, že program prochází hodnoty z domény jednu po druhé a pro každou hodnotu ověřuje, zda má pro všechny unární omezující podmínky smysl. Pokud nemá, je odebrána.

Algoritmus 2 Algoritmus NC

In: CSP problém $P = (V, D, C)$ k vyřešení.
Out: CSP bez nekonzistentních hodnot (podle node consistency) v doménách.

```
1:  procedure NC3(V, D, C)
2:      for každou proměnnou v ∈ V do
3:          for každou hodnotu x v doméně proměnné v do
4:              for každou omezující podmínku c ∈ C do
5:                  if x nesplňuje podmínku c then
6:                      Odstraň x z domény.
7:                  end if
8:              end for
9:          end for
10:     end for
11: end procedure
```

1.5.2 Arc consistency

Definice 7 ([2, s. 16])**.** Proměnné x a y jsou *arc konzistentní*, existuje-li pro každou hodnotu z domény proměnné x hodnota z domény proměnné y tak, že všechny omezující podmínky mezi těmito dvěma proměnnými platí.

Diskrétní CSP problém se dá představit jako orientovaný graf (omezující podmínka je relace mezi proměnnými). Název této konzistenční techniky pochází z jednoho anglických názvů pro hranu („arc“).

Pro dosažení tohoto typu konzistence existuje několik algoritmů, z nichž poměrně jednoduchý a zároveň efektivní je algoritmus AC3 [10]. AC3 postupuje tak, že pro každou dvojici proměnných (x, y) odstraní z domény proměnné x všechny hodnoty nekonzistentní s hodnotami z domény proměnné y. Algoritmus si udržuje seznam hran ke zkontrolování a při odebrání alespoň jedné hodnoty z domény proměnné x jsou do zmíněného seznamu přidány všechny hrany směřující do uzlu reprezentujícího proměnnou x.

1.5.3 Hull consistency

Jak node consistency, tak i arc consistency, se hodí spíše pro CSP s diskrétními doménami, protože vždy pracují s konkrétními hodnotami. V případě spojitých domén je výhodnější nalézt obal, který zaručeně pokrývá všechny konzistentní hodnoty, i za cenu toho, že některé hodnoty v něm budou nadbytečné. Takto funguje právě hull consistency.

Definice 8 ([2, s. 572])**.** Omezující podmínka je *hull konzistentní*, pokud kartézský součin domény proměnných z dané podmínky tvoří intervalový obal všech řešení splňující danou podmínku. CSP problém P je hull konzistentní, jsou-li všechny jeho omezující podmínky hull konzistentní.

Jsou známy dva základní algoritmy k dosažení hull consistency - *HC3* a *HC4*.

Tyto algoritmy jsou vhodné především v případě, kdy se jedna proměnná nevyskytuje v mnoha podmínkách najednou. Pokud tomu tak je, může nastat situace podobná dependency problemu popsanému v kapitole 1.3. Existují jiné konzistenční techniky (např. *box consistency*), které si s tímto problémem poradí [6].

Tyto techniky se dají dále kombinovat například s *branch-and-prune* algoritmy, které umožňují rekurzivně rozpůlit nalezený box a tyto poloviční boxy dále zmenšovat pomocí HC algoritmů. Je tak možné získat několik menších boxů, které budou zahrnovat méně nadbytečných hodnot.

1.5.3.1 HC3

Algoritmus HC3 byl navržen jako první z algoritmů pro propagaci intervalů v roce 1997 v článku [11], autoři ho tehdy nazvali jednoduše jako „A narrowing algorithm“. Název HC3 byl vytvořen později v článku [12] (tento název byl vybrán pro podobnost s algoritmem AC3 používaným pro dosažení arc consistency), kde byl uveden i pokročilejší algoritmus HC4, který na rozdíl od HC3 nevyžaduje vstupní podmínky rozložené do primitivního tvaru.

Původní algoritmus HC3 navržený v článku [11] je uveden v pseudokódu č. 3, tvar algoritmu reálně použitý v této práci je uveden v implementační části.

Algoritmus 3 Algoritmus HC3

In: CSP problém $P = (V, D, C)$ k vyřešení.
Out: Informace o hull nekonzistenci, nebo zmenšený problém P.

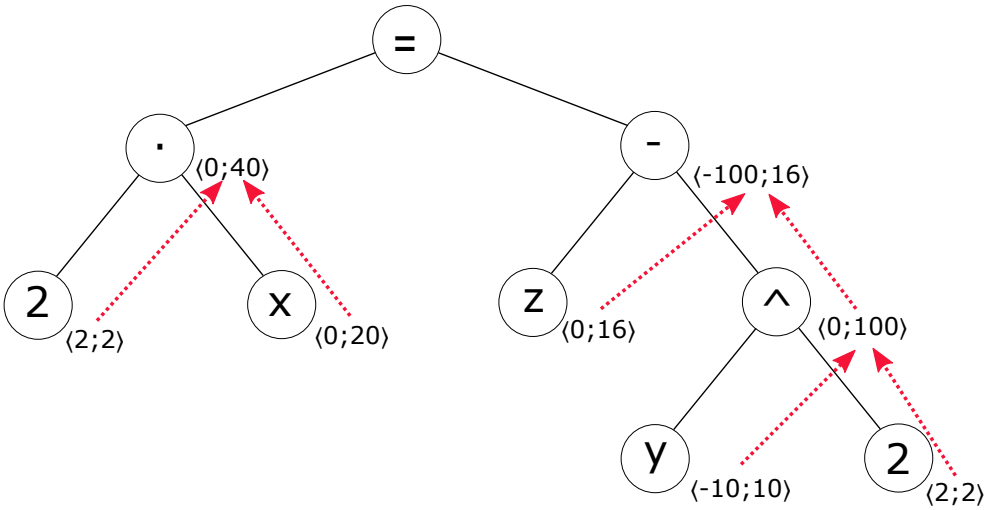
```
1: procedure HC3(V, D, C)
2:     Q ← C
3:     while Q ≠ ∅ do
4:         Vyber podmínku c z Q.
5:         Q ← Q \ {c}
6:         {x_1, ..., x_n} ← proměnné z V vyskytující se v c.
7:         for každé x_i ∈ {x_1, ..., x_n} do
8:             D'_i ← zredukuj doménu D_i proměnné x_i podle podmínky c.
9:             if D'_i = ∅ then
10:                return CSP je nekonzistentní.
11:            end if
12:            if D'_i ≠ D_i then
13:                D_i ← D'_i
14:                Přidej do Q podmínku c a všechny další podmínky z C, které
   obsahují proměnnou x_i.
15:            end if
16:        end for
17:    end while
18:    return P
19: end procedure
```

1.5.3.2 HC4

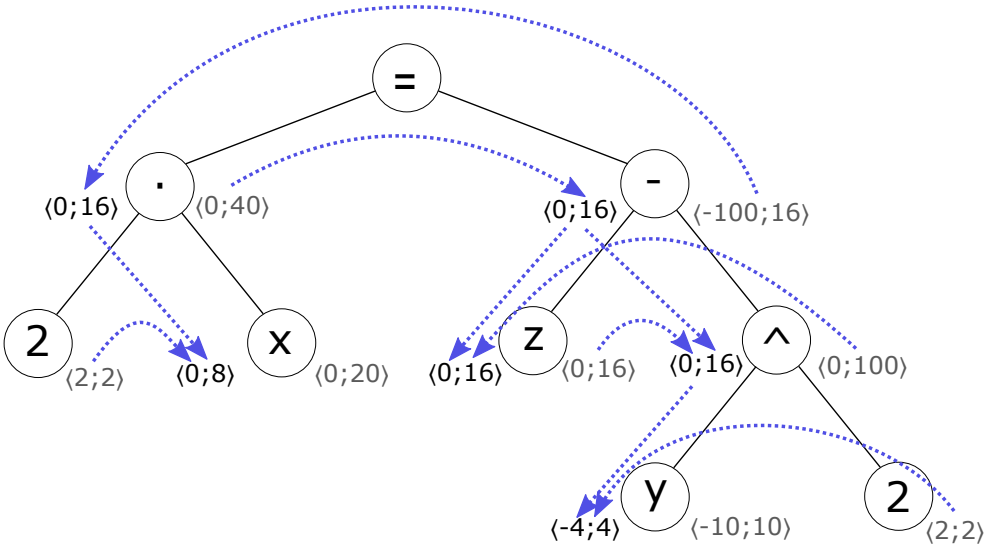
Výhoda algoritmu HC4 oproti HC3 spočívá ve faktu, že umí přímo pracovat s komplexními podmínkami a není tak potřeba jejich rozklad, což ve výsledku zkracuje i dobu běhu algoritmu [12]. Nevýhodou je ovšem mnohem složitější implementace, která se skládá ze dvou fází: dopředného průchodu (*ForwardEvaluation*) a zpětného průchodu (*BackwardEvaluation*).

Celý algoritmus funguje tak, že prochází syntaktický strom výrazu a při dopředném průchodu od listů ke kořeni odhaduje domény rodičovských uzlů podle domén synů (s pomocí intervalové aritmetiky). Průběh pro omezující podmínku $2x = z - y^2$ s doménami $D_x = \langle 0; 20\rangle$, $D_y = \langle -10; 10\rangle$ a $D_z = \langle 0; 16\rangle$ ukazuje obrázek č. 1.2.

Při zpětném průchodu (obrázek č. 1.3) od kořene k listům se pak zpřesňuje výsledná doména - podle rodičovského uzlu se zúží domény synů. Algoritmus u kořene provede stromu z předchozího příkladu provede průnik (kvůli rovnosti v kořeni stromu) domén obou synů ($\langle 0; 40\rangle \cap \langle -100; 16\rangle$), které byly vypočteny při dopředném průchodu. Tímto je kořen zpracován a zpětný průchod se může spustit rekurzivně pro oba podstromy. V případě levého podstromu je novým kořenem uzel s operací násobení a doménou $\langle 0; 16\rangle$. Nyní je již snadné



Obrázek 1.2: Dopředný průchod u algoritmu HC4



Obrázek 1.3: Zpětný průchod u algoritmu HC4

zúžit doménu proměnné x na její finální hodnotu, $\langle 0; 8\rangle$. Stejným postupem se podaří zúžit i domény proměnných y a z na $\langle -4; 4\rangle$, resp. $\langle 0; 16\rangle$.

