Математичка гимназија, Београд

Матурски рад из предмета Рачунарство и информатика

Употреба програмирања ограничења (CLP) у програмским језицима треће генерације

Леа Ирт

Ментор: Филип Хаџић

Београд, мај 2022

Садржај

Глава 1

Увод

Математичке игре су логички проблеми код којих су правила и исходи прецизно математички дефинисани. Популаран су облик разоноде, па их осим у математичким часописима, врло често срећемо у дневним листовима, магазинима или на друштвеним мрежама. Примери математичких игара су Судоку, Ајнштајнова загонетка, Какуро, магични квадрати и слично. Једноставно су формулисане и разумљиве чак и људима са елементарним знањем математике. Међутим, ако бисмо покушали да их решимо уз помоћ рачунара, то не би било нимало једноставно.

Програмирање ограничења[1] је занимљива парадигма за формулисање и решавање проблема које је могуће написати декларативно, у облику ограничења над скупом промењивих. Решавање таквих проблема своди се на проналажење комбинација вредности које задовољавају дата ограничења. Логичко програмирање ограничења (енг. *constraint logic programming*, скраћено CLP) комбинује логичко програмирање и принцип решавања проблема дефинисањем ограничења, без експлицитног навођења алгоритма. CLP је већ дуги низ година уграђен у логички програмски језик Prolog, али је остао непознат у најпопуларнијим програмским језицима данашњице.

У последње време појавиле су се бројне независне имплементације CLP које омогућавају коришћење програмирања ограничења у разним програмским језицима на начин сличан оном који је до сада био познат искључиво у Prolog-у. Циљ матурског рада је испитивање могућности примене логичког програмирања ограничења у популарним програмским језицима треће генерације.

Глава 2

Еволуција програмских језика

2.1 Програмски језици прве генерације

Прва генерација програмских језика (1GL) обухвата језике на машинском нивоу.[2,15] Они су конструисани над бинарном азбуком и није потребно превођење, што програме чини брзим и ефикасним за извршавање, али је програмирање тешко и подложно грешкама. Будући да свака фамилија процесора има свој машински језик, још једна од мана ове генерације јесте што су везани за конкретан рачунар.

2.2 Програмски језици друге генерације

Другу генерацију програмских језика (2GL) чине асемблерски језици. У поређењу са машинским лакши су за разумевање и учење. Ипак, како су команде једноставне, и мање комплексни програми садрже велики број инструкција. Ови језици више су прилагођени хардверу него програмеру, што значи да сваки програмер мора бити добро упознат са начином рада рачунара за који пише програм.

2.3 Програмски језици треће генерације

Трећа генерација програмских језика (3GL) користи команде на енглеском језику, што их чини разумљивијим. Инструкције су сложеније, па се исти програм реализује далеко мањим бројем команди него у језицима прве и друге генерације. Коришћење ових језика елеминисало је потребу да програмер познаје асемблерски језик и хардвер. Међутим, неопходни су напреднији компајлери, и то различити за сваки тип рачунара. Начин програмирања у програмским језицима ове генерације је процедурални, што значи да се од програмера тражи да напише детаљан алгоритам за решавање проблема. Шездесетих година 20. века ушли су у широку примену, а данас су то програмски језици у убедљиво најмасовнијој употреби. Овој генерацији припадају C#, C++, Java и многи други.

2.4 Програмски језици четврте генерације

Четврта генерација програмских језика (4GL) је настала са идејом да се учење програмирања олакша. Команде су комплексније, а програмирање брже и мање подложно грешкама. Највећа мана јесте што нису флексибилни, односно, ограничени су на скуп команди које програм подржава. Најпознатији језик ове генерације који се још увек користи је SQL.

2.5 Програмски језици пете генерације

Пета генерација програмских језика (5GL) је базирана на принципима вештачке интелигенције. То значи да програмер декларативно описује проблем, а начин решавања у потпуности препушта програмском језику. Иако су у теорији најједноставнији од свих програмских језика, никада нису ушли у масовну употребу. Најпознатији програмски језици ове генерације су Prolog и Lisp.

Глава 3

Програмски језик Prolog

Prolog[16] је логички програмски језик који је нашао примену у вештачкој интелигенцији и рачунарској лингвистици.

Темељи се на логици првог реда, формалној логици, и за разлику од многих других програмских језика, Prolog је првенствено замишљен као декларативни програмски језик. Програми у Prolog-у не садрже инструкције које су типичне за језике треће генерације. Логика програма изражена је у терминима релација, представљених у виду чињеница и правила. Извршавање ових програма такође није стандарно, већ корисник интерактивно поставља упите на које програм логичким путем покушава да дође до одговора.

Језик је настао 1972. године, као плод сарадње истраживача Роберта Ковалског (Универзитет у Единбургу), Алена Кормерера и Филипа Русела (Универзитет у Марсеју).

3.1 Логичко програмирање ограничења (CLP)

Логичко програмирање ограничења (CLP) је модул који проширује Prolog.[1] Укључивањем овог модула дозвољава се дефинисање ограничења, која Prolog процесуира заједно са осталим логичким предикатима. Погодан је за велике комбинаторне проблеме оптимизације и зато је користан за апликације у индустријским окружењима, као што је аутоматизовано распоређивање ресурса или организација производње. Већина Prolog имплементација укључује модул за решавање логичких ограничења за коначне домене (CLPFD), а често и модуле за друге домене, попут рационалних (CLPQ) или реалних бројева (CLPR).

3.2 Пример примене CLP у програмском језику Prolog

Програм 3.1: Решавање криптоаритметичког проблема SEND + MORE = MONEY у програмском језику Prolog.[1] Криптоаритметички проблеми су математичке игре у којима се решавају једначине код којих су цифре бројева замењене одређеним словима.

```
:- use_module(library(clpfd)).

solve([S,E,N,D],[M,O,R,E],[M,O,N,E,Y]):-
    Vars = [S,E,N,D,M,O,R,Y],        % all variables in the puzzle
    Vars ins 0..9,                   % they are all decimal digits
    all_different(Vars),             % they are all different
    1000*S + 100*E + 10*N + D +
    1000*M + 100*O + 10*R + E #=
    10000*M + 1000*O + 100*N + 10*E + Y,
    S #\= 0,
    M #\= 0,
    labeling([], Vars).
```

Протумачимо делове програма. Коришћење модула за логичка ограничења за коначне домене CLPFD потребно је најавити наредбом:

```
:- use_module(library(clpfd)).
```

Затим дефинишемо промењљиве и њихов домен. Промењљиве су слова $\{S, E, N, D, M, O, R, Y\}$, а њихов домен цифре од 0 до 9:

```
  Vars = [S,E,N,D,M,O,R,Y],
  Vars ins 0..9,
```

Следи дефинисање ограничења. Уграђеним предикатом `all_different` захтевамо да вредности свих промењливих буду различите:

```
  all_different(Vars),
```

Аритметички израз $SEND + MORE = MONEY$ треба да буде испуњен:

```
  1000*S + 100*E + 10*N + D +
  1000*M + 100*O + 10*R + E #=
  10000*M + 1000*O + 100*N + 10*E + Y,
```

Почетне цифре бројева морају бити различите од 0:

