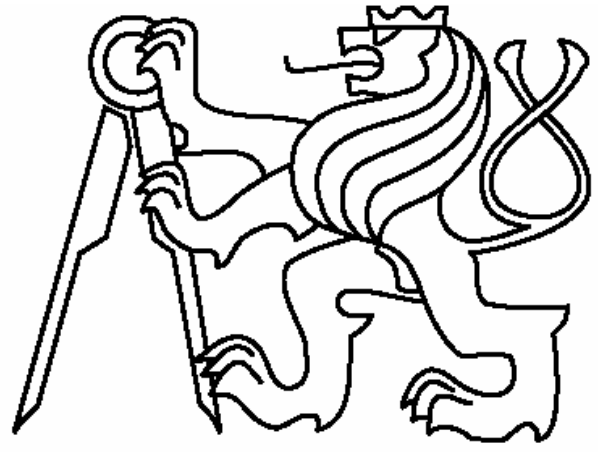
České vysoké učení technické v Praze
Fakulta elektrotechnická



Diplomová práce

Moderní metody řešení problému pokrytí

Bc. Lukáš Krejčík

Vedoucí práce : Ing. Petr Fišer

Studijní program: Elektrotechnika a informatika, strukturovaný, Navazující magisterský

Obor: Výpočetní technika

květen 2008

Poděkování

Mé poděkování patří ing. Petru Fišerovi za odborné vedení, trpělivou pomoc a veškerý čas, který mi věnoval při zpracovávání této diplomové práce.

Prohlášení
Prohlašuji, že jsem svou diplomovou práci vypracoval samostatně a použil jsem pouze podklady uvedené v přiloženém seznamu.
Nemám závažný důvod proti užití tohoto školního díla ve smyslu §60 Zákona č.121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Praze dne Bc. Lukáš Krejčík

Declaration

I hereby declare that I have completed this master thesis independently and that I have listed all the literature and publications used.
I have no objection to usage of this work in compliance with the act §60 Zákona č.121/2000 Sb. (copyright law), and with the rights connected with the copyright act including the changes in the act.

V Praze dne Bc. Lukáš Krejčík

Abstract

In this work is programmed current best exact covering problem solver AURAII, and on the basis of experimental results is built an adaptation mechanism with ability to select best covering method for given problem. Covering problem occurs in several fields of computer science, logic synthesis and reliability analysis. Solving the covering problem may be sometimes rather time-consuming, when based on exploring whole state space. In this work are programmed and tested new methods solving the covering problem with accent to solving speed and created adaptation mechanism able to select best solving method for given problem, with accent to quality of solution and solving time. I have been given materials mentioned in the References. Final adaptation mechanism is integrated into program BOOM.

Anotace

V této diplomové práci je naprogramována současná nejlepší exaktní metoda řešení problému pokrytí AURA II a na základě experimentálních výsledků vytvořen adaptační mechanizmus, schopný určit pro daný problém pokrytí nejlepší metodu pro nalezení řešení problému pokrytí. Problém pokrytí se objevuje v mnoha oblastech počítačového výzkumu, syntéze logických obvodů a analýze spolehlivosti. Řešení problému pokrytí je obecně velmi zdlouhavá záležitost, založená na prohledávání celého vyhledávacího prostoru. V této práci jsou naprogramovány a otestovány nové metody, které pro zadanou matici naleznou její minimální pokrytí, důraz je kladen na rychlost, a vytvořen adaptační mechanizmus schopný zvolit nejlepší metodu pro daný problém, s ohledem na kvalitu řešení a výpočetní čas. K tomuto účelu mi byly poskytnuty na prostudování materiály uvedené v seznamu literatury. Výsledný adaptační mechanizmus je začleněn do programu BOOM.

Obsah

Seznam tabulek

1 Úvod

Problém pokrytí má použití v mnoha oblastech počítačového výzkumu, syntéze logických obvodů, dvouúrovňové minimalizaci logických obvodů, analýze spolehlivosti a přesného kódování.

Problém pokrytí je obecně velmi zdlouhavá a časově náročná záležitost, pro jeho řešení byly navrženy algoritmy s různou složitostí a kvalitou řešení.

Řešení problému pokrytí může být dosaženo pomocí rekurzivního algoritmu, který zkouší smysluplné kombinace prvků řešení, dokud není nalezeno minimální pokrytí. Řešení tímto způsobem je velmi časově náročné. Redukce výpočetního času může být dosaženo například zamezením prohledávání prostoru v místech, kde se řešení nenachází.

Při hledání minimálního řešení máme však ještě jiné možnosti, např. použití heuristik popsaných v [1], [4], která vyhledá nějaké řešení pokrytí v rozumném čase, avšak za cenu horšího řešení. Pro problémy pokrytí o velkém počtu prvků je to obvykle jediné řešení z důvodu časové náročnosti.

Tato práce navazuje na mnou napsanou bakalářskou práci „Moderní metody řešení problému pokrytí“ z roku 2005, v které byly implementovány jednoduché heuristiky Natural a ContribAdd pro nalezení řešení problému pokrytí, z té práce je zde převzata kapitola 2. Dále jsou zde testovány heuristiky implementované v práci Kamila Kopejtka z roku 2007, „Moderní metody řešení problému pokrytí“.

Zde v této práci je uveden a implementován současný nejlepší exaktní algoritmus AURA II pro řešení problému unátního pokrytí, popsaný v [2], s mnou navrženými vylepšeními urychlující výpočet, dále je tento algoritmus patřičně otestován na generovaných a reálných maticích.

Pro problémy velkého rozsahu, kde již není možné kvůli časové náročnosti algoritmus AURA II použít, jsou zde dále otestovány heuristiky popsané v [1], [4], [6], [9], [10], [11] a pomocí takto získaných experimentálních výsledků je vytvořen adaptační mechanizmus schopný určit pro daný problém pokrytí nejvhodnější algoritmus pro nalezení řešení problému pokrytí s ohledem na výpočetní čas. Výsledný mechanizmus je zaimplementován do programu BOOM.

2 Problém pokrytí

V této kapitole shrneme základní pojmy, definice problému pokrytí a možnosti řešení.

Definice 2.1: *Nechť* $X=\{x_1,x_2,\ldots,x_n\}$ *je množina řádků a* $Y=\{y_1,y_2,\ldots,y_m\}$ *je množina sloupců v matici A o rozměrech mxn ,* <u>*cena*</u> *je funkce definovaná na* $Y \mid cena(y) \rightarrow R$, *která každému* $y \in Y$ *přiřadí kladné reálné číslo. Prvek* $y_j \in Y$ *pokrývá prvek* $x_i \in X$ *pokud A[i,j]=1,jinak A[i.j]=0. Problém pokrytí <X,Y,cena> spočívá v nalezení podmnožiny S množiny Y s nejmenší cenou, takové, že pro každý prvek x množiny X existuje prvek y množiny Y tak, že x R y.*

Nechť je dána matice *A* , všechny položky jsou 1 nebo 0. Řešení problému pokrytí spočívá v nalezení minimální podmnožiny sloupců matice *A* tak, že každý řádek matice *A* je pokrytý alespoň jedním sloupcem podmnožiny. Řádek x_i je pokrytý sloupcem y_j pokud *Aij = 1*. Sloupce jsou ohodnoceny kladným reálným číslem, zvaném cena sloupce. Označme si *UCP(A)* jako problém unátního pokrytí matice A.

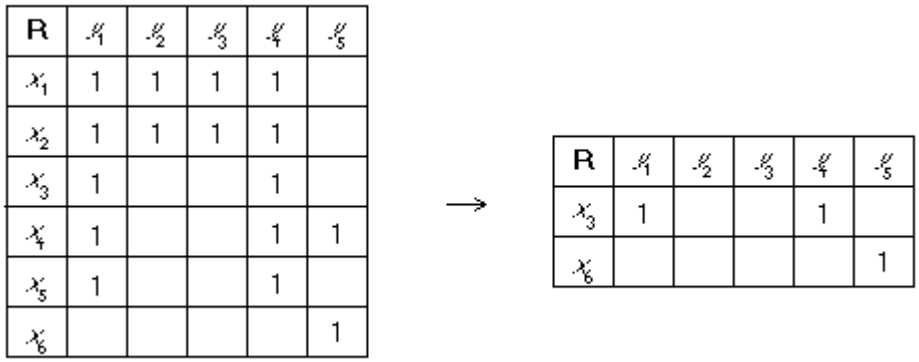
2.1 Dominance

Při řešení problému pokrytí můžeme zjistit, že řádek x_1 obsahuje jedničky všude, kde je obsahuje řádek x_2 a ještě na dalších místech. Při pokrývání to znamená, že pokud vybereme do řešení jakýkoliv sloupec pokrývající x_2, určitě pokryjeme i řádek x_1. Řádek x_1 může tedy být z matice odstraněn. Říkáme, že x_2 dominuje x_1. V následujícím odstavci si popíšeme, jak se odstraňují tyto dominance.

2.1.1 Dominance řádků

Definice 2.1.1: *Nechť <X,Y,cena> je problém pokrytí. Řádek x' množiny X všech řádků dominuje x právě tehdy, když všechny prvky množiny sloupců Y které pokrývají x' pokrývají také x .*

Jinými slovy,pokud nějaký řádek *x'* obsahuje ve sloupci jedničku stejně tak jako řádek *x* (*x* může obsahovat jedničky i na jiných místech), tak *x'* dominuje *x* , vždycky,když je pokryt řádek *x'*, bude pokryt i řádek *x*. Řádek *x* tedy může být vyškrtnut z matice. Algoritmus vždy vezme jeden řádek *x'* z matice a porovná ho s ostatními. Pokud nalezne řádek *x* obsahující stejný nebo vyšší počet jedniček jako řádek *x'*, je šance že řádek x' bude dominantní. Dále zjistíme, jestli *x* obsahuje jedničky všude, kde je obsahuje *x'*, vyškrtne přepojením spojového seznamu řádků tento řádek z matice. Jelikož se porovnává každý řádek s každým, operační složitost algoritmu v nejhorším případě je $O(m^2)$, kde *m* je počet řádků matice. Na obrázku 2.1.1 je znázorněna redukce matice před a po odstranění dominance řádků.

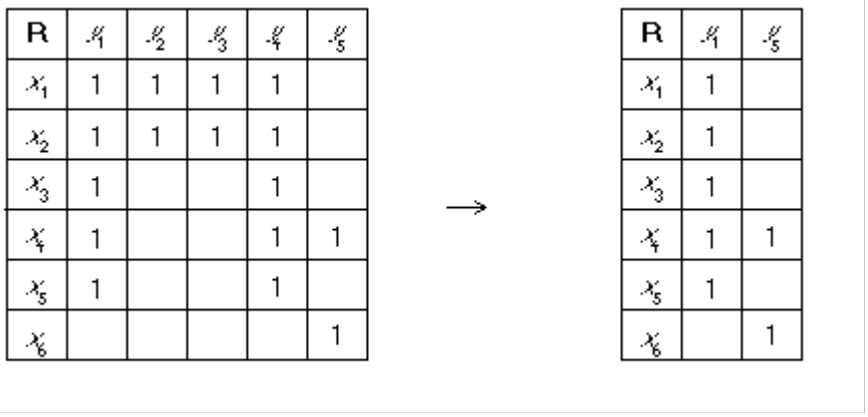


Obrázek 2.1.1 – Odstranění dominancí na množině řádků

2.1.2 Dominance sloupců

Definice 2.1.2: *Nechť <X,Y,cena> je problém pokrytí. Sloupec y' množiny všech sloupců Y dominuje sloupci y právě tehdy, když všechny prvky množiny řádků X pokryté y jsou také pokryté y'* .

To znamená, že pokud nějaký sloupec *y'* obsahuje jedničky všude, kde je obsahuje sloupec *y* (*y'* může obsahovat jedničky i na více místech), tak *y'* dominuje *y*. Algoritmus vezme sloupec *y'* z matice a porovná ho s ostatními sloupci. Pokud sloupec *y* obsahuje stejný nebo menší počet jedniček jako sloupec *y'*, je možnost že *y'* bude dominantní. Zjistíme, zda *y* obsahuje jedničky všude, kde je obsahuje *y'*, a je splněna podmínka *cena*(*y'*) = *cena*(*y*), *y* bude vyjmut z matice. Pokud podmínka splněna není, mohli bychom ztratit minimální řešení. Opět se porovnává každý sloupec s každým, takže operační složitost v nejhorším případě, kdy se nevyškrtne žádný sloupec, je $O\ (\ n^2\ )$, kde *n* je počet sloupců matice. Na obrázku 2.1.2 je znázorněna redukce matice před a po odstranění dominance sloupců.

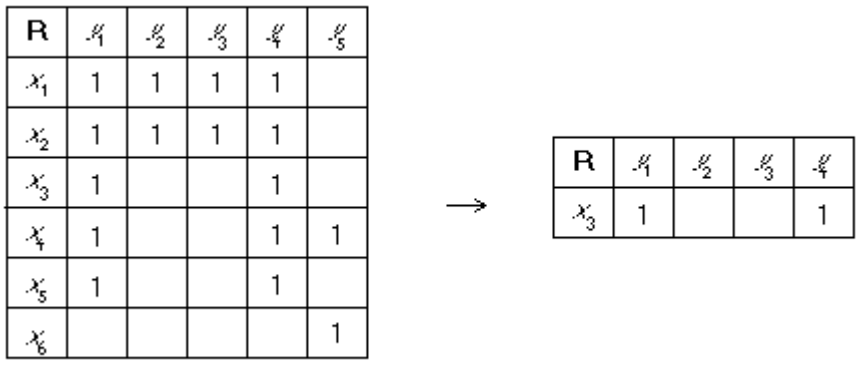


Obrázek 2.1.2 – Odstranění dominancí na množině sloupců

2.1.3 Nezbytné sloupce

Sloupec *y* množiny sloupců *Y* je nezbytný, pokud je jediný, který pokrývá řádek *x* množiny řádků *X*. Tyto sloupce vždycky musí patřit do řešení, můžeme je proto z matice vyřadit, a minimální řešení původního problému získáme přidáním těchto sloupců do

minimálního řešení redukovaného problému. Při řešení dominancí, pokud narazíme na řádek obsahující pouze jednu jedničku,přidáme sloupec který ji obsahuje do řešení. Na obrázku 2.1.3 je znázorněna redukce matice před a po odstranění dominance řádků, s výběrem nezbytných sloupců. Sloupec y_5 byl vybrán do řešení, protože řádek x_6 byl pokrytý pouze tímto sloupcem.



Obrázek 2.1.3 – Odstranění dominancí na množině řádků s výběrem nezbytných sloupců

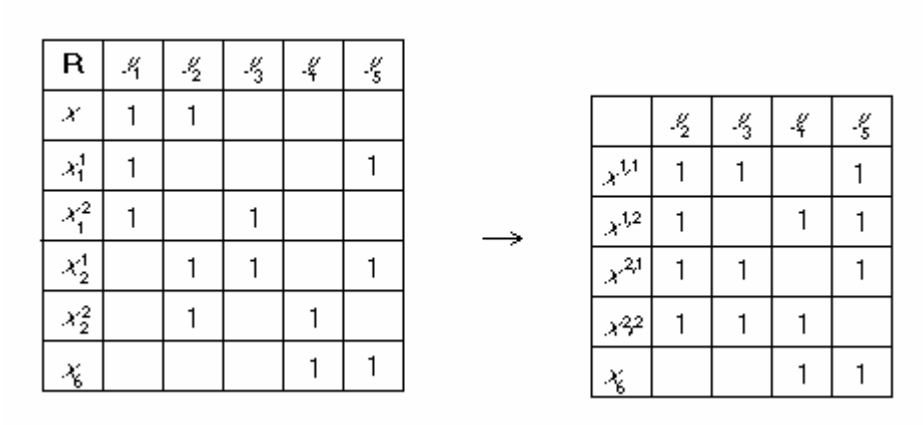
Při řešení dominancí sloupců můžou však v matici vzniknout nové dominance řádků a naopak. Musíme tedy zajistit opakované volání funkcí řešící dominance, dokud se v matici nějaké budou vyskytovat. Při nalezení nezbytných sloupců musíme tyto zahrnout také do řešení. Na matici aplikujeme algoritmy popsané výše tak dlouho,dokud je matice měněna řešením dominancí.