1.6 Heuristiky

Heuristiky při řešení problémů s omezujícími podmínkami pomáhají rozhodnout, podle jaké omezující podmínky zužovat domény a v jakém pořadí je zužovat. Jsou stále předmětem výzkumu a neexistuje žádný důkaz, který by prokazoval nějakou heuristiku lepší, než všechny ostatní.

1.6.1 Typy heuristik

Heuristiky pro řešení NCSP se dají rozdělit do následujících dvou skupin [13]:

- Heuristiky rozhodující podle statických (neměnných) vlastností proměnných, například počtu jejich výskytů, nebo aritmetických operací, kterých se účastní.
- Heuristiky rozhodující podle domén proměnných - zúžit širokou doménu může být výhodnější než dále zužovat malou doménu. Tyto vlastnosti jsou dynamické, tj. mění se během průběhu algoritmu.

Jedním z cílů této práce je otestovat vliv heuristik na efektivitu řešení různých problémů a v následujících podkapitolách je uvedeno několik nápadů na možné heuristiky (část jich je převzata z [13]), z nichž některé jsou využity v praktické části. Seznam není vyčerpávající, jistě jich lze vymyslet o mnoho více. Konkrétní heuristiky využité v této práci jsou uvedeny v praktické části v kapitole 3.4. Všechny heuristiky uvedené v této práci se vztahují k algoritmu HC3.

1.6.1.1 Statické parametry

Heuristiky rozhodující podle statických parametrů se mohou řídit podle:

- dominance či nedominance proměnné,
- vstupní množiny podmínek:

  počtu podmínek, ve kterých se proměnná vyskytuje,

  počtu dominantních proměnných v podmínce,

  operace použité v omezující podmínce (sčítání/násobení).

1.6.1.2 Dynamické parametry

Heuristiky využívající dynamické parametry mohou brát ohled na:

- intervaly:

  jejich velikost,

  zda pokrývají kladné i záporné hodnoty,

- zda se omezující podmínka již použila ke zredukování domény.

U dynamických parametrů se může vyplatit ukládat posloupnost změn hodnot tak, jak se měnily během běhu programu. V praxi se ale při rozhodování nepoužívá celá posloupnost, jen její část. Následující parametry pracují s historií změn:

- poměr velikosti domény či problému k původní velikosti,
- poměr aktuální velikosti k velikosti v minulém průběhu smyčky,
- číslo průběhu smyčky, kdy se naposledy používala podmínka ke zredukování domény.

1.7 Charakteristiky problémových instancí pro algoritmus HC3

Problémy k řešení mají některé vlastnosti, které přispívají k nižší efektivitě při řešení. Mohou jimi být například:

- výskyt jedné proměnné ve více omezujících podmínkách,
- počáteční velikosti domén,
- počet proměnných,
- počet omezujících podmínek.

Kapitola 2

Cíl práce

V první části byla vysvětlena veškerá potřebná terminologie a je tedy nyní možné přesněji popsat cíle práce. Hlavním cílem práce je vytvoření jednoduchého řešiče pro numerické CSP problémy ve funkcionálním jazyce `F#`. Program komunikuje s uživatelem přes příkazovou řádku a problémy řeší s využitím branch-and-prune algoritmu a algoritmu HC3.

Dalším cílem práce je nalezení parametrů algoritmu HC3, které ovlivňují jeho účinnost a rychlost. V řešiči jsou implementovány heuristiky (nalezené v literatuře či vymyšlené), které mají za účel pomoci rozhodnout, v jakém pořadí je nejlepší domény zužovat, atp. Posledním cílem práce je experimentálně porovnat účinnost jednotlivých heuristik na sadě benchmarkových úloh a zjištěné výsledky vyhodnotit.

Kapitola 3

Implementace

Jako součást bakalářské práce byl vytvořen řešič numerických CSP s názvem *HullSolver*, který jako jediný volně dostupný řešič umožňuje porovnávat efektivitu použitých heuristik. Zdrojové kódy jsou dostupné pod licencí GNU GPL na serveru GitHub[2], případně jsou k nalezení i na přiloženém CD. Manuál popisující kompilaci, spuštění a použití programu je uveden na konci práce v příloze B.

Tato část práce nejprve popisuje použité technologie a algoritmy v programu HullSolver, následně se věnuje popisu architektury programu a na závěr uvádí několik existujících řešení, které řeší podobné problémy.

3.1 Použité technologie

Jako jazyk pro implementaci byl zvolen funkcionální jazyk `F#` (čteno jako F Sharp) běžící na platformě .NET, který se v posledních letech stal velmi populární pro vědecké využití. Tento jazyk byl poprvé uveden v roce 2005 společností Microsoft, v roce 2013 byla ale založena nezisková společnost `F#` Software Foundation, která má na starosti jeho další vývoj. Od té doby se z jazyka stal open-source a díky projektu Mono je možné aplikace napsané v `F#` spouštět i na jiných platformách, než pouze na Microsoft Windows.

3.2 Algoritmy

S vedoucím práce bylo dohodnuto, že řešič bude podporovat omezující podmínky s operacemi sčítání, odčítání a násobení.

Řešení vstupních problémů probíhá s využitím algoritmu HC3 s malou úpravou pro pestřejší možnosti využití heuristik. Zatímco originální algoritmus vždy vybere z množiny podmínek jednu náhodnou a zredukuje domény všech proměnných v ní obsažených, upravený algoritmus na vstupu přijímá množinu

[2] `https://github.com/jakubkottnauer/hull-solver`

všech dvojic (c, x) (kde c je omezující podmínka definovaná nad proměnnou x) a z této množiny heuristicky vybírá jednu dvojici. Následně podle podmínky c zúží doménu proměnné x. Omezující podmínka c se tedy na vstupu objeví tolikrát, nad kolika proměnnými je nadefinována - v případě programu HullSolver vždy právě třikrát, neboť program podporuje pouze ternární omezující podmínky. Algoritmus 4 uvádí pseudokód takto upraveného algoritmu HC3.

Oba algoritmy nalézající se v této kapitole přijímají seznam dvojic omezujících podmínek a proměnných z CSP, a množinu domén proměnných z CSP. Tyto parametry spojené dohromady odpovídají CSP problému, jen je u těchto algoritmů praktičtější definovat CSP v této alternativní formě.

Algoritmus 4 Upravený algoritmus HC3

In: Seznam P dvojic (c, x), kde c je omezující podmínka definovaná nad proměnnou x, a množina domén D z CSP.
Out: Informace o nekonzistenci, nebo zmenšený problém P.

1: **procedure** HC3(P, D)
2: $Q \leftarrow P$
3: **while** $Q \neq \emptyset$ **do**
4: Pomocí heuristiky vyber z Q pár $p = (c, x)$.
5: $Q \leftarrow Q \setminus \{p\}$
6: $D'_x \leftarrow$ Zredukuj doménu $D_x \in D$ proměnné $x \in p$ podle c.
7: **if** $D'_x = \emptyset$ **then**
8: **return** CSP je nekonzistentní.
9: **end if**
10: **if** $D_x \neq D'_x$ **then**
11: $D_x \leftarrow D'_x$
12: Přidej do Q pár p a všechny další dvojice z P, které obsahují x.
13: **end if**
14: **end while**
15: **return** P
16: **end procedure**

Na řádku č. 10 v algoritmu 4 jsou pro rovnost porovnávány dvě domény. Vzhledem k tomu, že jsou zde domény tvořeny reálnými intervaly, nemusí být jejich meze celá čísla a v počítači tak nebudou uložena přesně (v programu je pro reprezentaci mezí využit 64-bitový typ `double`). Rovnost je v programu ověřována nepřesně, jak ukazuje následující kód (jako `ZERO_EPSILON` je použita hodnota 10^{-30}):

```
member this.EqualTo y =
  abs(this.a - y.a) < ZERO_EPSILON
  && abs(this.b - y.b) < ZERO_EPSILON
```

Nad algoritmem HC3 je postaven jednoduchý branch-and-prune algoritmus (viz Algoritmus 5), který zároveň tvoří hlavní funkci řešiče. Tento algoritmus přibližně kopíruje strukturu algoritmu 1 (str. 8). Při spuštění je mu předána množina dvojic omezujících podmínek a proměnných v nich obsažených.

Algoritmus 5 Upravený algoritmus Solve

In: Seznam P dvojic (c, x), kde c je omezující podmínka definovaná nad proměnnou x, a množina domén D z CSP.
Out: Seznam nalezených boxů obsahující všechna řešení.

```
1:  procedure SOLVE(P, D)
2:      if Problém není dostatečně malý then
3:          P' ← HC3(P)
4:          (P1, P2) ← rozděl box tvořený doménami dominantních proměnných z D na poloviny.
5:          Solve(P1)
6:          Solve(P2)
7:      else
8:          Vypiš/ulož box tvořený proměnnými z P jako jeden z výsledků.
9:      end if
10: end procedure
```

V použité variantě algoritmu `Solve` je kontrolováno, zda je problém již dostatečně zmenšen (řádek č. 2). V programu se konkrétně kontroluje, jestli se již podařilo domény všech dominantních proměnných dostatečně zúžit vzhledem k původní velikosti. Jako koeficient pro porovnání se využívá parametr `eps`, který je specifikován uživatelem při spuštění řešiče přepínačem `-p` (viz manuál).

```
member this.AllFraction eps =
  dominantVariables
  |> List.forall(fun v ->
    (v.Domain.Length / v.OriginalDomain.Length) < eps)
```

Druhým prvkem algoritmu `Solve`, jehož implementace je potřeba upřesnit, je rozpůlení boxu na řádku č. 4. Box, jak je popsáno v kapitole 1.2, ohraničuje řešení a je tvořen doménami proměnných z CSP. Algoritmus box rozpůlí tím, že rozpůlí doménu jedné dominantní proměnné. Ve vstupním souboru uživatel označí dominantní proměnné a algoritmus je půlí metodou round-robin ve stejném pořadí, jako jsou zapsány ve vstupním souboru (při rozpůlení poslední dominantní proměnné přeskočí opět na první).

