Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Náročnosť riešenia maľovaných krížoviek

Bakalárska práca

2016
Kristína Komanová

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Náročnosť riešenia maľovaných krížoviek

Bakalárska práca

Študijný program:	Informatika
Študijný odbor:	2508 Informatika
Školiace pracovisko:	Katedra informatiky
Školiteľ:	RNDr. Michal Forišek, PhD.

Bratislava, 2016
Kristína Komanová

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Kristína Komanová
Študijný program: informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: slovenský
Sekundárny jazyk: anglický

Názov: Náročnosť riešenia maľovaných krížoviek
Difficulty of solving Nonogram puzzles

Cieľ: Hlavným cieľom práce je skúmať náročnosť riešenia hlavolamu "maľovaná krížovka" človekom. K tomuto cieľu by sa mala práca čo najviac priblížiť postupným plnením nasledovných čiastočných cieľov:
1. získať a spracovať prehľad existujúcich teoretických výsledkov súvisiacich s problémom
2. identifikovať heuristiky používané ľuďmi pri riešení konkrétnych inštancií
3. implementovať sadu nástrojov použiteľnú pri analýze inštancií, vrátane simulácie postupov používaných ľuďmi
4. analyzovať dáta z praxe a snažiť sa identifikovať súvis medzi v praxi pozorovanou náročnosťou inštancie, jej syntaktickými parametrami, úspešnosťou konkrétnych heuristických metód a inými parametrami.

Vedúci: RNDr. Michal Forišek, PhD.
Katedra: FMFI.KI - Katedra informatiky
Vedúci katedry: doc. RNDr. Daniel Olejár, PhD.

Dátum zadania: 25.10.2015

Dátum schválenia: 27.10.2015 doc. RNDr. Daniel Olejár, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Poďakovanie: V prvom rade by som sa chcela poďakovať Mišofovi, vedúcemu mojej bakalárky, ktorý ma viedol tým správnym smerom a mal so mnou trpezlivosť. Rasťovi za poskytnuté dáta do mojej práce. Mojim rodičom, ktorí akceptovali, že na bakalárku sa ma nemajú radšej pýtať. Majke, za jej odborné či povzbudzujúce slová, ktorými sa ma snažila upokojiť, Marcelovi, Sysľovi, Usamovi, Hopkovi, rovnako ako mojim milým spolužiakom, ktorí ak sa nesnažili útočiť na môj počítač počas mojej neprítomnosti, boli mi prínosom. Vďaka patrí Admanovi, Marekovi, Peťovi, Michalovi aj Wiďovi, ktorí tu boli pre mňa vždy, keď som to potrebovala. Samozrejme sa chcem poďakovať aj (bývalému) kamarátovi Žabovi, vďaka ktorému som množstvo času, ktorý som mohla venovať bakalárke musela stráviť nad riešením úloh z TEA. No a na záver Dominike a Tomášovi, ktorí sa starali o moju psychickú pohodu.

Abstrakt

Maľované krížovky sú obľúbeným logickým hlavolamom. Aj keď pravidlá na vyriešenie nie sú zložité, vyriešenie hlavolamu nám niekedy trvá chvíľu, inokedy nad iným, rovnako veľkým, strávime hodiny. Z výpočtového hľadiska sa *Maľované krížovky* zaraďujú medzi NP-úplné problémy. Ale aj tak existuje pomerne veľká skupina krížoviek, ktoré vieme doriešiť pomocou polynomiálnych algoritmov. V práci sa budeme sústreďovať na súvislosti medzi riešením človekom a algoritmom. Krížovky sa pokúsime vyriešiť viacerými metódami, ktorých výsledky použijeme na predikciu času vyriešenia krížovky človekom, čo porovnáme s reálnymi dátami.

Kľúčové slová: maľované krížovky, NP-úplné problémy, riešenie človekom, lineárna regresia

Abstract

Nonograms are popular logic puzzle. Though rules are not difficult and solution of some may take a moment, sometimes solution of same sized puzzle may take hours. From computing point of view nonograms belong to NP-complete problems. Still relatively big set of nonograms can be solved with polynomial algorithm. This thesis will focus on connections between solution used by human and algorithm. We will try to solve nonograms with multiple methods, that will be used for prediction of time, that human would need to solve puzzle and compare results with real data.

Keywords: nonograms, NP-complete problems, human behavior modeling, linear regression

Obsah

Zoznam obrázkov

Úvod

Bakalárska práca vznikla z podnetu prevádzkovateľa portálu `Griddlers.net`, čo je stránka na online riešenie rôznych logických hlavolamov. Používatelia si tu môžu vytvoriť profil, čím získajú možnosť ukladať si nedoriešené krížovky, či dostávať za vyriešené krížovky body. Práve tieto body celkom nekorešpondovali s dátami o priemerných časoch vyriešenia jednotlivých krížoviek. Tie doteraz počítali lineárne, v závislosti od veľkosti a času riešenia krížovky ich riešiacim algoritmom. Preto mi dáta, ktoré mali k dispozícii poskytli a ja som mohla skúsiť iné prístupy, ktoré by čas vedeli predpovedať presnejšie.

V práci som sa snažila nájsť postup riešenia maľovaných krížoviek, ktorý by sa čo najviac podobal prístupu, aký používa človek. Ukázalo sa, že dáta, ktoré získame po riešení krížoviek sadou logických pravidiel, sa dajú celkom dobre použiť na predpovedanie času riešenia.

Naša práca bude pozostávať z niekoľkých častí. V prvej kapitole predstavíme *maľované krížovky*, ukážeme si základné pravidlá, ktoré musí spĺňať vyriešená krížovka a jej rôzne variácie. Druhá sa pozerá na maľovanú krížovku z výpočtového hľadiska, prezentujeme niekoľko typov algoritmov, akými možno získať riešenie krížovky. Hoc sa maľované krížovky zaraďujú k NP-úplným problémom, existujú polynomiálne algoritmy, ktorými vieme vyriešiť veľa krížoviek, hlavne také, ktoré sa nachádzajú bežne v časopisoch. Jeden takýto algoritmus si bližšie popíšeme. Tretia kapitola prezentuje logické pravidlá, aké približne používa človek, keď krížovku rieši. Vyriešenie krížovky pomocou nich by nám teda malo dať lepšiu predstavu o náročnosti vyriešenia krížovky človekom. V štvrtej kapitole popisujeme metódy, akými sme spracovali vstupné dáta a riešili krížovky, ktoré máme k dispozícii. Keďže krížoviek, na ktorých môžeme naše techniky skúšať je cez 40 000, vyskúšali sme na predpoveď času použiť techniky strojového učenia. Konkrétne metódy, ktoré sme používali sú popísané v piatej kapitole, kde sú aj výsledky, ku ktorým sme sa dopracovali.

Kapitola 1

Maľované krížovky

V tejto kapitole sa zoznámime s maľovanou krížovkou, jej variáciami a základnými pravidlami riešenia.

1.1 Logické hlavolamy

Maľované krížovky, sudoku, kakuro, mozaika, lampy, zebra. Všetky tieto krížovky majú čosi spoločné. Niektorým ľuďom spríjemňujú voľné chvíle a ich riešenie pomáha rozvíjať logické myslenie. Čo ešte majú spoločné je, že sú to hlavolamy, ktorých zadaním sú čísla. Niektoré majú v tých číslach zakódovaný obrázok, ktorý sa zjaví až po vylúštení krížovky. Sú v rôznych veľkostiach, obtiažnostiach, farbách a nájsť ich môžeme v časopisoch, zvyčajne pomenovaných po samotnej krížovke. Takisto sú vydávané v knižnej podobe, ale existujú aj špeciálne stránky, ktoré ponúkajú členstvo a možnosť si nedokončené krížovky uložiť či pokračovať v riešení neskôr. Rovnako ako aplikácie pre smartfóny, či iné herné prístroje. Každoročne sa tiež organizujú olympiády v riešení jednotlivých hlavolamov, kde sa súťažiaci snažia krížovku vyriešiť za čo najkratší čas.

1.2 História

Maľovaná krížovka (nonogram, griddler - angl.) je japonským hlavolamom vynájdeným v roku 1988 pod názvom "Window Art Puzzles". S názvom Nonograms prišiel týždenník *The Sunday Telegraph* v roku 1990 vo Veľkej Británii, kde sa tieto hlavolamy začali pravidelne publikovať. V roku 1998 nechali svojich čitateľov vymyslieť nové meno pre krížovky. Vyhral názov *Griddlers*, no tieto hlavolamy vo svete dostávajú rôzne pomenovania: Japanese puzzles, Paint by Numbers, Pic-a-Pix, Picross a iné. V roku 1995 bola maľovaná krížovka implementovaná ako jedna z hier pre Nintendo, ktorá ponúkala maľované krížovky do veľkosti *25x20*. Neskôr dokonca vyšla aj vo verzii 3D. (Viac o histórii a vývoji maľovaných krížoviek môžte nájsť na stránkach [1] a [2])

1.3 Pravidlá

Zadanie Maľovanej krížovky spočíva v prázdnej štvorčekovej sieti, kde sú hrubšou čiarou zvýraznené štvorce *5x5* a na začiatku každého riadka a stĺpca je napísaných niekoľko čísel *(budeme ich nazývať **indície**)*. Naším cieľom je štvorčekovú sieť podľa zadania vyfarbiť a odmenou sa stane vzniknutý obrázok. Tento obrázok musí spĺňať nasledovné podmienky: nech je *maľovaná krížovka* veľkosti m x n. Potom

- každý zo štvorčekov musí byť vyfarbený (čierny) alebo prázdny (biely)
- ak riadok/stĺpec obsahuje k indícií: $s_1, s_2, ..., s_k$, potom musí obsahovať k súvisle vyfarbených blokov v danom riadku/stĺpci, pričom prvý má dĺžku s_1, druhý dĺžku s_2, atď.
- medzi každými dvoma blokmi s_i a s_{i+1} je minimálne jeden štvorček prázdny [5]

Zásadným pravidlom pri riešení krížovky je, že nikdy nehádame. Každý štvorček, ktorý zafarbíme alebo necháme prázdny (zvyčajne si ho riešitelia označujú krížikom alebo bodkou), je výsledkom nejakých logických záverov, že to políčko musí by určite zafarbené alebo naopak, že je určite prázdne.

1.4 Varianty maľovaných krížoviek

Trojuholníkové

Trojuholníkové maľované krížovky sa od obyčajných odlišujú tým, že niektoré štvorčeky sú vyplnené len do polovice. Štvorček sa vždy delí na polovicu diagonálne, takým smerom, ako je to uvedené v legende krížovky. Tieto trojuholníky sa môžu vyskytovať len na konci nejakého súvislého bloku, a ak sú bloky jasne od seba odlíšiteľné, nemusí byť medzi nimi voľné políčko. Prítomnosť koncových trojuholníkov často pomáha rýchlejšie určiť začiatok či koniec jednotlivých indícií, čím sa krížovka napriek navonok zložitejšiemu zadaniu stáva ľahšie riešiteľnou.

Farebné

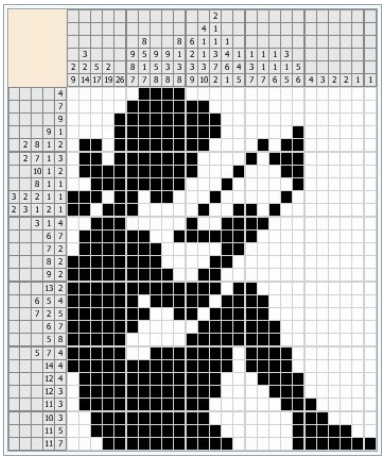
V legende sú indície písané farbou, akou sa v krížovke má tento blok vyfarbiť. Opäť sa ruší pravidlo minimálne jedného prázdneho štvorčeka medzi blokmi, avšak len ak bloky nasledujúce za sebou majú rozdielne farby. Ak je ich farba rovnaká, voľný štvorček tam musí byť, aby sa bloky dali od seba odlíšiť.

Triddlers

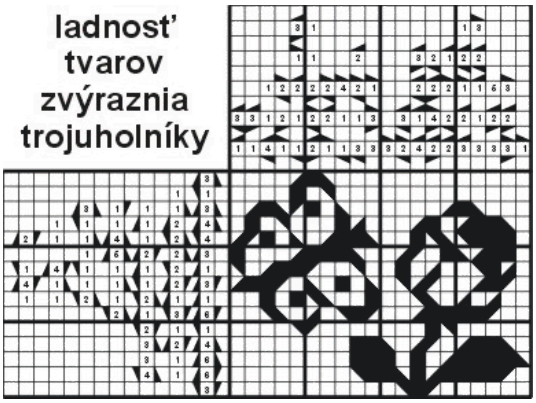
Tentokrát sa štvorčeková sieť mení za trojuholníkovú a legenda ku krížovke je napísaná z troch strán. Platí pravidlo jedného voľného trojuholníka medzi indíciami. Aj keď sa tieto krížovky nevyrábajú veľmi veľké (v porovnaní s ostatnými), horšie sa sleduje línia riadkov.

Zebra

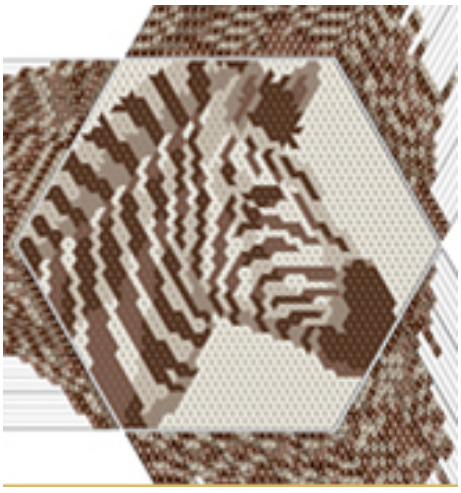
Pri klasickej maľovanej krížovke nám legenda určovala bloky na vyfarbenie. Pri *zebre* sú v legende písané niektoré indície na biele a iné na čierne pozadie. Tie na čiernom pozadí nám určujú veľkosti blokov určených na vyfarbenie, tak ako pri klasickej maľovanej krížovke. Na bielom pozadí sú naopak veľkosti za sebou nasledujúcich prázdnych štvorčekov. Tento typ krížoviek je asi najťažší na riešenie, aj pri malej krížovke sa dá veľmi ľahko pomýliť. Princíp striedania čísel pre vyfarbené a voľné políčka totiž veľmi mätie riešiteľa.



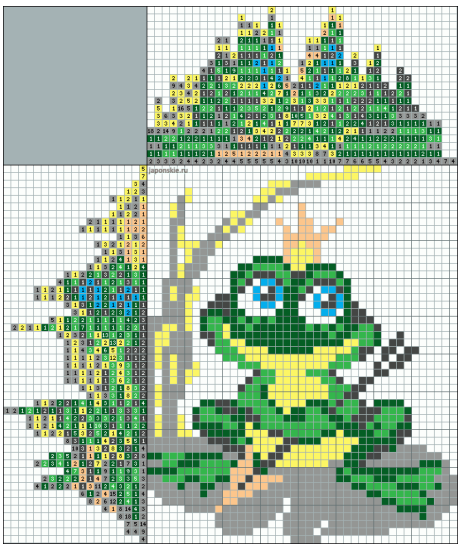
Obr. 1.1: Klasická maľovaná krížovka



Obr. 1.2: Trojuholníková krížovka



Obr. 1.3: Triddlers



Obr. 1.4: Farebná krížovka

Kapitola 2

Algoritmy riešenia

V tejto kapitole si predstavíme maľované krížovky z výpočtového hľadiska. Ukážeme si niekoľko typov algoritmov, akým je možné dospieť k riešeniu maľovanej krížovky.

2.1 Výpočtová zložitosť

Maľované krížovky ako aj veľa iných podobných logických hlavolamov zaraďujeme do NP-úplných problémov. To znamená, že ak máme riešenie, existuje algoritmus, ktorým jeho správnosť vieme overiť v polynomiálnom čase, ale algoritmus, ktorý by riešenie generoval v polynomiálnom čase, by dokazoval P=NP. Dôkaz pre NP-úplnosť nájdeme v [10]. To, či existuje ďalšie riešenie pre krížovku, ak už jedno máme *(Another solution problem)*, skúmali v [9] a podarilo sa im dokázať, že aj táto otázka patrí medzi NP-úplné problémy.

2.2 Riešenie hrubou silou

Pre každý riadok vygenerujeme všetky možné uloženia čiernych políčok. Následne kontrolujeme, či takéto riešenie je validné aj pre stĺpce. Takýmto algoritmom vieme nájsť riešenie zakaždým, pokiaľ existuje. Ak má krížovka viac riešení, vieme prísť na všetky. Avšak toto riešenie je časovo zložitejšie *(exponenciálna časová zložitosť)* a počas overovania nenaberáme žiadne nové informácie, ktoré by nám výpočet mohli uľahčiť. (Viac v článkoch [5], [12]).

2.3 Genetický algoritmus

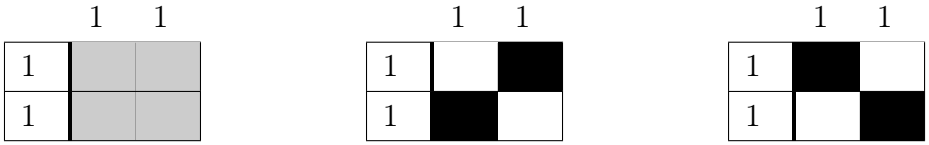
Ako už z názvu vyplýva, jeho podstata je založená na princípe biologickej evolúcie. Používajú sa techniky, ktoré napodobňujú evolučné procesy známe z biológie - dedičnosť, mutácia, prirodzený výber a kríženie. Takýmto algoritmom sa podarilo riešiť

krížovky oveľa rýchlejšie ako predchádzajúcou metódou (viac v [12]),keďže sa však nejedná o exaktný prístup, krížovky nie sú vždy doriešené správne.

Iteratívne postupy

Ofarbenie štvorčekov nehádame, ale postupne zafarbujeme tie, o ktorých sme zistili, že môžu byť len jednej farby. Pri týchto spôsoboch sa pozeráme vždy len na 1 riadok krížovky a na základe neho sa snažíme pridať novú informáciu, ktorá by pomohla k doriešeniu krížovky. Iteruje sa nad riadkami a stĺpcami krížovky. Jedna iterácia je prejdenie cez všetky riadky a stĺpce 1-krát. Môžeme si teda definovať krížovky, ktorým stačí tento iteratívny spôsob na vyriešenie. Takéto krížovky veľkosti mxn vieme vyriešiť za najviac $m \cdot n$ iterácii, pretože v každej iterácii musíme aspoň o 1 novom štvorčeku vedieť povedať, či má byť zafarbený alebo je prázdny, inak by sme sa dostali do cyklu a krížovku nevyriešili. Z toho vyplýva, že všetky krížovky, ktoré takto vieme doriešiť musia mať len 1 riešenie. Keby mali viac, nastal by spor s tým, že ofarbujeme len políčka o ktorých vieme s istotou povedať akej farby budú a zároveň by bola krížovka doriešená.