2.1.4 Gimpelova redukce

V matici se nám může stát, že některý řádek x je pokrytý pouze dvěma sloupci y_1 a y_2, které mají navíc stejnou cenu. Označme $\{x, x_1^1,..., x_1^n\}$ a $\{x, x_2^1,..., x_2^m\}$ množinu řádků pokrytých sloupcem y_1 resp. y_2. Gimpelova technika popsaná v [3] spočívá v odstranění sloupce y_1, odstranění $n + m + 1$ řádků pokrytých sloupci y_1 a y_2 a přidáním nových $n.m$ řádků $(1 \le i \le n)$ a $(1 \le j \le m)$ takových, že řádek x^{ij} je pokryt množinou sloupců

$$\{y \in Y \mid x_1^i \in y \vee x_2^j \in y\} - \{y_1\}$$

Nechť S je minimální řešení nově vzniklé matice. Pak minimální řešení původní matice je odvozen následovně. Pokud S je takové že $S \cap \{y \in Y \mid x_1^i \in y\} \neq 0$ pro $1 \le i \le n$,pak přidáme y_2 do S. V opačném případě přidáme y_1 do S.



Obrázek 2.1.4.1 – Gimpelova redukce

Příklad 2.1.4.1: Uvažujme matici na obrázku 2.1.4.1. Řádek *x* je pokryt pouze dvěma sloupci, y_1 a y_2, sloupce mají stejnou cenu. Všechny řádky pokryté sloupci y_1 a y_2 budou z matice odstraněny, dále bude odstraněn sloupec y_1.V tomto případě $n = m = 2$, musíme tedy přidat $n.m = 4$ řádky. Matice je získána následujícím způsobem: řádek $x^{1,1}$ je pokryt sloupci, které pokrývají řádky x_1^1 a x_2^1 bez sloupce y_1 ,řádek $x^{1,2}$ je pokryt sloupci, které pokrývají řádky x_1^1 a x_2^2 bez sloupce y_1 , řádek $x^{2,1}$ je pokryt sloupci, které pokrývají řádek x_1^2 a x_2^1 bez sloupce y_1 , a řádek x_2^2 je pokryt sloupci, které pokrývají řádky x_1^2 a x_2^2 bez sloupce y_1 . Výsledkem redukce je matice na pravé straně obrázku 2.1.4.1. Jedno minimální řešení je $S = \{y_3, y_5\}$, které protíná obě množiny $S \cap \{y \in Y \mid x_1^i \in y\}, 1 \le i \le 2$, tzn. $\{y_1, y_5\}$ a $\{y_1, y_3\}$, tudíž přidání y_2 do *S* vytvoří $\{y_2, y_3, y_5\}$, což je minimální řešení původní matice. Jiné minimální řešení je $S = \{y_2, y_5\}$. Množina S neprotíná množinu $\{y_1, y_3\}$, a přidání y_1 do *S* vytvoří $\{y_1, y_2, y_5\}$, což je stále minimální řešení původní matice.

Gimpelova redukce vytváří novou matici, které obsahuje o 1 méně sloupců (víme, že počet volání rekurze algoritmu je $2^{\wedge n}$, *n* je počet sloupců matice), ale zvýší počet řádků o $m.n - n - m - 1$. Přidáním těchto řádků nám vzniknou nové dominance řádků a sloupců, což vede k dalšímu zmenšení matice. Nicméně, z praktického hlediska je lepší provádět Gimpelovu redukci v případě, že je zaručena menší matice, čili pokud je $m.n - n - m - 1 \le 0$.

3 Řešení problému pokrytí

Pokud na matici aplikujeme výše uvedené techniky, dostaneme matici do stavu, kdy ji tyto procesy již více nezmenší. Zbylá matice se nazývá cyklické jádro, <*X,Y,cena*>. Pokud je prázdné, množina všech nezbytných prvků byla nalezena během procesů popsaných výše.

3.1 Metoda větví a hranic

Pokud cyklické jádro prázdné není, je třeba dále hledat minimální řešení. Nalezení přesného řešení problému pokrytí můžeme dosáhnout pomocí rekurzivního algoritmu, metodou větví a hranic popsané v [1],[2], spočívajícím ve výběru sloupců do řešení a hledáním smysluplné kombinace sloupců tak, aby byly pokryty všechny řádky a zároveň cena řešení byla co nejmenší. Algoritmus je schopen vyloučit z řešení takové kombinace sloupců, jejichž cena je vyšší než cena zatím nejlepšího nalezeného řešení.

Běh algoritmu je reprezentován binárním stromem. Kořen stromu představuje počáteční problém, hrany znamenají rekurzivní volání algoritmu, cesty do jednotlivých uzlů reprezentují částečné nalezené řešení. List je dosažen v případě, že jsme nalezli řešení, nebo jsme překročili cenu nejlepšího nalezeného řešení. Branch and bound je popsán algoritmem 3.1.1.

```
Branch_And_Bound (A, Cesta, uBound )
{

        solve_dominance();
        /* odstraní dominantní řádky a sloupce, vybere do řešení nezbytné sloupce */
        if ( A= ∅ ) bestSolution = Path to node
        else {
                j=select_column(A);
                Path1 = Path_to_node + j;
                A' = A – {j + row covered by j}
                Solution1 = Branch_And_Bound (A',Path1, uBound)
                if ( Solution1 ≠ ∅ ) uBound = Solution1
                A'' = A – {j}
                Solution2 = Branch_And_Bound (A'',Path_to_node, uBound)
                bestSolution = min(Solution 1, Solution 2)
        }
        Return bestSolution
}
```

Algoritmus 3.1.1 - Branch and Bound

Tento algoritmus můžeme vylepšit několika způsoby. Předpokládejme, že známe hodnotu $L(A_N)$, reprezentující minimální hodnotu řešení potřebnou pro pokrytí matice *A*. Tato hodnota nám poskytuje spodní hranici řešení matice *A*. V takovém případě, pokud

$$|cena\_nejlepšího\_řešení| \leq L(A_N) + |cesta\_do\_uzlu|$$

můžeme zastavit rekurzi a navrátit se zpět.

Snížit počet volání rekurze můžeme dvěma způsoby. Snížením horní hranice, čili nalezením lepšího řešení, než je doposud nalezené a zvýšením spodní hranice $L(A_N)$. Otázkou je, jak nalézt tuto spodní hranici. Předpokládejme, že v matici nalezneme množinu řádků takovou, že každý řádek z této množiny je pokryt sloupci, které nepokrývají žádné jiné řádky z této množiny. Nazvěme tuto množinu množina nezávislých řádků. Čím přesněji nalezneme množinu nezávislých řádků, tím větší bude spodní hranice. Cílem je nalézt maximální množinu nezávislých řádků, nazvěme ji *MSIR* (*maximum set of independent rows*). Právě velikost *MSIR* nám dává spodní hranici pro řešení problému $L(A_N)$. Je zřejmé, že do řešení musíme vybrat minimálně |*MSIR*| sloupců pro pokrytí matice. Nyní označme :

$$K(A_N) = |cena\_nejlepšího\_řešení| - L(A_N) + |cesta\_do\_uzlu|$$

Pokud je $K(A_N) \leq 0$ je třeba zastavit rekurzi, v opačném případě je cena částečného nalezeného řešení dosti daleko od ceny nejlepšího nalezeného řešení a očekáváme velké množství rekurzivního volání algoritmu. Hledání množiny *MSIR* však znamená nalézt řešení dalšího NP-úplného problému, a čas pro hledání *MSIR* by jistě převýšil čas uspořený nalezením přesné spodní hranice. Musíme proto pro nalezení *MSIR* zvolit rychlou heuristickou funkci. Jednoduchá heuristika pro nalezení *MSIR* je v [1]. Čím přesněji však nalezneme *MSIR*, tím vyšší bude spodní hranice a dosáhneme urychlení výpočtu.

3.2 Vylepšená heuristika pro nalezení MSIR

Jednoduchá heuristika popsaná v [1] vybírá z matice řádek s nejméně jedničkami, protože je vysoká šance, že pokud budeme vybírat tyto řádky do množiny *MSIR*, budou mezi sebou nezávislé.

Pro dosažení lepších výsledků jsem navrhl lepší algoritmus, který vybírá řádky podle tzv. řádkového příspěvku *RP*, definovaného

$$RP(x_i) = \sum_{j=1}^{N} A[i, j] \cdot SP(y_j)$$

kde *N* je počet sloupců matice *A*, a *SP* je tzv. sloupcový příspěvek, definovaný

$$SP(y_j) = \sum_{i=1}^{M} A[i, j].$$

Čím nižší bude $RP(x_i)$, tím vyšší šance bude na nezávislost řádků. Jinými slovy, vybíráme řádky, které jsou minimálně pokryty a toho pokrytí je tvořeno sloupci, které pokrývají nejméně řádků. Vylepšenou heuristiku pro nalezení *MSIR* zobrazuje algoritmus 3.2.1.

```
FindMSIR (A) {
        bestRow = firstRow (A)
        for each row ∈ A {
                r = find_row_with_min1 (A)
                if (length (r) < bestRow ) bestRow = r
                else if ( length (r) = bestRow )
                        if ( RP(r) < RP( bestRow ) bestRow = r
        }
}
```

Algoritmus 3.2.1 – Vylepšení heuristiky pro nalezení MSIR

Praktické výsledky jsou uvedeny v tabulce 3.2.1. Bylo měřeno několik matic generovaných programem Boom, v tabulce je uveden počet sloupců cyklického jádra, počet řádků cyklického jádra a hustota matice. Naměřené časy ukazují, že touto heuristikou dosáhneme přesnější nalezení množiny *MSIR* a tím urychlení výpočtu.

Počet řádků	Počet Sloupců	Hustota	Puvodní čas	Vylepšená heuristika MSIR
49	89	7,43	12,41	7,84
117	96	4,67	55,18	46,43
65	102	5,2	5,19	2,23
86	116	3,69	64,74	34,23
101	134	3,47	73,62	51,06

Tabulka 3.2.1 – Porovnání časů heuristik pro nalezení MSIR

3.3 Vylepšení metody větví a hranic

V této kapitole si stručně shrneme rekurzivní algoritmus metoda větví a hranic (*Branch and Bound*), popsaný v [1],[2], vylepšený počítáním spodní hranice, do kterého následně zahrneme režim „negativního“ přístupu.. Vstupní parametry takového algoritmu jsou:

- Matice A
- Cesta do uzlu, reprezentující částečné nalezené řešení
- Spodní hranici (*lBound*)
- Horní hranici, danou nejlepším nalezeným řešením (*uBound*)

Takto vylepšený *Branch_And_Bound* je popsán algoritmem 3.3.1.

```
Branch_And_Bound (A, Path, lBound, uBound )
{

    solve_dominance();
    /* odstraní dominantní řádky a sloupce, vybere do řešení nezbytné sloupce */
    find_MSIR(A);
    /*Nalezne MSIR pro výpočet spodní hranice*/
    lBoundNew = cost(Path) + cost(MSIR)

    if (lBoundNew ≥ uBound ) bestSolution = ∅
    if ( A= ∅ ) bestSolution = Path
    else {
        j=select_column(A);
        Path 1 = Path + j;
        A' = A – {j + row covered by j}
        Solution1 = Branch_And_Bound (A',Path1, lBoundNew, uBound)
        if (Solution 1 ≠ ∅ ) uBound = Solution 1
        if (Solution 1 ≠ ∅ & cost(Solution 1) = lBoundNew ) return Solution 1
        /*pokud jsme již dosáhli lBound, nebudeme větvit dále*/
        A'' = A – {j}
```

```
            Solution 2 = Branch_And_Bound (A'', Path, lBoundNew, uBound)
            bestSolution = min(Solution 1, Solution 2)
        }
        Return bestSolution
}
```

Algoritmus 3.3.1 - Branch and Bound s výpočtem spodní hranice

Algoritmus nejprve zavolá funkci *solve_dominance*, která z matice odstraní dominantní sloupce a řádky, a vybere do řešení všechny nezbytné sloupce, které následně odstraní z matice. Dále pokud cyklické jádro není prázdné, spočítá MSIR a spočítá novou spodní hranici. Pokud jsme již přesáhli spodní hranicí cenu nejlepšího nalezeného řešení, algoritmus vrátí prázdné řešení, v opačném případě začne větvení algoritmu podle výběru sloupců. Vybraný algoritmus tedy používá pro řešení problému pokrytí „pozitivní“ přístup.

Metodu větví a hranic můžeme vylepšit dále, snížením horní hranice. Čím nižší zřejmě bude horní hranice, tím méně větvení bude třeba. Jelikož takto navržený algoritmus vybírá sloupce z matice *A* pro větvení postupně, nalezne pravděpodobně nejprve dosti neoptimální řešení UCP(A). Předpokládejme však, že již první nalezené řešení by bylo „téměř“ optimální, horní hranice by tím pádem byla nižší a počet větvení algoritmu by byl snížen. Navrhl jsem tedy pro výběr sloupce z matice A heuristiku, která se bude snažit vybírat takové sloupce, jejichž přidání do částečného řešení nám pomůže nalézt řešení rychleji a přesněji. Pro výběr jsem použil dvě strategie, první z nich je realizována funkcí *select_column (A)*, která vybírá sloupce podle počtu řádků pokrytých sloupcem, vybere se sloupec pokrývající nejvíce řádků. Funkce *select_column_contrib (A)* používá pro výběr sloupce strategii použitou v heuristické funkci *ContribAdd* popsanou v [1], jejíž použití nám zaručuje, že první nalezené řešení bude rovné jako řešení nalezené touto heuristickou funkcí. Tato strategie výběru vybírá sloupec s nejvyšší silou pokrytí *SP(y)*, kde

$$SP(y_j) = \sum_{i=1}^{N} A[i, j] \cdot SP(x_i)$$ *,s ohledem na definici 2.1*

a

$$SP(x_i) = \frac{1}{\sum_{j=1}^{M} A[i, j]}$$ *,s ohledem na definici 2.1*

Je tedy vybrán takový sloupec, který pokrývá nejvíce takových řádků, které jsou pokryty nejméně sloupci.

Příklad 3.3.1: Uvažujme následující matici.

s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	s_9
		1		1			1	1
				1		1		
				1	1			1
1		1	1					
					1	1	1	
			1					1
	1							

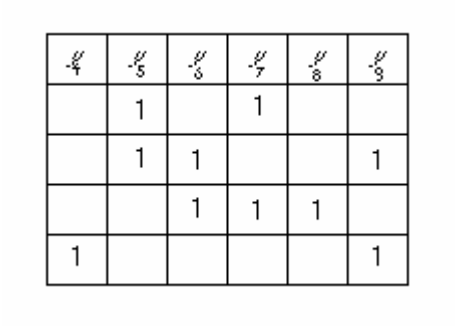
Pokud budeme do řešení vybírat sloupce postupně, bude do nalezení řešení třeba 6 kroků, viz níže.

s_2	s_3	s_4	s_5	s_6	s_7	s_8	s_9
	1		1			1	1
			1		1		
			1	1			1
				1	1	1	
		1					1
1							

Výběr sloupce č.1

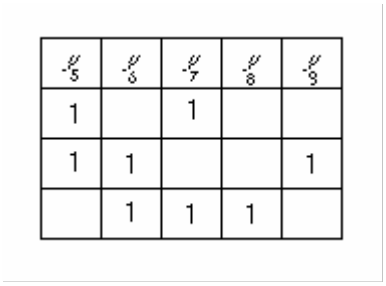
s_3	s_4	s_5	s_6	s_7	s_8	s_9
1		1			1	1
		1		1		
		1	1			1
			1	1	1	
	1					1

Výběr sloupce č.2



	1		1		
	1	1			1
		1	1	1	
1					1

Výběr sloupce č.3



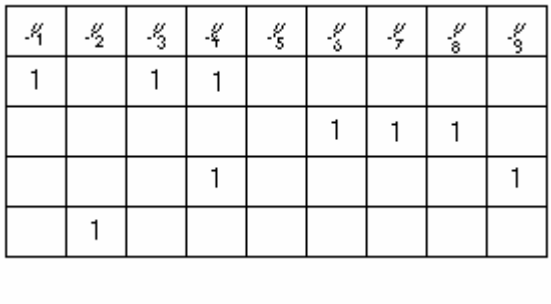
1		1		
1	1			1
	1	1	1	

Výběr sloupce č.4

1	1	1

Výběr sloupce č.5, dále výběr slupce č.6.

Poté již zbude prázdná matice a horní hranice je nastavena na 6 po 6ti krocích. Pokud však zvolíme strategii výběru sloupců podle počtu pokrytých řádků sloupcem, bude výběr vypadat takto:



1		1	1					
					1	1	1	
			1					1
	1							

Výběr sloupce č.5

s_2	s_3	s_7	s_8
	1	1	1
1			

Výběr sloupce č.4

s_3	s_7	s_8
1	1	1

Výběr sloupce č.2, dále např. č.6

Vidíme, že počet kroků se snížil z 6 na 4 a horní hranice byla nastavena na 4. Tyto strategie výběru nám tedy zároveň zajistí rychlejší zmenšování matice při větvení.