3.2.1 Efektivita algoritmů

Nejslabším místem použité varianty algoritmu HC3 je řádek č. 4, protože výpočet složité heuristiky může algoritmus velmi zpomalit. Druhým potenciálně rizikovým místem je řádek č. 6, protože závisí na konkrétní implementaci zužovací funkce. V aktuální verzi programu HullSolver by ale nemělo docházet k problémům díky tomu, že podporuje pouze sčítání a násobení v omezujících podmínkách, a ty se dají zužovat podle jednoduchých pravidel (navíc je přítomna optimalizace pro zužování podle druhé mocniny).

U branch-and-prune algoritmu `Solve` jsou rovněž dvě potenciálně riziková místa, a to řádky č. 2 a 4. Konkétní implementace popsána výše by ale neměla způsobovat žádné výkonnostní problémy.

3.3 Architektura

Celý program se skládá ze čtyř souborů - `DomainTypes.fs`, `Heuristics.fs`, `Solver.fs`, `Program.fs`. Při kompilování programu napsaného v `F#` záleží na pořadí souborů a v tomto případě jsou soubory zpracovávany v tomto pořadí. Braním ohledu na pořadí souborů kompilátor zabraňuje tvorbě kruhových závislostí mezi částmi programu, protože soubor později ve frontě může využívat pouze typy z již zpracovaného souboru. Kód se tak stává modulárnější a přehlednější, protože přirozeně dochází ke striktnímu oddělení „high-level“ částí programů od „low-level“ částí.

Dalším důsledkem je odlišná organizace kódu v podobných jazycích jako `F#` od klasických procedurálních či objektově orientovaných jazyků. V `F#` se typicky nevytváří samostatný soubor pro každý typ (typ přibližně odpovídá třídě z objektově orientovaného programování), ale všechny typy se shlukují do jednoho modulu často nazývaného `DomainTypes`. V případě HullSolveru jsou doménovými typy `Interval`, `Variable`, `Constraint`, `Problem` a `Heuristic`, z nichž poslední se pro přehlednost nachází v samostatném souboru `Heuristics.fs`.

Zatímco první dva soubory obsahují především deklarace typů a definují jejich funkce, soubor `Solver.fs` tvoří hlavní výpočetní jádro programu - zde jsou implementace branch-and-prune algoritmu a algoritmu HC3. Posledním souborem je `Program.fs`, který se už stará jen o ošetření a zpracování vstupu a spouští řešící algoritmy.

3.4 Použité heuristiky

Níže jsou uvedeny heuristiky, které jsou využité v programu HullSolver v algoritmu č. 4 na řádce 4. Každá heuristika má svůj krátký název, kterým je identifikována ve výsledcích experimentů. Heuristiky převzaté z literatury jsou označeny odkazem na použitý zdroj.

Všechny heuristiky se chovají stejně v tom, že v případě existence více než jedné dvojice splňující jejich kritéria vždy vyberou dvojici nacházející se v seznamu Q jako první (prohledávání seznamu probíhá zleva doprava).

- *rand* je „heuristika“ vybírající pseudonáhodnou dvojici,
- *fifo* simuluje FIFO frontu - vybírá dvojici, která je ve frontě nejdéle,
- *dom/nondom-first* [13] se snaží nejprve redukovat domény dominantních, resp. nedominantních, proměnných - proměnné vyskytující se v původních nerozložených podmínkách,
- *small/large-int-first* [13] se snaží dále zúžit nejužší, resp. nejširší domény,
- *shrunk-most/least-first* [13] vybírá nejprve proměnné, jejichž domény byly nejvíce, resp. nejméně, od začátku běhu algoritmu zmenšeny (vypočítává se kvocient z původní a aktuální velikosti domény),
- *max-right-cand* [13] vybírá proměnnou s nejvyšší pravou mezí domény. Např. proměnná s doménou $(1; 6)$ může být vybrána dříve než proměnná s doménou $(1; 3)$, protože má vyšší pravou mez,
- *min-right-cand* [13] vybírá proměnnou s nejmenší pravou mezí domény,
- *fail-first* vybírá proměnnou, která je přítomna v největším počtu omezujících podmínek,
- *prefer-add/mult* vybírá omezující podmínku s operací sčítání, resp. násobení.

3.5 Výstup programu a indikátory

V průběhu řešení problémů je sledováno několik tzv. indikátorů. To jsou různé zajímavé metriky, pomocí kterých se dá následně porovnat efektivita heuristik. Sledovány jsou následující indikátory:

- doba běhu,
- počet rozpůlení řešení,
- počet zúžení intervalů (kolikrát byla zavolána funkce pro zúžení domény podle podmínky),
- poměr objemu řešení k objemu vstupního CSP.

Doba běhu je závislá na konkrétním hardware a může se měnit během jednotlivými spuštěními programu, pro hrubé porovnání heuristik ale poslouží dobře.

3.6 Možná vylepšení programu

Program v této době neumí sám dekomponovat omezující podmínky v komplexním tvaru a je proto nutné je zadávat v primitivním tvaru. HullSolver rovněž neprovádí detekci redundantních podmínek.

Dalším vylepšením by mohlo být přidání grafického rozhraní (vykreslování grafů s řešením).

3.7 Existující řešení

Vzhledem k obecnosti termínu CSP existuje velké množství aplikací řešících problémy s omezujícími podmínkami - lze nalézt nespočet řešičů sudoku i řešičů SAT problému. Vzniklo také několik obecných řešičů, například *Gecode*[3] a *Microsoft Solver Foundation*[4].

Řešení NCSP problémů pomocí propagace intervalů je již poměrně specifická oblast a tak těchto řešičů není mnoho. Na službě GitHub patří mezi nejznámější řešiče projekt *JaCoP*[5], dále existují například *IASolver*[6], *RSolver*[7] a *RealPaver*[8]. První dva projekty jsou napsány v Javě, třetí v jazyce OCaml a čtvrtý v C. Žádný z nich ale není dále vyvíjen.

[3] http://www.gecode.org
[4] https://msdn.microsoft.com/en-us/library/ff524509(v=vs.93).aspx
[5] `https://github.com/radsz/jacop`
[6] `http://www.cs.brandeis.edu/~tim/Applets/IAsolver.html`
[7] `http://rsolver.sourceforge.net`
[8] `https://github.com/lcgutierrez/Realpaver-0_4-Windows`

Kapitola 4

Výpočetní experimenty

V této části práce jsou provedeny samotné výpočetní experimenty pomocí programu HullSolver. Experimenty byly provedeny na sadě různě komplikovaných benchmarkových problémů, z nichž část byla vymyšlena pro potřeby otestování implementace a část byla převzata z databáze polynomiálních problémů dostupné na adrese `http://homepages.math.uic.edu/~jan/demo.html`.

Experimenty byly provedeny tak, že pro každou heuristiku uvedenou v 3.4 byl spuštěn test se všemi dostupnými testovacími soubory a výsledky byly zaznamenány do tabulky. Použitá hodnota parametru `eps` z kapitoly 3.2 byla 0,001.

4.1 Testovací prostředí

Veškeré experimenty s programem HullSolver byly provedeny na počítači s operačním systémem MS Windows 10 Pro 64-bit v běhovém prostředí .NET 4.6.1 v následující hardwarové konfiguraci:

- Intel Core i5-4570 3,2 GHz (4 jádra)
- 8 GB RAM DDR3 1600 MHz
- nVidia GeForce GTX970

4.2 Testovací úlohy

Následuje seznam testovacích úloh s krátkým popisem a údajem o počtu primitivních podmínek. Úlohy převzaté z databáze zmíněné výše jsou označeny hvězdičkou (*):

- *conform1** - soustava tří polynomiálních rovnic stupně 3 o třech neznámých (24 podmínek),

- *mickey** - soustava dvou polynomiálních rovnic stupně 2 o dvou neznámých (6 podmínek),
- *minus* - testovací rovnice pro ověření správnosti algoritmu $10(x - x + x - x + x - x + x) = 100$ (7 podmínek),
- *polyn1* - rovnice $x^4 - 20x^2 + 64 = 0$ (4 podmínky),
- *polyn2* - rovnice $x^3 - 2x = 0$ (4 podmínky),
- *quadfor2** - soustava čtyř rovnic stupně 3 o čtyřech neznámých (14 podmínek),
- *solotarev** - soustava čtyř rovnic stupně 2 o čtyřech neznámých (16 podmínek),
- *wright** - soustava pěti rovnic stupně 2 o pěti neznámých (26 podmínek).

Tabulky s výsledky jsou uvedeny v kapitole 4.3. V těchto tabulkách jsou uvedeny naměřené hodnoty pro indikátory z kapitoly 3.5, přičemž jsou kvůli omezené šířce stránky následovně zkráceny názvy některých sloupců:

- # půlení - počet rozpůlení řešení,
- # zúžení - celkový počet zúžení všech domén dohromady,
- poměr objemu - poměr objemu nejmenšího nalezeného boxu ku objemu původního CSP.

O kapitolu dále (4.4) jsou k nalezení grafy vykreslené na základě dat z tabulek. Tyto grafy jsou využity pro snazší porovnání heuristik.