Obr. 2.1: Príklad krížovky, ktorá sa nedá doriešiť Iterujúcou metódou, ale má len 1 riešenie.



Obr. 2.2: Príklad krížovky, ktorá má 2 riešenia.

2.4 Logické pravidlá

Logické pravidlá napodobňujú riešenie krížovky ľuďmi. Využívajú techniky, aké používa človek pri riešení krížovky. Každé z nich rieši nejakú konkrétnu situáciu, takže keď iterujeme cez riadky a stĺpce, pokúsime sa pravidlá postupne aplikovať. Týmto pravidlám sa budeme podrobnejšie venovať v samostatnej kapitole 3.

2.4.1 Modifikácia

V [5] tieto pravidlá použili spolu s prehľadávaním do hĺbky. Ak už nemožno použiť žiadne z logických pravidiel, skúsili vygenerovať všetky riešenia pre prvý nedokončený riadok a ďalej pokračovať v používaní logických pravidiel. Takto postupovali, kým nedošlo k chybe (situácia, kedy nesedí ofarbenie riadkov a stĺpcov) alebo opäť nevieme aplikovať žiadne z pravidiel. Docielili, že sa im podarilo doriešiť každú krížovku a zároveň to bol dostatočne efektívny algoritmus oproti algoritmom používajúcim hrubú silu.

2.5 Dynamické programovanie

Tento postup je založený na princípe, kedy sa snažíme z aktuálnej konfigurácie pre jeden riadok povedať čo najviac novej informácie. Toto riešenie má polynomiálnu časovú zložitosť. Spolu so školiteľom sme takýto algoritmus zostrojili a následne som ho vo svojej práci implementovala.
Pri riešení postupujeme nasledovne. Vstupom je aktuálne riešenie jedného riadka a zoznam prislúchajúcich indícii.

Krok 1: Ako i označíme počet indícii ktoré v aktuálnom spracovávaní berieme do úvahy a ako j počet políčok, s ktorými pracujeme. Platné rozsahy indexov sú $i \in \langle 0, p_i \rangle$ a $j \in \langle 0, n \rangle$, kde p_i je počet indícií v danom riadku a n je počet stĺpcov. Pre každú dvojicu $P_1(i, j)$ potrebujeme zistiť, či to je validná konfigurácia, inak povedané, či sa prvých i indícii dá uložiť na prvých j políčok. Začíname s $i = 0$ aj $j = 0$, a postupne iterujeme až k maximálnym hodnotám.
Triviálny prípad - ak $i = 0$ tak $P_1(i, j)$ je *true* práve vtedy, keď na prvých j políčkach nie je žiadne čierne. Aby sme zakaždým nemuseli pozerať všetkých j políčok, využijeme už predpočítané hodnoty. Výsledok bude rovnaký, ako na predchádzajúcom políčku, ak posledné nie je čierne. V tom prípade je odpoveď *false*. Odpoveď na otázku $P_1(0, 0)$ je *true*.
Rekurzia - pozeráme sa na posledné políčko v spracovávanom prefixe.

Ak je **biele**, potom odpoveď na otázku $P_1(i, j) = P_1(i, j - 1)$.

Odpoveď na otázku $P_1(1, 5)$ je *true*. Nezaujíma nás v tejto chvíli, či sa ostatné indície do zvyšku riadka zmestia, pozeráme sa len na prvých j políčok, a či tam môže byť prvých i indícií.

Ak je toto políčko **čierne**, pokúsime sa uložiť poslednú indíciu od tohto miesta doľava (to je možné, ak sa na tomto intervale nenachádzajú biele políčka) a odpoveď

bude *konjunkcia* s $P_1(i-1, j-c-1)$, kde c je dĺžka poslednej indície, ktorú sme do riadka umiestnili a pred ním musí byť minimálne jedno políčko biele.

Ak položíme otázku $P_1(1, 4)$, dostali by odpoveď *false* už pri kontrole, či sa tam dá uložiť indícia 1. V prípade $P_1(1, 5)$ je však už odpoveď *true*.

Poslednou možnosťou je, že políčko je zatiaľ **neurčené**. Vtedy je odpoveďou *disjunkcia* prípadov, kedy by bolo políčko biele alebo čierne. Celé to prebieha nasledovne:

```
public void step1(int num, Inicializacia inic) {
      for (int i = 0; i < inic.zadanie.get(num).size() + 1; i++) {
          for (int j = 0; j <= inic.p_stlpcov; j++) {
              if (i == 0) {
                  if (j == 0) { //triviálny prípad
                      step_1[i][j] = true; continue;
                      }
                  if (inic.riesenie.get(num).get(j - 1).value == 1) {
                      step_1[i][j] = false; continue;
                  } else {
                      step_1[i][j] = step_1[i][j - 1]; continue;
                  }
              }
              if (j == 0) {
                  if (inic.zadanie.get(num).get(i - 1) != 0) {
                      step_1[i][j] = false; continue;//ak má krížovka nulový riadok
                  } else {
                      step_1[i][j] = true; continue;
                  }
              }
              switch (inic.riesenie.get(num).get(j - 1).value) {
                  case 0:
                      step_1[i][j] = step_1[i][j - 1];
                      break;
                  case 1:
                      step_1[i][j] = uloz_indiciu(num, i, j, inic);
                      break;
                  case 3:
                      step_1[i][j] = step_1[i][j - 1] || uloz_indiciu(num, i, j,
                          inic);
                      break;
              }
          }
      }
  }
```

Každú odpoveď si postupne ukladáme, takže odpoveď na už vypočítanú otázku získame v čase $O(1)$. Ak ukladáme indíciu, tak musíme pozrieť interval, či sa na ňom nenachádzajú biele políčka. To vieme v čase $O(c)$, kde c_i je dĺžka indície i. Ak by sme sa otázkou pýtali na konfiguráciu mimo rozsahu, odpoveďou je $false$.

Krok 2: Uplatníme rovnaký prístup ako v kroku 1, len s tým rozdielom, že budeme hľadať platné konfigurácie z opačnej strany. Takže otázka $P_2(i, j)$ teraz znamená, či posledných i indícii sa dá uložiť na posledných j políčok.

Krok 3: Využijeme predpočítané prefixové (P_1) a sufixové (P_2) hodnoty z krokov 1 a 2. V tomto kroku, pre každé doteraz neurčené políčko v riešení zisťujeme, či môže byť biele. Povedzme, že sa pýtame na políčko j. Ak môže byť biele, tak musí existovať také i, že obe otázky $P_1(i, j-1), P_2(p_i - i, n-j)$ sú $true$. Odpovede si ukladáme do samostatného poľa, takže nakoniec máme pre každé políčko označené, či môže byť biele alebo nie. V najhoršom prípade bude krok 3 trvať $O(n \cdot (n+1))$. To je v prípade, že sú všetky políčka v riešení doteraz neurčené a musíme sa spýtať na všetky možnosti i (a zakaždým by sme museli dostať odpoveď, že to nie je validné uloženie - to je v prípade, keď indície úplne pokryjú riadok).

Krok 4: Teraz zistíme, ktoré políčka môžu byť čierne. Na začiatku si inicializujeme celé pole na $false$ hodnoty. Iterujeme cez indície. Vnútorný cyklus prechádza cez stĺpce riadka. Ako prvé zistíme, či v aktuálnom stĺpci môže začínať indícia. Skontrolujeme, či sa tam indícia vôbec zmestí, či predchádzajúce políčko nie je čierne, či sa biele políčka nevyskytujú na intervale, kam chceme uložiť indíciu. A nakoniec skontrolujeme, či vieme zvyšné indície validne umiestniť pred a za túto indíciu. Ak políčko j s indíciou i všetky podmienky spĺňajú, v poli označíme všetky políčka, kde umiestňujeme indíciu na $true$. Ak v hociktorom kroku podmienkou neprejdeme, pokračujeme ďalšou iteráciou.

```
public void step4(int num, Inicializacia inic) {

    for (int w = 0; w < inic.zadanie.get(num).size(); w++) {//w==ktorú indíciu
        spracovávame
        int indi = inic.zadanie.get(num).get(w);
        for (int i = 0; i < inic.p_stlpcov; i++) {

            if (i + indi > inic.p_stlpcov) {
                break; //indícia sa tam ani nezmestí
            }
            if (!(i == 0 || inic.riesenie.get(num).get(i - 1).value != 1)) {
                continue;
            }
            if (!(i + indi >= inic.p_stlpcov || inic.riesenie.get(num).get(i +
                indi).value != 1)) {
                continue;
            }              //za uloženou indíciou môžeme dať biele políčko
```

```
                if (!(super.najdi_biele(num, i, i + indi - 1, inic).isEmpty())) {
                    continue;
                }
                if (step_1[w][Math.max(0, i - 1)]
                        && step_2[inic.zadanie.get(num).size() - w - 1][Math.max(0,
                            inic.p_stlpcov - indi - i - 1)]) {
                    Arrays.fill(c_black, i, i + indi, true);
                }
            }
        }
    }
```

Krok 5: V poslednom kroku sa pozeráme na doposiaľ neurčené políčka a ak to políčko môže byť iba čierne alebo naopak len biele, pričom používame výsledky z krokov 3 a 4, tak ho tak označíme aj v konečnom riešení.

Tu je ukážka, ako predpočítavame výsledky v jednotlivých krokoch:

	j =	0	1	2	3	4	5	6	7	8	9	10
	2 3							■				
krok 1	i = 0 :	✓	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗
	i = 1 :	✗	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓
	i = 2 :	✗	✗	✗	✗	✗	✗	✓	✓	✓	✓	✓
krok 2	i = 0 :	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗
	i = 1 :	✗	✗	✗	✓	✓	✗	✗	✓	✓	✓	✓
	i = 2 :	✗	✗	✗	✗	✗	✗	✓	✓	✓	✓	✓
	krok 3:	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	
	krok 4:	✓	✓	✗	✓	✓	✓	✗	✓	✓	✓	
	výsledok :					■	■					

Obr. 2.3: Ukážka priebehu algoritmu Dynamické programovanie

V prvom riadku máme zadanie aj s jeho aktuálnym riešením. Krok 1: ak je $i = 0$, tak je odpoveď *true*, kým sa neobjavilo čierne políčko. Ak máme $i = 1$, tak sa snažíme uložiť 1. indíciu, ktorá má dĺžku 2. Na to treba aspoň 2 políčka a keďže je v riadku iba 1 čierne políčko, všetky ostatné volania $P_1(1, j)$, kde $j > 6$, vracajú hodnotu $P_1(1, j-1)$, ktorá je *true*. Ak máme $i = 2$, tak to je rovnaký prístup, ale prvýkrát sa indície podarí uložiť až na 6. políčku. Krok 2 je rovnaký, len z opačnej strany. Ukážme si teda aspoň $i = 1$. Indícia, ktorú teraz ukladáme má dĺžku 3. Na uloženie teda potrebujeme aspoň 3 políčka. Políčka $8 - 10$ sú zatiaľ neurčené, preto to je možné. 7. políčko je biele, takže odpoveďou je $P_2(1, 3) =$ *true*. 6. políčko je však čierne, a indíciu 3 chceme uložiť na políčka $6 - 8$. To nie možné, lebo sa tu nachádza biele políčko. Teda toto

volanie vracia *false*. Políčko 5 je opäť neznáme, vraciame teda disjunkciu prípadov, kedy by bolo *biele* a *čierne*. Pri bielom je odpoveďou $P_2(1,6)$, čo je false a ak je čierne, pokúsime sa indíciu uložiť. To sa nám znova nepodarí. Až pri otázke $P_2(1,7)$ je odpoveď opäť true. Krok 3: Odpoveď *false* na políčku 5 sme dostali nasledovne: majme všetky indície za týmto políčkom. Takže sa pozrieme na odpovede $P_1(0,4) = true$ a $P_2(2,5) = false$. Ďalšie rozdelenie je $P_1(1,4) = true$ a $P_2(1,5) = false$. Poslednou možnosťou je $P_1(2,4)$ *false* a $P_2(0,5) = false$. Ani v jednom prípade neboli obe rozdelenia *true*, preto na tomto mieste nemôže byť biele políčko a zaznačíme si hodnotu *false*. V kroku 4 si vlastne zaznačujeme, kde všade sa môže objaviť biele políčko. 1. indíciu ukladáme pri iterácii keď i = 2 a i = 6, vtedy sa pýtame na $P_2(1,7)$ a $P_2(1,3)$. Pri zvyšných, sa buď nepodarilo indíciu uložiť alebo otázka bola *false*. 2. indíciu uložíme pri i = 6 (otázka $P_1(1,2)$) a i = 10 (otázka $P_1(1,6)$). Krok 5 je už jednoduchý. Na indexoch kde sa nachádzajú rozdielne hodnoty a v riešení je nedoriešené políčko, zafarbíme políčka podľa toho, kde je hodnota *true*.

Kapitola 3

Logické pravidlá

Túto kapitolu venujem prezentovaniu logických pravidiel, ktoré som implementovala v jazyku **JAVA** a pomocou nich riešila maľované krížovky. Hľadala som niečo, čo by sa približovalo správaniu ľudí, keď riešia maľované krížovky. Štvorčeky zafarbujú nabielo alebo načierno na základe nejakých logických záverov, a presne to sa snažia zachytávať tieto pravidlá. Mnohé z pravidiel však riešia veci, ktoré robí človek automaticky, bez toho, aby si to uvedomil. Algoritmické riešenie to ale musí mať presne zadefinované.

V článku [5] prezentovali sadu logických pravidiel, ktorými skúšali vyriešiť maľované krížovky. Im však išlo o vytvorenie efektívneho algoritmu na riešenie krížoviek. My ju teraz použijeme na získavanie informácií o zložitosti jednotlivých krížoviek. Oproti článku [5], kde jednotlivé pravidlá popisovali matematickými definíciami, sa snažíme ukázať niektoré krajné prípady, do ktorých sa krížovka môže dostať. Prezentujeme jednotlivé správanie pravidiel v týchto prípadoch. S väčšinou situácii sme sa museli vysporiadať počas implementácie pravidiel, kedy sme museli rozhodnúť, aké správanie je v danej chvíli vhodnejšie.

3.1 Označenie

Pre jednoznačnosť si hneď na začiatku zadefinujme pojmy, akými jednotlivé časti maľovaných krížoviek budeme ďalej nazývať:

- **krížovka** - skrátené označenie pre maľovanú krížovku
- **indície** - označenie pre jednotlivé čísla v zadaní
- **čierne políčka** - políčka, o ktorých v krížovke vieme, že budú vyfarbené
- **biele políčka** - políčka, o ktorých vieme, že budú prázdne

- **neznáme políčka** - tie políčka, o ktorých zatiaľ nevieme povedať, či budú vyfarbené alebo prázdne. Na začiatku majú všetky túto hodnotu.
- **hranice indícií** - políčka, na ktoré môžu jednotlivé indície siahať
- **sekvencia bielych/čiernych** - súvislý rad za sebou vyfarbených políčok tej istej farby, v rovnakom riadku.
- **blok** - výsledná sekvencia patriaca konkrétnej indícii
- c_{is},c_{ie} - označenie prvého a posledného políčka, ktoré patrí do hraníc indície i
- LB_i - dĺžka indície i.

3.2 Inicializácia

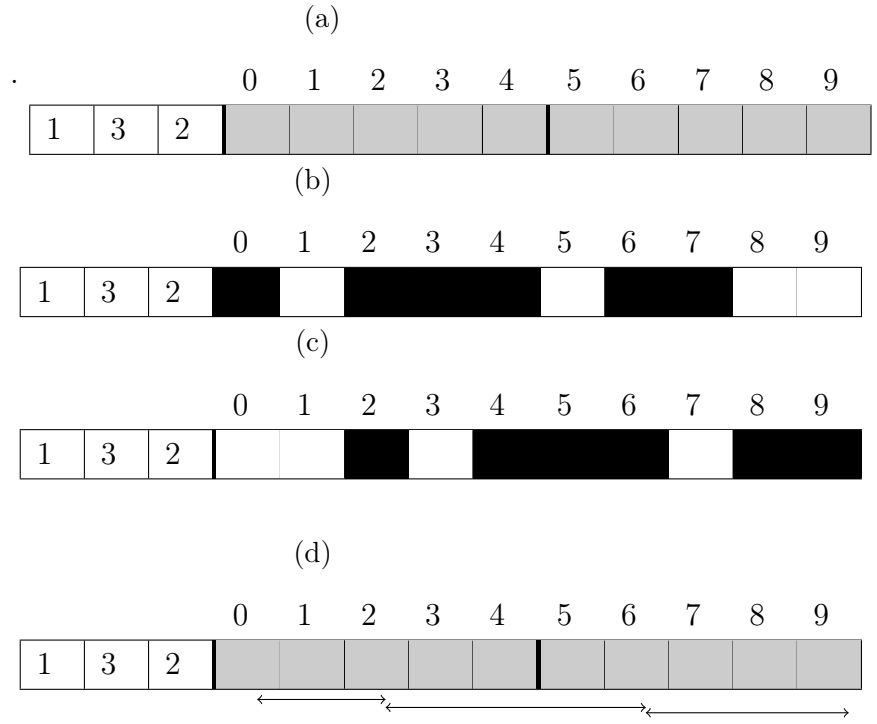
Pri riešení krížoviek týmito logickými pravidlami si potrebujeme pamätať množstvo vecí. To, čo človek robí spôsobom *pozriem-vidím* my musíme explicitne zadefinovať. Preto budeme mať štruktúru *Inicializácia*, v ktorej si uložíme všetky potrebné informácie.

```
public class Inicializacia {
    public int ID_nono;
    public ArrayList<ArrayList<Integer>> zadanie;
    public int p_stlpcov;
    public int[][][] pole_hodnot;
    public ArrayList<ArrayList<MyInt>> riesenie = new ArrayList<>();

    public Inicializacia(ArrayList<ArrayList<Integer>> zadanie, int p_stlpcov,
          int ID) {/*vytvára stĺpcovú štruktúru*/}

    public Inicializacia(ArrayList<ArrayList<Integer>> zadanie, int p_stlpcov,
          Inicializacia rr) {/*vytvára riadkovú štruktúru*/}
}
```

Ako si môžeme všimnúť, máme 2 konštruktory na vytváranie *Inicializácie*. Je to kvôli tomu, že potrebujeme, aby riadky a stĺpce zdieľali to isté *riešenie* a mohli ho upravovať. Na to, aby sme mohli krížovku vyriešiť potrebujeme inštancie oboch typov. Obe obsahujú rovnaké údaje, takže odteraz ak budeme vravieť o *riadku krížovky*, môžeme týmto termínom pomenovať aj stĺpec krížovky. V konštruktore si inicializujeme *riešenie* na počiatočné hodnoty reprezentujúce *neznáme políčko* (alebo len nastavíme referenciu na *stĺpcové riešenie*). Taktiež nastavíme *hranice indícii*. Pre každú indíciu si uložíme číslo políčka, kde môže začať a kde skončiť. Je to reprezentácia toho, ako by to

vyzeralo, keby každé číslo bolo najviac vľavo alebo najviac vpravo, ako mu to zadanie pre jednotlivé riadky dovoľuje.