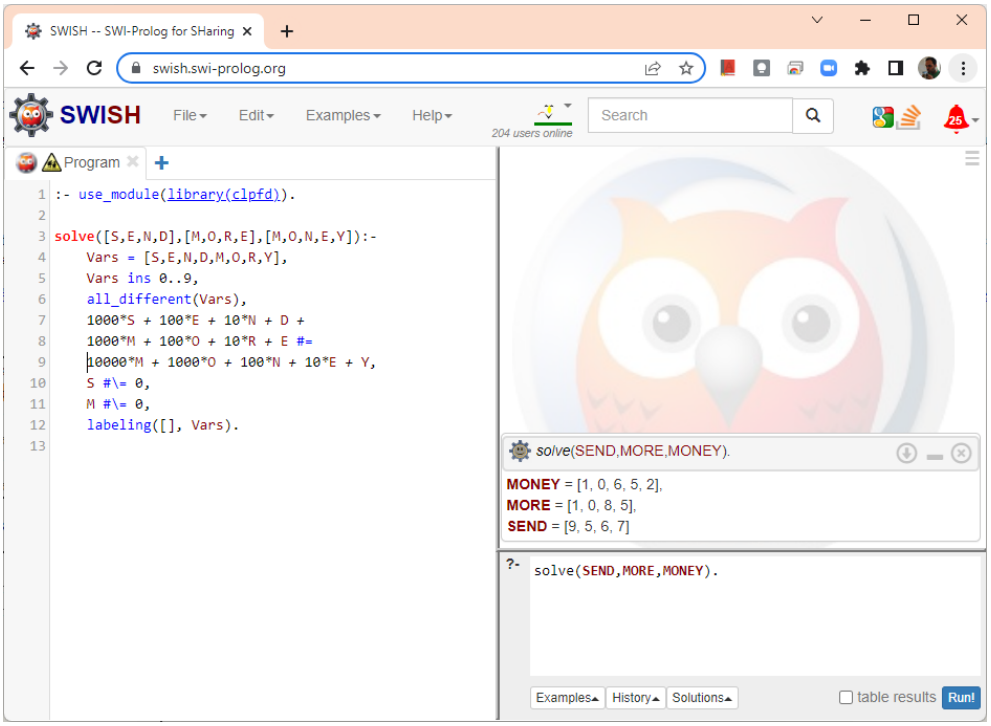
```
S #\= 0,
M #\= 0,
```

Генеришемо листу конкретних вредности промењљивих које одговарају свим задатим ограничењима:

```
labeling([], Vars)
```

То је уједно и последња команда програма.

Програм тестирамо у једној од имплементација програмског језика Prolog (слика 3.1).



Слика 3.1: Тестирање криптоаритметичког проблема SEND + MORE = MONEY са CLPFD у SWI-Prolog

Глава 4

Дефиниција логичких проблема ограничења

Дефиниција Проблем задовољења ограничења (енг. *constraint satisfaction*, скраћено CSP) представља[10] уређену тројку $\mathcal{A} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$, где је:

- $\mathcal{X} = \{x_1, \ldots, x_n\}$ скуп променљивих,
- $\mathcal{D} = \{\mathcal{D}_1, \ldots, \mathcal{D}_n\}$ скуп домена у којима променљиве смеју узимати вредности и
- $\mathcal{C} = \{C_1, \ldots, C_m\}$ скуп ограничења која променљиве морају задовољавати.

Разликујемо три типа ограничења:

- екстензионална ограничења - ограничења дефинисана експлицитном листом комбинација вредности променљивих које долазе у обзир
- аритметичка ограничења - ограничења дефинисана аритметичким релацијама, као што су $<$, $>$, $\leq$, $\geq$, $=$, $\neq$, ...
- логичка ограничења - ограничења дефинисана уграђеним логичким предикатима, попут *allDifferent*, *atMost*,...

Циљ је променљивим доделити конкретне вредности из домена, које испуњавају сва задата ограничења.

Глава 5

Начини решавања логичких проблема ограничења

Логички проблеми ограничења у пракси имају веома широк спектар могуће употребе, па зато не постоји јединствена мотода за решавање свих проблема те врсте, већ се примењују различите[10] технике и алгоритми. Тако се на пример проблеми у којима су променљиве реални бројеви решавају потпутно другачије од оних у којима су променљиве целобројне. Најчешће коришћене технике су бектрекинг и пропагација ограничења, а у случајевима када је број комбинација велики смислено је користити и хеуристичке* алгоритме попут локалног претраживања.

5.1 Бектрекинг

Бектрекинг[9] (енг. *backtracking*) је рекурзивни алгоритам који представља приступ грубе силе у тражењу решења, где се испробавају све могуће комбинације. Постепено се граде кандидати за решење, а одбацују се сви кандидати за које се испостави да не воде до тачног решења. Због сложености неких проблема, алгоритам се често споро извршава, па се алгоритми прилагођавају датом проблему. У практичкој примени код решавања логичких проблема ограничења је бектрекинг алгоритам обично основни алгоритам за претраживање, али се због ефикасности комбинује и са осталим техникама.

5.2 Пропагација ограничења

Техника пропагације ограничења[12]је метода која се користи за упрошћавање логичких проблема ограничења. Тачније, то је метода која намеће локалну конзистентност променљивих и/или ограничења. Пропагација

*Хеуристика је техника решавања која је бржа од класичних метода и користи се за налажење приближног решења када класични методи не могу да нађу тачно решење.

ограничења има различите намене. Прво, претвара проблем у онај који је еквивалентан, али је обично једноставнији за решавање. Друго, може доказати да ли проблем има решење. Најпопуларнији метод пропагације ограничења је AC-3 алгоритам, који спроводи конзистентност грана (енг. *arc consistency*).

Пример 1: Решавање једноставних аритметичких ограничења

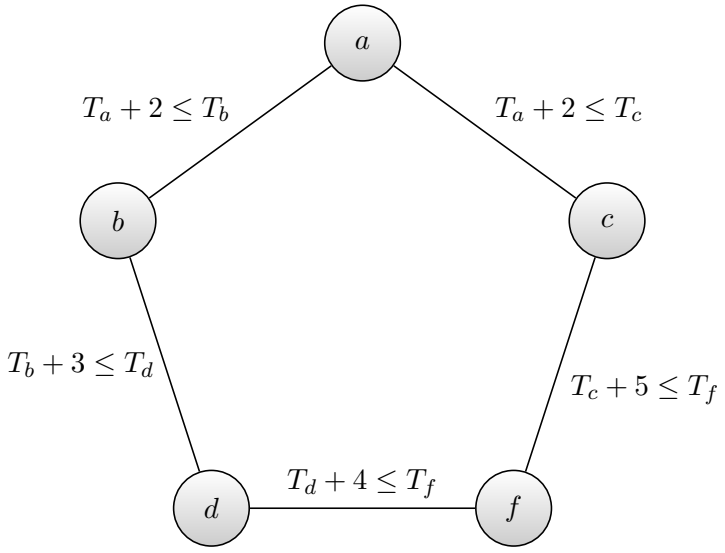
Замислимо једноставан проблем[1] са расподелом задатака a, b, c и d. Нека су вредности T_a, T_b, T_c и T_d ненегативни реални бројеви којима обележавамо почетке извођења тих задатака и нека је T_f време завршетка извођења свих задатака.

Ограничења:

$T_a + 2 \leq T_b$	задатак a траје 2 сата и претходи b
$T_a + 2 \leq T_c$	a претходи c
$T_b + 3 \leq T_d$	b траје 3 сата и претходи d
$T_c + 5 \leq T_f$	c мора да буде завршен пре T_f
$T_d + 4 \leq T_f$	d мора да буде завршен пре T_f

Циљ је израчунати најкраће време за које могу да се заврше сви задаци, тј. минимализација T_f.