4.3 Tabulky

problém	# půlení	# zúžení	poměr objemu	čas (s)
conform1	16	12441	6.777e-10	0.5688
mickey	30	227	5.722e-11	0.0462
minus	42	816	0.0009277	0.0554
polyn1	35	645	0.0007813	0.0458
polyn2	15	146	2.278e-38	0.0417
quadfor2	89	3772	6.248e-17	0.1432
solotarev	99	15162	1.489e-15	0.6056
wright	2286	96850	3.458e-19	4.8842

Tabulka 4.1: Výsledky pro heuristiku *rand*

problém	# půlení	# zúžení	poměr objemu	čas (s)
conform1	16	8185	6.777e-10	0.3547
mickey	30	224	5.722e-11	0.0443
minus	42	760	0.0009277	0.0483
polyn1	35	608	0.0007813	0.0463
polyn2	15	131	0.0007813	0.0382
quadfor2	89	3495	6.248e-17	0.1377
solotarev	95	11728	1.489e-15	0.5022
wright	2286	93533	3.458e-19	4.2201

Tabulka 4.2: Výsledky pro heuristiku *fifo*

problém	# půlení	# zúžení	poměr objemu	čas (s)
conform1	16	7995	6.777e-10	0.6239
mickey	30	240	5.722e-11	0.0433
minus	42	767	0.0009277	0.0538
polyn1	35	608	0.0007813	0.0467
polyn2	15	125	0.0007813	0.0390
quadfor2	89	3126	6.248e-17	0.1771
solotarev	95	9493	1.489e-15	0.6660
wright	2286	88059	3.458e-19	8.4097

Tabulka 4.3: Výsledky pro heuristiku *dom-first*

problém	# půlení	# zúžení	poměr objemu	čas (s)
conform1	16	10194	6.777e-10	0.7317
mickey	30	225	5.722e-11	0.0435
minus	42	800	0.0009277	0.0529
polyn1	35	606	0.0007813	0.0506
polyn2	15	128	0.0007813	0.0465
quadfor2	89	3552	6.248e-17	0.2243
solotarev	95	20800	1.489e-15	1.2040
wright	2286	108977	3.458e-19	10.4618

Tabulka 4.4: Výsledky pro heuristiku *nondom-first*

problém	# půlení	# zúžení	poměr objemu	čas (s)
conform1	16	23673	6.777e-10	1.5826
mickey	30	255	5.722e-11	0.0513
minus	42	826	0.0009277	0.0639
polyn1	35	646	0.0007813	0.0541
polyn2	15	127	0.0007813	0.0433
quadfor2	89	6672	6.248e-17	0.3859
solotarev	95	162967	1.489e-15	11.7988
wright	2286	98378	3.458e-19	11.3865

Tabulka 4.5: Výsledky pro heuristiku *max-right-cand*

problém	# půlení	# zúžení	poměr objemu	čas (s)
conform1	16	23306	6.777e-10	1.4782
mickey	30	217	5.722e-11	0.0481
minus	42	769	0.0009277	0.0668
polyn1	35	638	0.0007813	0.0525
polyn2	15	478	9.243e-35	0.0479
quadfor2	169	66938	6.248e-17	3.6092
solotarev	116	540970	3.669e-16	34.8202
wright	2286	96594	3.458e-19	9.6712

Tabulka 4.6: Výsledky pro heuristiku *min-right-cand*

problém	# půlení	# zúžení	poměr objemu	čas (s)
conform1	16	15528	6.777e-10	1.3386
mickey	30	260	5.722e-11	0.0758
minus	42	809	0.0009277	0.0838
polyn1	35	647	0.0007813	0.0563
polyn2	15	130	0.0007813	0.0488
quadfor2	169	7682	6.248e-17	0.5212
solotarev	95	136896	1.489e-15	9.3976
wright	2286	119420	3.458e-19	13.4664

Tabulka 4.7: Výsledky pro heuristiku *large-int-first*

problém	# půlení	# zúžení	poměr objemu	čas (s)
conform1	16	36655	6.777e-10	2.2830
mickey	30	215	5.722e-11	0.0435
minus	42	708	0.0009277	0.0559
polyn1	35	643	0.0007813	0.0486
polyn2	15	477	9.243e-35	0.0451
quadfor2	169	159	6.248e-17	0.0458
solotarev	116	118221	3.669e-16	8.2039
wright	2286	86619	3.458e-19	9.3518

Tabulka 4.8: Výsledky pro heuristiku *small-int-first*

problém	# půlení	# zúžení	poměr objemu	čas (s)
conform1	16	15528	6.777e-10	1.0341
mickey	30	253	5.722e-11	0.0457
minus	42	809	0.0009277	0.0560
polyn1	35	644	0.0007813	0.0495
polyn2	15	137	0.0007813	0.0399
quadfor2	169	7696	6.248e-17	0.4336
solotarev	95	95883	1.489e-15	6.5600
wright	2286	127848	3.458e-19	15.9222

Tabulka 4.9: Výsledky pro heuristiku *shrunk-most-first*

problém	# půlení	# zúžení	poměr objemu	čas (s)
conform1	16	36041	6.777e-10	3.0409
mickey	30	221	5.722e-11	0.0445
minus	42	708	0.0009277	0.0585
polyn1	35	630	0.0007813	0.0500
polyn2	15	167	2.278e-38	0.0406
quadfor2	89	33264	6.248e-17	1.6815
solotarev	116	361640	3.669e-16	23.1700
wright	2286	79055	3.458e-19	10.0826

Tabulka 4.10: Výsledky pro heuristiku *shrunk-least-first*

problém	# půlení	# zúžení	poměr objemu	čas (s)
conform1	16	8376	6.777e-10	3.1547
mickey	30	236	5.722e-11	0.0526
minus	42	767	0.0009277	0.0766
polyn1	35	611	0.0007813	0.0523
polyn2	15	125	0.0007813	0.0415
quadfor2	89	3065	6.248e-17	0.5714
solotarev	95	8208	1.489e-15	2.2684
wright	2286	89847	3.458e-19	51.7957

Tabulka 4.11: Výsledky pro heuristiku *fail-first*

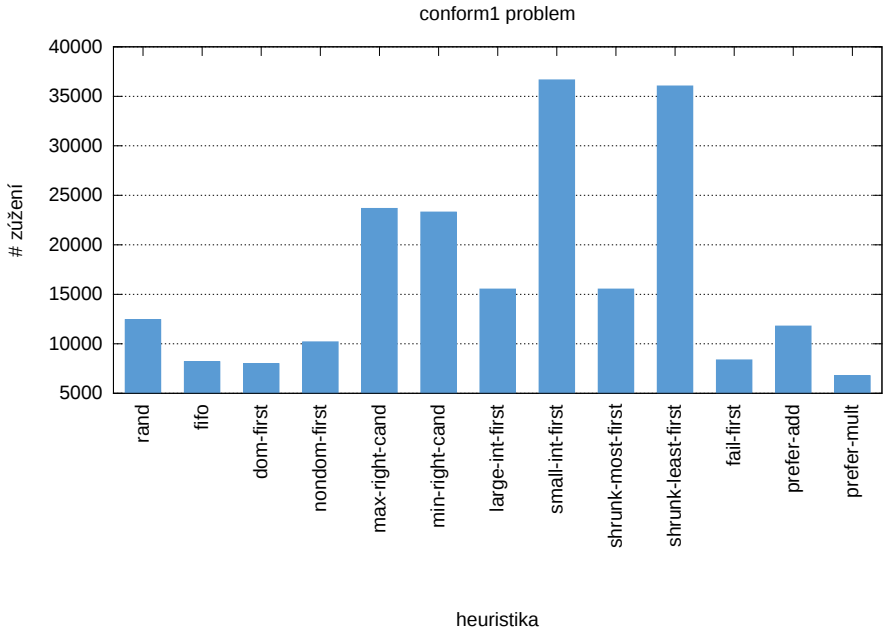
problém	# půlení	# zúžení	poměr objemu	čas (s)
conform1	16	11787	6.777e-10	0.6058
mickey	30	252	5.722e-11	0.0426
minus	42	760	0.0009277	0.0507
polyn1	35	639	0.0007813	0.0465
polyn2	15	130	0.0007813	0.0411
quadfor2	89	3502	6.248e-17	0.1508
solotarev	95	10434	1.489e-15	0.5193
wright	2286	141514	3.458e-19	6.9660

Tabulka 4.12: Výsledky pro heuristiku *prefer-add*

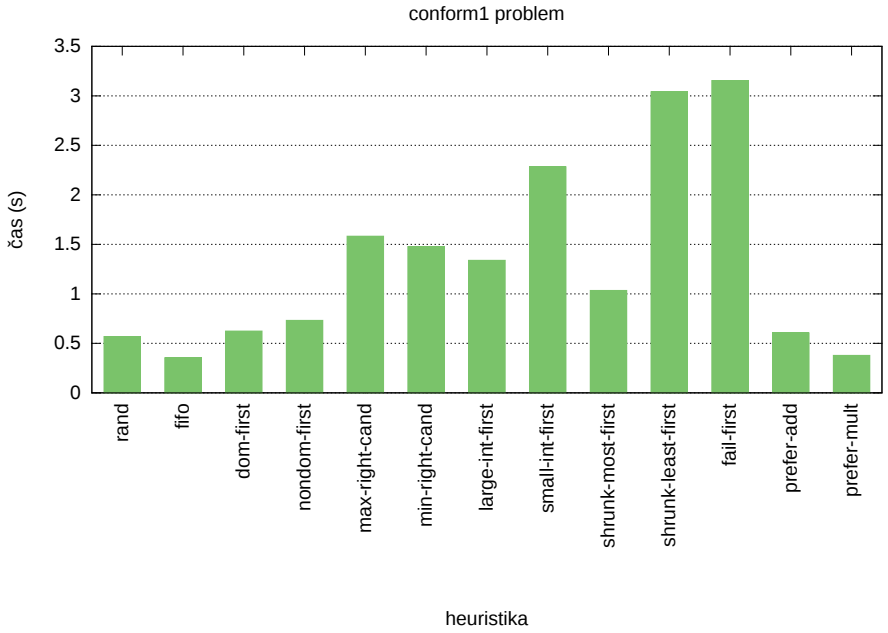
problém	# půlení	# zúžení	poměr objemu	čas (s)
conform1	16	6777	6.777e-10	0.3781
mickey	30	242	5.722e-11	0.0438
minus	42	760	0.0009277	0.0519
polyn1	35	605	0.0007813	0.0452
polyn2	15	127	0.0007813	0.0391
quadfor2	89	3519	6.248e-17	0.1817
solotarev	95	8526	1.489e-15	0.4209
wright	2286	86273	3.458e-19	3.9737