Na obrázku (b) je uloženie indícií najviac vľavo, na obrázku (c) zase najpravejšie uloženie. Hranice jednotlivých indícií teda sú: $[0, 2]$, $[2, 6]$ a $[6, 9]$ a budeme ich prezentovať tak, ako je ukázané na obrázku (d), šípkami pod príslušnými políčkami



Obr. 3.1: *Inicializácia* hraníc indícii krížovky.

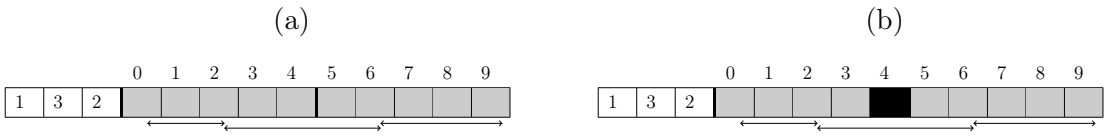
3.3 Pravidlá

Pravidlá sú rozdelené do 3 skupín. V prvej *(1 - 5)* máme pravidlá, ktoré sa snažia zistiť, či nejaké políčko možno zafarbiť nabielo alebo načierno. Do druhej skupiny *(6 - 8)* sú zaradené pravidlá, ktoré nám len aktualizujú hranice indícií, ale nič nezafarbujú. A do tej tretej *(9 - 13)* sa zaradili pravidlá, ktoré sú o trochu zložitejšie. Vedia sa aplikovať len ak sa jednoznačne priradí sekvencia k indícii. Dôležité je ešte podotknúť, že každé toto pravidlo sa vykonáva tak, že sa pozerá len na hodnoty v jednom riadku.

3.3.1 Prienik - Pravidlo 1

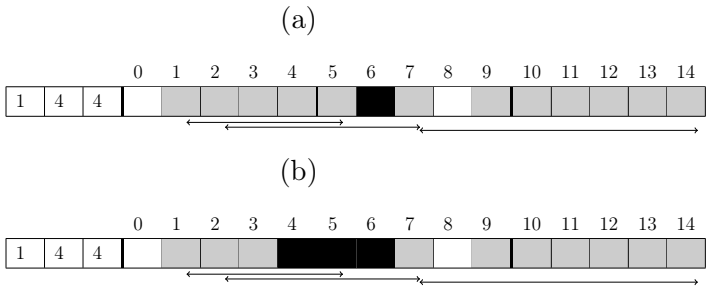
Ako už naznačuje názov, ide o pravidlo zaoberajúce sa prienikom všetkých možných uložení danej indície v riadku. Budeme ho počítať z hodnôt, ktoré sme si na začiatku inicializovali. Platí, že políčko c_j je vyfarbené, ak $r_{is} + u \leq j \leq r_{ie} - u$, kde $u =$

$(r_{ie} - r_{is} + 1) - LB_i$. Princíp tohto pravidla je rovnaký ako pri inicializácii, ale teraz nám už netreba prechádzať celý riadok. Stačia nám aktuálne hranice jednej indície, z ktorých vieme povedať, či nejaké políčka vieme s istotou zafarbiť.



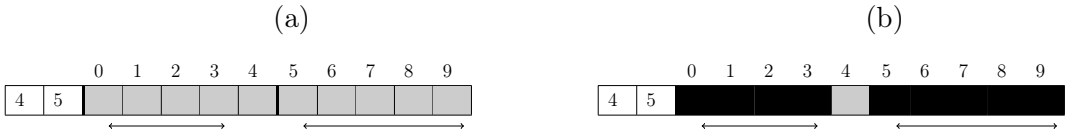
Obr. 3.2: Použitie pravidla *Prienik*

Riadok s hranicami $\{0, 2\}$, $\{2, 6\}$, $\{6, 9\}$. **(a)** ukazuje vstup pre pravidlo a **(b)** ukazuje riadok po aplikovaní.



Obr. 3.3: Použitie pravidla *Prienik*

Indície majú hranice $\{1, 5\}$, $\{2, 7\}$, $\{7, 14\}$. Môžeme si všimnúť, že pravidlo neprestavuje hranice, ani si nijak nevšíma zle nastavené hranice.

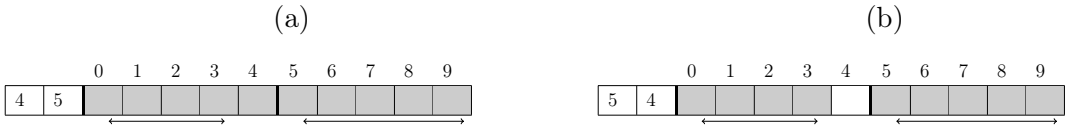


Obr. 3.4: Použitie pravidla *Prienik*.

Situácia, kedy je celý riadok vyplnený, až na políčko 4, ktoré zostáva neurčené.

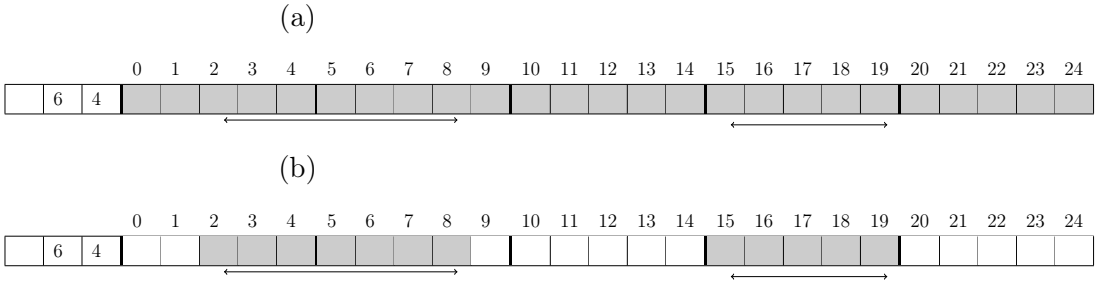
3.3.2 Jednoduché medzery - Pravidlo 2

Toto pravidlo vyfarbí na bielo všetky políčka, ktoré nepatria do hraníc žiadnej indície.



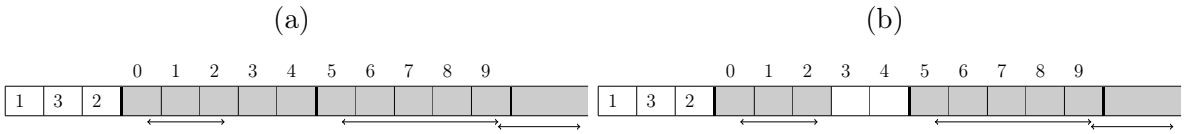
Obr. 3.5: Použitie pravidla *Jednoduché medzery*.

Môžeme si všimnúť, že je akýmsi doplnkom k pravidlu *Prienik*.



Obr. 3.6: Použitie pravidla *Jednoduché medzery*.

Tento riadok pokrýva všetky možnosti, ktoré pravidlo *Jednoduché medzery* rieši. Na bielo zafarbí štvorčeky pred prvou indíciou, medzi indíciami a za poslednou.

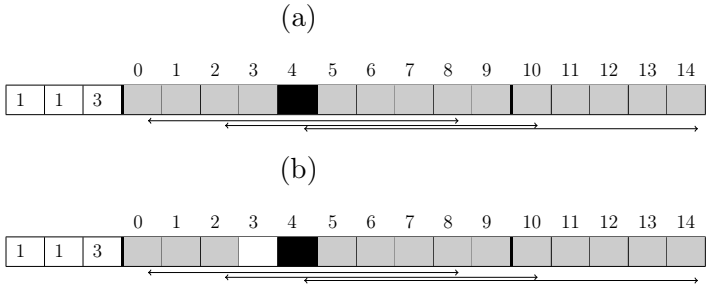


Obr. 3.7: Použitie pravidla *Jednoduché medzery*.

Tu sa pravidlo aplikuje len na štvorčeky medzi indíciami.

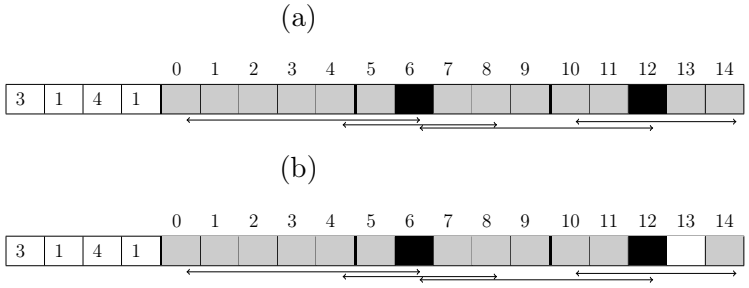
3.3.3 Jednotky - Pravidlo 3

Zaujímajú nás tie indície, ktorých začiatočné políčko c_{is} je čierne. Vyhľadáme všetky ostatné indície, ktoré majú toto políčko v hraniciach a ak sú všetky indície **jednotky**, tak potom políčko c_{is-1} musí byť biele. Nič nám to ale nevraví o tom, či políčko c_{is} bude začiatok indície i alebo nejakej inej. V každom prípade však musí byť medzi 2 indíciami aspoň jedno políčko prázdne, preto si c_{is-1} môžeme dovoliť zafarbiť nabielo. Podobnú úvahu spravíme aj z opačnej strany. Ak je posledné políčko indície c_{ie} vyfarbené a zvyšné indície, ktoré toto políčko obsahujú sú jednotky, tak políčko c_{i+1} bude biele.



Obr. 3.8: Použitie pravidla *Jednotky*.

Pravidlo je aplikované na začiatok tretej indície.

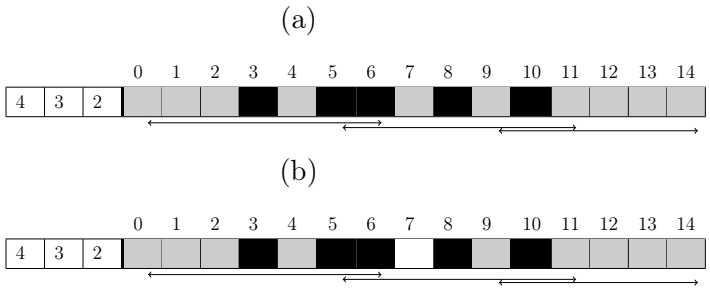


Obr. 3.9: Použitie pravidla *Jednotky*.

V prípade prvej indície sa pravidlo nemohlo aplikovať, v tretej, kde bolo posledné políčko v hraniciach vyfarbené a nasledovala len indícia veľkosti 1, sme ho už aplikovať mohli.

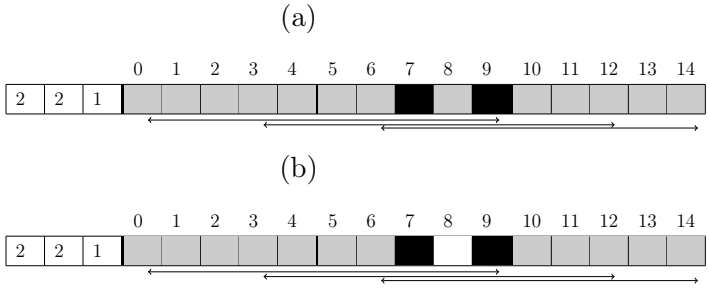
3.3.4 Vynútenie - Pravidlo 4

Tentokrát iterujeme cez všetky políčka v riadku a hľadáme situáciu, kedy políčka c_{j-1} a c_{j+1} sú čierne a políčko c_j je zatiaľ neurčené. Predstavme si situáciu, že ho zafarbíme. Môže nám vzniknúť sekvencia, ktorá bude mať dĺžku väčšiu ako ktorákoľvek indícia, ktorá sa môže vyskytnúť na týchto políčkach. Vtedy môžeme toto stredné políčko vyfarbiť nabielo. Ak ale vznikla sekvencia, ktorá by mohla prislúchať nejakej indícii, políčko c_j vrátime naspäť na zatiaľ neurčenú hodnotu.



Obr. 3.10: Použitie pravidla *Vynútenie*.

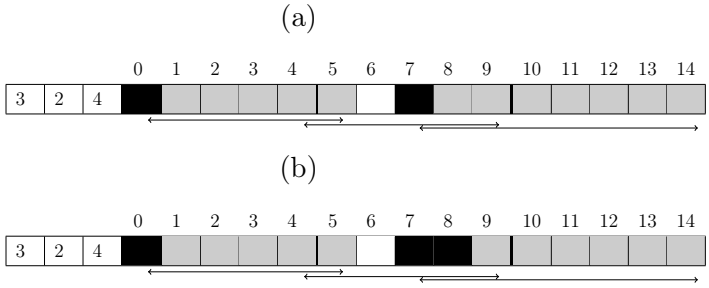
Ak by bolo políčko 4 čierne, získame sekvenciu dĺžky 4, čo môže byť sekvencia patriaca prvej indícii. Ak však zafarbíme 7. políčko, sekvenciu dĺžky 4 nemáme kam priradiť. Na to potrebujeme, aby obe koncové políčka patrili rovnakej indícii. Takže sekvencia 5-8 by nepatrila ku žiadnej indícii. Druhé políčko, ktoré týmto pravidlom skúmame je 9, ostáva neznáme, vzniknutá sekvencia by mohla patriť 2. indícii.



Obr. 3.11: Použitie pravidla *Vynútenie*.
Zafarbením políčka 8 by nám vznikla sekvencia veľkosti 3, čo nemôže patriť žiadnej indícii, preto je políčko zafarbené nabielo.

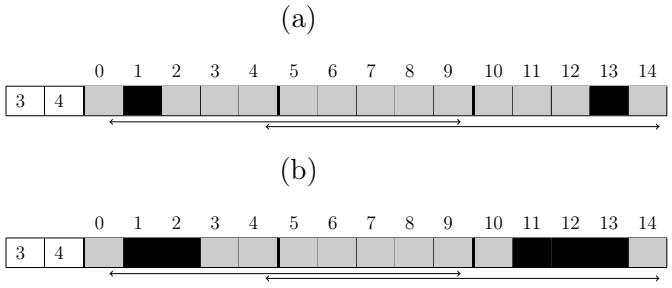
3.3.5 Lepidlo - Pravidlo 5

V riadku nájdeme konfiguráciu, kde sa nachádza za sebou neznáme alebo prázdne c_{j-1} a čierne políčko c_j. Ďalej nájdeme najmenšiu indíciu L_{min}, ktorá toto políčko obsahuje. Smerom doľava hľadáme biele políčko, ktoré je od c_i vzdialené najviac o L_{min}. Ak sme ho našli, tak políčka napravo od c_j vyfarbíme načierno, pokiaľ platí, že vzdialenosť od nášho bieleho políčka je ešte menšia alebo rovná L_{min}. To isté aplikujeme na hľadanie bieleho políčka doprava. Nakoniec, ak majú všetky indície, ktoré obsahujú políčko c_j dĺžku rovnakú ako dĺžka sekvencie pokrývajúca c_j, môžeme nastaviť políčko c_{j-1} a $c_j + L_{min}$ biele, aj keď ešte nevieme presne určiť, ktorej indícii táto sekvencia patrí. Ak počet indícií, ktoré siahajú na toto políčko je 1, táto sekvencia musí patriť výhradne jej, preto v tomto prípade môžeme upraviť hranice.
V tomto pravidle som najprv špeciálne riešila prípad prvého a posledného vyfarbeného políčka, kedy sme toto pravidlo nevedeli použiť klasicky, pretože sa tu hľadajú nejaké políčka pred a za a vyžadovalo si to veľa opatrení. Až neskôr som si všimla, že tento špeciálny prípad vyrieši kombinácia pravidiel. *Medzivýplň* len upraví hranice indície (ak sa tam vyskytuje len jedna čierna sekvencia), následne sa použije pravidlo *Prienik* - to vyplní všetky políčka, ktoré sa nachádzajú v hraniciach. Pravidlo *Update0* upraví hranice nasledujúcej indície, a nakoniec biele políčko, ktoré musí oddeľovať bloky indícií nastaví pravidlo *Jednoduché medzery*. Na prvý pohľad je to o čosi zložitejšie, ale keď som porovnala počet iterácii s použitím pravidla *Lepidlo* v špeciálnych prípadoch a bez použitia, hodnoty sa líšili v priemere o 2 iterácie. Ak sa čitateľ prepracoval až k pravidlu *Prvé čierne*, mohlo by sa mu zdať, že stačí použiť toto pravidlo. To však z prehľadávaných indícii vynecháva prvú indíciu. Aj keby sme ju špeciálne ošetrovali (počas pravidla sa pýtame na políčka, ktoré by boli mimo rozsahu), prípad, že je zafarbené posledné políčko to aj tak nerieši. Pravidlo *Prvé čierne* totiž prechádza iba z jednej strany.



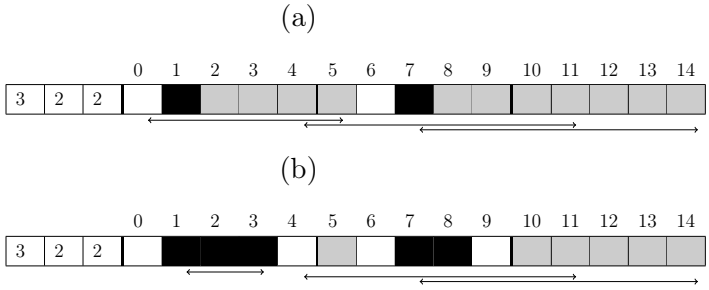
Obr. 3.12: Použitie pravidla *Lepidlo*.

Výnimka - políčko 1 ani 2 sa týmto pravidlom nezafarbia. Zafarbili sme políčko 8. Patriť môže buď druhej indícii alebo tretej. Keďže zatiaľ nevieme, ktorá to bude, môžeme zafarbiť len menší počet políčok a to 2.