Praktické výsledky jsou uvedeny v tabulce 3.3.1. Měření bylo provedeno na maticích zmíněných v kapitole 3.2. Naměřené hodnoty ukazují, že pomocí těchto heuristik dochází k velkému urychlení výpočtu, nejlepšího času je dosaženo při použití funkce *select_column_contrib (A).*

Počet řádků	Počet Sloupců	Hustota	Puvodní čas	Select column	Select column contrib
49	89	7,43	12,41	0,23	0,09
117	96	4,67	55,18	4,23	1,29
65	102	5,2	5,19	0,22	0,09
86	116	3,69	64,74	4,29	2,36
101	134	3,47	73,62	5,23	0,64

Tabulka 3.3.1 – Porovnání časů heuristik výběru sloupce

3.4 Negativní přístup

Předpokládejme však, že již není možnost, jak zlepšit nejlepší nalezené řešení, ale hodnota $K(A_N) > 0$. Algoritmus má dvě možnosti, první z nich je „***pozitivní***“, rekurzivní algoritmus musí pokračovat ve větvení, aby se přesvědčil, že se nikde nenalézá lepší řešení, ve snaze lepší řešení nalézt. Rekurzivní algoritmus však zpočátku nalezne řešení blízké minimálnímu řešení matice a většinu času pak právě tráví tím, že prochází vyhledávací prostor ve snaze přesvědčit se, že se nikde nenalézá lepší řešení. Druhá možnost, navržená v [2], je „***negativní***“, algoritmus se snaží dokázat, že se nikde již lepší řešení nenalézá. Pokud je tedy hodnota $K(A_N)$ nízká, je přirozenější pro algoritmus být „skeptický“ ohledně nalezení lepšího řešení.

Budeme se snažit při řešení problému pokrytí zahrnout obě techniky, pozitivní přístup a negativní přístup. K tomu musíme upravit rekurzivní algoritmus, zpočátku začneme řešit problém v režimu pozitivního přístupu, poté, co počet vzniklých podproblémů je dostatečně veliký, každý podproblém je řešen v režimu negativního přístupu. Čím menší je hodnota $K(A_N)$, tím přirozenější je řešení podproblému v režimu negativního přístupu.

Nechť je tedy *P* podproblém, který má být řešen v režimu negativního přístupu a předpokládejme, že pokud je cena *(P)* vetší než cena nejlepšího nalezeného řešení, tak *P* nemůže vrátit řešení lepší než je nejlepší nalezené řešení. Cílem negativního přístupu je dokázat, že neexistuje řešení *P* s menší cenou, než je cena nejlepšího nalezeného řešení.

Problém je dobré řešit postupně. Začneme s podproblémem *P'* problému *P* tak, že řešení *P'* budou reprezentována v kompaktní formě. Poté měníme *P'* s cílem přiblížit ho původnímu *P* a přepočítáváme množinu řešení takto změněného problému. Snažíme se nalézt co největší „překážky“ v postupu od *P'* k *P* ve snaze dokázat, že řešení *P'* nepřekoná tyto překážky a nemůže být rozšířeno na řešení *P*. Reprezentace řešení získaných takovým způsobem však roste exponenciálně s počtem řádků matice. Budeme se proto snažit řešení, která podobným způsobem dokazují neexistenci lepšího řešení *P*, než je nejlepší nalezené řešení, uchovávat v určitém balíku řešení.

3.5 Začlenění negativního přístupu do metody větví a hranic

Pokud však již není šance vylepšit stávající nejlepší nalezené řešení, ale hodnota $K(A_N) > 0$, je přirozenější pro algoritmus pokusit se dokázat, že se již nikde nenalézá lepší řešení. Provedeme tedy úpravu algoritmu, začleněním techniky „negativního“ přístupu, popsané v [2]. Pokud je hodnota $K(A_N)$ velmi malá, algoritmus použije „negativní“ přístup namísto „pozitivního“ větvení podle výběru sloupců a problém řešit postupně, jak již bylo zmíněno v kapitole 3.4. Upravený Branch and bound popisuje algoritmus 3.5.1. Tučný text odpovídá začlenění postupného řešení. „Negativní“ přístup je prováděn funkcí Raiser, která je volána v případě, že rozdíl mezi spodní a horní hranicí je rovno konstantě *maxRaiser*. Raiser je volán s hodnotou *n*, v případě, že spodní hranice daná velikostí *MSIR* není dosti vysoká k oříznutí větve vyhledávacího prostoru, ale zvýšení právě o *n* oříznutí dovolí. Zpočátku je hodnota *n* nastavena na hodnotu *maxRaiser*.

```
Branch_And_Bound (A, Path, lBound, uBound )
{

    solve_dominance();
    /* odstraní dominantní řádky a sloupce, vybere do řešení nezbytné sloupce */
    find_MSIR(A);
    /*Nalezne MSIR pro výpočet spodní hranice*/
    lBoundNew = cost(Path) + cost(MSIR)

    if (lBoundNew ≥ uBound ) bestSolution = ∅
    else if (uBound – lBoundNew <= maxRaiser ) {
            /* pokud je hodnota uBound – lBoundNew dostatečně malá, zvolíme
             "negativní" přístup/*

            n = uBound – lBoundNew; /*určíme n přesně*/
```

```
            return raiser (MSIR,n,A)
            }

    else if ( A=∅ ) bestSolution = Path
    else {
            j=select_column(A);
            Path 1 = Path + j;
            A' = A – {j + row covered by j}
            Solution1 = Branch_And_Bound (A',Path1, lBoundNew, uBound)
            if (Solution 1 ≠ ∅ ) uBound = Solution 1
            if (Solution 1 ≠ ∅ & cost(Solution 1) = lBoundNew ) return Solution 1
            /*pokud jsme již dosáhli lBound, nebudeme větvit dále*/
            A'' = A – {j}
            Solution 2 = Branch_And_Bound (A'', Path, lBoundNew, uBound)
            bestSolution = min(Solution 1, Solution 2)
    }
    Return bestSolution
}
```

Algoritmus 3.5.1 - Branch and Bound s režimem „negativního“ přístupu

Hodnota *n* by měla být nastavena na malou hodnotu, v rozmezí 2 - 4. Je to proto, že pokud hodnota *n* je velmi malá, je pro algoritmus přirozenější být skeptický ohledně nalezení lepšího řešení. Navíc můžeme využít faktu, že problém pokrytí *MSIR* má regulární prostor řešení.

Zvýšením spodní hranice i o takto malou hodnotu můžeme dosáhnout velkého omezení výpočetního času.

3.6 Postupné zlepšování spodní hranice

Zde si popíšeme způsob, jak postupně zlepšit spodní hranici problému. Mějme problém *P*, který rozložíme na podproblémy. Nechť *P'* je podproblém *P*. Pokud pro každý *P'* je cena minimálního řešení *P* vetší nebo rovna řešení *P'*, velikost minimálního řešení *P'* nám dává spodní hranici velikosti minimálního řešení *P*.

Mějme tedy jako vstupní problém nalezení minimálního řešení *UCP(A)*. Označme $\min(UCP(A))$ velikost minimálního řešení *UCP(A)*. Dále je třeba zvolit si vhodný podproblém, tedy matici vzniklou z původní matice. Nechť *A'* je tedy sub-matice původní matice *A*, obsahující některé řádky *A* a všechny sloupce matice *A*. Minimální řešení *A'* použijeme pro výpočet spodní hranice řešení *A*. Jako matici *A'* zvolíme matici obsahující řádky *MSIR* a sloupce původní matice *A*. *A' = MSIR(A)*. Pak velikost minimálního řešení *A'* je rovna počtu řádků *A'*, pokud jsou ceny sloupců jednotkové, jelikož každý řádek musí být pokryt jiným sloupcem.

Myšlenka postupné řešení navržená v [2] spočívá v postupném přibližování podproblému původnímu problému a postupném přepočítávání řešení. Je třeba matici *A'* postupně rozšiřovat o řádky matice *A*, které netvoří *MSIR*. Nechť *X(A)* je množina řádků matice *A*. Rozšíříme matici *A'* o jeden řádek z množiny *X(A) \ X(A')*. Označme *S* jako řešení

UCP(A). Může se nám stát, že sloupec y_j, obsažen v *S*, je redundantní. Je to právě v případě, že $S - \{y_j\}$ je stále řešení *UCP(A)*.

Definice 3.6.1: Řešení neobsahující redundantní sloupce je nazýváno *neredundantní*.

Označme *S(A',m)* množinu řešení *UCP(A')* která obsahuje všechny neredundantní řešení obsahující *m* nebo méně sloupců. Pokud *m = min (UCP(A'))*, pak *S(A',m)* dává přesně množinu všech minimálních řešení *UCP(A)*. Pokud známe množinu *S(A',m)* pro matici *A'*, spodní hranice daná maticí *A'* je rovna *m = min (UCP(A'))*.

Nyní rozšíříme matici o námi zvolený řádek A_r z původní matice *A*, který není obsažen v *A'*. Je patrné, že ne všechna řešení matice *A'* budou i řešení $A'+A_r$, neboť A_r může být pokryt i jinými sloupci, které nepokrývají řádky *A'*. Tedy množina řešení $S(A'+A_r, m) \subseteq S(A', m)$. Pokud budeme postupně přidávat řádky z A_{r1} až A_{rk} z *A* do *A'*, je tedy šance, že se množina řešení $S(A'+A_{r1}+..+A_{rk}, m)$ se bude zmenšovat, až do stavu kdy $S(A'+A_{r1}+..+A_{rk}, m) = \emptyset$, což znamená, že jsme zvýšili spodní hranici o 1. Pokud $S(A'+A_{r1}+..+A_{rk}, p) = \emptyset, p \geq m$, zvýšili jsme spodní hranici o *r – m + 1*. Tento postup je v algoritmu odpovídá tučnému textu, a je prováděn funkcí *raiser*. *Raiser* je volán pro matici A_N, pokud $|MSIR| + |Cesta\_k\_uzlu\_A_N| + n \geq |Nejlepší\_řešení|$. V opačném případě pokračuje větvení algoritmu podle výběru sloupců. Raiser nám buď zvýší spodní hranici o *n*, nebo pokud hranice zvýšena být nemůže, vrátí minimální řešení $S(A_N)$ daného $UCP(A_N)$, kde pak $S(A_N) \cup cesta(A_N)$ tvoří nové nejlepší řešení $UCP(A_N)$.

Nicméně, tento postup hledání řešení má nevýhodu, že počet nalezených řešení roste exponenciálně, tudíž bude třeba velké množství paměti k jejich uchování. V další kapitole si popíšeme řešení tohoto problému, pomocí uchování řešení v tzv. krychlích a větvení pomocí těchto balíků řešení.

4 Reprezentace a přepočítávání řešení

V minule kapitole jsme zmínili problém nárůstu počtu uchovávaných řešení sub-matice *A'* postupně rozšiřovanou o řádky původní matice *A*. Nejprve uvedeme techniku přepočítávání řešení, uvedenou v [2], s ohledem na rozšiřující se matici.

4.1 Přepočítávání řešení – operátor *Rec*

Mějme tedy *S* řešení *UCP(A')*. Nyní vezmeme řádek A_r z matice *A*, matice A neobsahuje řádky matice *A'*, a vložíme ho do *A'*. Zavedeme operátor *Rec(A'+A$_r$,S)*, reprezentující množinu řešení *UCP(A'+A$_r$)*. Je zřejmé, že pokud *S* je řešením *UCP(A'+A$_r$)*, pak *Rec(A'+A$_r$,S) = S*. Pokud A_r je pokryt jinými sloupci než jsou pokryty řádky matice *A'*, pak do množiny řešení *Rec(A'+A$_r$,S)* musíme ještě přidat sloupce pokrývající A_r. V takovém případě $Rec(A'+A_r, S) = \{S \cup \{y\}\}$, kde *y* jsou všechny sloupce pokrývající A_r. Operátor *Rec(A'+A$_r$, S)* nám tedy dává řešení *UCP(A'+A$_r$)*. Je zřejmé, že pokud *S* není řešení *UCP(A'+A$_r$)*, pak dostaneme počet řešení odpovídající počtu sloupců pokrývajících A_r. Uveďme si příklad.

Příklad 4.1.1: Nechť *(A'+A_r)* je matice na obrázku 4.1.1.

	$\mathscr{A}_1$	$\mathscr{A}_2$	$\mathscr{A}_3$	$\mathscr{A}_4$	$\mathscr{A}_5$
Ar		1		1	
A'	1	1			
			1		
					1

Obrázek 4.1.1 - matice(A'+A_r)

Matice *A'* má dvě neredundantní řešení, a sice $S_1 = y_1,y_3,y_5$ a $S_2 = y_2,y_3,y_5$. S_2 je řešením *(A'+A_r)*, do množiny řešení *UCP(A'+A_r)* tedy přidáme S_2. S_1 není řešením *UCP (A'+A_r)*, musíme tedy do tohoto řešení přidat sloupce pokrývající *Ar*. nově vzniklá řešení jsou $S'_{11} = S_1 \cup y_2$ a $S'_{12} = S_1 \cup y_4$. S'_{11} je redundantní řešení, jelikož $S'_{11} - \{y_1\}$ je také řešení *UCP (A'+A_r)*. *Rec(A'+A_r ,S)* tedy může obsahovat i redundantní řešení. Této problematice se budeme věnovat v následující kapitole.

4.2 Reprezentace řešení

Velikost množiny řešení získávaná postupem uvedeným v předchozí kapitole poroste exponenciálně s počtem přidaných řádků. Navíc pokud chceme zvýšit spodní hranici *MSIR* např. o *n = 3*, bude patrně třeba přidat jen málo řádek, abychom zvýšili řešení *S* jakožto řešení *UCP (MSIR)*. Problém však vznikne, pokud vezmeme jiné řešení *UCP (MSIR)*, řekněme *S'*, bude třeba také přidat jen velmi málo řádků, avšak pravděpodobně jiných, než v prvním případě. Hledáme tedy takové řádky, které zvýší všechna řešení *UCP(MSIR)* o *n*. Avšak jelikož jsou tyto množiny řádků odlišné pro různá řešení *UCP(MSIR)*, bude třeba přidat všechny řádky do matice *A'*.

Musíme tedy jednotlivá řešení třídit podle toho, zda mohou být zvyšována stejnými řádky z *X(A) \ X(MSIR)*. K tomu využijeme tzv. krychle řešení.

Definice 4.2.1: Nechť krychle je množina $C = D_1 x ... x D_d$, kde $D_i \cap D_j = \emptyset, i \neq j$ a $D_i \subset Y(A), 1 \leq i, j \leq d$. Podmnožiny D_i jsou domény krychle *C*.

Krychle *C* označuje množinu množin *d* sloupců. Potom množina neredundantních řešení matice *A' = MSIR (A)* je reprezentována jako $O(A_{i1}) x ... x O(A_{id})$, kde $A_{i1},...,A_{id}$ jsou řádky matice *A'* a *O(A)* je množina sloupců pokrývající řádek *A*. Nyní přidáme do *A'* řádek A_r z matice *A*, který netvoří *MSIR*. Nechť $C = D_1 x ... x D_d$ je krychle řešení *UCP (A')*. Pak podle definice operátoru *Rec* dostaneme

$$Rec(A'+A_r, C) = část1(C) \cup část2(C) x O(A_r)$$

kde *part1(C)* je množina řešení obsažených v *C*, která pokrývá A_r a *part2(C)* je množina řešení obsažených v *C*, která nepokrývá A_r. V příkladě uvedeném v předchozí kapitole *části1(C)* odpovídá řešení *S2* a *části2(C)* odpovídá řešení *S1*.