Tabulka 4.13: Výsledky pro heuristiku *prefer-mult*

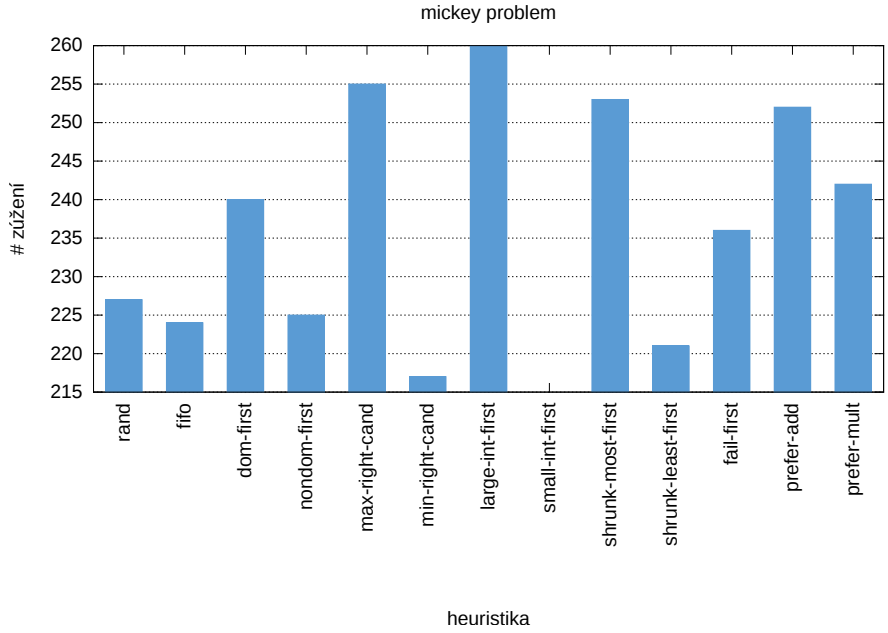
4.4 Grafy



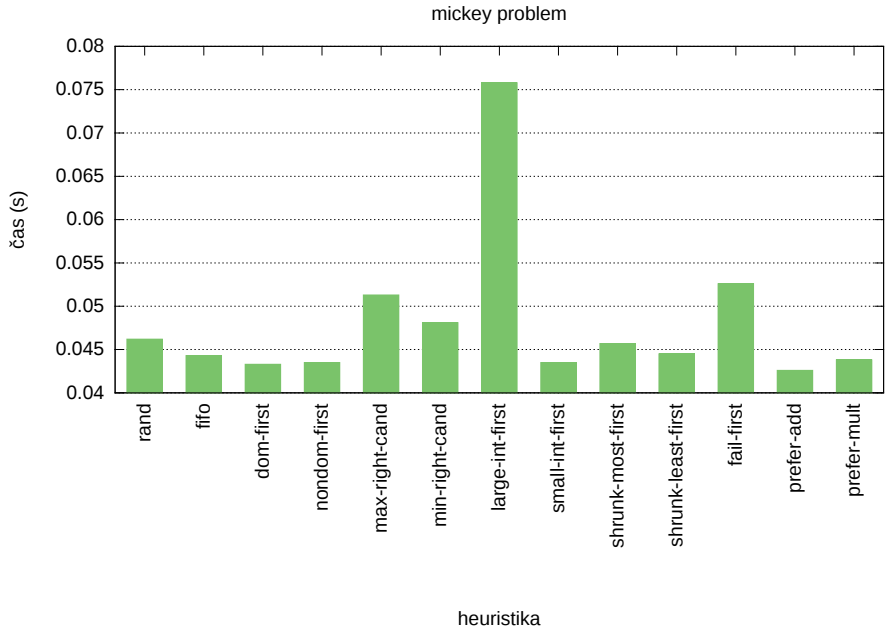
Obrázek 4.1: Porovnání heuristik pro problém *conform1* podle počtu zúžení



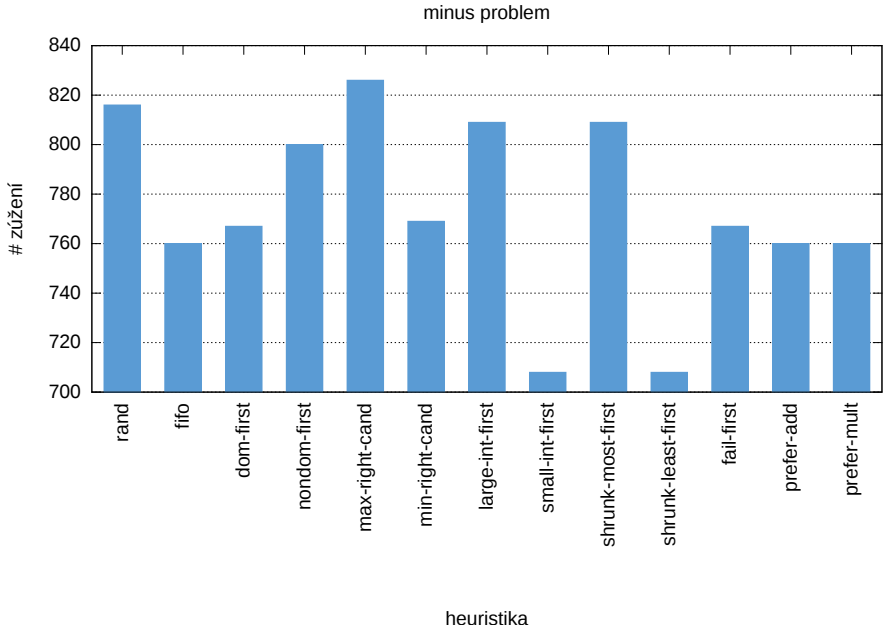
Obrázek 4.2: Porovnání heuristik pro problém *conform1* podle času



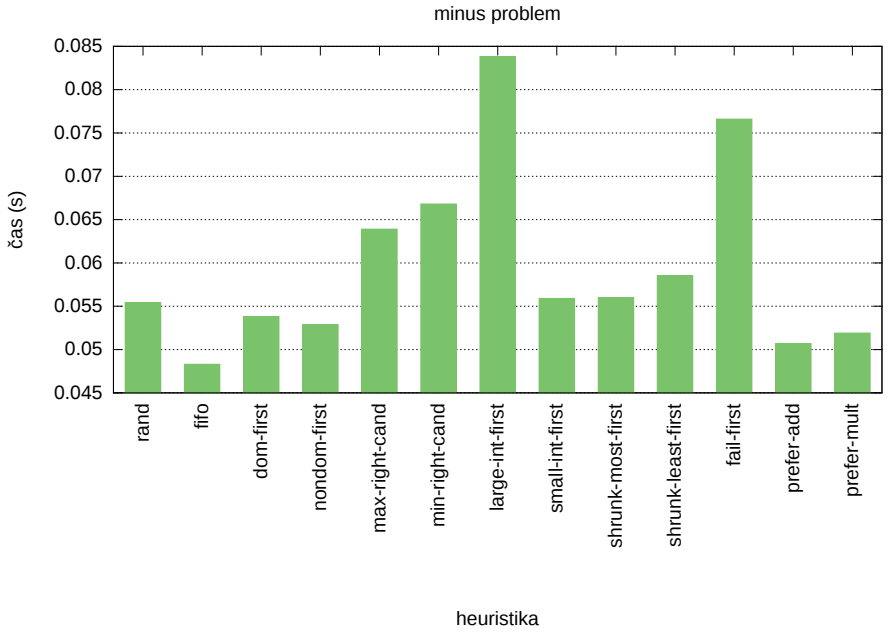
Obrázek 4.3: Porovnání heuristik pro problém *mickey* podle počtu zúžení



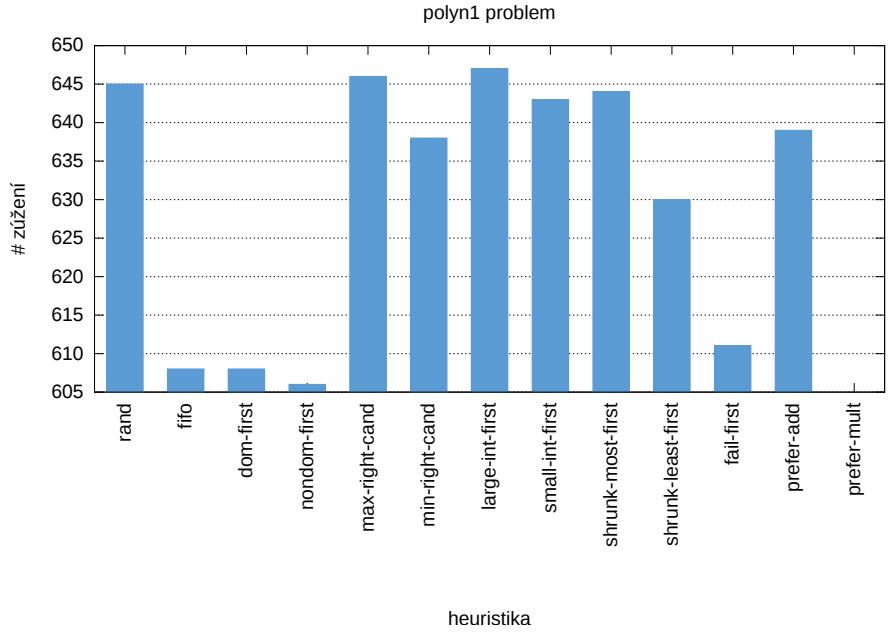
Obrázek 4.4: Porovnání heuristik pro problém *mickey* podle času