Obr. 3.13: Použitie pravidla *Lepidlo*.

Ukážka, že pravidlo rieši krajné indície okrem prvého a posledného zafarbeného štvorčeka v riadku. Takisto obrázok (3.13) ukazuje, že sa neprestavujú hranice.

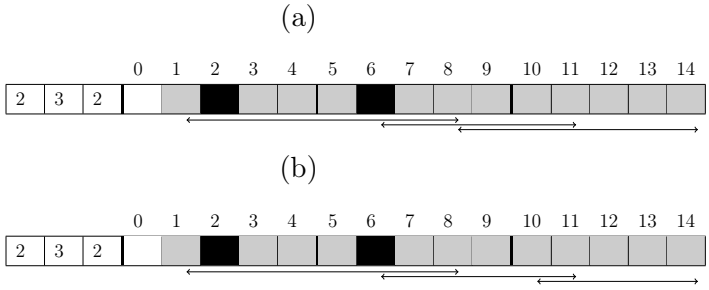


Obr. 3.14: Použitie pravidla *Lepidlo*.

Keďže všetky indície na políčkach 7-8 majú rovnakú dĺžku, môžeme 9. políčko nastaviť na biele. Hranice indícií však skracovať nemôžeme. Jediná situácia, kedy v *Lepidle* prestavujeme hranice je pri prvej indícii - sekvencia má rovnakú dĺžku ako zafarbená sekvencia a patrí už len jednej indícii.

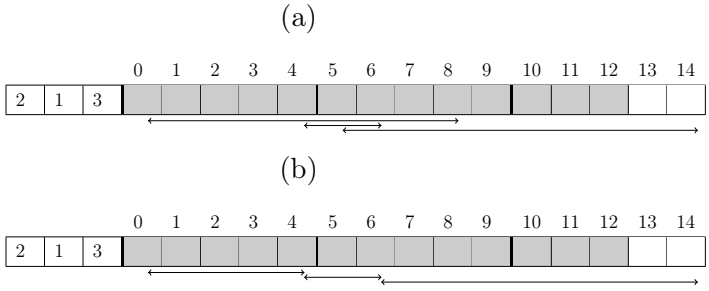
3.3.6 Update 0 - Pravidlo 6

Bez toho, aby sme sa pozerali na vyfarbené políčka v riešení, upravíme hranice indícií na nasledovné hodnoty: $c_{is} = c_{(i-1)s} + LB_{(i-1)+1}$ a $c_{ie} = c_{(i+1)e} - LB_{(i+1)-1}$.



Obr. 3.15: Použitie pravidla *Update0*.

Toto pravidlo upravuje hranice 2. indície. Tá má začiatok na 6. políčku. Takže políčko, kde táto indícia bude končiť môže byť najskôr políčko 8. Medzi jednotlivými indíciami musí byť minimálne jedno prázdne, čiže začiatok nasledujúcej indície môžeme posunúť na políčko 10. Ak by však tretia indícia mala začiatok hraníc na políčku vyššom ako je takto vypočítaná hodnota, hranice neupravujeme. To je prípad prvej indície. Druhá môže najskôr začínať na 4. políčku, v hraniciach je hodnota 6, čo je viac, tak hranice zostávajú.

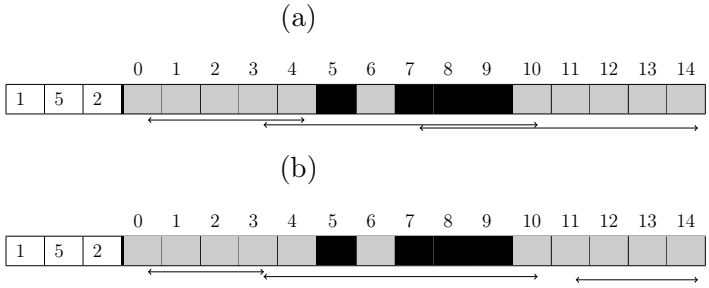


Obr. 3.16: Použitie pravidla *Update0*.

Pravidlu nevadí ani keď je indícia vnorená, iteruje sa zvlášť po začiatkoch indícií a koncových políčkach. Vždy sa len vyráta najmenšia možná pozícia od začiatku/konca a tá sa nastaví. Preto ignoruje posledné 2 biele políčka.

3.3.7 Update 1 - Pravidlo 7

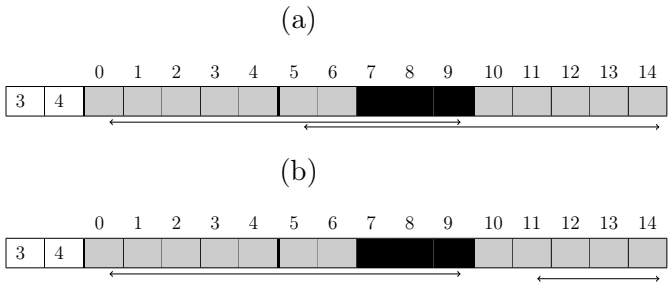
Ak je políčko c_{is-1} vyfarbené, môžeme začiatočnú pozíciu indície i posunúť na políčko c_{x+2}, kde c_x je koniec čiernej sekvencie obsahujúcej c_{is-1}. Pretože medzi 2 indíciami musí byť aspoň jedno políčko voľné a indícia i by nemohla začínať na políčku c_{is}. (V pravidlách z [5] sa hranice posúvajú len na políčko c_{is+1}. Ak by bolo aj c_{is} čierne, museli by sme pravidlo použiť znovu). Rovnaký princíp použijeme aj na koncové políčka.



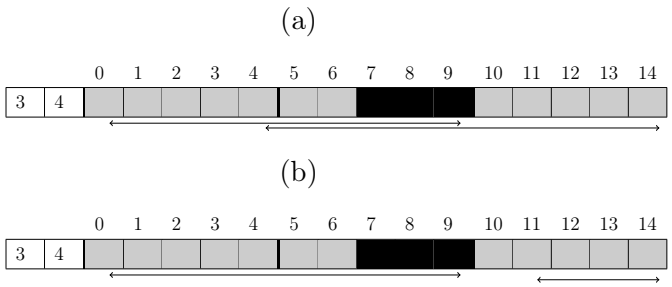
Obr. 3.17: Použitie pravidla *Update1*.
Zmenili sa hranice 1. a 3. indície.

3.3.8 Update 2 - Pravidlo 8

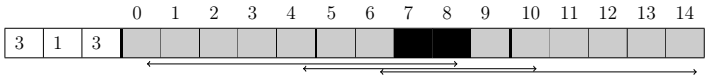
V hraniciach indícii i sa môžu vyskytnúť čierne sekvencie dlhšie ako táto indícia. Tie teda nebudú patriť k tejto indícii a môžeme ich odstrániť. Zbavovať sa môžeme dlhších sekvencií zo začiatku (konca), ak sa indícia pred túto sekvenciu nezmestí - obr. 3.18, alebo sekvencia patrí už len indícii, ktorá ju predchádza (nasleduje) - obr. 3.19. Keďže všetky krížovky majú mať riešenie, každá čierna sekvencia musí niektorej indícii prislúchať.



Obr. 3.18: Použitie pravidla *Update2*.
Indícia 2 sa pred čiernu sekvenciu $[7-9]$ nezmestí, lebo medzi blokmi indícii musí byť minimálne 1 políčko voľné.



Obr. 3.19: Použitie pravidla *Update2*.
Tentokrát sme hranice posunuli kvôli tomu, že čierna sekvencia $[7-9]$ mohla patriť už len predchádzajúcej indícii.

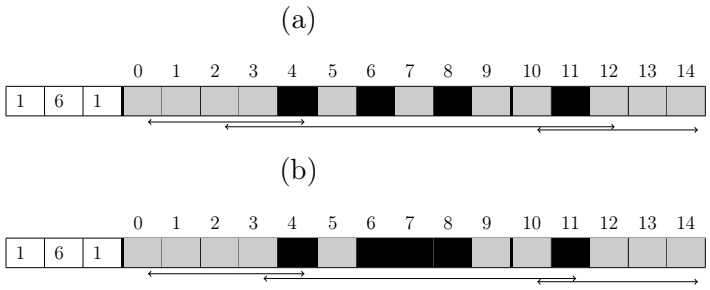


Obr. 3.20: Použitie pravidla *Update2*.

Teraz žiadne hranice posunúť nemôžeme. 2. indícia by sa mohla nachádzať aj pred čiernou sekvenciou $[7-8]$ aj za ňou. Nevieme povedať z ktorej strany hranice treba orezať. V pravidle neprejde podmienkou, že čierna sekvencia, ktorú sa snaží odstrániť zo zoznamu patrí len indícii pred ňou.

3.3.9 Medzivýplň - Pravidlo 9

Pre každú indíciu najprv nájdeme interval, ktorý sa nijak nekryje s ostatnými indíciami. Na tomto intervale hľadáme prvé a posledné čierne políčko. Ak sa nachádzajú nejaké prázdne políčka medzi nimi, vyfarbíme ich. Na záver upravíme hranice tejto indície.



Obr. 3.21: Použitie pravidla *Medzivýplň*.

Na intervale $[5-9]$ sa 2. indícia nekryla s ostatnými. Všetky čierne políčka nachádzajúce sa v ňom preto patria jej. Vznikla sekvencia, ktorá nám mierne upravuje hranice tejto indície.

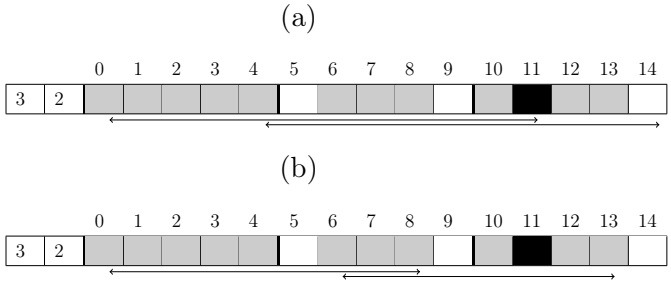
Obr. 3.22: Použitie pravidla *Medzivýplň*.

Hranice sa upravia aj keď sa na intervale nachádza len jedna indícia.

3.3.10 Oklieštenie - Pravidlo 10

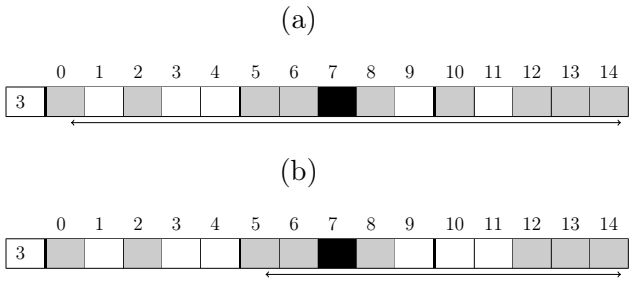
Podobné pravidlo ako *Vynútenie*, ale teraz chceme pre každú indíciu odstrániť z jej hraníc sekvencie ohraničené bielymi políčkami, ktorých dĺžka je menšia ako indícia. Čiže naša indícia by sa tam už nezmestila. Keďže intervaly vieme odstrániť z hraníc, len ak sú na krajoch, môže sa nám stať, že nám v strede zostanú sekvencie, kam sa síce žiadna indícia nezmestí, ale napravo a naľavo od nej sa uložiť dá. Toto pravidlo teda

bude riešiť aj takéto situácie a príliš krátke sekvencie (obsahujú len doteraz neurčené políčka) môžeme zafarbiť nabielo.



Obr. 3.23: Použitie pravidla *Oklieštenie*.

Pri iterovaní prvou indíciou sme našli 3 sekvencie: $[0-4], [6-8], [10-11]$. Na posledný z intervalov sa táto indícia dĺžky 3 nezmestí, keďže je na konci, môžeme túto sekvenciu z hraníc odstrániť. V hraniciach 2. indície sa nachádzajú tiež 3 sekvencie: $[4-4], [6-8], [10-13]$. Pri prechádzaní cez tieto sekvencie sa vždy najskôr začiatok indície nastaví na začiatok tejto sekvencie aby sme predišli tomu, že tam zbytočne budeme mať na začiatku/konci biele políčka, ktoré by sme nevedeli odstrániť žiadnym pravidlom. Preto sa z hraníc odstránilo posledné políčko.



Obr. 3.24: Použitie pravidla *Oklieštenie*.

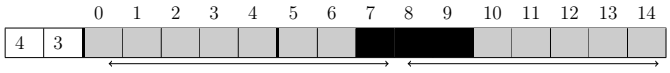
Odstránili sme všetky sekvencie, kam sa indícia s dĺžkou 3 nezmestí. Okrem toho nám v hraniciach zostala sekvencia s dĺžkou 1 $[10-10]$ o ktorej vieme, že nemôže patriť tejto indícii, ale nevieme ju z jej hraníc odstrániť. Keďže nepatrí žiadnej ďalšej indícii, môžeme ju vyfarbiť nabielo. Ale sekvencie, ktoré sme z hraníc odstránili už nabielo nevyfarbujeme, ak majú byť biele - nepatria žiadnej indícii, nabielo ich zafarbí pravidlo *Jednoduché medzery*.

Nekryjúce sa

Nasledujúce 3 pravidlá sú navrhnuté na situácie, kedy sa jednotlivé **hranice susedných indícii neprekrývajú**. Na použitie týchto pravidiel musia susedné indície spĺňať vzťah: $r_{(i-1)e} < r_{is}$.

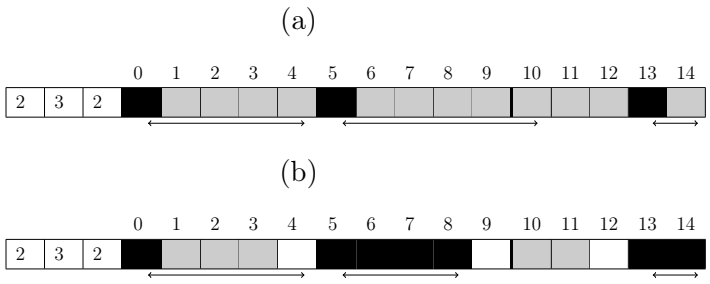
3.3.11 Prvé čierne - Pravidlo 11

Ak je políčko r_{is} čierne, potom môžeme vyfarbiť čiernou aj nasledujúce políčka za r_{is}, podľa dĺžky indície. Nakoniec upravíme koncovú pozíciu indície a nastavíme nasledujúce políčko aj políčko $r_{i(s-1)}$ na bielu farbu. Toto pravidlo je veľmi podobné ako pravidlo *Lepidlo*. Avšak to vyhľadáva biele políčka pred/za nájdeným čiernym políčkom a nezaoberá sa tým, ktorej indícii prislúcha. Toto pravidlo tiež neprechádza prvou indíciou, ktorá sa určite nekryje s predchádzajúcou, keďže žiadna nie je. Vieme to však nahradiť použitím iných pravidiel.



Obr. 3.25: Použitie pravidla *Prvé čierne*.

Ak sa zamyslíme nad takouto situáciou, kedy je prvé políčko čierne a sekvencie sa nekryjú, ale aj políčko pred ním je čierne, zistíme, že to nemôže nastať. Vtedy by táto čierna sekvencia neprislúchala nikomu, lebo celú ju ani jedna z indícii nemá v hraniciach a medzi indíciami musí byť aspoň jedno políčko voľné. Ak by nastala takáto situácia, je problém niekde v krížovke, lebo toto nie je validná konfigurácia.

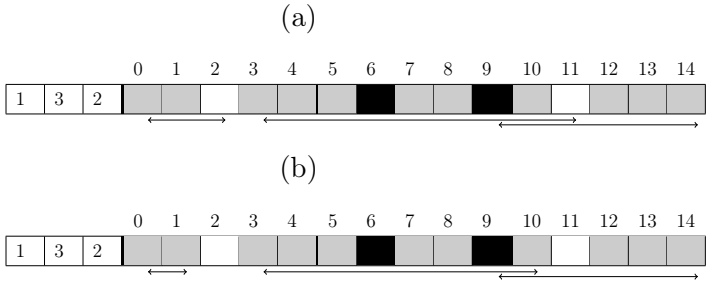


Obr. 3.26: Použitie pravidla *Prvé čierne*.

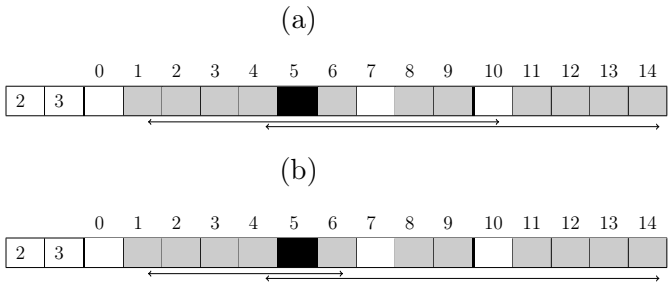
Pravidlo nerieši prípad prvej indície. Všetky ostatné však vyfarbí podľa dĺžky indície, nastaví biele políčka pred a vzad (ak to je možné) a upraví hranice.

3.3.12 Prekážka - Pravidlo 12

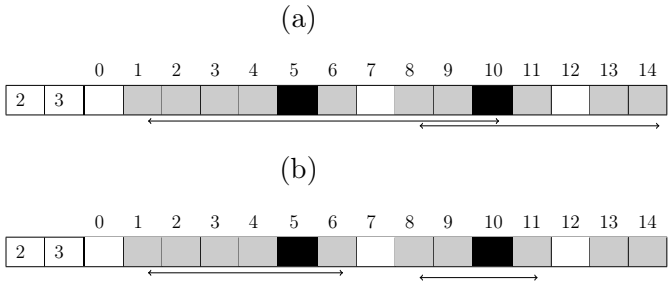
Ak nastane situácia, že sa biele políčko c_w nachádza za čiernym v hraniciach patriacich jednej indícii, môžeme nastaviť pozíciu r_{ie} na index c_{w-1}. Keďže predchádzajúca indícia nezasahuje do tohto intervalu, nájdené čierne políčko musí určite patriť indícii i, alebo patrí niektorej nasledujúcej indícii, ale potom sa indícia i musí celá nachádzať pred políčkom c_w.



Obr. 3.27: Použitie pravidla *Prekážka*.
Pravidlo upravilo hranice prvých 2 indícii, 3. sa však zľava kryje s predchádzajúcou, preto nie je možné pravidlo aplikovať.