Направимо граф ограничења (слика 5.1).



Слика 5.1: Граф ограничења

Пропагацијом услова сужавамо домен променљивих и постижемо конзистентност ограничења на гранама (слика 5.2).

Корак	Грана	T_a	T_b	T_c	T_d	T_f
Старт		$0 \dots 10$	$0 \dots 10$	$0 \dots 10$	$0 \dots 10$	$0 \dots 10$
1	(T_b, T_a)		$2 \dots 10$			
2	(T_d, T_b)				$5 \dots 10$	
3	(T_f, T_d)					$9 \dots 10$
4	(T_d, T_f)				$5 \dots 6$	
5	(T_b, T_d)		$2 \dots 3$			
6	(T_a, T_b)	$0 \dots 1$				
7	(T_c, T_a)			$2 \dots 10$		
8	(T_c, T_f)			$2 \dots 5$		

Слика 5.2: Табеларни приказ пропагације услова

Пример 2: Решавање *allDifferent* логичких ограничења

У овом проблему имамо пет особа које желе да позајме књигу из библиотеке.[20] У библиотеци има 5 књига. Сваку особу занимају само неке од тих књига:

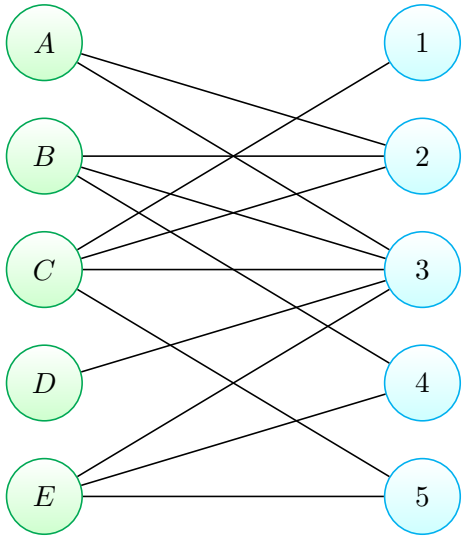
- *Особу A занимају књиге* 2 *и* 3.
- *Особу B занимају књиге* 2, 3 *и* 4.
- *Особу C занимају књиге* 1, 2, 3 *и* 5.
- *Особу D занима књига* 3.
- *Особу E занимају књиге* 3, 4 *и* 5.

Потребно је поделити књиге тим особама тако да свако добије по једну.

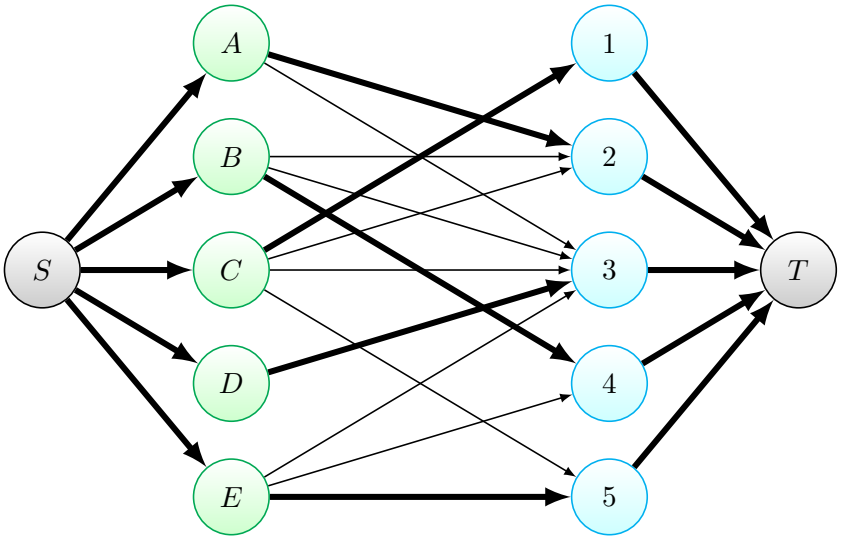
Запишимо проблем у стандардном CSP облику. Променљиве су $A \in \{2, 3\}$, $B \in \{2, 3, 4\}$, $C \in \{1, 2, 3, 5\}$, $D \in \{3\}$ и $E \in \{3, 4, 5\}$, а тражено ограничење је *allDifferent*$(\{A, B, C, D, E\})$. Проблем представимо у облику бипартитног графа (слика 5.3).

Са једне стране тог графа доцртаћемо чвор S (*source*) и повезати га са променљивим $(A \dots E)$, а са друге стране доцртаћемо чвор T (*target*) и повезати га са свим могућим вредностима $(1 \dots 5)$. Претварањем неусмереног графа у усмерени добија се такозвана транспортна мрежа (слика 5.4). На транспортној мрежи уз помоћ неког од стандардних алгоритама (нпр. Форд-Фулкерсоновим[11] алгоритмом) проналазимо највећи проток.

Ако је највећи проток једнак кардиналности скупа вредности онда су све вредности различите што је еквивалентно *allDifferent* ограничењу.



Слика 5.3: Бипартитни граф који приказује променљиве (са леве стране) и њихове могуће вредности (са десне стране)



Слика 5.4: Транспортна мрежа која одговара бипартитном графу на слици 5.3. Подебљаним линијама су обележене гране које припадају максималном протоку. Максималан проток је 5 што одговара укупном броју свих различитих вредности па је тако услов $allDifferent$ испуњен.

5.3 Локално претраживање

Метода локалне претраге[13] је хеуристички метод за решавање рачунарски тешких проблема оптимизације. Локална претрага се може користити за проблеме који се могу формулисати као проналажење решења које максимизира критеријум међу бројним решењима кандидата. Алгоритми локалне претраге се крећу од решења до решења у простору кандидата решења (простор за претрагу) применом локалних промена, све док се не пронађе решење за које се сматра да је оптимално или док не истекне временско ограничење.

Глава 6

Примена CLP у програмском језику C#

6.1 CLP у програмском језику C#

Логичко програмирање ограничења није саставни део програмског језика C#, али је зато преко *NuGet* платформе објављена *open-source* имплементација логичког програмирања ограничења названа *Decider*.[4] Библиотеку је једноставно укључити у C# пројекат наредбом:

```
dotnet add package decider
```

Нема много информација о ауторима тог решења и начину имплементације. Међутим, употреба ове библиотеке је врло једноставна, а тестирањем на разним примерима утврђено је да ефикасно решава и сложеније проблеме над коначним доменима (CLPFD), али не подржава решавање проблема са реалним бројевима (CLPR).

6.2 Примери примене у математичким играма

6.2.1 Криптоаритметички проблеми

Криптоаритметички проблеми[18] су математичке игре у којима су цифре бројева у некој једначини представљене словима или симболима. Свако слово представља јединствену цифру. Циљ је пронаћи цифре тако да је дата математичка једначина исправна.

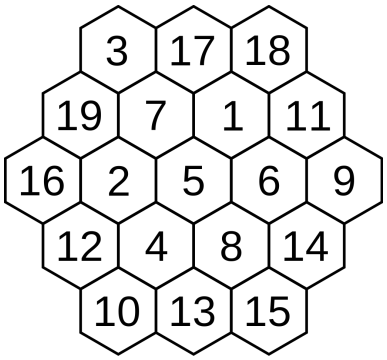
Ове проблеме једноставно је поставити у CLP. Садрже мали број променљивих и комбинују аритметичка и логичка ограничења.