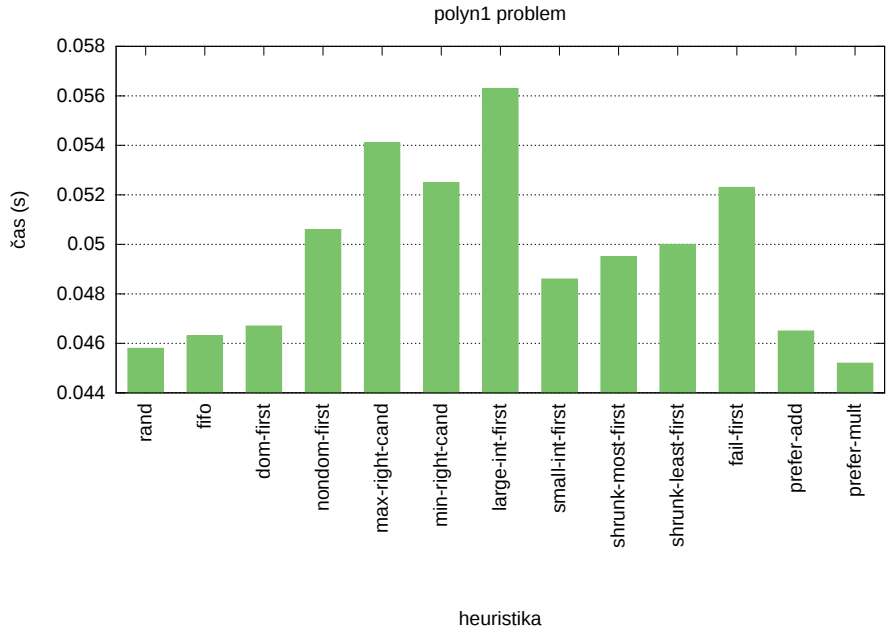
Obrázek 4.5: Porovnání heuristik pro problém *minus* podle počtu zúžení



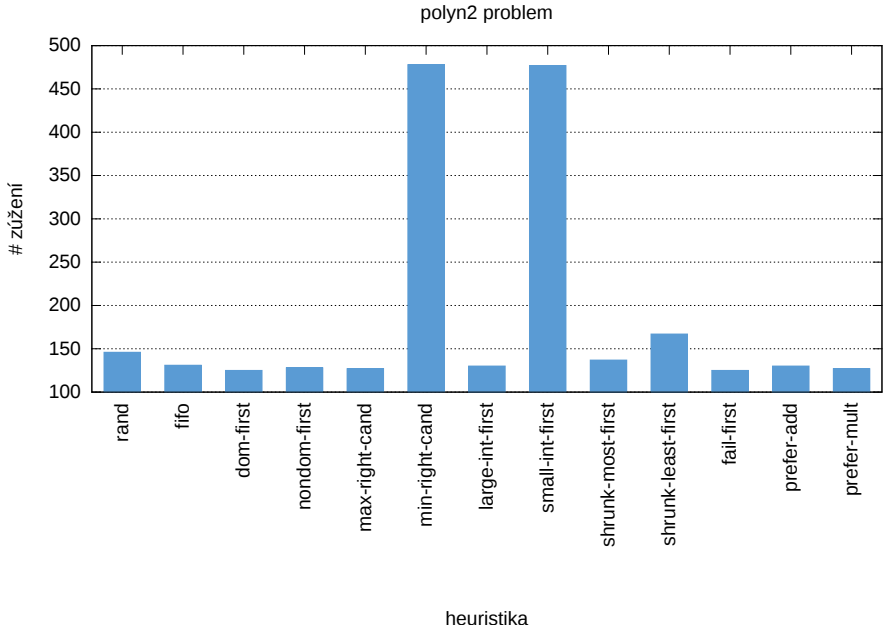
Obrázek 4.6: Porovnání heuristik pro problém *minus* podle času



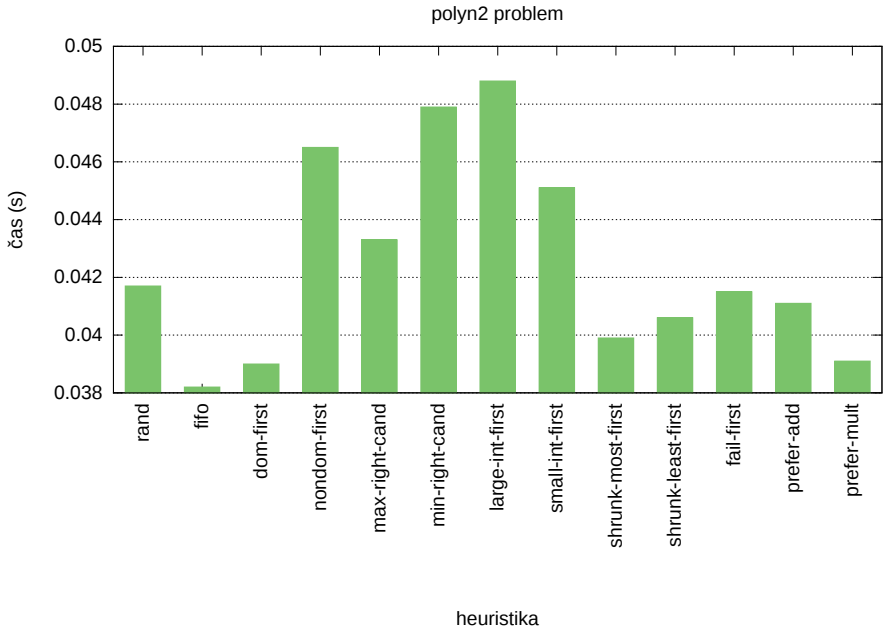
Obrázek 4.7: Porovnání heuristik pro problém *polyn1* podle počtu zúžení



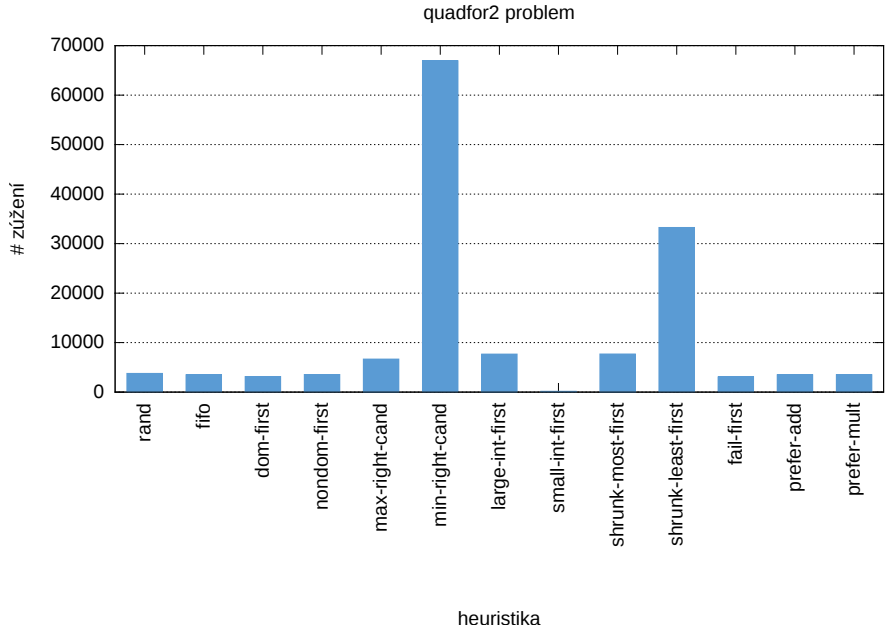
Obrázek 4.8: Porovnání heuristik pro problém *polyn1* podle času



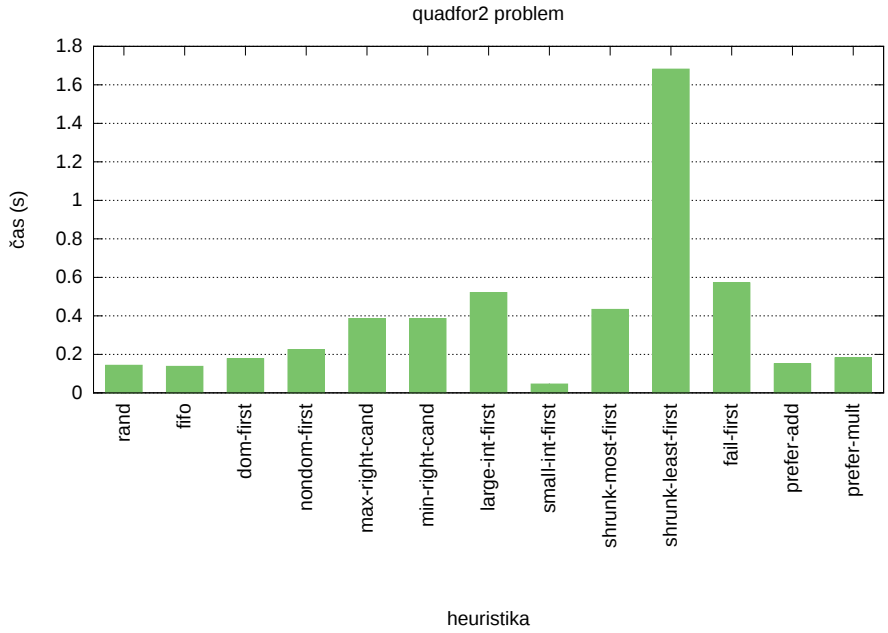
Obrázek 4.9: Porovnání heuristik pro problém *polyn2* podle počtu zúžení