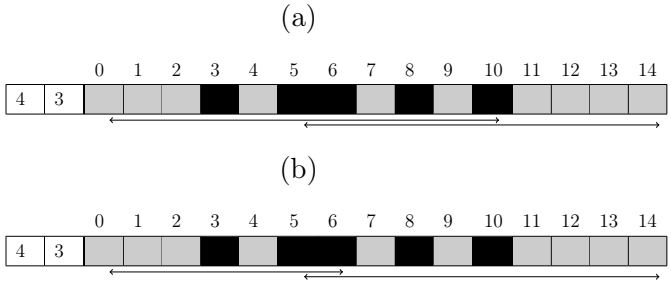
Obr. 3.28: Použitie pravidla *Prekážka*.
Hranice prvej indície sme posunuli na políčko 7. Keby sme toto pravidlo odľahčili o podmienku, že sa môže aplikovať len na neprekrývajúce sa indície, z druhej indície by sme odstránili sekvenciu $[11 - 14]$, kde sa indícia môže nachádzať a týmto spôsobom by sme spravili chybu.



Obr. 3.29: Použitie pravidla *Prekážka*.
Keďže najskôr spracovávame prvú indíciu a až potom nasledujúcu, môže sa nastať rovnaká situácia ako tu. Aj keď sa druhá indícia kryla s predošlou, po použití pravidla sa hranice prvej indície posunuli na políčko 6 a už viac sa indície neprekrývali. Mohli sme teda pravidlo *Prekážka* použiť aj na druhú indíciu.

3.3.13 Rozdelenie - Pravidlo 13

V jednom intervale sa nachádza viac ako 1 čierna sekvencia. Zoberme si prvé dve. Ak by ich spojením vznikla sekvencia s dĺžkou väčšou ako je dĺžka indície, ktorú prechádzame, potom obe sekvencie nemôžu patriť k tejto indícii. Preto interval orežeme z pravej strany (keďže vpredu už je nejaké políčko čierne, orezávame sprava), nastavením r_{je} na pozíciu pred 2. čiernou sekvenciou. Ak by spojením týchto 2 sekvencií vznikla postupnosť kratšia ako je dĺžka indície, no v intervale sú ešte ďalšie čierne sekvencie, opakujeme postup s 1. a 3. sekvenciou. Ak to ešte stále vyhovuje, posunieme sa na 4. atď. Ak nám v niektorom kroku vznikne sekvencia dlhšia ako je dĺžka indície, upravíme hranice tak, ako by sme to spravili pri spojení 1. s 2.



Obr. 3.30: Použitie pravidla *Rozdelenie*.

V hraniciach prvej indície sa nachádzali 4 čierne sekvencie. Ak skúsime spojiť 1. s 3. vznikne sekvencia dlhšia ako 4, čo je dĺžka aktuálnej indície. Druhá indícia sa však stále kryje s prvou, takže toto pravidlo naň nemôžeme aplikovať.

3.4 Prehľad pravidiel

Ak by sme si chceli vyskúšať, ako vyriešiť krížovku pomocou týchto pravidiel, tak po jednom prečítaní, by to bolo asi náročné. Preto som zostavila tabuľku s označením, čo jednotlivé pravidlá v riadku hľadajú a ktoré hodnoty v riešení alebo v hraniciach prestavujú.

ID	pravidlo	hľadá čierne	hľadá biele	nastavuje čierne	nastavuje biele	prestavuje c_{is}	prestavuje c_{ie}	iteruje cez
1	Prienik	✗	✗	✓	✗	✗	✗	indície
2	Jed. medzery	✗	✗	✗	✓	✗	✗	hranice indícií
3	Jednotky	✓	✗	✗	✓	✗	✗	indície 2x
4	Vynútenie	✓	✗	✓	✗	✗	✗	stĺpce
5	Lepidlo	✓	✓	✓	✓	(✓)	(✓)	stĺpce
6	Update0	✗	✗	✗	✗	✓	✓	hranice indícií
7	Update1	✓	✗	✗	✗	✓	✓	indície
8	Update2	✓	✗	✗	✗	✓	✓	indície
9	Medzivýplň	✓	✗	✓	✗	✓	✓	indície
10	Oklieštenie	✓	✓	✗	✗	✗	✓	indície
11	Prvé čierne	✓	✗	✓	✓	✗	✓	indície
12	Prekážka	✓	✓	✗	✗	✗	✓	indície
13	Rozdelenie	✓	✗	✗	✗	✗	✓	indície

Kapitola 4

Návrh algoritmu

V tejto kapitole je popísané riešenie krížovky pomocou pravidiel z predchádzajúcich kapitol. Sú tu demonštrované postupy, akými som získavala informácie o krížovkách pre ďalšie spracovanie.

4.1 Zdroj dát

Dáta, ktoré používam, som získala od prevádzkovateľa stránky `Griddlers.net`. Táto stránka je vytvorená pre ľudí obľubujúcich logické hlavolamy, ktoré sa tu dajú online riešiť. Prevažnú časť tvoria maľované krížovky rôznych tvarov, veľkostí, obtiažností a farieb, ale je možné nájsť aj sudoku, kakuro, osemsmerovky a iné menej známe hlavolamy. Dôležité je poznamenať, že všetky krížovky na tomto portáli majú **len jedno riešenie**. Ak má krížovka viac riešení, proces na nájdenie všetkých riešení sa komplikuje. Viac o tejto problematike sa dočítame v [11]. Maľované krížovky, ktorými sa v práci zaoberáme sú *dvojfarebné* a *bez trojuholníkov na konci*. Na portáli sú viaceré krížovky poskladané z niekoľkých menších, avšak v databáze je každá z týchto častí reprezentovaná ako samostatná krížovka, má vlastné ID. Pre každú krížovku som si vytiahla rozmery, počet riešiteľov, stredný čas vyriešenia krížovky (= medián, do dát sa započítala len doriešená krížovka). Počet krížoviek, ktoré som takto dostala bolo **42 316**. Z nich bolo niekoľko krížoviek, ktoré nemali stredný čas riešenia (ten sa prepočíta každý deň, pre krížovky, ktoré majú aspoň 3 vyriešenia). Dobrých dát, ktoré som mohla ďalej spracovávať bolo nakoniec **42 277**.

4.2 Vstup

Každé zadanie krížovky mám uložený ako samostatný súbor *number.txt*, kde *number* je ID číslo krížovky. ID číslo reprezentuje krížovku aj na portáli `Griddlers.net`, takže zadania jednotlivých krížoviek bolo jednoduché vyhľadať. Na stiahnutie zadaní a roz-

parsovanie *html* súboru, som použila knižnicu *Jsoup*. Vytváranie samotného `txt` súboru prebieha v dvoch krokoch. Na začiatok súboru si uložíme údaje z databázy *puzzles.sql*, toto nám zabezpečuje funkcia

```
Files_creator.tahaj(Path file)
```

kde **file** je naša databáza, s údajmi o krížovkách. Funkcii

```
Files_creator.spracuj(String id, PrintWriter writer)}
```

posielame referenciu na číslo krížovky a otvorený dokument, kam chceme zadanie uložiť. Tá nám z portálu stiahne html súbor krížovky, zadanie nájde a uloží do dokumentu. Najprv indície, ktoré nám určujú stĺpce a potom riadky. Tu je ukážka takého dokumentu:

```
Puzzle number: 9580
width:  5
height:  5
solved_count:  11388
AVE_TIME:  '00:00:14'
ave_time_count:  6501

top_values
[4]
[2, 2]
[1]
[2, 2]
[4]

left_values
[3]
[2, 2]
[1, 1]
[2, 2]
[2, 2]
```

4.3 Vyriešenie krížovky

Jednotlivé krížovky je možné vyriešiť viacerými spôsobmi. Najprv si ukážeme ako vyriešiť jednu konkrétnu krížovku. Na to nám slúži funkcia

```
Vyries.tuto_vyries(int cislo, int[] pravidla, boolean uloz);,
```

ktorej ako argumenty dávame ID číslo krížovky a pravidlá pomocou ktorých chceme krížovku vyriešiť. Posledný parameter `uloz` nastavíme na `true`, ak chceme výsledok

riešenia uložiť. Každé pravidlo má svoje **číslo**. Takže ak chceme používať pravidlá 5,6 a 10, pravidlá budú mať na týchto miestach v poli nenulovú hodnotu. V našom prípade by to teda vyzeralo:

```
int[] pravidla = {0,0,0,0,5, 6,0,0,0,10, 0,0,0};
Vyries.tuto_vyries(303, pravidla, true);
```

Logických pravidiel, ktorými môžeme krížovku riešiť je 13, predstavené boli v predchádzajúcej kapitole (3). Ak by sme krížovku chceli vyriešiť Dynamickým programovaním (pozri: 2.5), môžeme použiť takúto sadu pravidiel, ktorá tomu zodpovedá:

```
int[] pravidla = {0,0,0,0,0, 0,0,0,0,0, 0,0,0, 14}
```

. Kombinovať pravidlo dynamické programovanie s inými pravidlami už nie je nutné, na vyriešenie všetkých krížoviek, ktoré máme k dispozícii stačilo samotné pravidlo `dynamické programovanie`, ale taká možnosť je. Tu je zoznam pravidiel aj s ich referenčnými číslami:

1. Prienik
2. Jednoduché medzery
3. Jednotky
4. Vynútenie
5. Lepidlo
6. Update 0
7. Update 1
8. Update 2
9. Medzivýplň
10. Oklieštenie
11. Prvé čierne
12. Prekážka
13. Rozdelenie

Funkcia `tuto_vyries` si vytvorí inštanciu triedy ***Vyries***, ktorá sa už stará o celé vyriešenie krížovky. Vytvorí si *Inicializácie* (pozri: 3.2) na riadky a stĺpce krížovky, ktorej zadanie nájde na adrese `nonograms\id.txt`. Následne zavolá funkciu `ries()`.

```
,

public class Vyries {

    int cislo;
    ArrayList<Pravidla> pravidla;
    Inicializacia inic_S,inic_R;
    int[] statistika;
    boolean uloz;

    public Vyries(int Id_nono, int[] pravidla, boolean uloz){}

}
```

Funkcia `ries()` iteruje najprv cez celú krížovky a na každý riadok postupne skúša aplikovať pravidlá, ktorými sa snažíme krížovku vyriešiť. Ak sa v riadku nachádzajú

už len jednotky a nuly, znamená to, že je riadok doriešený, čo si zaznačíme, aby sme ho zbytočne znova neprechádzali. Každé z pravidiel má návratovú hodnotu, ktorá nám hovorí, či sa pravidlo podarilo použiť, či v riadku nastala nejaká zmena. Iterovanie končíme, keď je krížovka vyriešená, alebo prešla jedna celá iterácia (riadky + stĺpce) a nenastala zmena v žiadnej štruktúre. Po dokončení riešenia, ak sme zvolili `uloz = true`, sa výsledný obrázok uloží.

4.4 Automatické vyriešenie krížoviek

Ak chceme vyriešiť nejakú množinu krížoviek, použijeme jednu zo statických funkcií, v závislosti od toho, aké výsledky nás zaujímajú.

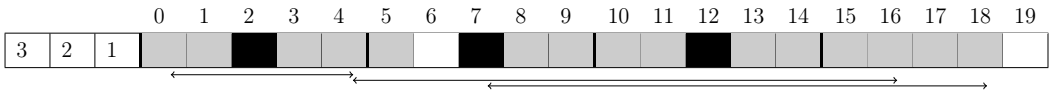
- `Vyries.tieto_ries(file zoznam, int[] pravidla, boolean uloz);`
  - zoznam má obsahovať ID čísla krížoviek, každé na samostatnom riadku. Krížovky sa vyriešia `pravidlami` a ak nastavíme `uloz = true`, po skončení funkcie `Ries()` sa zavolá funkcia `uloz_stav()` (pozri: 4.5(2)). Jednotlivé výsledky o krížovkách nájdeme v priečinku `Memento`.

- `Vyries.tieto_ries_LG(file zoznam, boolean uloz);`
  - touto funkciou sa vyriešia krížovky použitím všetkých 13 logických pravidiel. Dáta ktoré môžeme získať, ak dáme `uloz=true` sa uložia do súboru `Results_LG.txt` pomocou funkcie `uloz_vysledky()`.

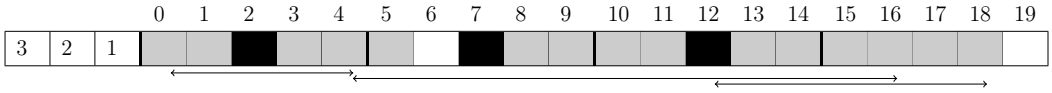
- `Vyries.ries_dvojako(zoznam);`,
  - každá krížovka sa rieši dvoma spôsobmi. Najprv dynamickým programovaním a následne použitím všetkých 13 logických pravidiel. Výsledky sa opäť získavajú funkciou `uloz_vysledky()` a môžeme ich nájsť v súbore `Both_ways.txt`.

- `Vyries.ries_podmnoziny(zoznam);`,
  - táto funkcia má za úlohu nájsť minimálnu podmnožinu logických pravidiel, pomocou ktorých sa krížovku podarí vyriešiť. Funkcia má ako výstup súbor `Sets.txt`, kde na začiatku riadka je ID číslo krížovky a ďalej nasledujú najmenšie podmnožiny, ktorými sa krížovka dá vyriešiť. Takže ak je takou množinou $[1, 2, 7, 10]$, tak tam už nenájdeme množinu $[1, 2, 5, 6, 7, 10]$ a ani $[1, 2, 10]$.

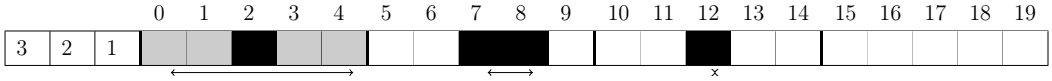
Z krížoviek, ktoré som mala k dispozícii sa mi použitím pravidla **Dynamické programovanie** podarilo vyriešiť všetkých **42 277**. Ak som na riešenie použila celú **sadu pravidiel**, nedoriešených zostalo **5 645**. Skúsila som sa teda bližšie pozrieť na konečný stav takýchto krížoviek, či by som ako ľudský riešiteľ ešte nevedela dať nejakú informáciu, ktorá by mi pomohla ďalej v riešení. Našla som situáciu, ktorú by človek vyriešiť zvládol, no žiadne z pravidiel to nezachytávalo:



Je jasné, že každé zo štvorčekov musí patriť inej indícii. Jeden zo spôsobov, ako by sme to mohli vyriešiť je, že upravíme pravidlá tretej skupiny, aby riešili situácie aj prechodom z opačnej strany riadka. Konkrétne tu by sa nám zišlo pravidlo *Rozdelenie*. 3. indícia sa sprava nekryje s nasledujúcou, preto čierne políčko č.7 už nebude patriť do jej hraníc. Z toho dostávame situáciu:



A teraz už opäť môžeme použiť známe pravidlá. *Lepidlo, Medzivýplň, Jednoduché medzery* až sa dopracujeme k takémuto stavu:



Pre každú krížovku som taktiež našla minimálnu množinu pravidiel, ktorá ju dokáže vyriešiť. Keď som potom porovnala počty krížoviek ktoré boli doriešené touto metódou a metódou kedy sme použili všetky pravidlá, zistila som, že tieto počty nie sú rovnaké. Odlišovali sa v jednej krížovke. Tá bola doriešená pomocou pravidiel `1,2,3,4,6,7,8,9,10,13`, ale nepodarilo sa ju vyriešiť, ak sme používali všetky pravidlá. Problémy spôsobovalo pravidlo *Lepidlo*. To sa nachádza v prvej skupine pravidiel, takže bolo na riadok aplikované medzi prvými. Tým mohlo odstrániť nejakú informáciu, ktorá bola potrebná na použitie neskoršieho pravidla. Krížovka, u ktorej som takéto správanie našla mala číslo `44 218`.

4.5 Uloženie dát

Ukladanie údajov zabezpečuje trieda `Memento`. Keďže máme niekoľko možností na vyriešenie krížoviek, tak aj dáta, ktoré si budeme ukladať sa budú mierne líšiť.

1. `uloz_riesenie()` nám uloží dokončený obrázok, so štatistikou, ktoré pravidlo sa koľkokrát použilo a koľko iterácii sa počas riešenia vykonalo. Obrázok zobrazuje riešenie ako $\{X, ., ?\}$, kde `?` sú nedoriešené políčka. Túto funkcia sa volá v prípade jednotlivého riešenia krížoviek. Výsledok nájdeme v súbore **MEMid.txt**.

2. `uloz_stav()` oproti predošlej funkcii sa nám ukladajú aj konečné nastavenia hraníc. V prípade, že nie je krížovka doriešená si môžeme pozrieť, v akom stave

sa nachádzala a či ju naozaj nie je možné doriešiť. Funkciu volá `tieto_vyries()`, takže pre každú krížovku sa nám vytvorí **MEMid.txt** a tieto súbory nájdeme v priečinku **MEMENTO**.

3. `uloz_vysledky()` nám o každej krížovke do súboru uloží na samostatný riadok tieto dáta: **ID krížovky, šírka, výška, plocha (=šírka*výška), obsah vyfarbenej časti (vypočítané zo zadania), priemerný čas v sekundách, počet riešiteľov, počet nulových riadkov, počet iterácii, štatistika použitia jednotlivých pravidiel**.
   V prípade, že sa funkcia používa pri riešení krížovky dvoma spôsobmi, tak sa najprv uloží počet iterácii a použitia pravidla pre dynamické programovanie a za tým výsledky použitia logických pravidiel.

4. `uloz_sety()` pre každú krížovku sa uložia najmenšie podmnožiny, ktorými sa ju podarilo doriešiť. Takže je riadok prázdny, ak sa krížovku nepodarilo celú vyriešiť.

Štatistika logických pravidiel je zobrazovaná ako pole dĺžky 15, kde prvá hodnota je počet iterácii a za ňou na pozíciách odpovedajúcich poradovým číslam jednotlivých pravidiel, je počet ich použití.

Kapitola 5

Analýza

V tejto časti by som rada uviedla výsledky, k akým sme dospeli na základe analyzovania dát získaných rôznymi formami popísanými v predchádzajúcich kapitolách. Naším cieľom bolo vedieť čo najpresnejšie odhadnúť čas riešenia krížovky človekom. To sme skúšali rôznymi typmi regresie.

5.1 Lineárna regresia

Lineárna regresia je štatistická metóda na modelovanie vzťahu (korelácie) medzi veličinami. Ich vzťah sa snaží čo najlepšie odhadnúť pomocou priamky, čo sa vykonáva metódou najmenších štvorcov.
$\hat{y}\ (w, x) = w_0 + w_1 x_1 + ... + w_p x_p$ $w = (w_0, ...w_p)$ je vektor parametrov priamky, ktoré najlepšie odhadujú koreláciu medzi odhadovanou hodnotou y, a zvyšnými veličinami $x_1 ... x_p$.
V našom prípade sa pokúšame nájsť vzťah medzi **riešením krížovky človekom** a zvyšnými dátami, ktoré máme dostupné. Aby sme overili, ako dobre sa nám podarilo tento vzťah nájsť, využijeme metódy strojového učenia.