Програм 6.1: Решавање криптоаритметичког проблема SEND + MORE = MONEY у програмском језику C#

```
using Decider.Csp.BaseTypes;
using Decider.Csp.Global;
using Decider.Csp.Integer;

var s = new VariableInteger("s", 1, 9);
var e = new VariableInteger("e", 0, 9);
var n = new VariableInteger("n", 0, 9);
var d = new VariableInteger("d", 0, 9);
var m = new VariableInteger("m", 1, 9);
var o = new VariableInteger("o", 0, 9);
var r = new VariableInteger("r", 0, 9);
var y = new VariableInteger("y", 0, 9);

var variables = new[] { s, e, n, d, m, o, r, y };

var constraints = new List<IConstraint>
{
    new AllDifferentInteger(variables),
    new ConstraintInteger(  s*1000 + e*100 + n*10 + d
                          +   m*1000 + o*100 + r*10 + e ==
                          m*10000 + o*1000 + n*100 + e*10 + y)
};

var state = new StateInteger(variables, constraints);
var searchResult = state.Search();

Console.WriteLine("    {0} {1} {2} {3} ", s, e, n, d);
Console.WriteLine("  + {0} {1} {2} {3} ", m, o, r, e);
Console.WriteLine("  ---------");
Console.WriteLine("  {0} {1} {2} {3} {4} ", m, o, n, e, y);
```

6.2.2 Магични шестоугао

Магичне фигуре један су од популарних проблема забавне математике. Постоје једноставнији примери, попут магичног квадрата 3×3, а има и тежих, као што је магични шестоугао.[14] Магични шестоугао је фигура у којој су бројеви распоређени у шестоуглу, а њихови збирови у сва три правца морају бити једнаки (слика 6.1). Оригинални задатак са 19 бројева осмислио је Клифорд Адамс, који га је интензивно решавао готово пола века.



Слика 6.1: Магични шестоугао

Проблем је једноставно записати у CLP облику. Промењиве су поља магичног шестоугла, домен промељивих су бројеви од 1 до 19. Лако је изра-

чунати, да су потребне суме једнаке $\frac{1}{5}(1+2+3+\cdots+19) = 38$. Ограничења су да бројеви морају бити различити и да свих 15 збирова буде једнако 38.

Решења магичног шестоугла су симетрична, па се поред стандарних аритметичких и логичких ограничења, уводе и ограничења која елиминишу односиметрична и централно симетрична решења.

Код већих магичних квадрата потребно је коришћење сложенијих алгоритама, па су згодни за тестирање ефикасности имплементације CLP.

Програм 6.2: Решавање магичног шестоугла у програмском језику C#

```
using Decider.Csp.BaseTypes;
using Decider.Csp.Global;
using Decider.Csp.Integer;

var p = new VariableInteger[19];
var constraints = new List<IConstraint>();

for(int i = 0; i < 19; i++)
    p[i] = new VariableInteger("p_" + i, 1, 19);
constraints.Add(new AllDifferentInteger(p));

var s = new VariableInteger("s", 38, 38);

constraints.Add(new ConstraintInteger(p[0]+p[1]+p[2]==s));
constraints.Add(new ConstraintInteger(p[3]+p[4]+p[5]+p[6]==s));
constraints.Add(new ConstraintInteger(p[7]+p[8]+p[9]+p[10]+p[11]==s));
constraints.Add(new ConstraintInteger(p[12]+p[13]+p[14]+p[15]== s));
constraints.Add(new ConstraintInteger(p[16]+p[17]+p[18]==s));

constraints.Add(new ConstraintInteger(p[0]+p[3]+p[7]==s));
constraints.Add(new ConstraintInteger(p[1]+p[4]+p[8]+p[12]==s));
constraints.Add(new ConstraintInteger(p[2]+p[5]+p[9]+p[13]+p[16]==s));
constraints.Add(new ConstraintInteger(p[6]+p[10]+p[14]+p[17]==s));
constraints.Add(new ConstraintInteger(p[11]+p[15]+p[18]==s));

constraints.Add(new ConstraintInteger(p[2]+p[6]+p[11]==s));
constraints.Add(new ConstraintInteger(p[1]+p[5]+p[10]+p[15]==s));
constraints.Add(new ConstraintInteger(p[0]+p[4]+p[9]+p[14]+p[18]==s));
constraints.Add(new ConstraintInteger(p[3]+p[8]+p[13]+p[17]==s));
constraints.Add(new ConstraintInteger(p[7]+p[12]+p[16]==s));

constraints.Add(new ConstraintInteger(p[0]<p[2]));
constraints.Add(new ConstraintInteger(p[0]<p[7]));
constraints.Add(new ConstraintInteger(p[0]<p[11]));
constraints.Add(new ConstraintInteger(p[0]<p[16]));
constraints.Add(new ConstraintInteger(p[0]<p[18]));

constraints.Add(new ConstraintInteger(p[2]<p[7]));

var state = new StateInteger(vars, constraints);
var searchResult = state.Search();

Console.WriteLine("{0} {1} {2}",p[0],p[1],p[2]);
Console.WriteLine("{0} {1} {2} {3}",p[3],p[4],p[5],p[6]);
Console.WriteLine("{0} {1} {2} {3} {4}",p[7],p[8],p[9],p[10],p[11]);
Console.WriteLine("{0} {1} {2} {3}",p[12],p[13],p[14],p[15]);
Console.WriteLine("{0} {1} {2}",p[16],p[17],p[18]);
```

6.2.3 Судоку

Судоку[17] је логичка загонетка у облику квадратне мреже димензија 9×9, раздељене на блокове димензија 3×3. У свако поље могуће је уписати цифру од 1 до 9. Неколико цифара је на почетку унето, а циљ је попунити сва остала поља. У свакој врсти, колони и блоку све цифре морају бити различите. Игра је настала 1986. године у Јапану, а глобалну популарност стекла је почетком 21. века.

У CLP је лако дефинисати наведена правила. Интересантно је што ова математичка игра не користи аритметичка ограничења, већ искључиво логичка *allDifferent* ограничења.