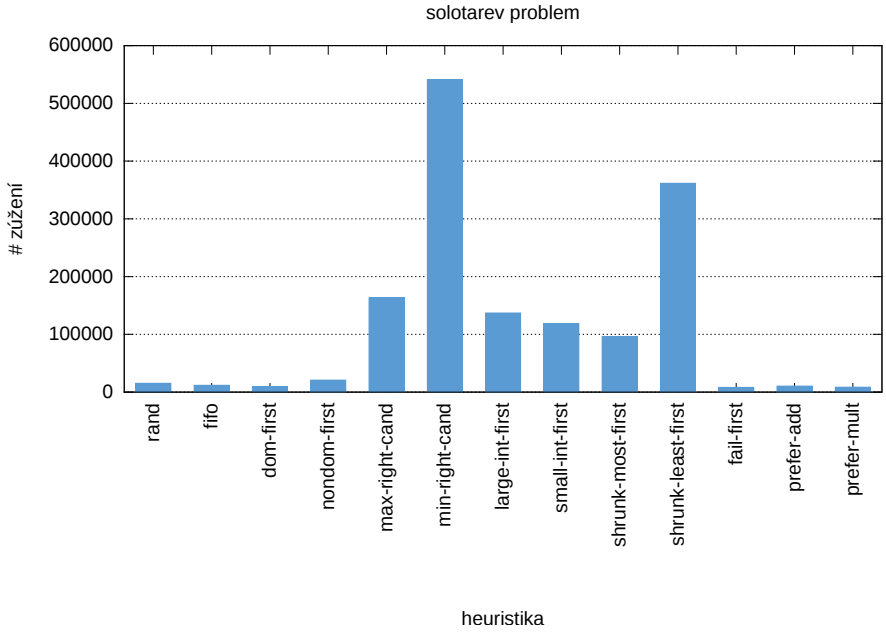
Obrázek 4.10: Porovnání heuristik pro problém *polyn2* podle času



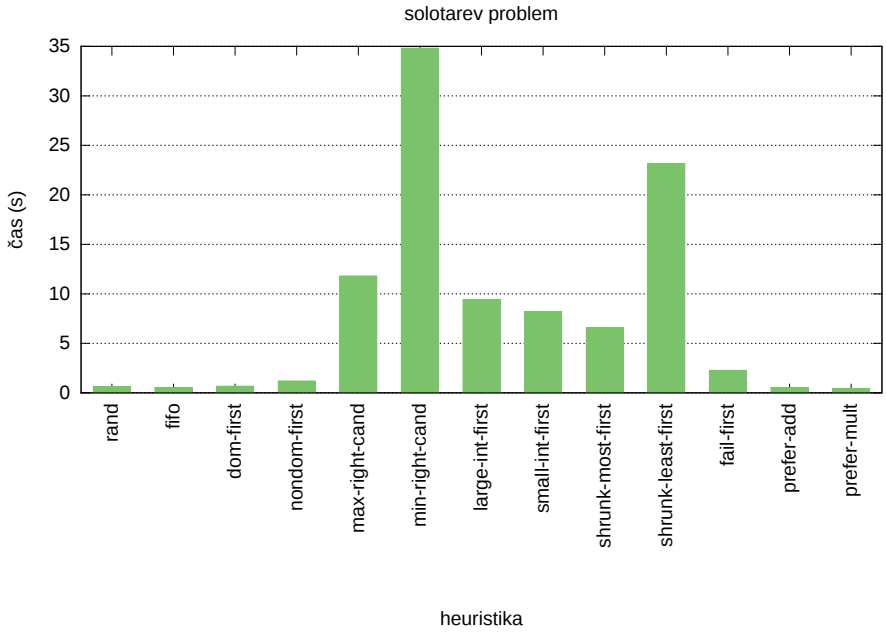
Obrázek 4.11: Porovnání heuristik pro problém *quadfor2* podle počtu zúžení



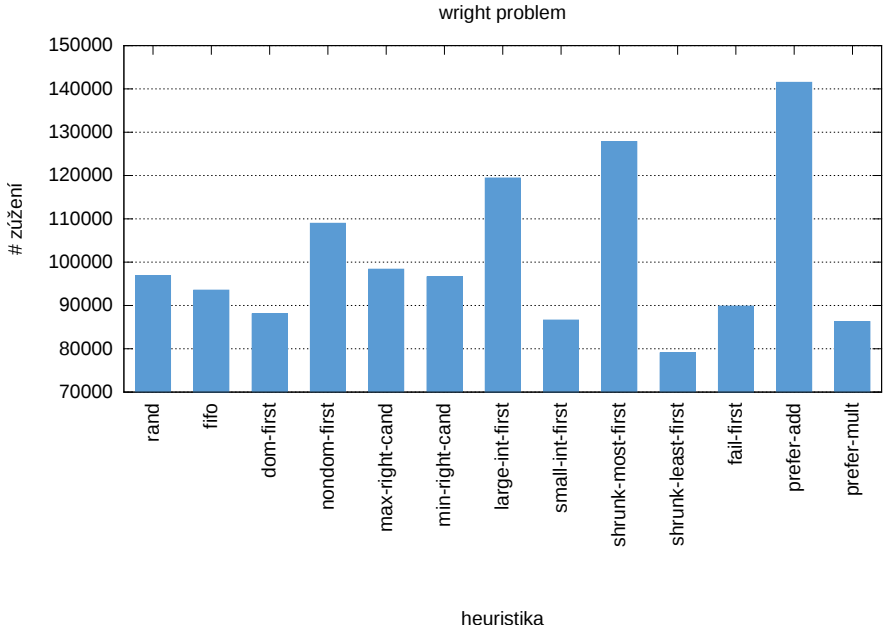
Obrázek 4.12: Porovnání heuristik pro problém *quadfor2* podle času



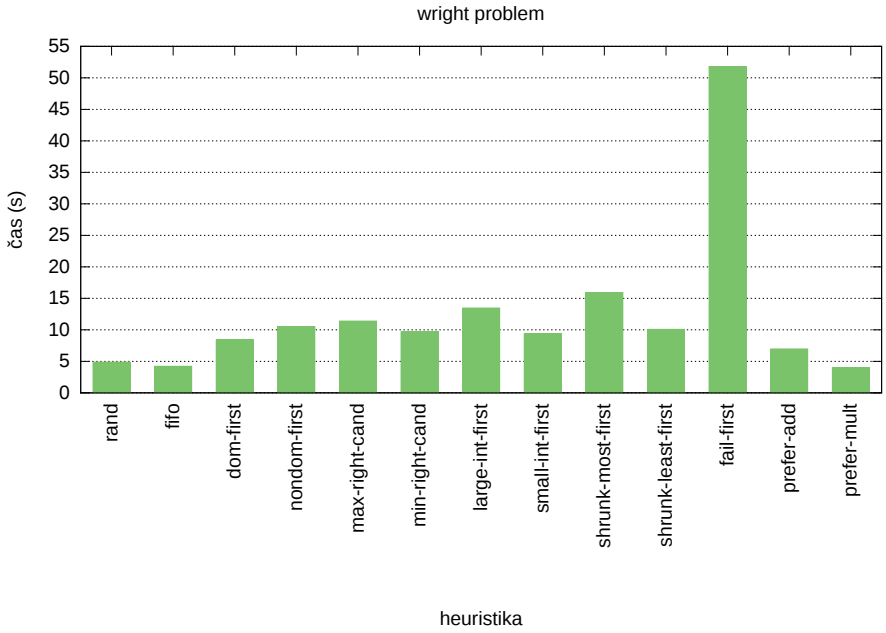
Obrázek 4.13: Porovnání heuristik pro problém *solotarev* podle počtu zúžení



Obrázek 4.14: Porovnání heuristik pro problém *solotarev* podle času



Obrázek 4.15: Porovnání heuristik pro problém *wright* podle počtu zúžení



Obrázek 4.16: Porovnání heuristik pro problém *wright* podle času

4.5 Vyhodnocení výsledků

Následující odstavce porovnávají heuristiky z pohledu hodnot naměřených pro různé indikátory. Platí, že „lepší“ či „dobrá“ heuristika má tyto hodnoty nižší než „horší“ či „špatná“ heuristika.

V tabulkách výše je vidět, že se u jednotlivých heuristik v rámci jedné testovací úlohy příliš neměnily hodnoty ve sloupcích „# půlení“ a „poměr objemu“. Bylo možné očekávat, že obě hodnoty budou ve většině případů stejné - v případě počtu půlení bude pravděpodobně na vině fakt, že při půlení je proměnná, jejíž doména se bude půlit, vybírána metodou round-robin a pořadí proměnné je při každém spuštění algoritmu stejné. Dojde tedy k tomu, že algoritmus HC3 vždy zmenší problém, jak nejvíce dokáže (důležité je, že tohoto dosáhne nehledě na použitou heuristiku), poté se box rozpůlí podle proměnné (opět podle stejné proměnné nehledě na použitou heuristiku) a toto se rekurzivně opakuje. Rozdílné hodnoty u několika málo vstupů mohou být způsobeny nepřesným porovnáváním čísel s plovoucí řádovou čárkou. Hodnoty ve sloupci „poměr objemu“ se pro jeden problém většinou nemění, protože je při startu algoritmu pevně dáno, kolikrát se má objem počátečního boxu zmenšit (hodnota `eps`).

Další indikátor, čas, je zobrazen v grafech z předchozí kapitoly. Heuristika *fifo* (simuluje FIFO frontu a pouze v čase $\mathcal{O}(1)$ vrátí první prvek ze seznamu) dosahuje ve srovnání s ostatními heuristikami velmi dobrých výsledků, co se času týče, v pěti z osmi případů. To poukazuje na skutečnost, že výpočet ostatních heuristik, které dvojici pro další zpracování vyhledávají v celém seznamu v čase $\mathcal{O}(n)$, zpomaluje běh programu.

Heuristikou nejvíce zpomalující běh programu se ukázala být heuristika *large-int-first* (výrazně pomalejší než ostatní byla v polovině případů), která vrací dvojici, jejíž proměnná má nejširší doménu. To je logické, protože cílem algoritmu je zúžit domény dominantních proměnných pod určitou mez a při výběru vždy té nejširší domény se program může zdržovat se zužováním domén, které nemusí být nutně potřeba zúžit.