5.2 Dataset

Dáta, ktoré budeme používať na analýzu sú kombináciou viacerých foriem vyriešenia krížoviek popísaných v predošlej kapitole. Pre úplnosť uvediem ešte raz všetky stĺpce, ktoré budeme analyzovať: 1-šírka, 2-výška, 3-plocha, 4-obsah vyfarbenej časti, **5-priemerný čas**, 6-počet nulových riadkov, 7-počet iterácii pri DYN, 8-počet použitia DYN, 9-počet iterácii pri LR, 10-22-jednotlivé počty použitia LR, 23-či bola krížovka doriešená pomocou LR, 24-36-zastúpenie jednotlivých LR v minimálnych setoch
DYN je skratka pre dynamické programovanie a LR pre logické pravidlá.

Dáta som si ešte pred začatím náhodne preusporiadala a rozdelila na trénovaciu sadu (32 277 krížoviek) a testovaciu (10 000 krížoviek).

5.3 Použité modely

Na analýzu som použila knižnicu jazyka `Python - scikit - learn` [7]. Z nej som využila tieto modely lineárnej regresie:

- `Linear Regression`
- `Ridge`
- `Ridge CV`
- `Lasso`
- `Elastic Net`
- `Lasso Lars`

Lineárna regresia sa snaží vyrobiť parametre, ktoré by čo najlepšie kopírovali naše dáta na trénovacej množine a následne ich otestovala na sade testovacej. Kvalitu predpovede vo všetkých prípadoch meriame tzv. koeficientom determinovanosti R^2. Túto hodnotu nám vie použitá knižnica priamo spočítať volaním metódy `score()`. Koeficient je definovaný ako $(1 - SS_{res}/SS_{tot})$, kde SS_{res} je suma štvorcov rezíduí $\sum_{n=1}(y_i - f_i)^2$ a SS_{tot} je celková suma štvorcov $\sum_{n=1}(y_i - \bar{y})^2$. Funkcia f je funkcia modelu, ktorý sme si zvolili, vektor $y = [y_1, \cdots, y_n]$ predstavuje hodnoty, ktoré chceme predikovať a $\bar{y}$ je aritmetický priemer hodnôt y. Najlepší možný koeficient je 1, môže však dosahovať aj záporné hodnoty. Pri analýze každý z týchto modelov dosahoval veľmi podobné výsledky, preto konečné výsledky budem prezentovať len použitím metódy **Linear Regression**.

5.4 Výsledky

Ako prvé som skúsila predpovedať čas na len na základe dát získaných z portálu `Griddlers.net`. Tie obsahovali: `šírka krížovky, výška, plocha, vyfarbená časť, počet nulových riadkov`. Pomocou lineárnej regresie sa mi podarilo získať koeficient determinovanosti na testovacej sade **0.48854** - viď tab.5.1. Pre zvýšenie úspešnosti som skúsila doplniť dáta, ktoré sme dostali riešením krížovky celou sadou pravidiel a dynamickým programovaním (4.4 - funkcia `ries_dvojako()`). Analýzou tejto sady dát sa nám podarilo dosiahnuť koeficient **0.89158**. Parameter ktorý mal najväčšiu váhu, bol záznam, či sa podarilo krížovku doriešiť pomocou logických pravidiel **(23)** alebo

nie. Ďalšie parametre s vysokými kladnými koeficientami boli počet iterácii v dynamike **(7)** a počet použitia pravidla Vynútenie **(13)**. So zápornými hodnotami to boli rozmery krížovky **1, 2** viď tab.5.2.
Keďže máme ešte údaje o minimálnych množinách pravidiel pre krížovky, pokúsili sme sa použiť aj tie. Pre každú krížovku a každé pravidlo sme spočítali jeho výskyt v minimálnych množinách a vydelili sme to počtom minimálnych množín pre danú krížovku.

Príklad: 402 [1, 2, 4, 5, 6, 10] [1, 2, 3, 5, 6, 10] [1, 2, 3, 4, 6, 7, 8, 9, 10]
Krížovka 402 má 3 sady pravidiel, pomocou ktorých sa krížovka dá úspešne doriešiť. Pravidlá $1, 2, 4, 10$ sa nachádzajú v každej sade, preto ich výsledná hodnota bude 1. Pravidlá $3, 4, 5, 6$ sa vyskytujú v sadách, preto budú mať hodnotu $\frac{2}{3}$. Tie pravidlá, ktoré sa v žiadnej sade nevyskytujú budú mať hodnotu 0.
Výsledok: 1, 1, 0.6666, 0.6666, 0.6666, 1.0000, 0.3333, 0.3333, 0.3333, 1, 0, 0, 0

Po doplnení o túto sadu dát sa nám podarilo len trochu zlepšiť výsledky, oproti predchádzajúcej analýze. Koeficient determinovanosti bol **0.89183**. Medzi najvýznamnejšie parametre patrili práve doplnené dáta o setoch. Najväčšiu váhu mal posledný stĺpec **(36)**, ďalšie zaujímavé stĺpce boli: **(23, 26, 27, 32, 34, 35)**. Z môjho pozorovania, vyplýva, že nutnosť použitia týchto pravidiel na doriešenie krížovky (inak majú tieto parametre hodnotu 0) vie pravidlá špecifikovať, aká je ich náročnosť, keďže tieto hodnoty sú v rozmedzí $(0, 1)$, nedá sa ich váha porovnávať s počtom použitia pravidiel, ktoré dosahujú hodnoty v desiatkach.

Pri používaní všetkých metód lineárnej regresie sme pri nízkych časoch často predpovedali záporné hodnoty, ktoré nám výsledok zbytočne kazili, keďže z charakteru našich dát je jasné, že záporné hodnoty nikdy nechceme dosahovať (viď priložené grafy). Preto sme sa rozhodli vyskúšať ešte jednu metódu a to Random Forest Regression.

5.5 Random Forest Regression

Táto metóda využíva rozhodovacie stromy na predpovedanie hodnôt pre sadu dát. Parameter, ktorý sme skúšali meniť, bol počet rozhodovacích stromov, ktoré si môže počas regresie vytvoriť - *parameter n_ estimators*. Kvalita predpovede sa oproti lineárnej regresii o niečo zlepšila. Výsledky môžeme nájsť v tabuľkách. Táto metóda nám však nedáva koeficienty parametrov ako pri lineárnej regresii, ale sú to hodnoty *(feature importances)*, ktoré zodpovedajú, ako často sa dané premenné používali pri testovaní metódy. Tieto hodnoty nám sama počíta knižnica *scikit*.

5.6 Porovnanie

V nasledujúcich tabuľkách uvádzam porovnanie výsledkov našimi vybranými metódami. Prvý riadok obsahuje typ regresie, ktorý sme použili, v prípade Random Forest Regression je pri názve číslo, ktoré reprezentuje, koľko stromov sme pri regresii dovolili vytvoriť *(n_ estimators)*. V prípade lineárnej regresie sú v stĺpcoch uvedené koeficienty, ktoré určujú regresnú priamku. Pri Random Forest Regression sa v stĺpcoch nachádzajú dôležitosti znakov *(feature importances)*. Grafy, ktoré sa nachádzajú v prílohe A, zobrazujú, ako dobre sa metóda dokázala naučiť predpovedať hodnoty. Na osi X-ovej sú skutočné hodnoty a na osi Y-ovej sú predikované hodnoty. *Graf prvého typu:* do grafu sa zaznačí bodka pre každú krížovku v datasete. Zobrazujem grafy na testovacej aj trénovacej množine. *Graf druhého typu:* prezentuje výsledky zoradené podľa času skutočného vyriešenia krížovky - ten je v grafe pospájaný červenou krivkou. Modrými čiarami sú znázornené predikované časy - čo nám dokopy ukazuje ako veľmi sa predpovedané hodnoty vychyľujú.

Tabuľka 5.1: Analýza použitá výlučne na dátach z portálu.

		lin.regresia	rand.forest 100	rand.forest 150
trén. skóre		0.49476	0.82911	0.82943
test. skóre		0.48854	0.48907	0.49308
koeficienty	1	-8.1739	0.02009	0.01934
	2	-7.2157	0.01581	0.01673
	3	1.2320	0.16837	0.16782
	4	-0.0313	**0.72283**	**0.72229**
	6	**-40.9563**	0.07291	0.07382

Tabuľka 5.2: Predikcia časov na základe údajov o vyriešení krížoviek dvoma spôsobmi.

		lin.regresia	rand.forest 100	rand.forest 150
trén. skóre		0.89294	0.98701	0.98705
test. skóre		0.89158	0.90447	0.90514
koeficienty jednotlivých stĺpcov	1	**-13.5358**	0.00286	0.00312
	2	**-12.3412**	0.00111	0.00113
	3	0.2335	0.00283	0.00269
	4	-0.0273	**0.01076**	**0.01086**
	6	-6.0160	0.00205	0.00200
	7	**7.3072**	0.00609	0.00622
	8	6.7510	**0.80627**	**0.80635**
	9	-1.1509	0.00560	0.00554
	10	6.2112	**0.02384**	**0.0288**
	11	-0.4153	0.00973	0.00994
	12	5.7024	0.00690	0.00687
	13	**7.7379**	0.00676	0.00683
	14	-1.5725	**0.01515**	**0.01467**
	15	0.1546	0.00579	0.00566
	16	4.8018	**0.05745**	**0.05747**
	17	6.2292	0.00659	0.00662
	18	-1.0452	0.00546	0.00538
	19	-1.4913	0.00746	0.00741
	20	-5.7830	0.00380	0.00388
	21	2.7881	0.00525	0.00513
	22	3.1320	0.00500	0.00506
	23	**40.1231**	0.00327	0.00327

Tabuľka 5.3: Analýza spravená na všetkých dátach, ktoré sme získali

		lin.regresia	rand.forest 100	rand.forest 150
trén. skóre		0.89332	0.98716	0.98740
test. skóre		0.89183	0.90514	0.90567
koeficienty jednotlivých stĺpcov	1	-13.5350	0.00312	0.00296
	2	-12.3141	0.00105	0.00104
	3	0.2334	0.00267	0.00257
	4	-0.0283	**0.01087**	**0.01070**
	6	-6.1183	0.00195	0.00194
	7	7.4136	0.00612	0.00612
	8	6.7389	**0.80567**	**0.80575**
	9	-1.9034	0.00499	0.00508
	10	6.3233	**0.02330**	**0.02348**
	11	-0.3521	0.00851	0.00883
	12	5.7412	0.00669	0.00646
	13	7.7549	0.00659	0.00647
	14	-1.5894	**0.01464**	**0.01434**
	15	0.1598	0.00492	0.00519
	16	4.7767	**0.05703**	**0.05719**
	17	6.1656	0.00617	0.00608
	18	-0.9903	0.00554	0.00534
	19	-1.4998	0.00687	0.00707
	20	-5.7139	0.00362	0.00364
	21	2.4836	0.00494	0.00487
	22	2.7415	0.00476	0.00494
	23	**32.0461**	0.00012	0.00015
	24	19.8002	0.00016	0.00015
	25	9.7121	0.00007	0.00008
	26	**86.5156**	0.00131	0.00133
	27	**-147.3935**	0.00229	0.00201
	28	-13.3767	0.00119	0.00131
	29	26.6514	0.00060	0.00063
	30	-21.7642	0.00130	0.00125
	31	16.4235	0.00083	0.00088
	32	**-35.1313**	0.00064	0.00058
	33	22.9010	0.00025	0.00031
	34	**-129.3255**	0.00011	0.00010
	35	**44.0327**	0.00046	0.00048
	36	**117.1058**	0.00063	0.00068

Záver

O probléme maľovaných krížoviek je zdokumentovaných viacero výsledkov. V mojej práci sme chceli ukázať prístup k riešeniu krížoviek podobný, ako keď sa krížovku pokúša vyriešiť človek. Keďže človek rieši krížovku na základe logických dedukcií, rozhodli sme sa používať sadu logických pravidiel. Pre porovnanie sme implementovali algoritmus s polynomiálnou časovou zložitosťou, ktorá je založená na princípe dynamického programovania. Takto vyriešené krížovky nám poskytli informácie, na ktoré sme použili metódy strojového učenia. Na dátach, ktoré sme získali, sa nám podarilo predikovať testovacie dáta s **koeficientom determinovanosti** 0.90567**.**

Ako pokračovanie práce, za účely vylepšenia tohto koeficientu by bolo možné vymyslieť nové logické pravidlá, ktoré zvládnu vyriešiť aj zvyšných 5645 krížoviek. Taktiež je možné skúmať predikciu krížoviek, ak by sme riadky neriešili iterovaním, ale viac špecificky. Keďže všetky krížovky, ktoré sme mali k dispozícii bolo možné doriešiť pomocou nášho polynomiálneho algoritmu, bolo by zaujímavé skúmať krížovky, ktoré ním vyriešiť nedokážeme. Možné pokračovanie vidím aj v skúmaní zvyšných variantov maľovaných krížoviek ako zebra, farebné maľované krížovky či maľované krížovky s trojuholníkmi.

Literatúra

[1] História maľovaných krížoviek. http://puzzlygame.com/pages/nonogram_history/. Dostupné z http://puzzlygame.com/pages/nonogram_history/.

[2] Nonogram. Dostupné z https://en.wikipedia.org/wiki/Nonogram.

[3] activityworkshop.net. Nonograms - tutorial. [Citované 2002-April] Dostupné z http://activityworkshop.net/puzzlesgames/nonograms/tutorial.html.

[4] Daniel Berend, Dolev Pomeranz, Ronen Rabani, and Ben Raziel. Nonograms: Combinatorial questions and algorithms. *Discrete Applied Mathematics*, pages 30–42, 2014.

[5] Ling-Hwei Chen Chiung-Hsueh Yu, Hui-Lung Lee. An efficient algorithm for solving nonograms. 2009. [Published 2009-11-13] Dostupné z http://debut.cis.nctu.edu.tw/Publications/pdfs/J54.pdf.

[6] griddlers.net. Griddlers. Dostupné z https://www.griddlers.net/.

[7] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[8] Rosettacode.org. Nonogram solver. [Citované 2015-01-11] Dostupné z http://rosettacode.org/wiki/Nonogram_solver.

[9] Nobuhisa Ueda and Tadaaki Nagao. Np-completeness results for nonogram via parsimonious reductions. 1996. Dostupné z http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.57.5277&rep=rep1&type=pdf.

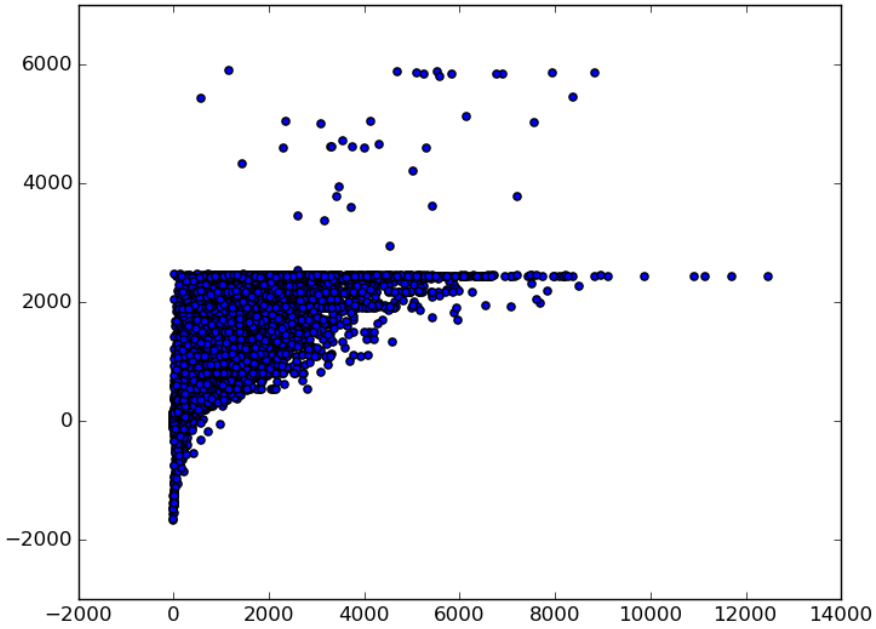
[10] Jan N. van Rijn. Playing games,the complexity of klondike, mahjong,nonograms and animal chess. 2012. Dostupné z http://liacs.leidenuniv.nl/assets/2012-01JanvanRijn.pdf.

[11] Wen-Li Wang and Mei-Huei Tang. Simulated annealing approach to solve nonogram puzzles with multiple solutions. 2014. Dostupné z `http://www.sciencedirect.com/science/article/pii/S1877050914013015`.

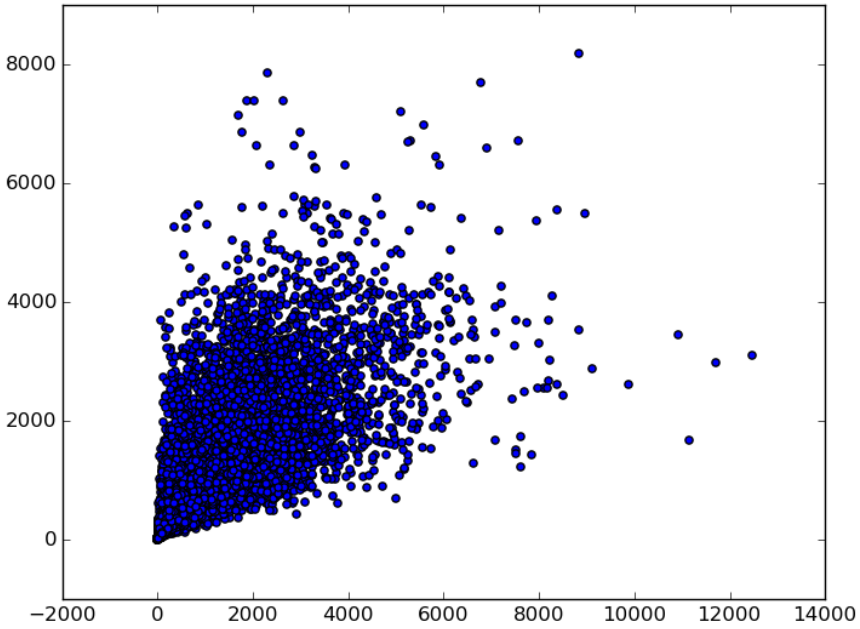
[12] W. A. Wiggers. A comparison of a genetic algorithm and a depth first search algorithm applied to japanese nonograms. 2009. [Published 2009-11-13] Dostupné z `http://debut.cis.nctu.edu.tw/Publications/pdfs/J54.pdf`.