Při výpočtu množiny $Rec(A'+A_r, C)$ může nastat několik případů. V prvém případě se může stát, že všechna řešení z C pokrývají přidaný řádek A_r. Pak $Rec(A'+A_r, C) = C$. V druhém případě se může stát, že žádné z řešení z C nepokrývá přidávaný řádek A_r, potom $Rec(A'+A_r, C) = CxO(A_p) = D_1 x...xD_d\, xO(A_p)$. Pokud nám řádek A_r pokrývají jen některá řešení z C, předpokládejme bez újmy na obecnosti prvních r řešení, pak krychle C bude rozdělena na následujících $r + 1$ neprotínajících se krychlí.

$$C_1 = D_1 \cap O(A_p) x D_2 x...xD_d$$
$$C_2 = D_1 \setminus O(A_p) x D_2 \cap O(A_p) x D_3 x...xD_d$$
$$C_3 = D_1 \setminus O(A_p) x D_2 \setminus O(A_p) x D_3 \cap O(A_p) x...xD_d$$
$$...$$
$$C_r = D_1 \setminus O(A_p) x...xD_{r-1} \setminus O(A_p) x D_r \cap O(A_p) x D_{r+1} x...xD_d$$
$$C_{r+1} = D_1 \setminus O(A_p) x...xD_{r-1} \setminus O(A_p) x D_r \setminus O(A_p) x D_{r+1} x...xD_d$$

Vidíme, že pro libovolný pár řešení $C_i, C_j, i \neq j, C_i \cap C_j = \emptyset$. Navíc vidíme, že prvních r krychlí nám dává řešení *UCP(A')*, které pokrývají A_r a krychle C_{r+1} dává řešení, které nepokrývá A_r.

4.3 Přepočítávání řešení – úprava operátoru *Rec*

V kapitole 4.1 jsme se setkali s problémem generování redundantních řešení operátorem *Rec*. Vraťme se k příkladu 4.1.1. Krychli C řešení matice A' tvoří $C = (y_1,y_2)x(y_3)x(y_5)$. Řešení pokrývající A_r tvoří sloupce y_2 a y_4. Z původní C nyní vygenerujeme krychle matice $A'+A_r$.

part1(C): $C_1 = (y_2) x (y_3) x (y_5)$
part2(C): $C_2 = (y_1) x (y_3) x (y_5) \cap (y_2, y_4)$

Vidíme, že pokud vytvoříme řešení S z C_2 takové, že $S = y_1,y_2,y_3,y_5$, je redundantní. Je to proto, že řešení S' získané z C_1 je obsaženo v S, konkrétně $S' = y_2,y_3,y_5$. Měli bychom tedy upravit operátor *Rec* tak, aby tato řešení nebyla generována, konkrétně aby v $část2(C) \cap O(A_r)$ nebyly obsaženy sloupce pokrývající řádek A_r obsažené v $část1(C)$.

Teorém 4.3.1: *Pokud výpočet operátoru Rec bude pozměněn následujícím způsobem*

$$\mathrm{Re}\, c(A'+A_r, C) = část1(C) \cup část2(C) x [O(A_r) \setminus (D_1 \cup ... \cup D_d)]$$

dostaneme všechna neredundantní řešení $A'+A_r$.

Důkaz je uveden v [2].

4.4 Zamezení opakovanému generování stejných řešení

Uvažujme nyní dále generovaná řešení operátorem *Rec*. Domény řádků *MSIR* jsou

$$D_1 = \{1,2\}$$
$$D_2 = \{3\}$$
$$D_3 = \{5\}$$

dále doména přidávaného řádku je

$$D_{Ar} = \{2,4\}$$

Vidíme, že přidávaný řádek protíná pouze první doménu. Generované krychle řešení jsou

$$C_1 = D_1' x D_2 x D_3 = \{2\} x \{3\} x \{5\}$$
$$C_2 = D_1'' x D_2 x D_3 = \{1\} x \{3\} x \{5\}$$
$$D_1' = D_1 \cap O(A_r)$$
$$D_1'' = D_1 \setminus O(A_r)$$
$$O*(A_r) = O(A_r) \setminus D_1$$

Generovaná řešení jsou $S_1 = (y_2, y_3, y_5)$ a $S_2 = (y_1, y_3, y_5, y_4)$, S_2 se liší od S_1 pouze záměnou sloupce y_2 za y_1 a přidáním y_4 z $O*(A_r)$. Nyní předpokládejme, že máme řešení *S'' UCP(A)* obsahující částečné řešení $S \cup S'$. Pak stejné řešení *S''* bude získáno jak z větve krychle C1, tak z větve krychle C2. Tedy jedno řešení získáme vícekrát, bude nutná další úprava operátoru Rec.

Ačkoli při získávání D_1'' odstraníme z D_1 sloupce pokrývající A_r, je stále možné rozšířit řešení z krychle C_1 přidáním sloupců z $D_1 \setminus O(A_r)$ a $O*(A_r)$, a rozšířit řešení z $C_2 x O*(A_r)$ přidáním sloupců z $D_1 \cap O(A_r)$, tudíž z obou větví můžeme získat stejná částečná řešení z $D_1 \cap O(A_r) x D_1 \setminus O(A_r) x D_2 x ... x D_d x O*(A_r)$.

Měli bychom tedy zabránit, abychom v případě větve C_2 rozšiřovali $C_2 x O*(A_r)$ o sloupce $D_1 \cap O(A_r)$. Řešení *S''* pak bude získáno pouze z větve C_1.

Lemma 4.4.1: Nechť *C* je krychle řešení *UCP(A')* a A_r je řádek z $X(A) \setminus X(A')$. Nechť *S* je řešení *UCP(A)* z *Gen(C)*, kde *Gen(C)* označuje všechna řešení *UCP(A)* která obsahuje částečné řešení z *C*. Předpokládejme bez újmy na obecnosti, že A_r protíná prvních *r* domén krychle *C*. Pak *S* může být získáno v jedné z *r + 1* větví odpovídající krychlím $C_k, 1 \le k \le r+1$, i když ve větvi každé krychle $C_k, 1 \le k \le r+1$ negenerujeme žádné řešení obsahující sloupce z $(D_1 \cup ... \cup D_{k-1}) \cap O(A_r)$

Důkaz je uveden v [2].

5 Algoritmus Raiser

Jak již bylo řečeno, metodu větví a hranic použitou pro řešení problému unátního pokrytí jsme upravili o režim „negativního“ přístupu, do kterého se metoda přepne v případě, že rozdíl mezi spodní a horní hranicí je roven hodnotě *maxRaiser*. V tomto režimu metoda volá funkci *raiser*, která se snaží dokázat neexistenci lepšího řešení, než je současné nejlepší nalezené. Funkce *raiser* je volána s parametrem *n*, který odpovídá rozdílu mezi horní a spodní hranicí.

5.1 Stručný popis funkce raiser

Pomocí raiseru provedeme postupné řešení problému. Začneme tedy s maticí *A'* jakožto podproblémem původní matice *A*, *A'* obsahuje řádky *MSIR* a všechny sloupce matice *A*. Z matice *A* tyto řádky odebereme. Nyní najdeme krychli matice *A'* skládající se z domén řádků *A'*, a začneme s větvením výběrem řádků. Matici *A'* postupně rozšiřujeme o vhodně vybrané řádky z $X(A) \setminus X(A')$ a provádíme třídění řešení do krychlí popsaným v kapitolách 4.1 – 4.4. Přidáním každého řádku A_r nám vznikne *p+1* krychlí, kde *p* je počet řádků *A'* protnutých přidávaným řádkem A_r. Toto větvení opakujeme pro každou z *p+1* krychlí, pro každou vybereme další řádek z $X(A) \setminus X(A')$ a větvíme dále.

Tento proces může být popsán výpočetním stromem, zvaným strom větvení krychlí, kořen stromu je vstupní problém odpovídající krychli *C* a maticím *A'* obsahující řádky *MSIR* a všechny sloupce původní matice, a maticí *A* z které jsme tyto řádky odebrali. Dále provedeme výběr řádku z druhé matice, řádek přidáme do *A'* a spočítáme všech *r+1* krychlí. Hrana grafu odpovídá rekurzivnímu volání raiseru, který je volán s odpovídající krychlí, a novým párem matic *A'* a *A*, v nichž byly provedeny úpravy množin řádků. Počet potomků každého uzlu je tedy roven počtu krychlí získaných v tomto uzlu. Rekurzi zastavíme, pokud

1. V matici *A* již nezbyly žádné řádky, které bychom mohli přidat do *A'* a odpovídající matice obsahuje k domén, kde $k < |MSIR| + n$, což znamená, že spodní hranice nemůže být zvýšena o *n*. V takovém případě vezmeme libovolné řešení z krychle jakožto nejlepší řešení *UCP(A)*.

2. Pokud je počet domén v uzlu větší než $|MSIR| + n$, v takovém případě, pokud je v každé větvi z uzlu dosaženo této hranice, tak spodní hranice může být navýšena na $|MSIR| + n$. Nemůžeme totiž získat levnější řešení, než je počet domén krychle.

Parametry funkce raiser jsou:

- *Matice A*, která je zpočátku rovna původní matici, z které jsme odebrali řádky MSIR.
- *Matice A'*, obsahující řádky *MSIR* a sloupce původní matice.
- *Krychle C*, odpovídající množině částečných řešení *A*, zpočátku odpovídá krychli *MSIR*
- parametr *n*, o který se snažíme zvýšit spodní hranici. Jedná se o vstupně výstupní parametr, zpočátku nastaven na velikost rozdílu mezi spodní a horní hranicí. Parametr *n* je zvolen jako globální proměnná.

- *uBound* je velikost nejlepšího nalezeného řešení. *uBound* je rovněž globální proměnná
- *lBound* je vstupní parametr rovný velikosti *MSIR*.
- *SolUCP* je výstupní parametr obsahující nové nejlepší řešení nalezené funkcí raiser, pokud spodní hranice nemůže být navýšena o *n*.

5.2 Příklad 1-raiseru

Uvažujme následující matici.

	x_1	x_2	x_3	x_4	x_5	x_6	x_7
				1	1		1
	1		1				
A		1	1	1		1	
						1	1
A'	1	1					
			1		1		

Obrázek 5.2.1 – Matice k příkladu 1-raiseru

Řádky MSIR tvoří matici *A'*, matici *A* tvoří řádky zbývající. Domény jednotlivých řádků jsou

$$D_4 = \{6,7\}$$
$$D_5 = \{1,2\}$$
$$D_6 = \{3,5\}$$

Krychle matice *A'* je rovna

$$C = \{1,2\}x\{3,5\}x\{6,7\}$$

Spodní hranice je tedy rovna 3, neboť *C* se skládá ze 3 domén. Raiser tedy voláme s těmito parametry.

Nyní funkce raiser vybere vhodný řádek z matice *A*, přidá ho do matice *A'* a provede výpočet *r+1* krychlí, kde *r* je počet protnutých řádků matice *A'*. Vybereme řádek A_3, pro který *r = 3*, a spočítáme 4 krychle řešení. Doména tohoto řádku je

$$D_{A3} = \{2,3,4,6\}$$

Odpovídající krychle jsou

$$C_1 = \{2\}x\{3,5\}x\{6,7\}$$
$$C_2 = \{1\}x\{3\}x\{6,7\}$$
$$C_3 = \{1\}x\{5\}x\{6\}$$
$$C_4 = \{1\}x\{5\}x\{7\}$$
$$část1(C) = C_1 \cup C_2 \cup C_3$$
$$část2(C) = C_4$$
$$\mathrm{Re}\,c(A'+A_3, C) = část1(C) \cup C_4 x(O(A_3) \setminus D) =$$
$$= (\{2\}x\{3,5\}x(6,7) \cup \{1\}x\{3\}\{6,7\} \cup \{1\}x\{5\}x\{6\} \cup \{1\}x\{5\}x\{7\}x\{4\})$$

Krychle C_1 představuje množinu řešení, v které je přidávaný řádek A_3 nezbytně pokryt sloupcem první domény C (možná i sloupci ostatních domén). Krychle C_2 představuje množinu řešení, kde A_3 je nezbytně pokryt sloupcem druhé domény (a možná sloupci poslední domény), krychle C_3 představuje množinu řešení neobsažených v C_1 a C_2, A_3 je nezbytně pokryt sloupcem třetí domény. Krychle C_4 představuje množinu řešení, která nepokrývá A_3 a není tudíž řešením *UCP* $(A'+A_3)$.

Nyní započne rekurzivní volání raiseru, kořen stromu bude mít 4 potomky, každý bude volán s odpovídající krychlí řešení $C_1, C_2, C_3, C_4 x(O(A_3) \setminus D)$ a párem matic $A'+A_3$ a A-A_3. V každém uzlu se opět vybere řádek z matice $A'+A_3$ a provede se výpočet odpovídajících krychlí. Následujme tedy první uzel, odpovídající krychli C_1. Algoritmus vybere např. řádek A_2, přidá ho do $A'+A_3$ a odebere z A-A_3. Doména řádku A_2 je

$$D_{A2} = \{1,3\}$$

Odpovídající krychle jsou

$$C_{11} = \{2\}x\{3\}x\{6,7\}$$
$$C_{12} = \{2\}x\{5\}x\{6,7\}$$

Nyní se opět provede větvení algoritmu, uzel bude mít 2 potomky, každý bude následován odpovídající krychlí $C_{11}, C_{12} x(O(A_2) \setminus D_{C1}), D_{C1} = D_{A2} \setminus C_1$ a párem matic $A' + A_3 + A_2$ a $A – A_3 – A_2$. Následujme tedy větev odpovídající krychli C_{11}. Provedeme výběr řádku A_1, a provedeme výpočet krychlí. Doména řádku A_1 je

$$D_{A1} = \{4,5,7\}$$

řádek nám tedy protíná pouze třetí doménu. Odpovídající krychle jsou

$$C_{111} = \{2\}x\{3\}x\{7\}$$
$$C_{112} = \{2\}x\{3\}x\{6\}$$

Dále nám již nezbývá žádný řádek v matici A, tudíž jsme dosáhli listu. Víme, že krychle kořene stromu měla 3 domény, a raiser byl volán s hodnotou 1, cílem tudíž bylo zvýšit spodní hranici o 1, neboli dokázat, že jakékoliv řešení, pokrývající vstupní matici A, má minimálně 4 domény. V našem případě C_{111} má pouze 3 domény, tudíž spodní hranice nemůže být zvýšena

na 4, ale raiser vrátí minimální řešení *A*, a to $S = \{2,3,7\}$, které zvolíme jako nejlepší nalezené řešení.

5.3 Raiser - Implementace

V této kapitole se blíže podíváme na úpravu metody větví a hranic, kterou rozšíříme o volání funkce raiser, a na implementaci funkce raiser samotné. Algoritmus 5.3.1 zobrazuje metodu větví a hranic, do které je začleněn negativní přístup zprostředkovaný funkcí raiser.

```
Aura (A, Cesta, lBound , uBound )
{
        solve_dominance();
        /* odstraní dominantní řádky a sloupce, vybere do řešení nezbytné sloupce */
        find_MSIR(A);
        /*Nalezne MSIR pro výpočet spodní hranice*/
        lBoundNew = cost(Path) + cost(MSIR)

        if (lBoundNew ≥ uBound ) bestSolution = ∅
        else if (difference ≤ maxRaiser ) {
                /*Pokud je difference malá, přepneme do režimu negativního přístupu*/
                C = find_cube_MSIR(MSIR)
                lowerBound=|MSIR|
                answer=raiser(C,difference,A,lowerBound,solRaiser,uBound)
                if (answer=1) bestSolution = ∅
                else bestSolution =  Path ∪ solRaiser
                }
        else if ( A= ∅ ) bestSolution = Path
        else {
                j=select_column(A);
                Path 1 = Path + j;
                A' = A – {j + row covered by j}
                Solution1 = Branch_And_Bound (A',Path1, lBoundNew, uBound)
                if (Solution 1 ≠ ∅ ) uBound = Solution 1
                if (Solution 1 ≠ ∅ & cost(Solution 1) = lBoundNew ) return Solution 1
                /*pokud jsme již dosáhli lBound, nebudeme větvit dále*/
                A'' = A – {j}
                Solution 2 = Branch_And_Bound (A'', Path, lBoundNew, uBound)
                bestSolution = min(Solution 1, Solution 2)
        }
        Return bestSolution
}
```