Програм 6.3: Решавање Судоку у програмском језику C#

```
using Decider.Csp.BaseTypes;
using Decider.Csp.Global;
using Decider.Csp.Integer;

var a = new int[] {
    8, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 3, 6, 0, 0, 0, 0, 0,
    0, 7, 0, 0, 9, 0, 2, 0, 0,
    0, 5, 0, 0, 0, 7, 0, 0, 0,
    0, 0, 0, 0, 4, 5, 7, 0, 0,
    0, 0, 0, 1, 0, 0, 0, 3, 0,
    0, 0, 1, 0, 0, 0, 0, 6, 8,
    0, 0, 8, 5, 0, 0, 0, 1, 0,
    0, 9, 0, 0, 0, 0, 4, 0, 0,
};

var p = new VariableInteger[81];
var constraints = new List<IConstraint>();

for(int i = 0; i < 9; i++)
    for(int j = 0; j < 9; j++) {
        int k = i*9 + j;
        if(a[k] == 0)
            p[k] = new VariableInteger("p" + k, 1, 9);
        else
            p[k] = new VariableInteger("p" + k, a[k], a[k]);
    }

for(int i = 0; i < 9; i++) {
    VariableInteger[] row = new VariableInteger[9];
    for(int j = 0; j < 9; j++)
        row[j] = p[i*9 + j];
    constraints.Add(new AllDifferentInteger(row));
}

for(int j = 0; j < 9; j++) {
    VariableInteger[] col = new VariableInteger[9];
    for(int i = 0; i < 9; i++)
        col[i] = p[i*9 + j];
    constraints.Add(new AllDifferentInteger(col));
}
```

```
for(int i = 0; i < 9; i+=3)
    for(int j = 0; j < 9; j+=3) {
        int k = i*9 + j;
        VariableInteger[] box = new[] {
            p[k + 0], p[k + 1], p[k + 2],
            p[k + 9], p[k + 10], p[k + 11],
            p[k + 18], p[k + 19], p[k + 20]
        };
        constraints.Add(new AllDifferentInteger(box));
    }

var state = new StateInteger(vars, constraints);
var searchResult = state.Search();

for(int i = 0; i < 9; i++) {
    for(int j = 0; j < 9; j++)
        Console.Write("{0} ", p[i*9 + j]);
    Console.WriteLine();
}
```

6.2.4 Ајнштајнова загонетка

Ајнштајнова загонетка[19] је логичка загонетка чији се настанак приписује Алберту Ајнштајну. У поставци задатка је 5 особа различитих националности, које воле различита пића и цигарете и имају различите кућне љубимце. Они живе у истој улици, а куће су им обојене различитим бојама. Потребно је закључити ко је власник рибе, на основу 15 задатих тврдњи.

За разлику од претходних примера где су вредности променљивих бројчане, овде су домени листе речи. Из тог разлога потребно је прво направити одговарајући модел.[7]

Програм 6.4: Модел направљен за Ајнштајнову загонетку у програмском језику C#

```
namespace Model
{
   public enum Pet {DOGS,BIRDS,HORSES,CATS,FISH}
   public enum Drink {TEA,COFFEE,MILK,BEER,WATER}
   public enum Color {RED,GREEN,WHITE,YELLOW,BLUE}
   public enum Nation {ENGLISHMAN,SWEDE,DANE,NORWEGIAN,GERMAN}
   public enum Cigarette {PALLMALL,DUNHILL,BLUEMASTER,PRINCE,BLEND}
}
```

Променљиве организујемо у облику структуре података мапа (*Map*, *Dictionary*) тако што правимо уређене парове свих вредности из сваке листе са променљивим. Након што је ово одрађено, логичка ограничења је једноставно поставити.

Програм 6.5: Решавање Ајнштајнове загонетке у програмском језику C#

```
using Decider.Csp.BaseTypes;
using Decider.Csp.Global;
using Decider.Csp.Integer;
using Model;

var pet = new Dictionary<Pet, VariableInteger>();
var drink = new Dictionary<Drink, VariableInteger>();
var color = new Dictionary<Color, VariableInteger>();
var nation = new Dictionary<Nation, VariableInteger>();
var cigarette = new Dictionary<Cigarette, VariableInteger>();

var vars = new VariableInteger[25];

int k = 0;
foreach (Pet value in Enum.GetValues(typeof(Pet))) {
   pet[value] = new VariableInteger("" + value, 1, 5);
   vars[k++] = pet[value];
}
foreach (Drink value in Enum.GetValues(typeof(Drink))) {
   drink[value] = new VariableInteger("" + value, 1, 5);
   vars[k++] = drink[value];
}
foreach (Color value in Enum.GetValues(typeof(Color))) {
   color[value] = new VariableInteger("" + value, 1, 5);
   vars[k++] = color[value];
}
foreach (Nation value in Enum.GetValues(typeof(Nation))) {
   nation[value] = new VariableInteger("" + value, 1, 5);
   vars[k++] = nation[value];
}
foreach (Cigarette value in Enum.GetValues(typeof(Cigarette))) {
   cigarette[value] = new VariableInteger("" + value, 1, 5);
   vars[k++] = cigarette[value];
}

var constraints = new List<IConstraint>();

constraints.Add(new AllDifferentInteger(pet.Values));
constraints.Add(new AllDifferentInteger(drink.Values));
constraints.Add(new AllDifferentInteger(color.Values));
constraints.Add(new AllDifferentInteger(nation.Values));
constraints.Add(new AllDifferentInteger(cigarette.Values));

// the Brit lives in the red house
constraints.Add(new ConstraintInteger(
    nation[Nation.ENGLISHMAN]==color[Color.RED]));

// the Swede keeps dogs as pets
constraints.Add(new ConstraintInteger(
    nation[Nation.SWEDE]==pet[Pet.DOGS]));

// the Dane drinks tea
constraints.Add(new ConstraintInteger(
    nation[Nation.DANE]==drink[Drink.TEA]));
```

```
// the green house is on the left of the white house
constraints.Add(new ConstraintInteger(
    color[Color.GREEN]+1==color[Color.WHITE]));

// the green house's owner drinks coffee
constraints.Add(new ConstraintInteger(
    color[Color.GREEN]==drink[Drink.COFFEE]));

// the person who smokes Pall Mall rears birds
constraints.Add(new ConstraintInteger(
    cigarette[Cigarette.PALLMALL]==pet[Pet.BIRDS]));

// the owner of the yellow house smokes Dunhill
constraints.Add(new ConstraintInteger(
    color[Color.YELLOW]==cigarette[Cigarette.DUNHILL]));

// the man living in the center house drinks milk
constraints.Add(new ConstraintInteger(
    drink[Drink.MILK]==3));

// the Norwegian lives in the first house
constraints.Add(new ConstraintInteger(
    nation[Nation.NORWEGIAN]==1));

// the man who smokes blends lives next to the one who keeps cats
constraints.Add(new ConstraintInteger(
    (cigarette[Cigarette.BLEND]+1==pet[Pet.CATS]) |
    (cigarette[Cigarette.BLEND]-1==pet[Pet.CATS])));

// the man who keeps horses lives next to the man who smokes Dunhill
constraints.Add(new ConstraintInteger(
    (cigarette[Cigarette.DUNHILL]+1==pet[Pet.HORSES]) |
    (cigarette[Cigarette.DUNHILL]-1==pet[Pet.HORSES])));

// the owner who smokes BlueMaster drinks beer
constraints.Add(new ConstraintInteger(
    cigarette[Cigarette.BLUEMASTER]==drink[Drink.BEER]));

// the German smokes Prince
constraints.Add(new ConstraintInteger(
    nation[Nation.GERMAN]==cigarette[Cigarette.PRINCE]));

// the Norwegian lives next to the blue house
constraints.Add(new ConstraintInteger(
    (nation[Nation.NORWEGIAN]+1==color[Color.BLUE]) |
    (nation[Nation.NORWEGIAN]-1==color[Color.BLUE])));

// the man who smokes blend has a neighbor who drinks water
constraints.Add(new ConstraintInteger(
    (cigarette[Cigarette.BLEND]+1==drink[Drink.WATER]) |
    (cigarette[Cigarette.BLEND]-1==drink[Drink.WATER])));

var state = new StateInteger(vars, constraints);
var searchResult = state.Search();

for(int i = 1; i <= 5; i++) {
   Console.Write("House {0}: ", i);
   foreach(Nation item in nation.Keys)
      if(nation[item].Value == i)
         Console.Write("{0}, ", item);
   foreach(Pet item in pet.Keys)
      if(pet[item].Value == i)
```

```
            Console.Write("{0}, ", item);
        foreach(Drink item in drink.Keys)
            if(drink[item].Value == i)
                Console.Write("{0}, ", item);
        foreach(Cigarette item in cigarette.Keys)
            if(cigarette[item].Value == i)
                Console.Write("{0}, ", item);
        foreach(Color item in color.Keys)
            if(color[item].Value == i)
                Console.Write("{0} ", item);
        Console.WriteLine();
    }
```

6.3 Пример примене у пословној апликацији

Из претходних примера примећујемо да је математичке игре прилично једноставно решити у CLP. Међутим, ову парадигму можемо применити и у многим реалним проблемима. Показаћемо то на једном измишљеном примеру из реалног живота.