Appendix A

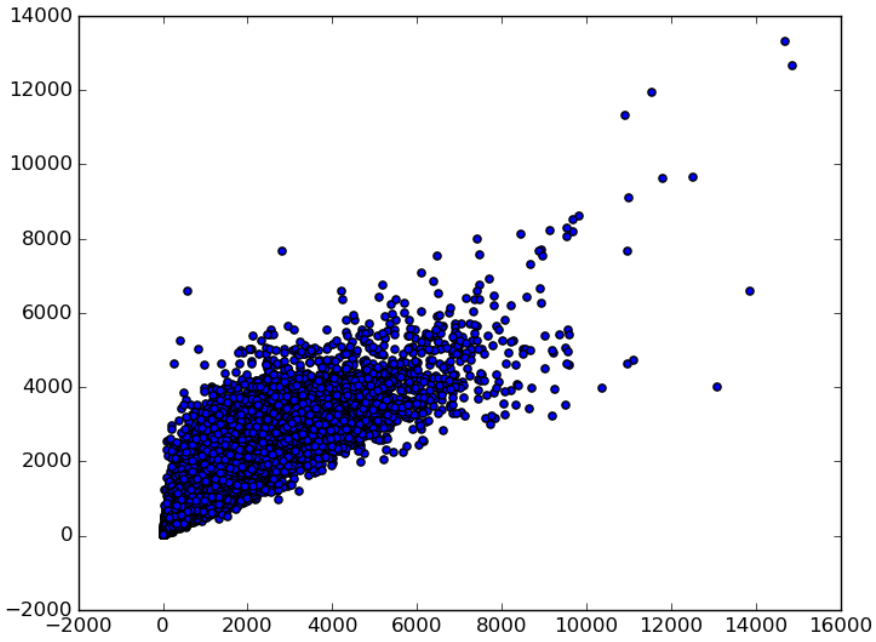
Príloha A obsahuje grafy reprezentujúce výsledky regresií z kapitoly 5



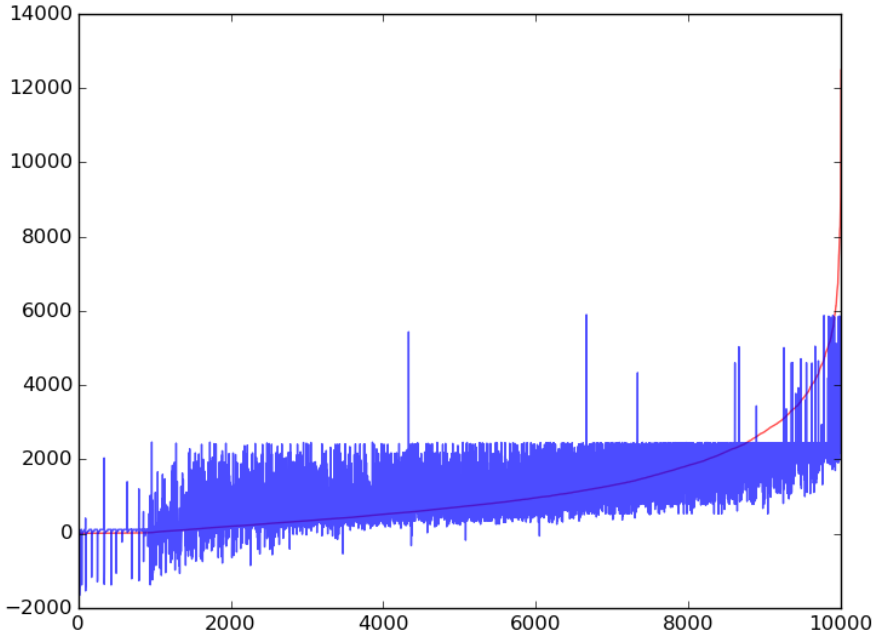
Graf prvého typu - **lineárna regresia** - k tab.5.1 - výsledky testovacích dát



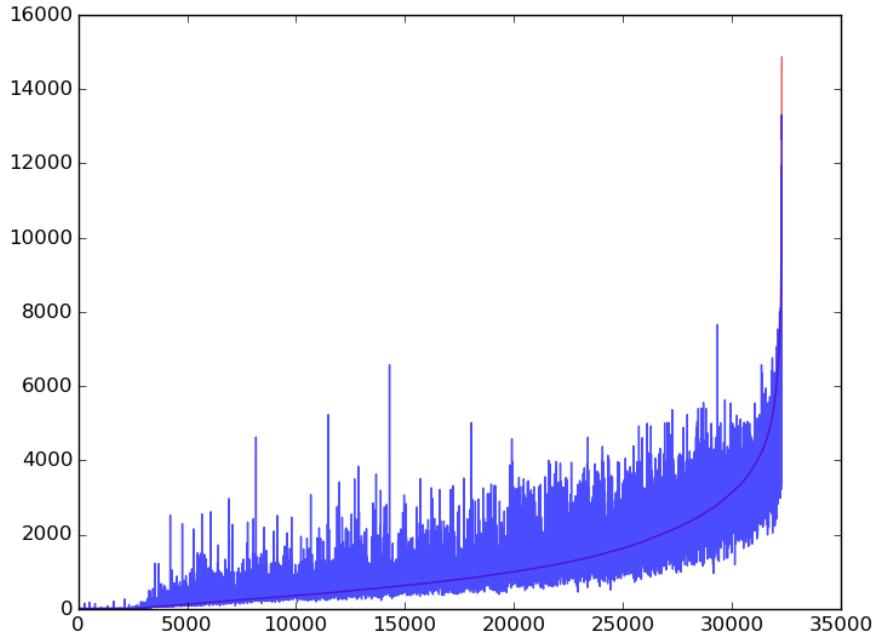
Graf prvého typu - **random forest regression -100** - k tab.5.1 - výsledky testovacích dát



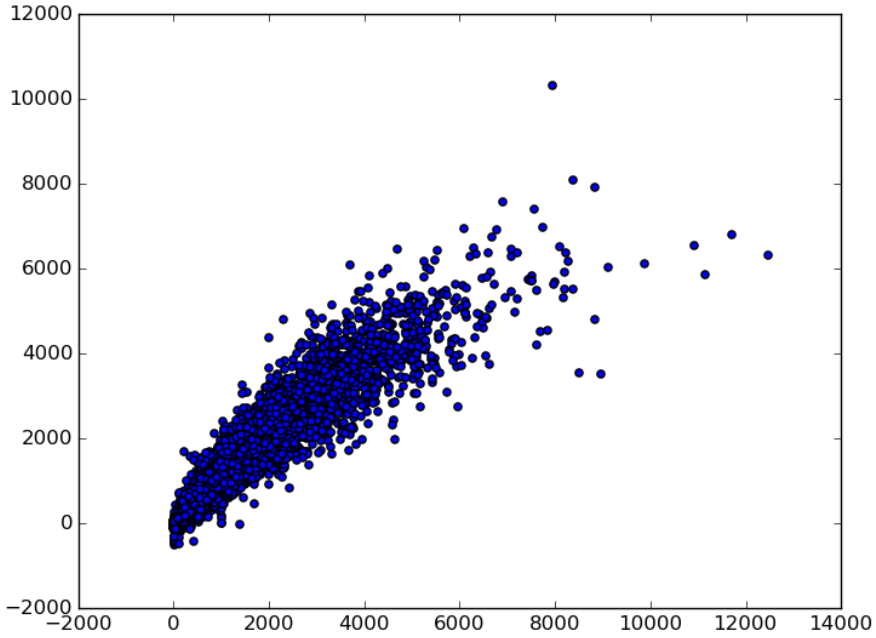
Graf prvého typu - **random forest regression -100** - k tab.5.1 - výsledky **trénovacích** dát



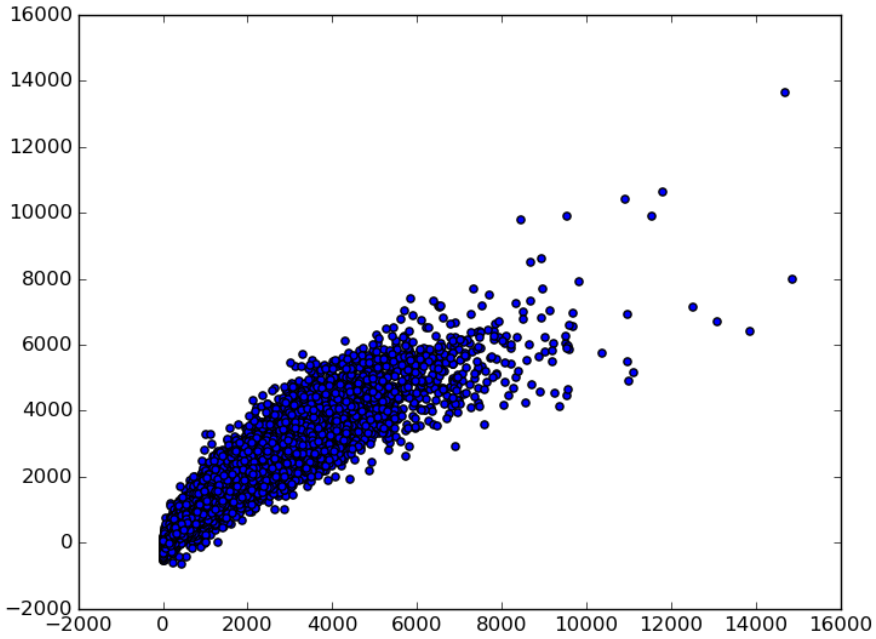
Graf druhého typu - **lineárna regresia** - k tab.5.1 - výsledky **testovacích** dát



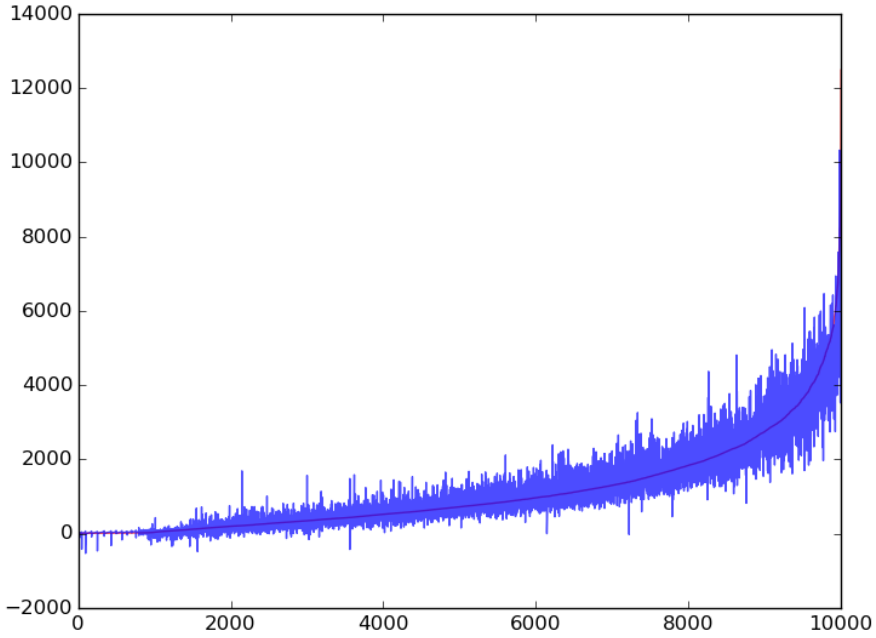
Graf druhého typu - **random forest regression -100** - k tab.5.1 - výsledky **testovacích** dát



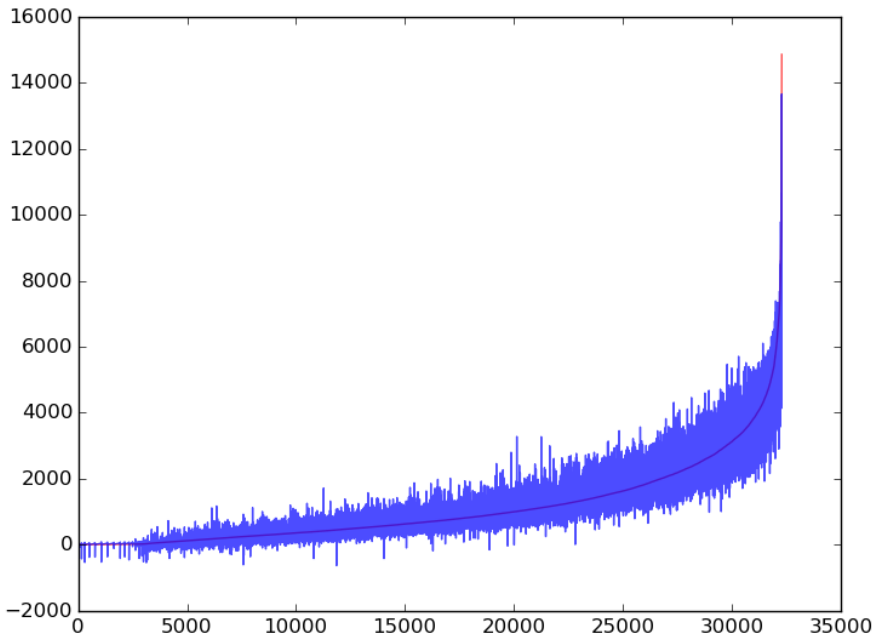
Graf prvého typu - **lineárna regresia** - k tab.5.2 - výsledky **testovacích** dát



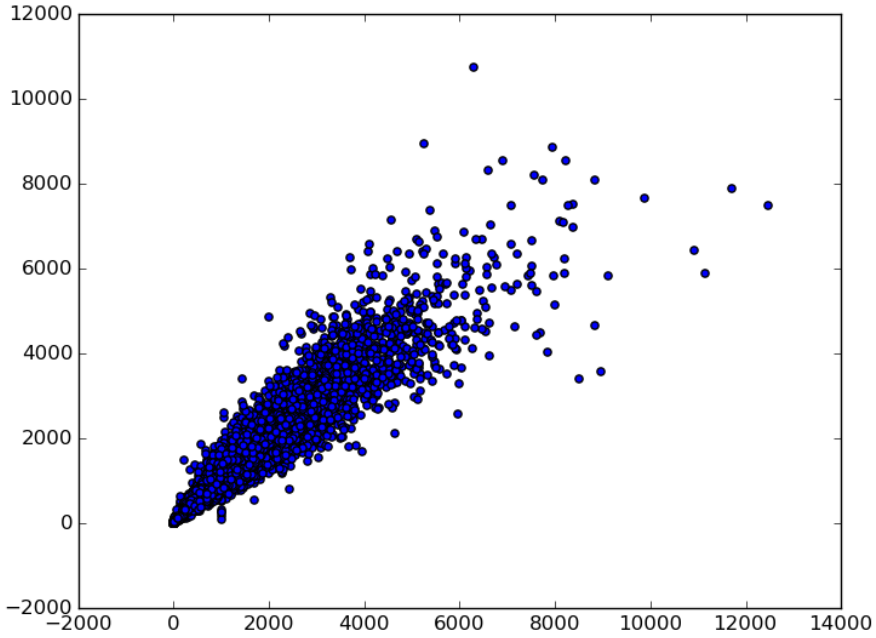
Graf prvého typu - **lineárna regresia** - k tab.5.2 - výsledky **trénovacích** dát



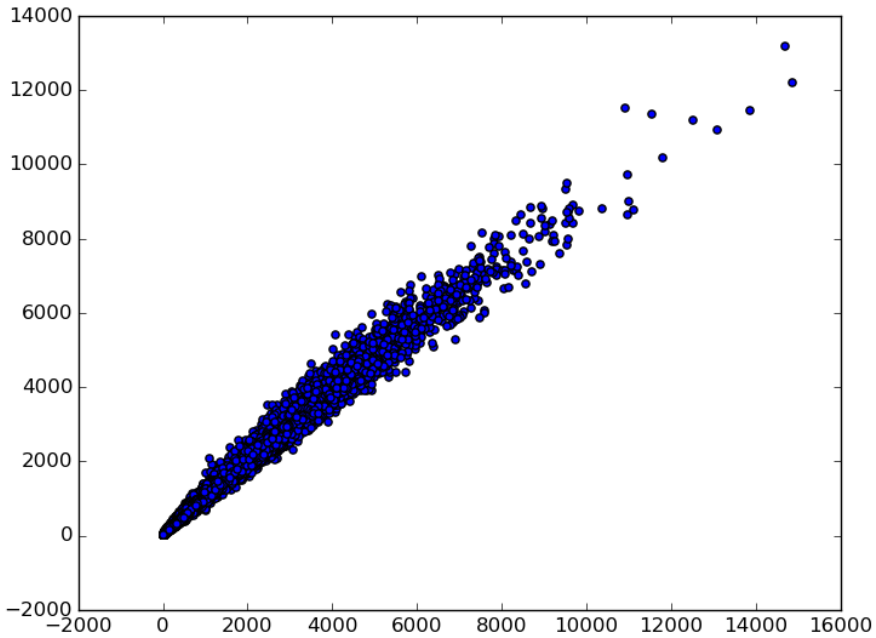
Graf druhého typu - **lineárna regresia** - k tab.5.2 - výsledky **testovacích** dát



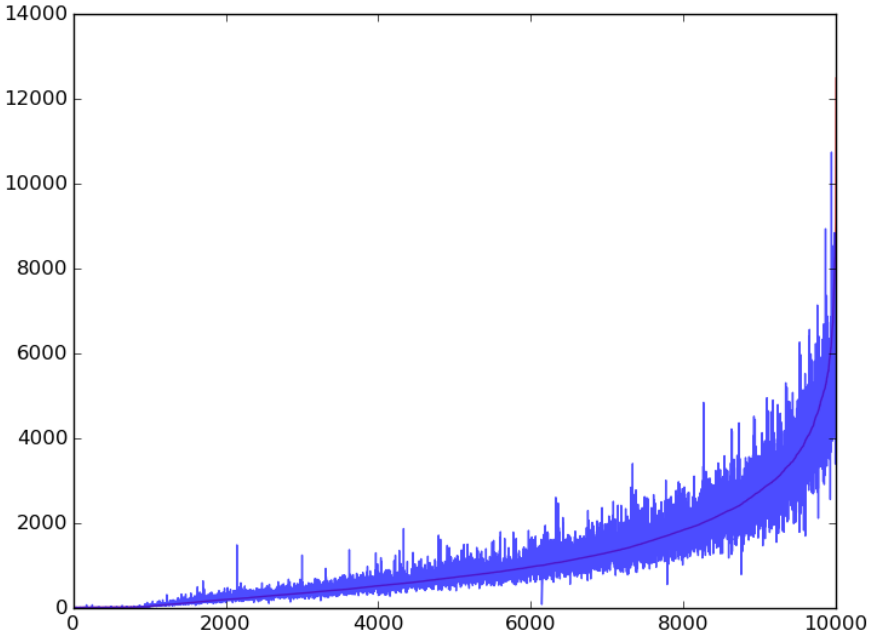
Graf druhého typu - **lineárna regresia** - k tab.5.2 - výsledky **trénovacích** dát