Algoritmus 5.3.1 – AURA

Algoritmus, pokud je rozdíl mezi spodní a horní hranicí menší nebo roven námi definované malé konstantě maxRaiser, se přepne do režimu negativního přístupu, nepokračuje větvením výběrem sloupců. Jak již bylo řečeno, negativní přístup je zprostředkován metodou raiser. Raiser vrací hodnotu 1, pokud spodní hranice může být zvýšena o *n*, jinak vrací

hodnotu 0 a nalezne lepší řešení UCP(A), než je doposud nejlepší nalezené. Algoritmus 5.3.2 zobrazuje implementaci funkce raiser.

```
raiser (CubeC, n, A, lbound, solRaiser, ubound) {
        stillToRaise = lbound + n – number_domains (CubeC)
        /*pokud jsme dostatečně navýšili spodní hranici, vrátíme 1*/
        if (stillToRaise<=0) return 1

        /*pokud je matice A prázdná, vrátíme řešení*/
        if ( A = 0 ) return found_solution ( CubeC, n, solRaiser, ubound )
        BSONIR = find_best_set_of_non_intersecting_rows (A, CubeC)
        for each row r ∈ BSONIR {
                CubeC = add_domain( CubeC, A, r )
                stillToRaise = stillToRaise – 1
                if ( stillToRaise <= 0 ) return 1
        }
        A = A \ BSONIR
        if ( A = 0 ) return found_solution ( CubeC, n, solRaiser, ubound )
        if ( stillToRaise = 1 ) {

                if (add_set_of_1intersecting_rows (A, CubeC ) = 1 ) return 1
                if (A = 0 ) return found_solution ( CubeC, n, solRaiser, ubound )
        }
        /*Vybereme další nejlepší řádek z matice A*/
        r = select_best_uncovered_row ( A, CubeC )
        A = A \ { r }
        /*rozdělíme CubeC na part1 a part2, podle řádku r */
        split_cube { CubeC, A, r, part1, part2 }
        /*přidáme k Cube_{k+1} část pokrývající řádek r*/
        Cube_{k+1}=add_domain (Cube_{k+1}, A, r )
        /*nyní větvíme podle vytvořených krychlí*/
        returnValue=1
        while ( part1 ∪ part2 ≠ ∅ ){
        Cube_j = select_next_cube( part1 ∪ part2 )
        /*Pokud bylo nalezeno lepší řešení, returnValue nastavíme na 0*/
        if ( raiser ( Cube_j, n, A, lbound, solRaiser ) = 0 )
                retunrValue=0
        }
        return returnValue
}
```

Algoritmus 5.3.2 – funkce raiser

Funkce raiser nejprve prověří, zda již počtem domén krychle nebyla překročena hodnota, o kterou jsme chtěli navýšit spodní hranici. Pokud ano, ukončíme raiser s hodnotou 1. Dále zkontrolujeme, zda máme ještě v matici *A* nějaké řádky pro výběr. Pokud ne, znamená to, že nemůžeme spodní hranici zvýšit o požadovanou hodnotu, ale můžeme z dané krychle získat řešení *UCP(A)* lepší, než současné nalezené. Řešení z dané krychle získáme funkcí *found_solution (Cube, n, solRaiser, ubound),* kterou zobrazuje algoritmus 5.3.3. Funkce zároveň upraví parametry *n* a *ubound* pro další větvení funkce raiser.

```
found_solution (Cube, n, solRaiser, ubound ) {
        /*funkce vybere libovolné řešení z dané krychle*/
        solRaiser = get_solution (Cube)
        newUbound = cost ( solRaiser )
        newN = n – ( ubound – newUbound )
        n = newN
        ubound = newUbound
        return 0
}
```

Algoritmus 5.3.3 – funkce found_solution

Dále bychom mohli pokračovat výběrem dalšího řádku A_r z matice *A*. Pomocí funkce *split_cube { CubeC, A, A_r, part1, part2 }* provedeme vypočet *r + 1* krychlí, kde *r* je počet domén krychle *CubeC* protnutých doménou řádku A_r. Funkce pak pokračuje větvením, každé rekurzivní volání raiseru odpovídá dané krychli $CubeC_i, 1 \le i \le r+1$ a páru matic *A, A'*. nutno podotknout, že pro každé volaní musíme mít stejnou matici *A'*, je tedy nutné ji při každém volání raiseru předávat jako kopii původní *A'*. V algoritmu však můžeme provést několik optimalizačních úprav.

5.4 Optimalizace funkce raiser

V této kapitole si popíšeme možné úpravy Aury popsané v [2]. Kořen stromu větvení krychlí má jako vstup krychli řešení pokrývající *MSIR*. Zřejmě čím přesněji *MSIR* nalezneme, tím méně řádek nám zbude v matici *A* z které *MSIR* odebereme a tím méně větvení algoritmu lze očekávat. V průběhu větvení se však může stát, že vybraný řádek A_r z matice *A* neprotíná žádnou doménu krychle řešení. V kořeni stromu je vstupní krychle tvořena doménami *MSIR*, tudíž všechny řádky matice $A \setminus MSIR$ musí protínat některou z domén vstupní krychle, vyplývá to z definice *MSIR*. Ale během větvení, kdy z domén krychlí odebíráme sloupce, může tato situace nastat. Nazvěme tuto množinu řádku *BSONIR (best set of non-intersecting rows).* Můžeme tedy tyto řádky nalézt, jejich doménu přidat do krychle *MSIR*, vyjmout z matice *A* a snížit hodnotu *stillToRaise*. Nalezení *BSONIR* však vyžaduje řešení dalšího NP-úplného problému, proto zvolíme rychlou heuristiku pro nalezení *BSONIR* prováděnou funkcí *find_best_set_of_non_intersecting_rows*, znázorněnou algoritmem 5.4.1, která nalezne řádek s nejvíce jedničkami, který neprotíná vstupní krychli a žádný z již nalezených řádků *BSONIR*.

```
find_best_set_of_non_intersecting_rows (A, Cube ) {
        emptyInterRows= ∅
        bestRow= ∅
        for each row r ∈ A {
                D = find all intersected domains
                if (D= ∅ ) {
                        emptyInterRows = emptyInterRows ∪ r
                if ( length ( bestRow ) < length (r))
                        bestRow=r
        }
        }
```

```
        if ( emptyInterRows = ∅ )
        return ∅
        else {
        BSONIR = ∅
                do {
        BSONIR = BSONIR ∪ bestRow
        emptyInterRows = emptyInterRows \ bestRow
        for each row r ∈ emptyInterRows {
                if ((r ∩ BSONIR) ≠ ∅)
                        emptyInterRows= emptyInterRows \ r
                else if ( length ( bestRow ) < length ( r ) )
                        bestRow = r
                }
                }while ( emptyInterRows ≠ ∅ )
        }
        return BSONIR
}
```

Algoritmus 5.4.1 – funkce find_best_set_of_non_intersecting_rows

Další možné vylepšení můžeme provést, pokud hodnota *stillToRaise* je rovna 1. Jelikož *stillToRaise = lbound + n – number_domains (Cube),* dalším navýšením počtu domén krychle dosáhneme oříznutí větve. Pokud vybraný řádek protíná pouze jednu doménu *D krychle řešení*, pak sloupce $D \setminus D_{Ar}$ vytvoří množinu řešení *part2(Cube),* která nepokrývá A_r, tudíž není řešením *UCP (A' + Ar)*. Přidáme tedy do *part2(Cube)* sloupce pokrývající A_r, tudíž přidáme do krychle další doménu, což povede k oříznutí větve. Můžeme tedy sloupce z *D* nepokrývající A_r vyjmout a řádek vyjmout z matice, neboť bude pokryt.

Obdobná situace nastane, pokud dva různé řádky protínají pouze jednu doménu *D*, ale každý je pokryt jinými sloupci. Předpokládejme, že $D = \{c_1,c_2,...,c_j\}$, řádek r_1 je pokryt sloupcem c_i a řádek r_2 je pokryt sloupce c_j. Pak pro pokrytí r_1 musíme vybrat c_i, a v krychli řešení pak nebude žádný sloupec pokrývající r_2, budeme muset přidat doménu do krychle, která r_2 pokrývat bude, což povede k oříznutí větve. Pokud tedy zjistíme, že dva řádky jsou pokryty pouze jednou doménou, a každý z nich je pokryt jinými sloupci, můžeme tuto větev oříznout.

Algoritmus 5.4.2 zobrazuje funkci funkci *add_set_of_1intersecting_rows (A, Cube),* která postupně vybírá řádky z matice *A* a zjišťuje počet protnutých domén s krychlí Cube. Pokud je počet protnutých domén roven jedné, odebere z domény krychle sloupce, které nepokrývají vybraný řádek. Pokud narazí na řádek, který neprotíná žádnou doménu krychle, může to být ze dvou příčin:

1. Řádek není pokryt z důvodu odebírání sloupců z domén krychle během větvení stromu, k jeho pokrytí bychom museli přidat další doménu do krychle
2. Jde právě o případ, kde dva různé řádky r_1 a r_2 protínají jednu doménu krychle, avšak jsou pokryty různými sloupci. Sloupce nepokrývající řádek r_1 jsme z této domény již odebrali, ale některé z těchto sloupců pokrývali r_2.

V obou případech jsme nuceni přidat další doménu do krychle, tudíž větev může být oříznuta.

```
add_set_of_1intersecting_rows (A, Cube){
do {
        reducingDomain=FALSE
        for each row r ∈ A {
                D = find number of intersected domains( Cube, r )
                if (|D|=1) {
                        reducingDomain = TRUE
                        d = find intersected domain ( Cube, r )
                        remove comunms not covering  ( d, r )
                        A = A \ r
                }
                else if ( |D|=0 ) {
                        return 1
                }
        }
  } while ( reducingDomains )
return 0
}
```

Algoritmus 5.4.2 – funkce add_set_of_1intersecting_rows

Jiné zlepšení spočívá ve výběru řádku pro větvení algoritmu. Cílem je snížit na minimum počet větvení algoritmu, musíme tudíž vybrat řádek protínající minimum domén krychle řešení. Funkce *select_best_uncovered_row(A,Cube),* zobrazená algoritmem 5.4.3 vybere z matice *A* řádek, který protíná minimum domén krychle Cube. Pokud narazíme na více řádků pokrytých stejným minimálním počten domén, vybereme řádek s větší váhou *w* definovanou

$$w = \prod_{k=1}^{m} \frac{| D_{ik}^{'} |}{| D_{ik} |}$$

kde *m* je počet domén krychle Cube protnutých vybraným řádkem A_r, D_{ik} je doména protnutá řádkem A_r a $D_{ik}^{'} = D_{ik} \setminus O(A_p)$. Váha řádku tedy odráží řešení z krychle Cube které nepokrývají A_r.

```
select_best_uncovered_row(A,Cube) {
        bestIntersectedRowNum = maxInt
        bestWeight = 0
        for each row r ∈ A {
          intersectedRowNum = 0
          weight = 1
          for each domain D ∈ Cube{
                if (r ∩ D){
                        intersectedRowNum = intersectedRowNum + 1
                D2 = rowMinus ( D,r )
                w = | D2 | / | D |
                weight = weight * w
        }
          }
          if (intersectedRowNum<bestIntersectedRowNum ) {
```

```
                bestIntersectedRowNum = intersectedRowNum
                bestWeight = weight
                bestRow = r
            } else if (intersectedRowNum = bestIntersectedRowNum) {
                if (weight > bestWeight ) {
                bestIntersectedRowNum = intersectedRowNum
                bestWeight = weight
                bestRow = r
            }
        }
        return bestRow
}
```

Algoritmus 5.4.3 – funkce select_best_uncovered_row

Vliv jednotlivých úprav na velikost výpočetního času je zaznamenán v tabulce 5.4.1. Měření bylo provedeno na několika reálných maticích generovaných programem BOOM. Algoritmus Aura bez vylepšení výběrem řádku protínajícího minimum domén krychle MSIR zde není uveden, neboť Aura bez tohoto vylepšení požaduje neúnosně velké množství pamětí, jelikož musí být vytvořeno velké množství krychlí. Je zde uvedena Aura s optimalizací výběrem řádku protínajícího minimum domén krychle MSIR, dále hledáním množiny BSONIR, a nakonec Aura se všemi vylepšeními.

Počet řádků	Počet Sloupců	Hustota	Best Row čas [s]	BSONIR čas [s]	Všechna vylepšení čas [s]
77	140	4,9	0,10	0,10	0,08
120	209	3	15,55	18,84	2,33
174	263	2,1	47,78	58,93	4,58
73	119	3	1,12	1,17	0,28
190	277	4,9	149,33	150,71	31,15

Tabulka 5.4.1 – Porovnání výpočetního času Aury s optimalizacemi

Vidíme však, že hledáním množiny BSONIR vzroste výpočetní režie zřejmě natolik, že výsledný čas je naopak vždy malinko zvýšen.

Kvalita řešení získaná Aurou byla pro matice s nižšími časovými nároky porovnána s řešeními získanými funkcí *Branch_and_Bound* s výpočtem spodní hranice pomocí *MSIR*, z důvodu ověření funkčnosti Aury. Ceny nalezených řešení byly vždy ekvivalentní.

5.5 Úprava pro nejednotkové ceny sloupců

Výše uvedené funkce však počítají s tím, že v matici *A* máme všechny sloupce ohodnocené cenou rovnou jedné, je tedy třeba příslušným způsobem funkce upravit, pokud jsou v matici *A* ceny sloupců různé od jedné. Pro takový případ jsem navrhl následující úpravy.

Nejprve provedeme úpravy ve funkci raiser. Hodnotu *stillToRaise* nemůžeme v každé větvi snižovat o počet domén ve vstupní krychli, neboť tento počet reprezentuje cenu řešení

možného získat z této krychle. Hodnota *stillToRaise* slouží pro signalizaci, že počtem domén, resp. cenou libovolného řešení již dosáhneme navýšení spodní hranice o požadovanou hodnotu. Pokud by měla nastat situace, kdy $stillToRaise \leq 0$, měli bychom při výběru libovolného řešení *S* ze vstupní krychle cenou řešení *S* přesáhnout *lbound + n*. Ceny řešení získaných z jedné krychle však budou různé, vybereme tedy řešení s nejnižší cenou, abychom měli zaručené, že všechna řešení tuto hodnotu přesáhnou. Struktura představující krychli řešení proto bude mít pomocnou proměnnou, do které při přidávání domény do krychle přičteme cenu nejlevnějšího sloupce, tato proměnná bude představovat nejlevnější řešení.

Dále je v raiseru nutné zjistit, kdy nastane vhodný okamžik pro volání funkce *add_set_of_1intersecting_rows*, neboť jistě při nejednotkových cenách sloupců bude více případů, kdy budeme moci optimalizaci provést, než v případě *stillToRaise = 1*. Připomeňme si hlavní myšlenku – pokud je *stillToRaise = 1*, a máme v matici jednotkové ceny sloupců, pak přidáním jakékoliv domény do krychle způsobíme oříznutí větve. Jelikož nyní máme ceny sloupců různé, oříznutí větve nastane v případě, že přidáme do krychle doménu s cenou převyšující nebo rovný *stillToRaise.* Pokud bychom narazili na řádek, jehož doména D_r protíná pouze jednu doménu D_i vstupní krychle, pak situace, kdy *cena* $(D_i \setminus D_r) \geq$ *stillToRaise* způsobí přidání další domény a následně oříznutí větve. Můžeme tedy tyto sloupce z D_i vyjmout. Tento fakt nám ovšem při různých cenách sloupců způsobí, že není jednoznačně daná situace kdy tuto podmínku testovat. Zřejmě pokud bychom stále přepočítávali rozdíly domén s cenami všech sloupců, byl by výpočet tak náročný, že by způsobil spíše zpomalení výpočtu, neboť režie vynaložená by převyšovala režii ušetřenou. Měli bychom tento výpočet provádět pouze v případě, kdy je velmi pravděpodobné, že situace nastane. Můžeme například testovat, zda sloupec nejnižší cenou protnuté domény D_i způsobí situaci, kdy $stillToRaise \leq 0$. V takovém případě můžeme všechny sloupce $D_i \setminus D_r$ z domény odebrat. Po tomto kroku je nezbytné znovu přepočítat hodnotu *stillToRaise*, neboť je počítána jako *lBound + n – cena (nejlevnejší sloupec domény),* a vyjmutím sloupců v předchozím kroku se *cena (nejlevnejší sloupec domény)* může změnit.