I výsledky pro poslední indikátor, počet zúžení, jsou zobrazeny v grafech. Zde již výsledky nejsou tak jasné, jako v případě času. Lze říct, že nejlepších výsledků dosáhla heuristika *prefer-mult*, která byla ve dvou případech znatelně lepší než ostatní a ve dvou případech přibližně stejně dobrá jako ostatní. Tato heuristika vybírá první dvojici, jejíž omezující podmínka obsahuje operaci násobení. Při bližším prozkoumání testovacích úloh si lze všimnout, že omezující podmínky problémů, u kterých tato heuristika zafungovala dobře, jsou z více než poloviny tvořeny podmínkami s operací násobení. Opačný jev, tedy že by heuristika *prefer-add* byla lepší než ostatní u problémů s majoritním zastoupením podmínek se sčítáním/násobením, nebyl pozorován.

Nejhorších výsledků z pohledu počtu zúžení dosáhla u složitějších úloh (*quadfor2*, *solotarev*) heuristika *min-right-cand*, v jednom případě byla nejhorší i *max-right-cand*. Velikost pravé meze domény tedy zřejmě neposkytuje

žádnou užitečnou informaci, která by zefektivnila průběh řešení. Špatných výsledků dosáhla také heuristika *large-int-first*, ze stejného důvodu jako u času.

Algoritmus využívající heuristiku *prefer-mult*, který by ukládal dvojice do seznamu již optimálně seřazené tak, aby je stačilo ze začátku seznamu v čase $\mathcal{O}(1)$ odebírat, by mohl u některého typu úloh dosahovat velmi dobrých výsledků. Naopak jistě se nevyplatí používat heuristiky, které se rozhodují podle velikosti jedné z mezí, nebo takové, které preferují největší domény. Otestování algoritmu na několika desítkách úloh by ale mohlo přinést jiné výsledky. Je také možné, že existují i o mnoho lepší heuristiky, které ale nebyly v této práci otestovány.

Závěr

Hlavním cílem bakalářské práce byla implementace řešiče numerických CSP problémů a otestování vlivu heuristik na efektivitu řešení numerických CSP pomocí propagace intervalů. Tento cíl byl splněn - v jazyce `F#` byl vytvořen volně dostupný program HullSolver, který je schopen pomocí algoritmu HC3 řešit problémy ve tvaru soustav polynomiálních rovnic s operacemi sčítání, odčítání a násobení. Pomocí této aplikace byl otestován vliv celkem třinácti heuristik na osmi benchmarkových úlohách a výsledky byly zpracovány. Několik heuristik se podařilo označit za špatné, zatímco některé se ukázaly být oproti ostatním velmi dobré.

V samotném programu je velký prostor pro další vývoj a byla by škoda v něm nepokračovat, protože neexistuje jiný program, který by se specializoval na testování heuristik při řešení NCSP problémů. Prvním krokem by mohlo být přidání podpory pro více aritmetických operací a přidání automatického rozkladu komplexních podmínek do primitivního tvaru. V rámci dalšího vývoje by bylo také možné přidat některé další algoritmy, které by si mohly lépe poradit s nalezenými problémovými instancemi, a přidat grafický výstup programu, neboť aktuální verze komunikuje s uživatelem pouze prostřednictvím příkazové řádky.

Literatura

[1] Wegner, P.; Doyle, J.: Editorial: Strategic Directions in Computing Research. *ACM Comput. Surv.*, ročník 28, č. 4, Prosinec 1996: s. 565–574, ISSN 0360-0300, doi:10.1145/242223.242227. Dostupné z: `http://doi.acm.org/10.1145/242223.242227`

[2] Rossi, F.; Beek, P. v.; Walsh, T.: *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. New York, NY, USA: Elsevier Science Inc., 2006, ISBN 0444527265.

[3] Vu, X.-H.: *Rigorous solution techniques for numerical constraint satisfaction problems*. Dizertační práce, IC, Lausanne, 2005.

[4] Russell, S. J.; Norvig, P.: *Artificial Intelligence: A Modern Approach*. Pearson Education, druhé vydání, 2003, ISBN 0137903952.

[5] Kueviakoe, I. K.; Lambert, A.; Tarroux, P.: Comparison of Interval Constraint Propagation Algorithms for Vehicle Localization. *Journal of Software Engineering and Applications*, ročník 5, č. 12B, 2012: s. 157–162.

[6] Rueher, M.: Solving Continuous Constraint Systems. In *Proceedings of 3IA'2005 International Conference in Computer Graphics and Artificial Intelligence*, 2005, ISBN 2-914256-07-8, s. 35–55. Dostupné z: `http://users.polytech.unice.fr/~rueher/Publis/3ia_rueher.pdf`

[7] Moore, R. E.; Kearfott, R. B.; Cloud, M. J.: *Introduction to Interval Analysis*. SIAM, 2009.

[8] Cleary, J. G.: Logical Arithmetic. *Future Computing Systems*, ročník 2, č. 2, 1987: s. 125–149.

[9] Hickey, T.; Ju, Q.; Van Emden, M. H.: Interval Arithmetic: From Principles to Implementation. *J. ACM*, ročník 48, č. 5, Září 2001: s. 1038–1068, ISSN 0004-5411, doi:10.1145/502102.502106. Dostupné z: `http://doi.acm.org/10.1145/502102.502106`

[10] Barták, R.: On-line Guide to Constraint Programming. 1998, [Online; cit. 2016-03-29]. Dostupné z: `http://ktiml.mff.cuni.cz/~bartak/constraints`

[11] Benhamou, F.; Older, W. J.: Applying interval arithmetic to real, integer and Boolean constraints. *Journal of Logic Programming*, ročník 32, č. 1, 1997: s. 1–24.

[12] Benhamou, F.; Goualard, F.; Granvilliers, L.; aj.: Revising Hull and Box Consistency. In *Int. Conf. on Logic Programming*, MIT press, 1999, s. 230–244.

[13] Feiten, L.: *Development and Analysis of Decision Heuristics for an Interval Constraint Solver Handling Non-linear Arithmetic*. Department of Computer Science, Albert-Ludwigs-Universitat Freiburg, 2010.

Příloha **A**

Seznam použitých zkratek

CSP Constraint Satisfaction Problem

NCSP Numerical Constraint Satisfaction Problem

Příloha B

Manuál k programu HullSolver

B.1 Kompilace

Pro zkompilování zdrojových kódu na Windows lze použít kompilátor `fsc.exe` (`fsc HullSolver.sln`) dodávaný společně s Microsoft Visual Studio. Pro spuštění je potřeba mít nainstalovaný .NET framework ve verzi minimálně 4.5.0.

Na Linuxu a OS X je potřeba nejprve nainstalovat Mono[9], které umožňuje kompilovat a spouštět .NET aplikace i na ostatních platformách, než jen Windows. Na Linuxu, resp. OS X jej lze nainstalovat příkazem `apt-get install mono-complete fsharp`, resp. `brew install mono`. Poté již půjde zdrojové kódy zkompilovat příkazem `xbuild HullSolver.sln`. Oba kompilátory uloží zkompilovanou aplikaci do adresáře `bin/Debug`.

B.2 Spuštění

Zkompilovaný program se spouští z příkazové řádky na Windows jednoduše jako `bin\Debug\HullSolver.exe`, v terminálu na Linuxu a OS X jako `mono bin/Debug/HullSolver.exe`. Program podporuje čtyři přepínače:

- `-f <path>` - cesta k souboru s problémem k vyřešení
- `-p <precision>` - kolikrát se musí velikost domény dominantní proměnné snížit, aby byla považována za řešení
- `-h <heuristic>` - použitá heuristika (seznam použitelných názvů je níže)
- `-l` - výstupem programu budou tabulky ve formátu LaTeX

[9] `http://www.mono-project.com`

V případě spuštění bez přepínače `-h` bude využita heuristika `rand` (pseudonáhodný výběr), pokud bude program spuštěn bez přepínače `-p`, bude využita výchozí hodnota 1.0. Pokud uživatel spustí program bez zadání vstupního souboru, bude vybídnut k jeho zadání.

V repozitáři se dále nachází dva dávkové soubory pro Windows, které umožňují pohodlnější opakované spouštění programu - spustí jej postupně se všemi heuristikami pro všechny vstupní problémy a naměřené hodnoty uloží do adresáře `./out`.

B.3 Vstup

Vstupem programu HullSolver jsou textové soubory, které deklarativně popisují NCSP problém, který chce uživatel vyřešit. V souboru jsou nejprve uvedeny dominantní proměnné následované seznamem omezujících podmínek v primitivním tvaru a nakonec jsou uvedeny domény jednotlivých proměnných. Podporovány jsou komentáře uvozené znaky `//`.

Příklad vstupního souboru:

```
// Dominantní proměnné
x y a b

// Omezující podmínky
x * x = a
x + y = b

// Domény proměnných
x in [1,10]
y in [0,100]
a in [16,16]
b in [10,10]
```

Pro intervaly je využita anglosaská notace, $[1, 10]$ odpovídá intervalu $\langle 1; 10\rangle$.

Program umí zpracovat pouze ternární omezující podmínky se sčítáním, odčítáním a násobením ve formátu naznačeném v příkladu. Jinými slovy musí být na levé straně rovnice právě jedna aritmetická operace mezi dvěma proměnnými a na pravé straně právě jedna proměnná. Například podmínku

$$x = y - 1, x \in \langle 0; 10\rangle, y \in \langle 0; 10\rangle$$

je možné zadat jako:

```
x y

y - a = x

x in [0,10]
y in [0,10]
a in [1,1]
```

Příloha C

Obsah přiloženého CD

- `README.txt` ..... popis obsahu CD
- `src`
  - `hullsolver` ..... zdrojové kódy programu
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `tests` ..... vstupní testovací úlohy
- `text`
  - `thesis.pdf` ..... text práce ve formátu PDF