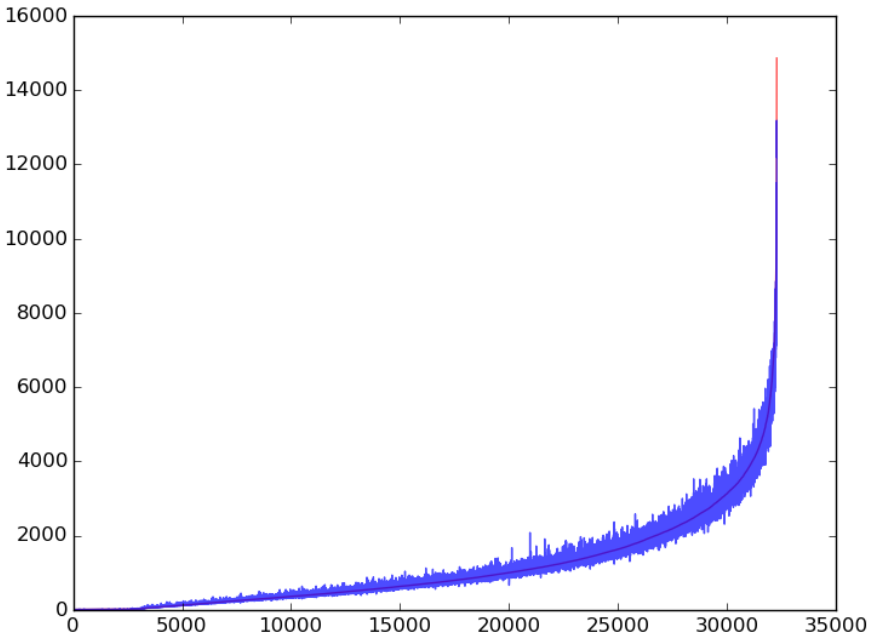
Graf prvého typu - **random forest regression -100** - k tab.5.2 - výsledky **testovacích** dát



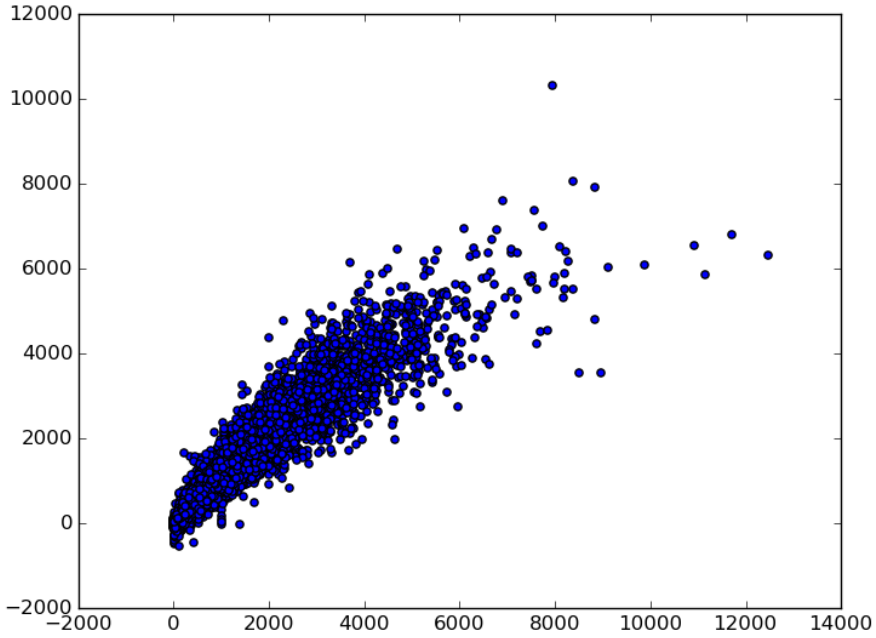
Graf prvého typu - **random forest regression -100** - k tab.5.2 - výsledky **trénovacích** dát



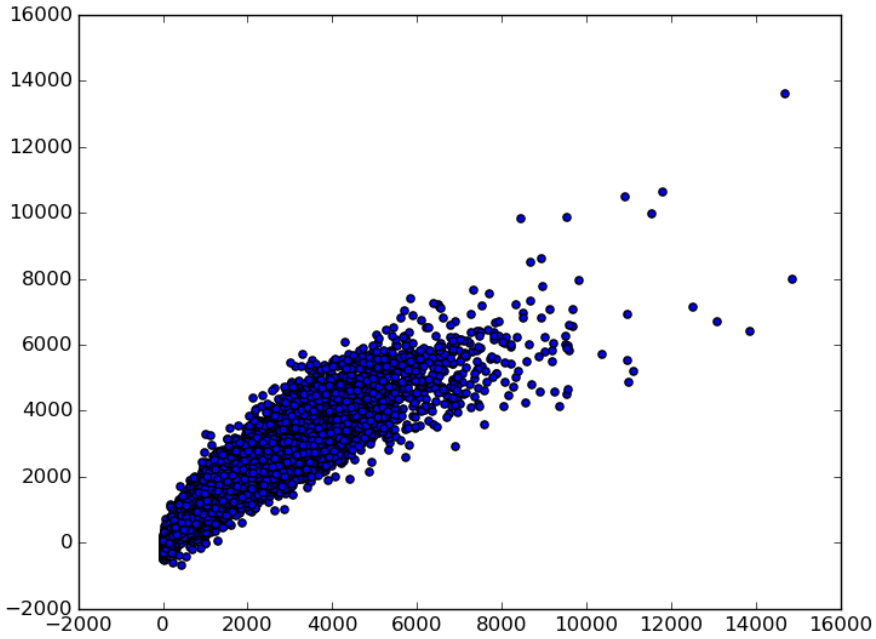
Graf druhého typu - **random forest regression -100** - k tab.5.2 - výsledky **testovacích** dát



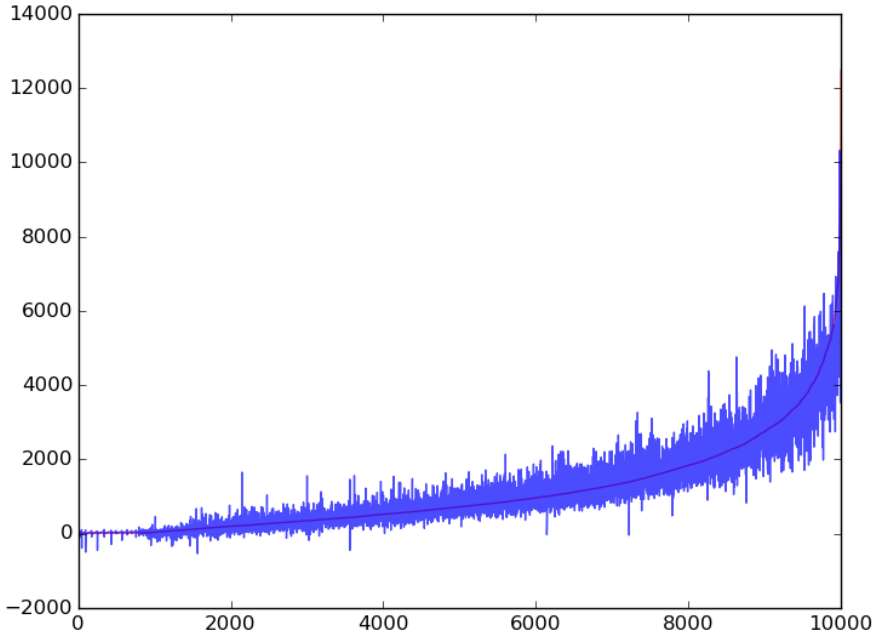
Graf druhého typu - **random forest regression -100** - k tab.5.2 - výsledky **trénovacích** dát



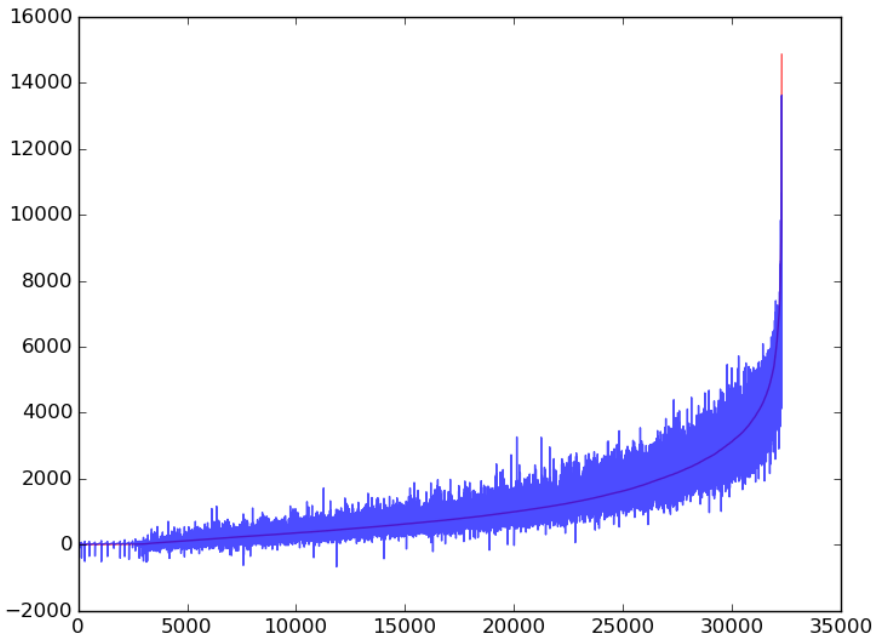
Graf prvého typu - **lineárna regresia** - k tab.5.3 - výsledky **testovacích** dát



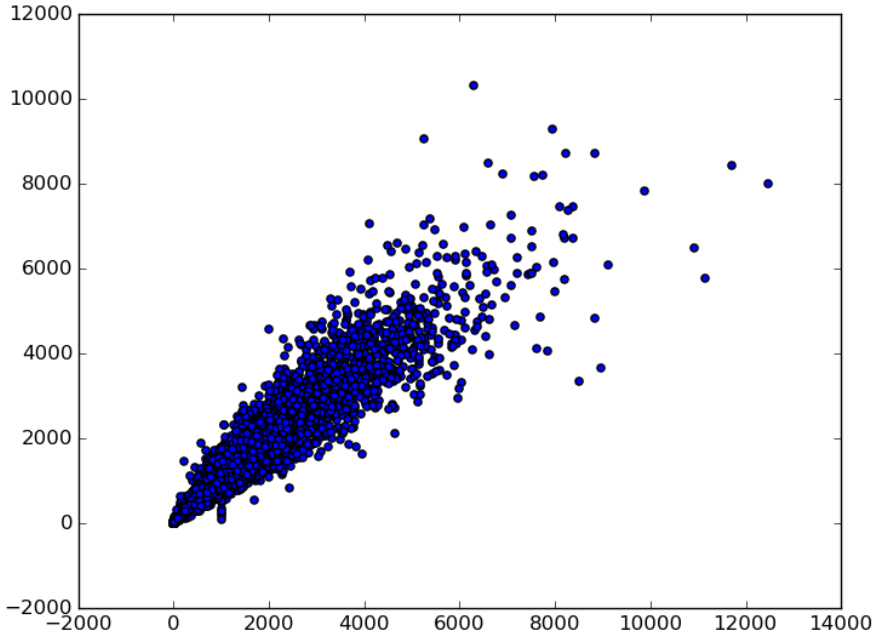
Graf prvého typu - **lineárna regresia** - k tab.5.3 - výsledky **trénovacích** dát



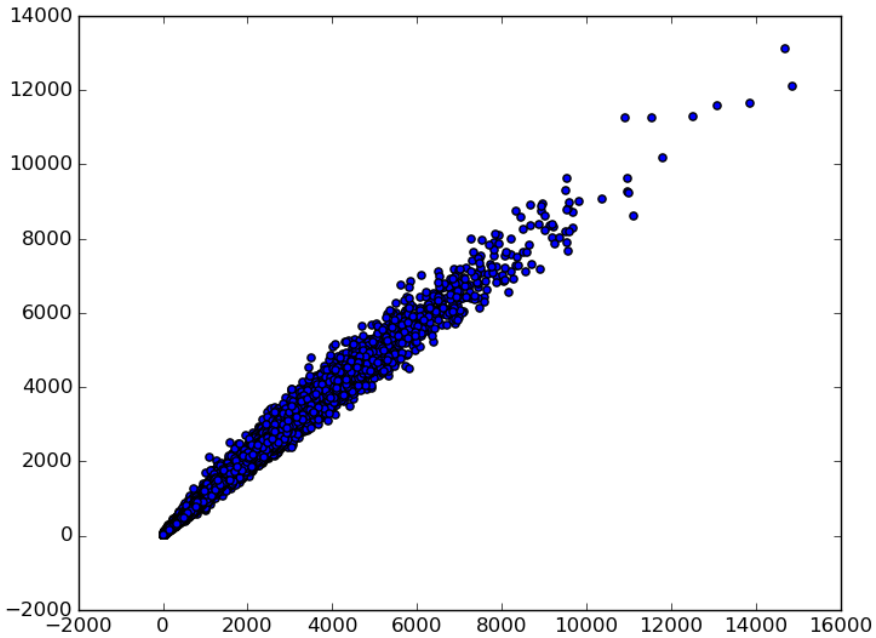
Graf druhého typu - **lineárna regresia** - k tab.5.3 - výsledky **testovacích** dát



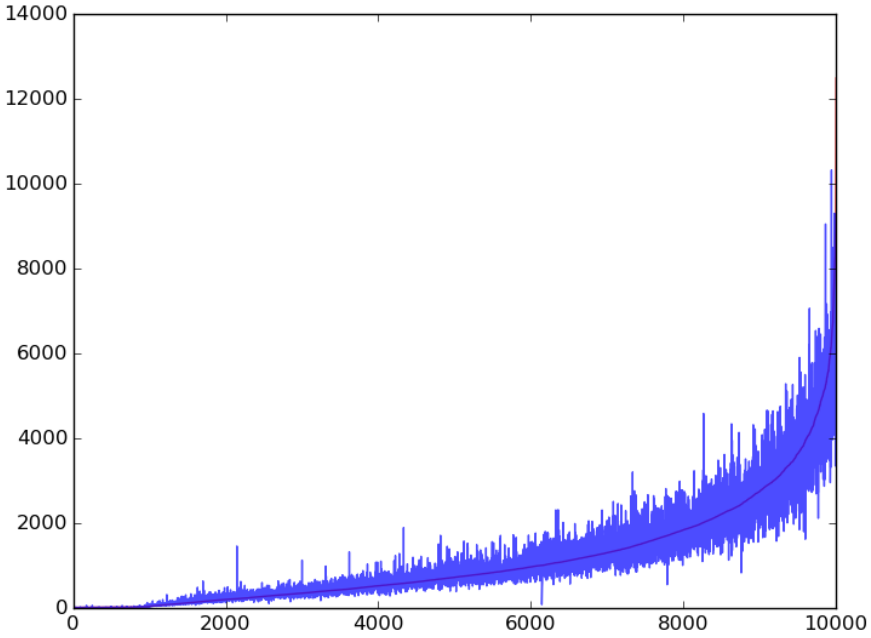
Graf druhého typu - **lineárna regresia** - k tab.5.3 - výsledky **trénovacích** dát



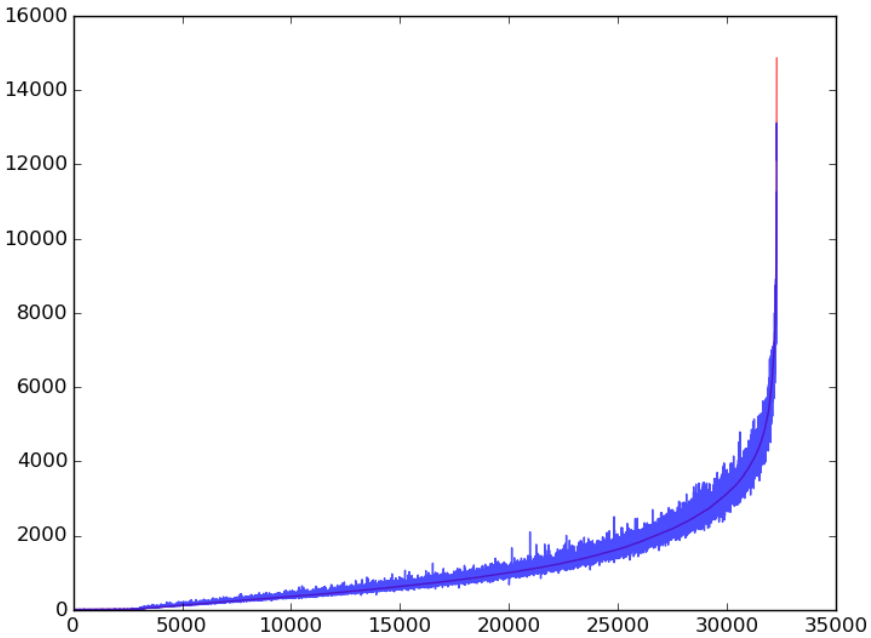
Graf prvého typu - **random forest regression -100** - k tab.5.3 - výsledky **testovacích** dát



Graf prvého typu - **random forest regression -100** - k tab.5.3 - výsledky **trénovacích** dát



Graf druhého typu - **random forest regression -100** - k tab.5.3 - výsledky **testovacích** dát



Graf druhého typu - **random forest regression -100** - k tab.5.3 - výsledky **trénovacích** dát

Appendix B

Príloha B obsahuje program, ktorý som počas práce vytvorila na riešenie krížoviek. Ďalej obsahuje súbor `parser.sh`, ktorým je možné vytvoriť zo súboru `Sets.txt` dáta, ktoré sa používali pri analýze. Súbor `reg.py` obsahuje program na regresiu, ktorou sme predikovali časy riešenia krížoviek. Všetky spomínané súbory možno nájsť na priloženom CD.