Velikost *MSIR* nemůže udávat počet nezávislých řádků, ale nejnižší cenu sloupců, které tyto řádky pokrývají. Poslední úprava se týká hodnoty maxRaiser a funkce *found_solution,* kdy *found_solution* musí z krychle vybírat do řešení sloupce s nejnižší cenou. Hodnotu *maxRaiser* musíme přizpůsobit ceně sloupců. Jelikož při libovolných cenách sloupců dosáhneme rozdílu *ubound - lbound* = 1 jen zřídka, v programu byla zvolena hodnota *maxRaiser*průměrná cena sloupců*, jejíž počítání je zajištěno funkcí *SetMaxRaiser*. Funkci raiser upravenou pro nejednotkové ceny sloupců zobrazuje algoritmus 5.5.1, tučně jsou zobrazeny úpravy pro nejednotkové ceny sloupců.

```
raiser (CubeC, n, A, lbound, solRaiser, ubound) {
    stillToRaise = lbound + n – mincost (CubeC)
    /*pokud jsme dostatečně navýšili spodní hranici, vrátíme 1*/
    if (stillToRaise<=0) return 1

    /*pokud je matice A prázdná, vrátíme řešení*/
    if ( A = 0 ) return found_solution ( CubeC, n, solRaiser, ubound )
    BSONIR = find_best_set_of_non_intersecting_rows (A, CubeC)
    for each row r ∈ BSONIR {
        CubeC = add_domain( CubeC, A, r )
        stillToRaise = stillToRaise – mincost (CubeC)
        if ( stillToRaise <= 0 ) return 1
```

```
        }
        A = A \ BSONIR
        if ( A = 0 ) return found_solution ( CubeC, n, solRaiser, ubound )
        /*Musíme se s cenami sloupců přiblížíme k stillToRaise*/
        if (stillToRaise ≤ mincost (CubeC)) {

                if (add_set_of_1intersecting_rows (A, CubeC ) = 1 ) return 1
                stillToRaise = stillToRaise – mincost (CubeC)
                if ( stillToRaise <= 0 ) return 1
                if (A = 0 ) return found_solution ( CubeC, n, solRaiser, ubound )

        }
        /*Vybereme další nejlepší řádek z matice A*/
        r = select_best_uncovered_row ( A, CubeC )
        A = A \ { r }
        /*rozdělíme CubeC na part1 a part2, podle řádku r */
        split_cube { CubeC, A, r, part1, part2 }
        /*přidáme k Cube_{k+1} část pokrývající řádek r*/
        Cube_{k+1}=add_domain (Cube_{k+1}, A, r )
        /*nyní větvíme podle vytvořených krychlí*/
        returnValue=1
        while ( part1 ∪ part2 ≠ ∅ ){
        Cube_j = select_next_cube( part1 ∪ part2 )
        /*Pokud bylo nalezeno lepší řešení, returnValue nastavíme na 0*/
        if ( raiser ( Cube_j, n, A, lbound, solRaiser ) = 0 )
                retunrValue=0
        }
        return returnValue
}
```

Algoritmus 5.5.1 – funkce raiser pro nejednotkové ceny sloupců

5.6 Pokročilé heuristické metody

Nalezení exaktního řešení však pro větší matice trvá velmi dlouhou dobu. V některých případech je použití heuristické funkce pro nalezení řešení UCP jediná možnost. V této kapitole si stručně popíšeme principy jednoduchých heuristik popsaných v [9], [10], [11] a pokročilých heuristických metod popsaných v [6].

Heuristika Natural, [9], [10], [11]

NaturalHeuristic (A, *Sol*)

```
        while ( CM == ∅ ) {
        1. Zjisti počet jedniček ve všech sloupcích.
        2. Vyber sloupec s nejvíce jedničkami.
        }
```

Algoritmus 5.6.1 – Heuristika Natural

Heuristika ContribAdd, [9], [10], [11]

ContribAddHeuristic (A, *Sol*)

```
while ( CM == ∅ ) {
        for each y_j ∈ Y find SP(y_j)
        select y with maximum SP, add to solution and remove it from matrix with
        rows it covers
}
```

Algoritmus 5.6.2 - Heuristika ContribAdd

Za zmínku ještě stojí mnou navržená úprava heuristiky ContribAdd pro nejednotkové ceny sloupců. Upravenou heuristiku zobrazuje algoritmus 5.6.3.

ContribAddHeuristic (A, *Sol*)

```
while ( CM == ∅ ) {
        for each y_j ∈ Y find SP(y_j)
        for each y_j ∈ Y SP(y_j) = SP(y_j)/Cost(y_j)
        select y with maximum SP, add to solution and remove it from matrix with
        rows it covers
}
```

Algoritmus 5.6.3 - Úprava heuristiky ContribAdd pro nejednotkové ceny sloupců

Takto upravená heuristika dosahuje výborných výsledků v přijatelném čase, výsledky jsou uvedené v kapitole 6.

Podstatou pokročilých heuristik, implementovaných v [4], jsou dvě funkce:

Nezbytnost (Essenciality) – funkce $e(y), y \in Y$ je heuristická funkce, pokud existuje řádek $r \in X$ který je pokryt y tak, že y je nezbytný sloupec, nabývá hodnoty TRUE, v opačném případě FALSE.
Hodnota(score) – Hodnota $p(y), y \in Y$ je heuristická funkce definovaná následovně pro jednotlivé heuristiky pro nalezení řešení UCP:

Pro heuristiku dle Servíta, [6]

$$p(y) = \sum_{i=0}^{m} \frac{a_{iy}}{c_y} \left(\sum_{j=0}^{n} \frac{a_{ij}}{c_j} \right)^{-1}$$

Pro heuristiku dle Bowmana, [6]

$$p(y) = \sum_{i=0}^{m} a_{iy} \left(\sum_{j=0}^{n} a_{ij} \right)^{-1}$$

pro heuristiku dle Neculy, [6]

$$p(y) = \sum_{i=0}^{m} \frac{a_{iy}}{c_y}$$

pro heuristiku dle Michalského, [6]

$$p(y) = \sum_{i=0}^{m} a_{iy}$$

pro heuristiku dle Rotha, [6]

$$p(y) = \frac{1}{c_y} \; jestliže \sum_{i=0}^{m} a_{iy} \neq 0, \; p(y) = 0 \; jestliže \sum_{i=0}^{m} a_{iy} = 0$$

Tyto heuristiky naleznou všechny nezbytné sloupce a přidají je do řešení, nezbytné sloupce jsou poté odstraněny z matice. Dále naleznou podle své hodnotící funkce sloupec s nejhorším ohodnocením a tento sloupec je odebrán z matice. Jelikož tento sloupec jistě nebyl nezbytný, všechny řádky jsou pokryté ještě jiným sloupcem. Dále pak hledáme opět všechny nezbytné sloupce a tento cyklus opakujeme dokud nejsou pokryty všechny řádky. Heuristiky můžeme pozměnit tak, že namísto hledání sloupců s nejhorším ohodnocením vyhledáme sloupce s nejlepším ohodnocením a tyto sloupce přidáme do řešení. Takto upravené heuristiky budeme nazývat *pozitivní.*

6 Experimentální výsledky

Nyní je třeba implementovaný algoritmus upravit pro integraci do programu Boom. Algoritmus by měl být schopen rozhodnout, zda řešení UCP(A) dané matice A bude s nejvyšší pravděpodobností nalezeno Aurou v rozumném čase a lze pro nalezení UCP(A) tento algoritmus použít, nebo zda bude nutné použít některou heuristickou funkci popsanou v [4]. K tomu musíme otestovat závislost běhu Aury a heuristik na parametrech matice, abychom pomocí těchto parametrů byli schopni zjistit, zda proběhne výpočet Aurou v únosném čase a rozhodnout, kdy je již třeba zvolit heuristickou funkci. Dále je třeba změřit závislost času běhu všech heuristických funkcí na rostoucích parametrech matice a rozhodnout, která heuristika bude nejlepší s ohledem na dané parametry matice. Pro měření je třeba generátoru matic schopného vygenerovat matice s proměnným vybraným parametrem při zachování ostatních parametrů konstantních.

6.1 Generátor matic

Pro generování matic určených k měření závislosti výpočetního času jsem navrhl generátor generující matice podle zadaných parametrů. Program používá generátor náhodných čísel, který na základě požadované hustoty *H* matice generuje s pravděpodobností *H* číslo 1, a s pravděpodobností *H-1* generuje číslo 0. Pravděpodobnost je počítána funkcí *random(100)*. Pokud je hodnota vygenerovaná touto funkcí menší než požadovaná hustota H, generátor vloží do matice 1, v druhém případě 0. Generátor je znázorněn algoritmem 6.1.1

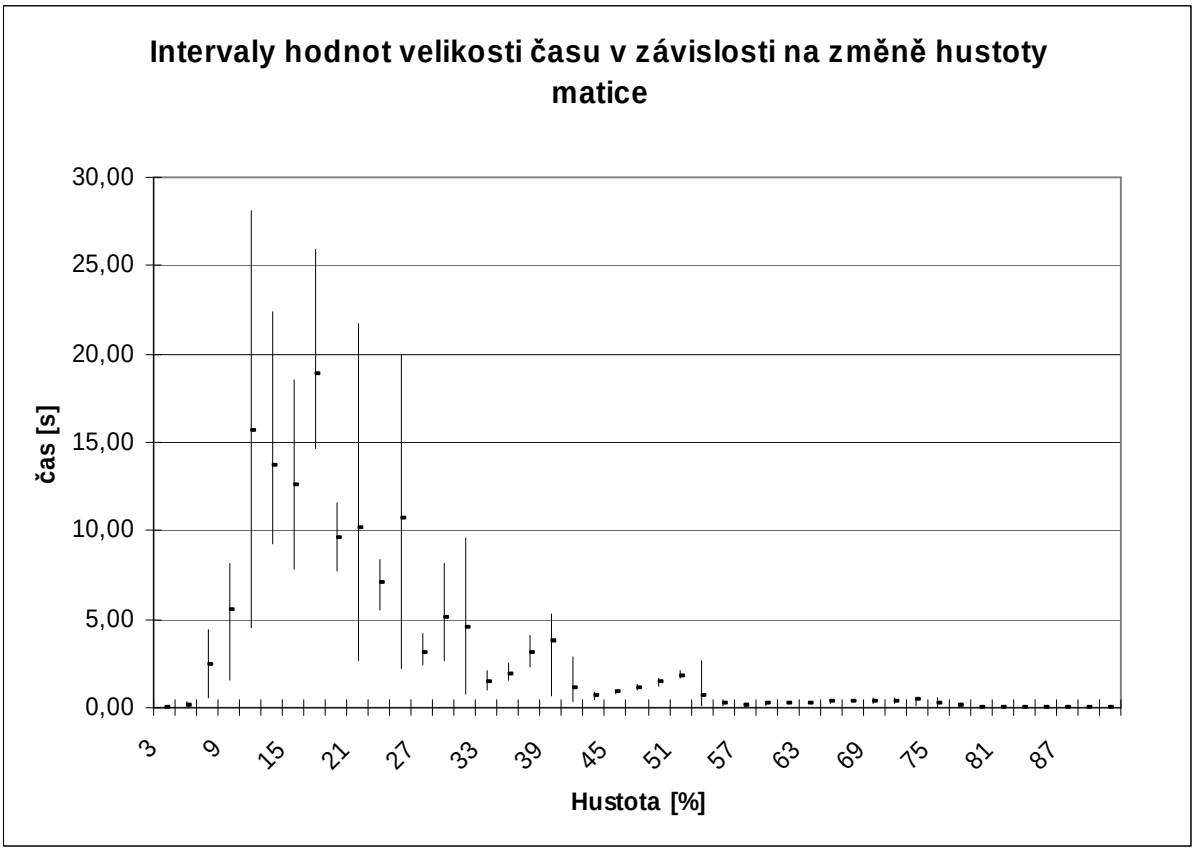
```
generator (NumColumns, NumRows, Density )
{
        while I < NumRows
        {
                while J < NumColumns
                {
                        if (random(100)<Density)
                         A[I,J] = 1
                        else
                         A[I,J] = 0
                }

        }
}
```

Algoritmus 6.1.1 – Generátor matic

6.2 Měření výpočetního času Aury – jednotkové ceny sloupců

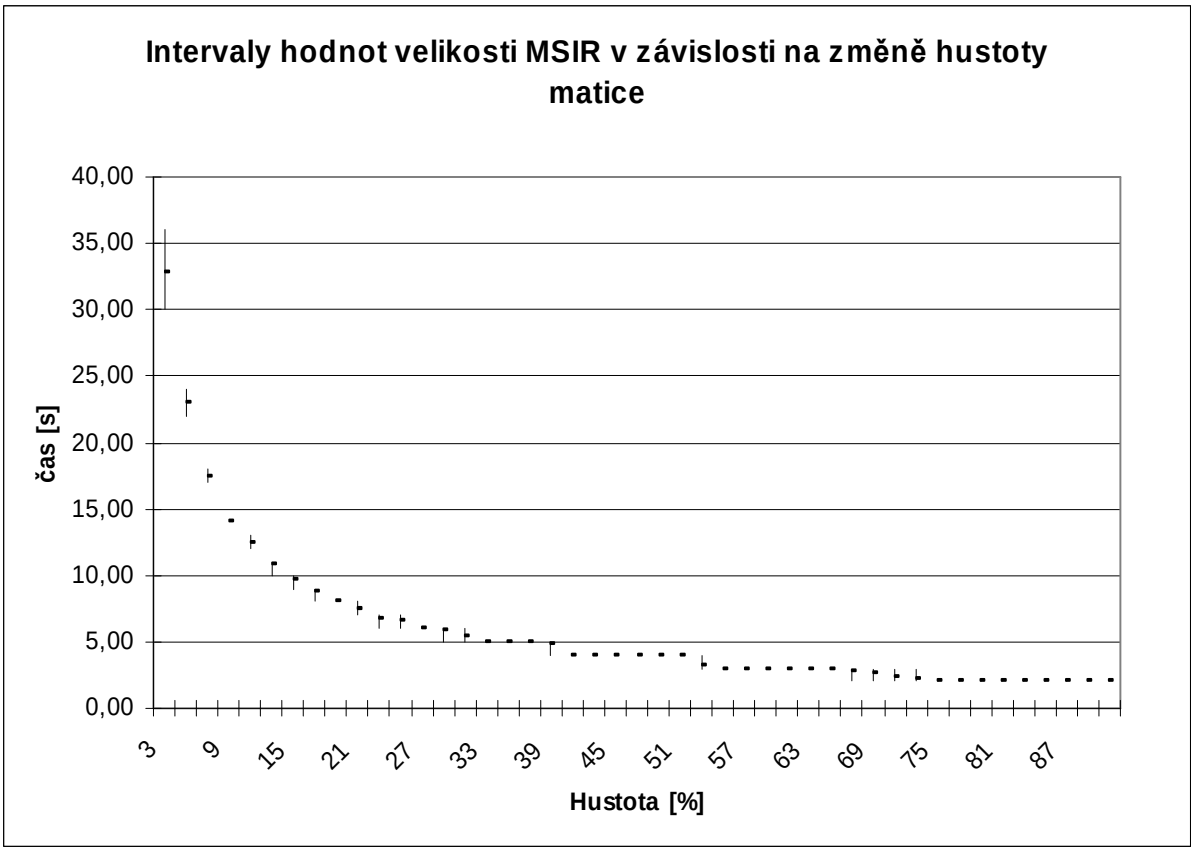
Nejprve je třeba změřit závislosti času běhu Aury na parametrech matice a na základě výsledků rozhodnout, pro jakou hodnotu parametrů již výpočetní čas přesáhne únosnou mez. Únosná mez je zvolena pro čas *t=3s*. Pro získání matic potřebných k měření jsem navrhl generátor popsaný v předchozí kapitole. Matice byly generované tak, aby mohla být změřena závislost změny jednoho parametru při konstantních ostatních parametrech. Jako parametry pro měření jsem zvolil počet řádků, počet sloupců a hustotu matice

Nejprve jsem provedl měření závislosti času na změně hustoty matice, při zachování stejného počtu řádků a sloupců. Měřeno bylo několik matic se stejnou velikostí 100 sloupců a 100 řádků, které byly generované pro narůstající hustotu od 3% do 93%. Naměřené hodnoty jsou zobrazeny v grafu 6.2.1. Je zobrazen interval maximálních a minimálních hodnot naměřených časů.



Graf 6.2.1 – Závislost času běhu Aury na změně hustoty matice

Z grafu je patrný největší nárůst času při nízké hustotě, a to při hustotě mezi 10% a 40%. Při hustotě nad 40% již výpočetní čas má tendenci klesat. Při nejnižší hustotě je výpočetní čas nízký, nejspíše z důvodu vysoké spodní hranice dané velikosti MSIR, závislost velikosti MSIR zobrazuje graf 6.2.2. Nárůst výpočetního času v oblasti mezi 10% - 30% je pravděpodobně způsoben poklesem velikosti MSIR s rostoucí hustotou a zároveň nízkým vznikem dominantních řádků a sloupců při větvení algoritmu. Pravděpodobnost vzniku dominancí je s rostoucí hustotou vyšší, tudíž dochází k zmenšování matice rychlejším tempem při větvení algoritmu a tím ke snížení výpočetního času. Při hustotě matice 10% - 25% lze tedy očekávat nárůst výpočetního času. Odhad velikosti matice, pro kterou bude AURA schopná nalézt řešení UCP v přijatelném čase musíme tomuto faktu přizpůsobit.