Компанија ACME[8] бави се израдом пројеката. Има велики број запослених, запослени раде на више пројеката истовремено и потребни су чести састанци у мањим групама, па зато организација тих састанака представља велики проблем. Компанија има на располагању неколико соба за састанке и жели да аутоматизује систем за заказивање. Радно време запослених најчешће је од 08:00 до 16:00, али постоје и изузеци. Узети у обзир и то да запослени имају паузе или обавезе ван зграде. Потребно је направити прототип апликације за заказивање састанака у току једног радног дана у првом слободном термину. Ако састанак није могуће реализовати, исписује се одговарајућа порука и не покушава заказивање у неком од наредних дана. Такође, нема података о боловањима, годишњим одморима и сличним одсуствима са радног места. Уколико прототип буде успешан, план компаније ACME је да га унапреди, интегрише и угради у Microsoft Teams.

Проблем је најбоље решити користећи принципе објектно-оријентисаног програмирања, јер је такво решење много лакше интегрисати са осталим апликацијама. За само распоређивање користићемо CLP.

Да бисмо направили апликацију, морамо прво дефинисати класе *Meeting* (садржи тему састанка, списак учесника, време почетка и краја састанка и собу у којој се одржава), *Room* (садржи само назив собе), *Person* (садржи име, почетак и крај радног времена и списак временских интервала када запослени није доступан) и *Time* (помоћна класа која садржи време почетка и краја радног времена, одсуства или састанка). Оне нису компликоване, па ћемо тај део кода изоставити.

Главна класа зове се *Scheduler* и садржи мапу запослених (кључ је име запосленог, а вредност објекат типа *Person*), листу соба и листу већ заказаних састанака.

```
public class Scheduler {
   Dictionary<string,Person> persons = new Dictionary<string,Person>();
   List<Room> rooms = new List<Room>();
   List<Meeting> meetings = new List<Meeting>();
```

Такође, у оквиру ове класе дефинисане су операције *addPerson* и *addRoom* које додају запослене и собе за састанке.

```
   public void addPerson(string name) {
      persons.Add(name, new Person(name));
   }

   public void addRoom(string name) {
      rooms.Add(new Room(name));
   }
```

Неки од запослених могу имати другачије радно време од уобичајеног (08:00-16:00), па из тог разлога правимо функцију *setWorkingTime* која конкреном запосленом мења радно време.

```
   public void setWorkingTime(string person, string from, string to) {
      persons[person].WorkingTime = new Time(from, to);
   }
```

Термине у којима запослени нису на свом радном месту уписујемо уз помоћ операција *setOutOfOffice*, *setOutOfOfficeFrom* и *setOutOfOfficeTo*.

```
   public void setOutOfOffice(string person, string from, string to) {
      persons[person].OutOfOffice.Add(new Time(from, to));
   }

   public void setOutOfOfficeFrom(string person, string from) {
      setOutOfOffice(person, from, "24:00");
   }

   public void setOutOfOfficeTo(string person, string to) {
      setOutOfOffice(person, "00:00", to);
   }
```

Нови састанак формирамо наредбом *scheduleMeeting*, у којој наводимо тему, листу учесника и предвиђено трајање састанка. Програм за сваку собу (позивајући помоћну функцију) проверава да ли је у њој могуће организовати састанак и који је најранији термин за састанак у тој соби. Алгоритам пореди термине свих соба и заказује састанак у оној у којој може најраније да се организује.

```
public Meeting? scheduleMeeting(String subject, string[] attendees,
         int duration) {

   Meeting? meeting = null;

   // for each room find first time available (if possible)
   // remember the one with minimum time
   foreach(Room room in rooms) {
      Meeting? meeting1 = scheduleMeeting(subject, attendees,
            duration, room);
      if(meeting1 == null)
         continue;
      if(meeting == null)
         meeting = meeting1;
      else if(meeting.Time.From >= meeting1.Time.From))
            meeting = meeting1;
   }

   if(meeting != null) {
      Console.WriteLine("{0} is scheduled at {1} in {2}",
         subject, meeting.Time, meeting.Room);
      meetings.Add(meeting);
   }
   else
      Console.WriteLine("{0} cannot be scheduled", subject);

   return meeting;
}
```

Помоћна функција за тражење првог могућег термина за састанак у датој соби се такође зове *scheduleMeeting*. У односу на претходну функцију, ова као додатни параметар има име собе. Она користи CLP за решавање проблема организације. Промењљиве су време почетка и завршетка састанка, а ограничења да термин састанка мора бити у складу са радним временом и паузама сваког учесника, као и са периодима када је неки од учесника на другом састанку.

```
private Meeting? scheduleMeeting(String subject, string[] attendees,
   int duration, Room room) {

     var variables = new List<VariableInteger>();
     var constraints = new List<IConstraint>();

     var start_time = new VariableInteger("start_time", 0, 1440);
     var end_time = new VariableInteger("end_time", 0, 1440);

     variables.Add(start_time);
     variables.Add(end_time);

     // connection between start and end meeting
     constraints.Add(
        new ConstraintInteger(start_time+duration==end_time));

     foreach(string attendee in attendees) {

         Person person = persons[attendee];
```

```
            // meeting time must match working hours of each attendee
            constraints.Add(new ConstraintInteger(
                  start_time>=person.WorkingTime.From));
            constraints.Add(new ConstraintInteger(
                  end_time<=person.WorkingTime.To));

            // we must check out-of-office time for each attendee
            foreach(Time tout in person.OutOfOffice)
                constraints.Add(
                   new ConstraintInteger(start_time >= tout.To
                      | end_time <= tout.From));
        }

        // we must check when attendees are on other meetings
        foreach(Meeting meeting in meetings) {

            bool found = false;

            foreach(string attendee in attendees)
                found = found | meeting.isAttendee(attendee);

            if(found)
                constraints.Add(new ConstraintInteger(
                     start_time >= meeting.Time.To
                   | end_time <= meeting.Time.From));
        }

        // room is already occupied
        foreach(Meeting meeting in meetings) {
           if(meeting.Room == room)
              constraints.Add(
                 new ConstraintInteger(start_time >= meeting.Time.To |
                                       end_time <= meeting.Time.From));
        }

        var state = new StateInteger(variables, constraints);
        var searchResult = state.Search();

        // cannot schedule meeting in this room
        if(searchResult == StateOperationResult.Unsatisfiable)
            return null;

        Meeting meeting1 = new Meeting(subject,
           new Time(start_time.Value, end_time.Value), room);

        foreach(string attendee in attendees)
            meeting1.add(persons[attendee]);

        return meeting1;
    }
```

Направљене су и две операције са именом *getMeetings* од којих једна враћа листу састанака у датој соби, а друга листу свих састанака датог запосленог. У одговору, састанци су поређани хронолошки.

Да погледамо сада како се користи програм. На почетку наведемо листу запослених и листу соба за састанке:

```
Scheduler scheduler = new Scheduler();

// add workers
scheduler.addPerson("ana");
scheduler.addPerson("bane");
scheduler.addPerson("voja");
scheduler.addPerson("gaga");
scheduler.addPerson("deki");
scheduler.addPerson("zoki");
scheduler.addPerson("zare");
scheduler.addPerson("igor");
scheduler.addPerson("jeca");
scheduler.addPerson("ceca");
scheduler.addPerson("miki");
scheduler.addPerson("mare");
scheduler.addPerson("lea");

// add rooms
scheduler.addRoom("room1");
scheduler.addRoom("room2");
```

Радно време запослених је од 8:00 до 16:00, али можемо за појединце да изменимо тај податак. Такође можемо да упишемо обавезе запослених ван зграде.

```
// workers with non-default working time
scheduler.setWorkingTime("lea", "09:00", "17:00");

// out of office
scheduler.setOutOfOffice("lea", "11:55", "12:30");
scheduler.setOutOfOfficeFrom("deki", "12:00");
scheduler.setOutOfOfficeTo("igor", "09:30");
```

Сада можемо да организујемо састанке:

```
// schedule 30min Scrum meeting with Ana, Lea and Igor
scheduler.scheduleMeeting("Scrum",
   new String[] { "ana", "lea", "igor"}, 30);

// ... and similar for all other meetings
scheduler.scheduleMeeting("Brainstorming",
   new String[] { "lea", "zoki", "zare", "ceca", "mare"}, 20);
scheduler.scheduleMeeting("Sales",
   new String[] { "miki", "mare", "voja"}, 90);
scheduler.scheduleMeeting("Management",
   new String[] { "voja", "lea", "deki", "gaga"}, 30);
scheduler.scheduleMeeting("Project 1",
   new String[] { "miki", "mare", "lea"}, 90);
scheduler.scheduleMeeting("Project 2",
   new String[] { "miki", "lea", "voja"}, 40);
scheduler.scheduleMeeting("Project 3",
   new String[] { "ana", "igor", "ceca"}, 30);
scheduler.scheduleMeeting("Project 4",
```

```
   new String[] { "jeca", "gaga"}, 130);
scheduler.scheduleMeeting("Project 5",
   new String[] { "ana", "lea", "igor"}, 330);
```

Програм на задате команде исписује следеће информације:

```
Scrum is scheduled at 09:30-10:00 in room2
Brainstorming is scheduled at 09:00-09:20 in room2
Sales is scheduled at 09:20-10:50 in room1
Management is scheduled at 10:50-11:20 in room2
Project 1 is scheduled at 12:30-14:00 in room2
Project 2 is scheduled at 14:00-14:40 in room2
Project 3 is scheduled at 10:00-10:30 in room2
Project 4 is scheduled at 11:20-13:30 in room1
Project 5 cannot be scheduled
```

Можемо једноставно да испишемо листу свих састанака по собама:

```
foreach(Room room in scheduler.Rooms) {
   Console.WriteLine("Meetings in {0}:", room);
   foreach(Meeting meeting in scheduler.getMeetings(room))
      Console.WriteLine("  {0} at {1}", meeting.Subject, meeting.Time);
}
```

Та информација се испише у следећем облику:

```
Meetings in room1:
  Sales at 09:20-10:50
  Project 4 at 11:20-13:30
Meetings in room2:
  Brainstorming at 09:00-09:20
  Scrum at 09:30-10:00
  Project 3 at 10:00-10:30
  Management at 10:50-11:20
  Project 1 at 12:30-14:00
  Project 2 at 14:00-14:40
```

На крају проверавамо и операцију која враћа листу састанака за дату особу:

```
Console.WriteLine("My meetings:");
foreach(Meeting meeting in scheduler.getMeetings("lea"))
   Console.WriteLine("  {0} at {1} in {2}", meeting.Subject,
      meeting.Time, meeting.Room);
```

Као резултат добија се:

```
My meetings:
  Brainstorming at 09:00-09:20 in room2
  Scrum at 09:30-10:00 in room2
  Management at 10:50-11:20 in room2
  Project 1 at 12:30-14:00 in room2
  Project 2 at 14:00-14:40 in room2
```

Глава 7

Примена CLP у програмским језицима C++, Python и Java

Сви CLP примери до сада направљени су у програмском језику C#. У овом поглављу показаћемо како се CLP користи у још неколико програмских језика треће генерације, конкретно у C++, Python и Java програмском језику.

7.1 CLP у програмском језику C++

За програмски језик C++ постоји *open-source* имплементација Naxos Solver[6] која је настала на Универзитету у Атини. Подржава решавање логичких проблема ограничења типа CLPFD. На званичном сајту је кратак приручник и неколико конкретних примера, па је лако направи једноставан пример.

Програм 7.1: Решавање криптоаритметичког проблема SEND + MORE = MONEY у програмском језику C++

```
#include <iostream>
#include "naxos.h"

using namespace std;
using namespace naxos;

int main(void)
{
   NsProblemManager pm;
   NsIntVar S(pm, 1, 9), E(pm, 0, 9), N(pm, 0, 9), D(pm, 0, 9),
            M(pm, 1, 9), O(pm, 0, 9), R(pm, 0, 9), Y(pm, 0, 9);
   NsIntVar send = 1000 * S + 100 * E + 10 * N + D;
   NsIntVar more = 1000 * M + 100 * O + 10 * R + E;
   NsIntVar money = 10000 * M + 1000 * O + 100 * N + 10 * E + Y;
   pm.add(send + more == money);
   NsIntVarArray letters;
   letters.push_back(S);
   letters.push_back(E);
   letters.push_back(N);
   letters.push_back(D);
```

```
   letters.push_back(M);
   letters.push_back(O);
   letters.push_back(R);
   letters.push_back(Y);
   pm.add(NsAllDiff(letters));
   pm.addGoal(new NsgLabeling(letters));
   if (pm.nextSolution()) {
      cout << "    " << send.value() << "\n"
           << " +  " << more.value() << "\n"
           << " = " << money.value() << "\n";
   }
}
```

7.2 CLP у програмском језику Python

Програмски језик Python има библиотеку за решавање логичких проблема ограничења *python-constraint*[5] која такође подржава само решавање проблема са ограничењима са коначним доменима. Аутори ове бибилиотеке су Густаво Нимејер и Себастијан Сел. Инсталира се помоћу управљача Python пакетима:

```
$ pip install python-constraint
```

Није тешко написати једноставан CLP програм.