Graf 6.2.2 – Závislost velikosti MSIR na změně hustoty matice

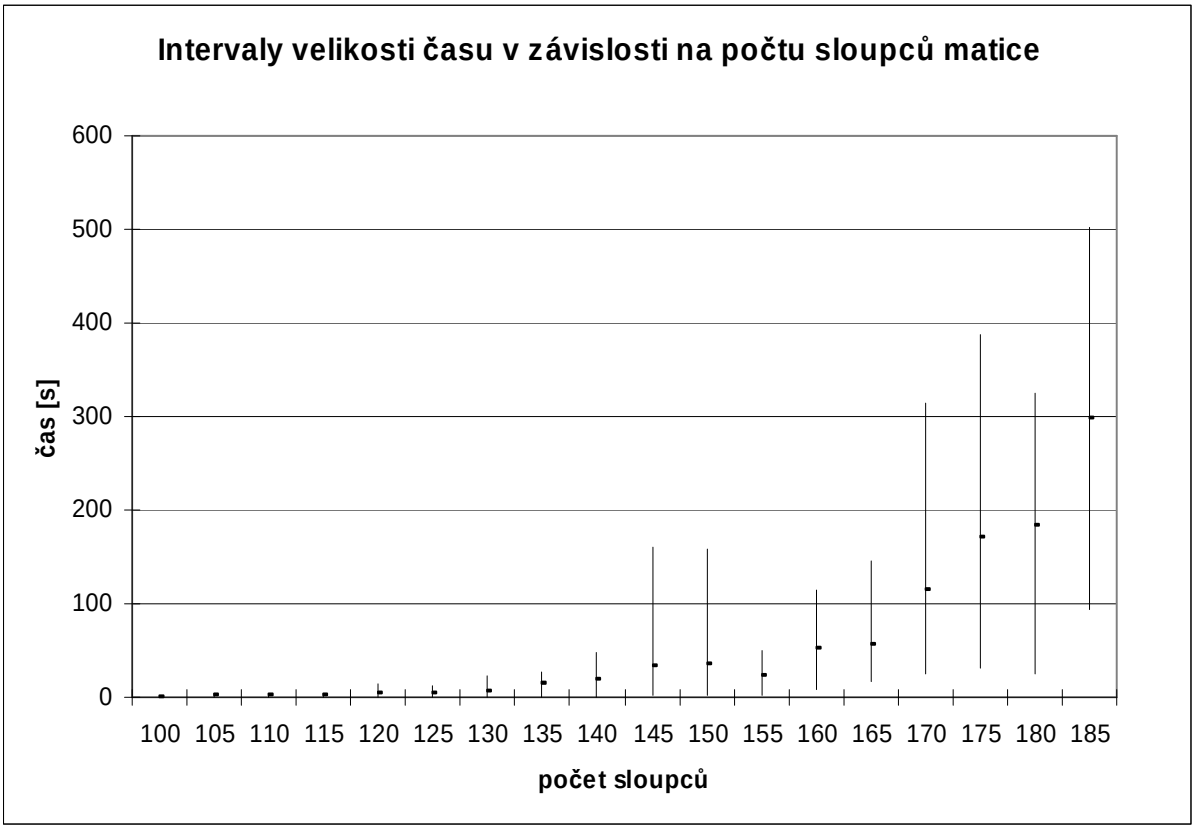
Z naměřených hodnot v grafu 6.2.1 vyplývá, že nejvyšší výpočetní čas pro nalezení řešení UCP je potřeba pro matice o hustotě v rozmezí přibližně 7 – 25%. Zbývá ještě otázka, jaké je rozmezí hustoty u reálných matic. Těmto hodnotám bude přizpůsobený odhad maximální velikosti matice, pro které budeme hledat řešení UCP exaktní metodou AURA.

Pro zjištění hodnot hustoty cyklických jader reálných matic byly získány matice programem BOOM. V tabulce 6.2.1 jsou uvedeny hustoty cyklických jader reálných matic. Jak vidíme, hustota cyklických jader reálných matic je v rozmezí 1% - 15% a ve většině případů převyšuje počet sloupců cyklického jádra nad počtem řádků, a to někdy i několikanásobně. V tomto rozmezí má výpočetní čas tendenci stoupat směrem k vyšším hodnotám, jak vyplývá z grafu 6.2.1. Pro další měření závislosti výpočetního času Aury na velikosti matice jsou zvoleny matice s hustotou 5% a 15%.

počet sloupců	Počet řádků	Hustota	počet sloupců	Počet řádků	Hustota [%]
179	75	2,99	45	40	7,56
275	88	2,62	47	40	7,61
299	176	2,08	99	50	7,35
320	181	2,04	115	51	7,01
498	491	0,62	182	84	4,59
524	108	10,41	210	87	4,50
716	100	5,56	221	49	16,47
940	1036	0,45	258	123	2,95
1206	159	3,50	313	131	2,87
1494	1599	0,29	455	184	2,26
1528	108	10,27	501	186	2,22
1648	100	5,49	633	49	16,05
1980	159	3,41	5027	958	0,98
3230	4059	0,21	13657	958	0,95

Tabulka 6.2.1 – Hustota cyklických jader reálných matic

Dále je nutné změřit závislost velikosti výpočetního času Aury na velikosti matice. Graf 6.2.3 zobrazuje závislost růstu výpočetního času na počtu sloupců matice při konstantním počtu řádků, bylo vygenerováno několik stejných matic, počet řádků je zvolen *m=120* a hustota *h=5%*. Je zobrazen interval maximálních a minimálních hodnot naměřených časů.



Graf 6.2.3 – Závislost času běhu Aury na počtu sloupců matice, hustota matice 5%

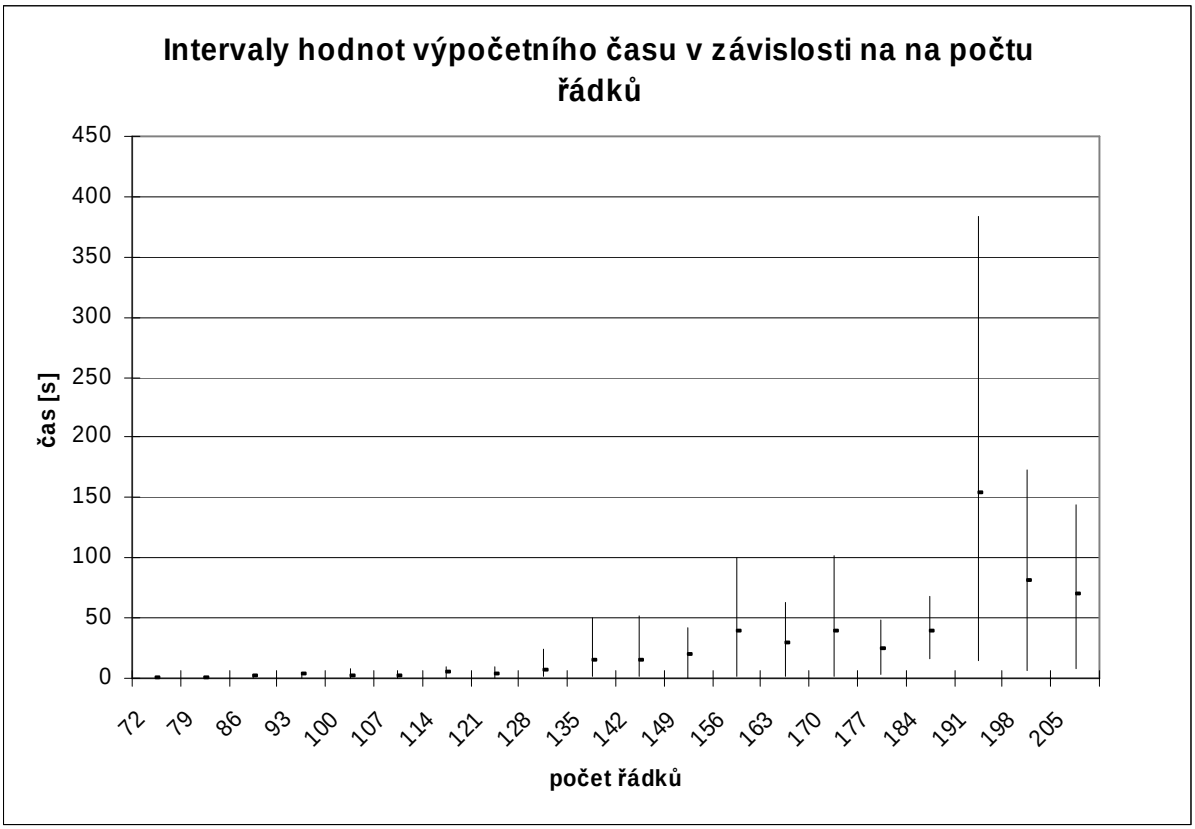
Počet sloupců volíme tak, abychom mohli odhadnout, pro které matice již čas potřebný k nalezení řešení UCP převyšuje požadovanou velikost 3 sekundy. Z grafu 6.2.3 vidíme, že pro počet sloupců *n=130* již čas znatelně roste do jednotek až desítek sekund, ale i při počtu sloupců *n=170* ještě stále můžeme nalézt řešení UCP v přípustném čase. Podrobnější výpis časů pro porovnání konkrétních hodnot je uveden v tabulce 6.2.2.

Počet řádků	počet sloupců	Matice1 čas [s]	Matice2 čas [s]	Matice3 čas [s]	Matice4 čas [s]	Matice5 čas [s]	Matice6 čas [s]
120	100	0,29	0,35	0,21	1,02	0,34	0,14
120	105	0,59	0,68	0,4	3,52	0,78	0,43
120	110	0,3	1,38	1,29	2,61	0,66	0,24
120	115	3,37	2,96	0,23	1,15	0,95	0,33
120	120	1,03	14,8	0,72	3,27	0,3	0,28
120	125	5,01	7,71	2,64	12,15	1,51	0,98
120	130	2,14	3,86	22,9	3,4	2,98	0,84
120	135	7,31	13,26	26,97	23,7	16,88	0,29
120	140	3,44	31,66	22,62	0,94	48,33	2,45
120	145	5,42	8,19	8,75	15,54	160,25	2,81
120	150	18,03	12,79	158,96	4,93	15,18	2,46
120	155	2,14	28,39	49,41	34,72	10,35	6,95
120	160	67,12	36,13	29,16	8,25	114,62	56,81
120	165	36,19	145,83	16,68	54,33	57,92	24,71
120	170	130,06	49,45	313,81	24,81	118	57,37
120	175	145,62	333,51	387,29	82,61	39,42	30,33
120	180	324,42	243,07	25,87	275,68	33,72	199,12
120	185	154,74	384,19	503,05	147,3	93,04	503,05

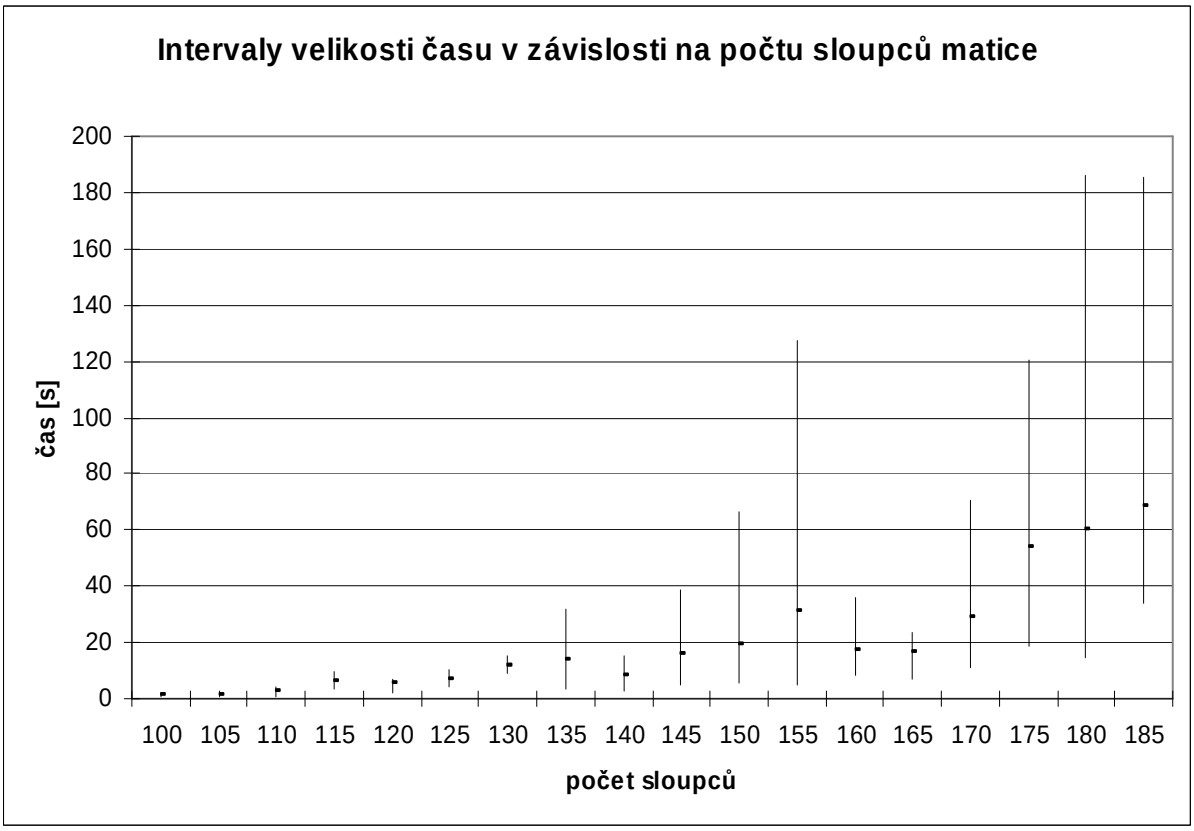
Tabulka 6.2.2 – Závislost výpočetního času Aury na počtu sloupců matice

Dále je nutné zjistit závislost velikosti času potřebného k nalezení řešení UCP na počtu řádků matice. Generováno bylo opět několik stejných matic. Graf 6.2.4 znázorňuje závislost velikosti času potřebného k nalezení řešení UCP na počtu řádků matice, generovány byly matice o počtu sloupců *n=115*, tento počet byl zvolen na základě odhadu z grafu 6.2.3, jelikož tato hodnota se blíží únosné hranici počtu sloupců matice pro kterou je ještě řešení UCP nalezeno v námi zvoleném přípustném čase.

Z grafu 6.2.3 vidíme, že výpočetní čas pro některé matice výpočetní čas přeroste únosnou hranici již při počtu řádků *m=130*, ale opět u některých matic ještě při počtu řádků *m=180* lze nalézt řešení v únosném čase. Graf 6.2.5 zobrazuje hodnoty času naměřené pro matice o rozměrech 80 řádků a 80 – 165 sloupců a hustotě 15%. Jak je vidět z těchto průběhů, časové nároky jsou pro takové parametry matic vyšší, jak napovídá graf 6.2.1.