Програм 7.2: Решавање криптоаритметичког проблема SEND + MORE = MONEY у програмском језику Python

```
import constraint

problem = constraint.Problem()

problem.addVariable("S", range(1, 10))
problem.addVariable("E", range(0, 10))
problem.addVariable("N", range(0, 10))
problem.addVariable("D", range(0, 10))
problem.addVariable("M", range(1, 10))
problem.addVariable("O", range(0, 10))
problem.addVariable("R", range(0, 10))
problem.addVariable("Y", range(0, 10))

def sum_constraint(s, e, n, d, m, o, r, y):
   if (  s*1000 + e*100 + n*10 + d)  \
     + ( m*1000 + o*100 + r*10 + e)  \
     == m*10000 + o*1000 + n*100 + e*10 + y:
        return True

problem.addConstraint(sum_constraint, "SENDMORY")
problem.addConstraint(constraint.AllDifferentConstraint())

for s in problem.getSolutions():
   print(" {}{}{}{}".format(s['S'],s['E'],s['N'],s['D']))
   print("+{}{}{}{}".format(s['M'],s['O'],s['R'],s['E']))
   print("{}{}{}{}{}".format(s['M'],s['O'],s['N'],s['E'],s['Y']))
```

7.3 CLP у програмском језику Java

За програмски језик Java постоји *open-source* библиотека Choko-solver.[3] Подржава не само целобројне промењљиве над коначним доменима, већ и реалне вредности (CLPR). Због тога користи и технику локалне претраге LNS (енг. *Large Neighborhood Search*). Библиотека се једноставно укључује са *maven* конфигурацијом:

```
        <dependency>
            <groupId>org.choco-solver</groupId>
            <artifactId>choco-solver</artifactId>
            <version>4.10.8</version>
        </dependency>
```

Иако та бибиотека пружа више могучности од осталих са којима смо радили, нисмо тестирали њене могућности, већ смо направили само једноставан пример апликације типа CLPFD.

За програмски језик Java је специфично и то да није могуће предефинисати основне рачунске операције да раде над објектима, па зато аритметички изрази не могу да се запишу једноставно као у C++ или у C#. На примеру можемо да видимо да су аутори те библиотеке нашли интересантно решење овог проблема употребом скаларног множења.

Програм 7.3: Решавање криптоаритметичког проблема SEND + MORE = MONEY у програмском језику Java

```
Model model = new Model();

IntVar S = model.intVar("S", 1, 9, false);
IntVar E = model.intVar("E", 0, 9, false);
IntVar N = model.intVar("N", 0, 9, false);
IntVar D = model.intVar("D", 0, 9, false);
IntVar M = model.intVar("M", 1, 9, false);
IntVar O = model.intVar("O", 0, 9, false);
IntVar R = model.intVar("R", 0, 9, false);
IntVar Y = model.intVar("Y", 0, 9, false);

model.allDifferent(new IntVar[]{S, E, N, D, M, O, R, Y}).post();

IntVar[] ALL = new IntVar[]{
    S, E, N, D,
    M, O, R, E,
    M, O, N, E, Y};
int[] COEFFS = new int[]{
    1000, 100, 10, 1,
    1000, 100, 10, 1,
    -10000, -1000, -100, -10, -1};
model.scalar(ALL, COEFFS, "=", 0).post();

Solver solver = model.getSolver();
solver.findSolution();

System.out.printf(" %d%d%d%d\n",
   S.getValue(), E.getValue(), N.getValue(), D.getValue());
System.out.printf("+%d%d%d%d\n",
   M.getValue(), O.getValue(), R.getValue(), E.getValue());
System.out.printf("%d%d%d%d%d\n",
   M.getValue(), O.getValue(), N.getValue(), E.getValue(), Y.getValue());
```

Глава 8

Закључак

У овом раду приказани су различити примери употребе логичког програмирања ограничења (CLP) у програмским језицима треће генерације. Иако CLP још увек није саставни део популарних програмских језика, показано је на који начин је његово коришћење могуће. Осим приказаних математичих игара, логичких загонетки и пословне апликације за заказивање састанака, ова парадигма може се применити у решавању многих других проблема математичко-логичке природе. Писање програма уз CLP је једноставно, па зато врло брзо можемо направити прототип решења. Међутим, може се десити да алгоритам буде преспор. У том случају немамо много могућности да оптимизујемо решење, за разлику језика треће генерације у којима је то могуће. Већина имплементација које сам нашла подржава искучиво рад над коначним доменима (CLPFD), а ретке су оне које подржавају рад над реалним доменима (CLPR).

Ово истраживање се може наставити у више праваца; могу се имплементирати неки нови оптимизацијски проблеми, истражити могућности рада над реалним доменима у оним програмским језицима треће генерације који то подржавају или направити сопствена имплементација CLP модула у неком програмском језику.

Литература

[1] Ivan Bratko. *Prolog programming for artificial intelligence*. Pearson education, 2001.

[2] Geeks for Geeks. Generation of programming languages. `https://www.geeksforgeeks.org/generation-programming-languages/`, 2022. [Online; accessed 22-May-2022].

[3] GitHub. Choco-solver - An Open-Source java library for constraint programming - GitHub. `https://github.com/chocoteam/choco-solver`, 2022. [Online; accessed 22-May-2022].

[4] GitHub. Decider, an open source .Net constraint programming solver. `https://github.com/lifebeyondfife/Decider`, 2022. [Online; accessed 22-May-2022].

[5] Sébastien Celles Gustavo Niemeyer. python-constraint - Constraint Solving Problem resolver for Python - GitHub. `https://github.com/python-constraint/python-constraint`, 2022. [Online; accessed 22-May-2022].

[6] Nikolaos Pothitos. Naxos Solver - A C++ Constraint Programming Library - GitHub. `https://github.com/pothitos/naxos`, 2022. [Online; accessed 22-May-2022].

[7] Tomasz Strzoda. CLP #2: Einstein's riddle. `https://itlookssoeasy.com/java/einstein-riddle/#`, 2019. [Online; accessed 22-May-2022].

[8] Wikipedia. ACME Corporation — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Acme\%20Corporation&oldid=1086892897`, 2022. [Online; accessed 22-May-2022].

[9] Wikipedia. Backtracking — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Backtracking&oldid=1077020832`, 2022. [Online; accessed 22-May-2022].

[10] Wikipedia. Constraint satisfaction problem — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Constraint\%20satisfaction\%20problem&oldid=1088971390, 2022. [Online; accessed 22-May-2022].

[11] Wikipedia. Ford–Fulkerson algorithm — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Ford\%E2\%80\%93Fulkerson\%20algorithm&oldid=1070199459, 2022. [Online; accessed 22-May-2022].

[12] Wikipedia. Local consistency — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Local\%20consistency&oldid=1082272762, 2022. [Online; accessed 22-May-2022].

[13] Wikipedia. Local search (optimization) — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Local\%20search\%20(optimization)&oldid=1075656392, 2022. [Online; accessed 22-May-2022].

[14] Wikipedia. Magic hexagon — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Magic\%20hexagon&oldid=1086155810, 2022. [Online; accessed 22-May-2022].

[15] Wikipedia. Programming language generations — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Programming\%20language\%20generations&oldid=1085967379, 2022. [Online; accessed 22-May-2022].

[16] Wikipedia. Prolog — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Prolog&oldid=1085505078, 2022. [Online; accessed 22-May-2022].

[17] Wikipedia. Sudoku — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Sudoku&oldid=1088183038, 2022. [Online; accessed 22-May-2022].

[18] Wikipedia. Verbal arithmetic — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Verbal\%20arithmetic&oldid=1088183178, 2022. [Online; accessed 22-May-2022].

[19] Wikipedia. Zebra Puzzle — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Zebra\%20Puzzle&oldid=1086311622, 2022. [Online; accessed 22-May-2022].

[20] WilliamFiset. Unweighted bipartite matching | network flow | graph theory. https://www.youtube.com/watch?v=GhjwOiJ4SqU, 2022. [Online; accessed 22-May-2022].