Graf 6.2.4 – Závislost času běhu Aury na počtu řádků matice, hustota matice 5%



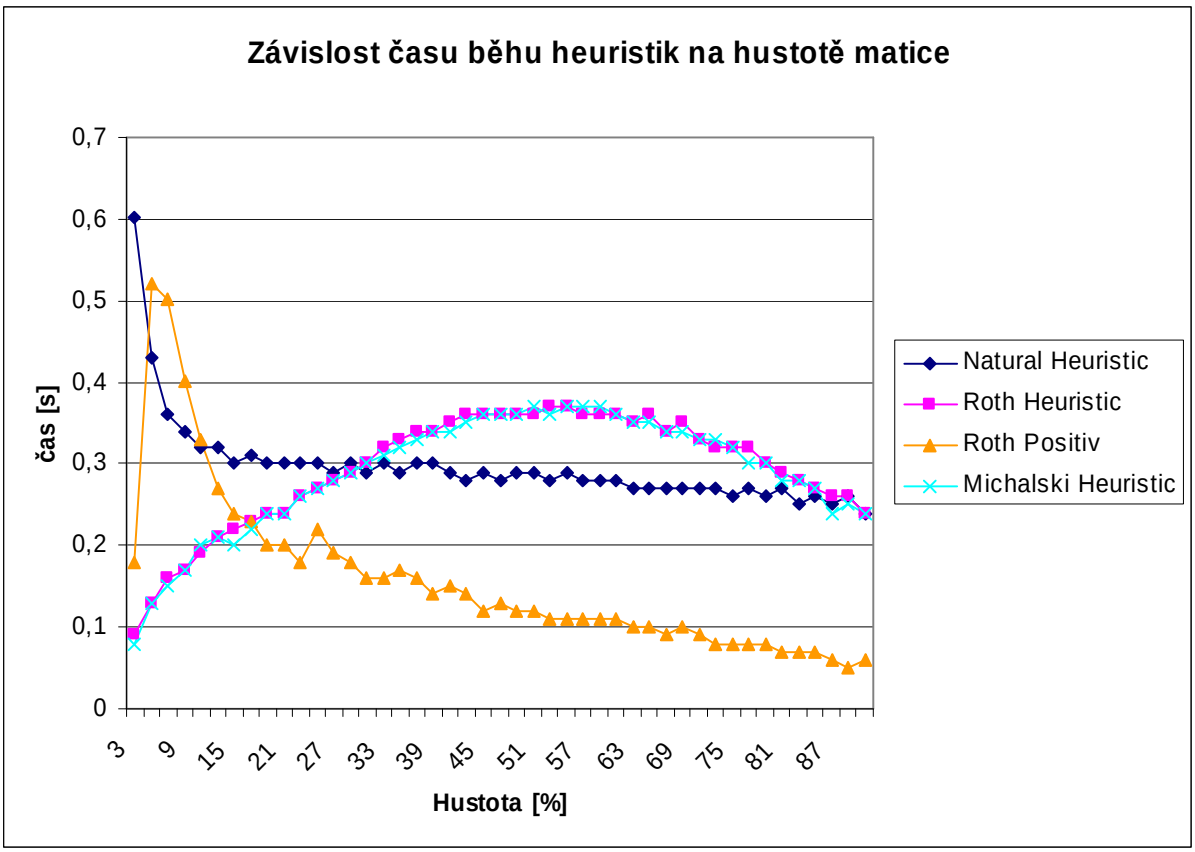
Graf 6.2.5 – Závislost času běhu Aury na počtu sloupců matice, hustota matice 15%

Z naměřených hodnot tedy vyplývá, že pro matice o rozměrech přes n=200 a m=200 a hustotou 5% nelze s velkou pravděpodobností očekávat nalezení řešení UCP v únosném čase. Při hustotě matice 15% je čas ještě vyšší. Pro vytvářený adaptační mechanizmus tedy volím maximální velikost matice 200 sloupců a 200 řádků a hustota do maxima 5% a 180 sloupců a 180 řádků s hustotou nad 5%, pro který ještě bude zvolena metoda Aura pro nalezení řešení UCP, Aura bude upravená tak, aby výpočet byl ukončen, pokud výpočetní čas přesáhne 3 sekundy, nebo jiný uživatelem zvolený čas, pak vrátí nejlepší nalezené řešení UCP. Jak vidíme z grafů uvedených v kapitole 6.3, ze všech použitelných metod tento způsob stále vrací řešení s nejmenší cenou.

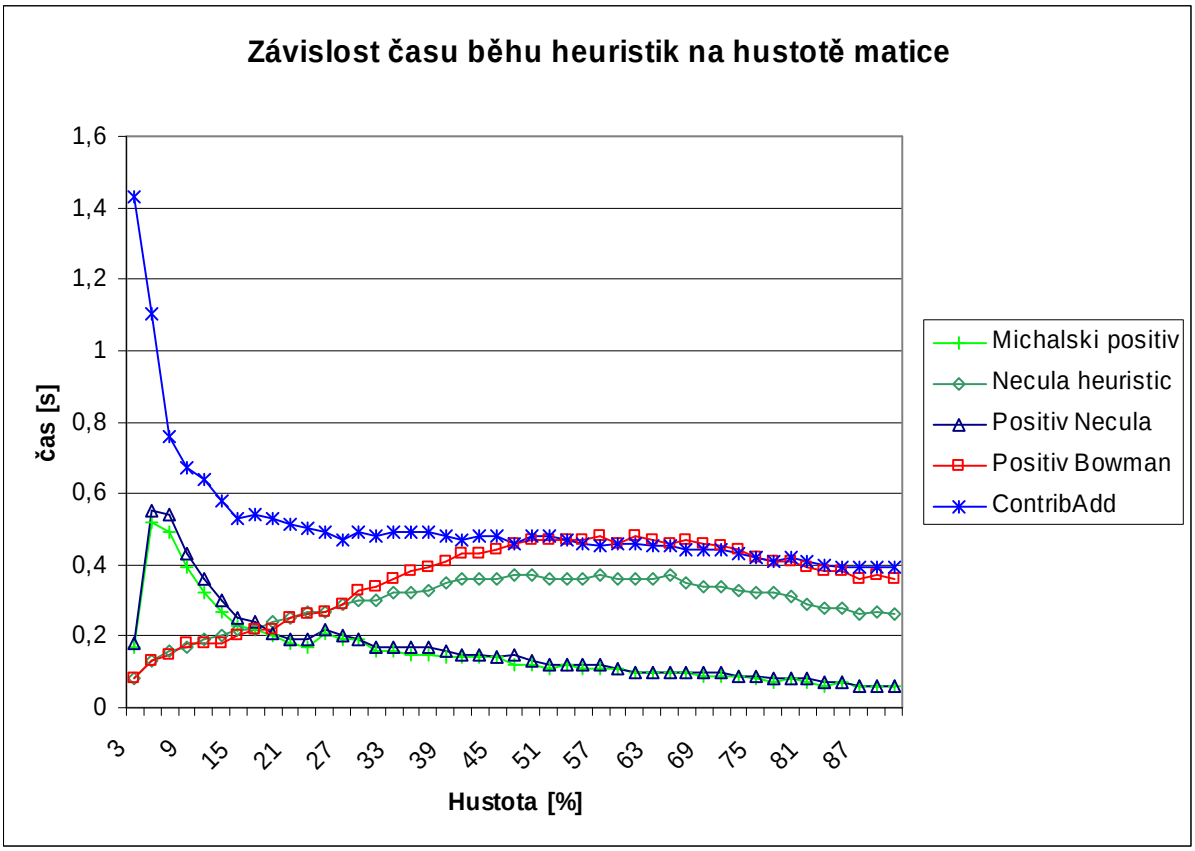
6.3 Měření výpočetního času heuristik – jednotkové ceny sloupců

Pokud již matice *A* svými parametry přesáhne možnost nalezení řešení UCP(A) Aurou, musíme zvolit pro nalezení řešení heuristickou funkci. V této kapitole otestujeme závislost všech heuristických funkcí na měnících se parametrech matice a z výsledků se pokusíme rozhodnout, kterou heuristiku bude nejlepší zvolit pro nalezení UCP(A) podle těchto parametrů, aby bylo dosaženo co nejmenšího času a zároveň řešení s cenou neblíže optimu. Průběhy jednotlivých heuristik jsou pro přehlednost rozděleny do několika grafů, ve stejných grafech jsou heuristiky s podobně vysokými časovými nároky.

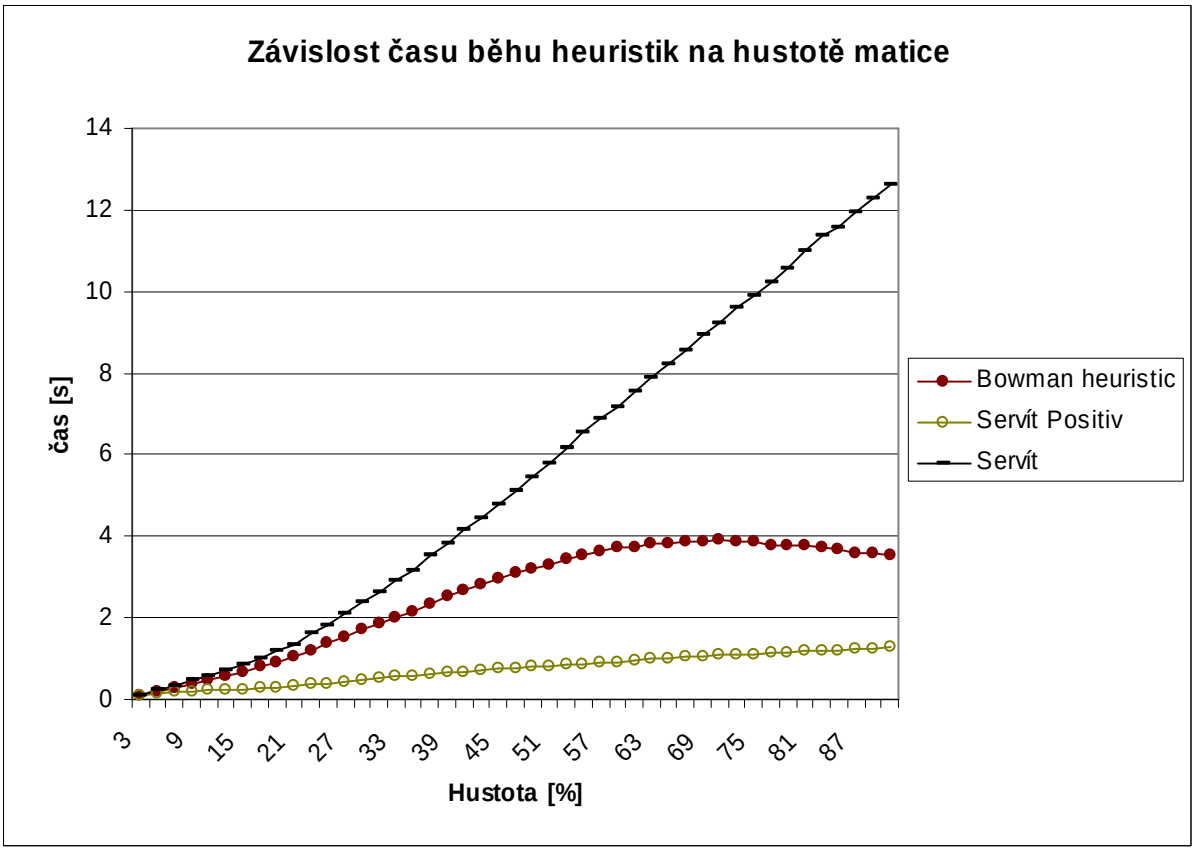
Nejprve je třeba změřit obecné závislosti hodnot výpočetního času heuristik na parametrech matice pro získání přehledu kvality a použitelnosti jednotlivých heuristik. Grafy 6.3.1, 6.3.2 a 6.3.3 znázorňují časovou závislost heuristik na hustotě matice, generované matice jsou o rozměrech m=150 řádků a n=150 sloupců.



Graf 6.3.1 – Závislost času běhu heuristik na změně hustoty matice



Graf 6.3.2 – Závislost času běhu heuristik na změně hustoty matice



Graf 6.3.3 – Závislost času běhu heuristik na změně hustoty matice

Z naměřených hodnot vidíme, že u heuristik Natural, Roth positiv, Michalski positiv, Necula positiv a ContribAdd je jen při nízkých hustotách vidět slabý nárůst výpočetního času, ale od hodnoty 5% průběh klesá. Největší časovou náročnost pozorujeme u Servít heuristiky, je to dáno její strategií hodnocení výběru sloupců.

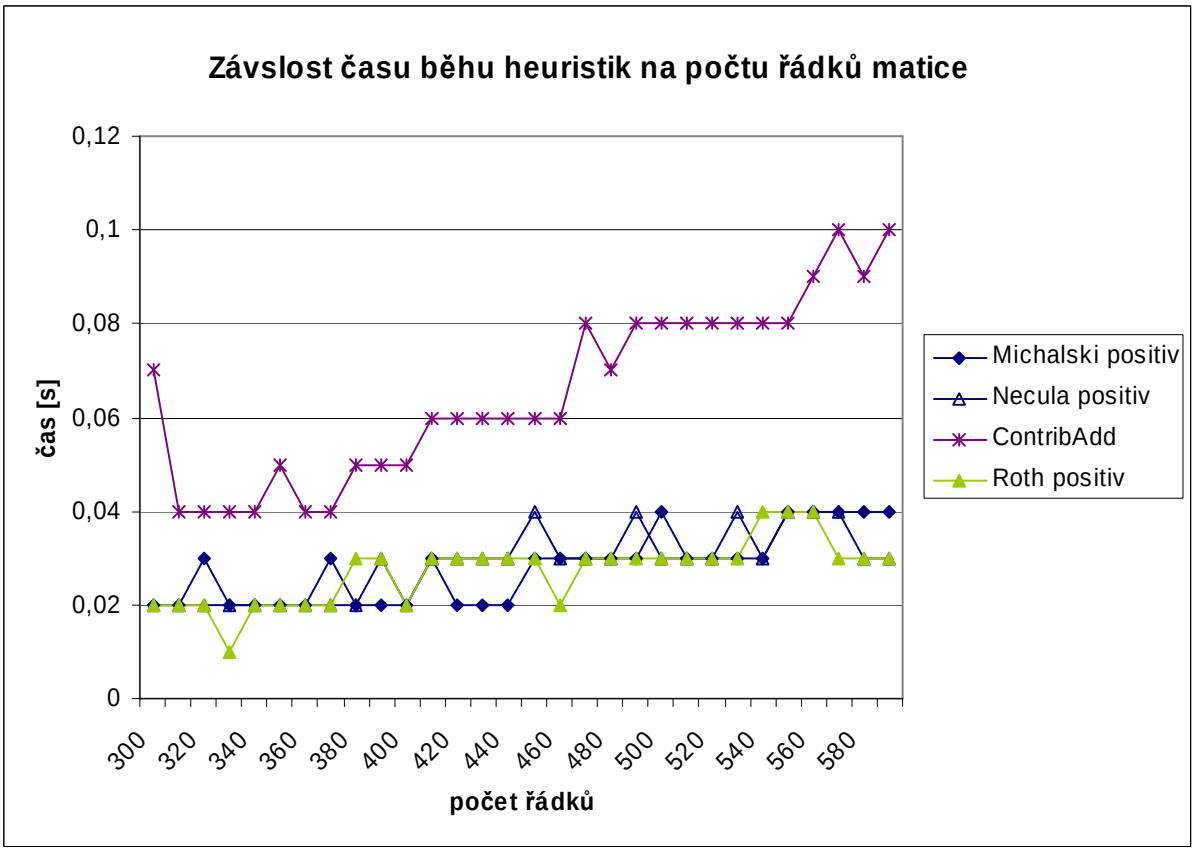
Nyní je tedy třeba změřit hustotu reálných matic. Matice byly získány programem BOOM, a byla měřena hustota celé matice, nikoliv jen cyklického jádra. Naměřené hodnoty jsou uvedeny v tabulce 6.3.1. Jelikož hodnoty hustoty reálných velkých matic se pohybují v rozmezí 1% - 15%, je třeba zvážit výběr jednotlivých heuristik pro nalezení řešení UCP.

počet sloupců	Počet řádků	Hustota [%]	počet sloupců	Počet řádků	Hustota [%]	počet sloupců	Počet řádků	Hustota [%]
11	10	25,45	2	4	50	17	15	36,86
11	10	25,45	4	4	75	22	24	20,45
19	108	10,33	4	4	75	33	60	12,32
28	34	4,62	4	11	36,36	38	77	22,69
29	297	11,31	6	10	26,67	46	65	6,12
32	297	11,32	8	11	30,68	48	36	12,27
32	297	11,32	8	11	30,68	54	62	7,41
40	100	6,4	11	49	19,85	62	62	12,49
44	34	5,55	20	43	10,7	67	73	15,66
46	34	5,5	26	58	9,88	77	42	12,96
56	98	4,76	30	56	8,27	82	71	14,14
76	149	3	36	58	9,91	85	82	13,96
101	160	3,63	36	49	7,99	93	62	13,11
130	203	2,29	38	43	10,4	95	104	8,35
138	241	1,94	38	58	9,75	104	90	14,96
211	98	4,33	39	43	10,26	126	105	8,53
244	98	4,25	46	94	6,73	128	100	7,64
310	149	2,8	48	49	7,65	128	90	11,18
348	149	2,77	49	49	7,66	159	124	8,62
381	241	1,92	62	105	4,52	171	122	7,94
407	241	1,89	108	94	6,33	203	203	6,15
524	108	10,41	108	94	6,31	219	938	12,2
541	203	2,18	115	105	4,34	240	237	4,5
600	203	2,16	119	105	4,35	243	830	14,81
734	100	5,5	128	56	7,55	318	281	3,9
1156	1495	0,33	143	56	7,32	352	156	5,66
1287	160	3,43	223	49	16,38	507	429	4,21
1536	108	10,25	265	958	1,03	874	1141	7,11
1778	100	5,38	653	49	15,95	1410	855	2,5
2168	160	3,31	5144	958	0,98	2025	851	6,15
7212	771	0,52	14032	958	0,95	3078	961	4,64

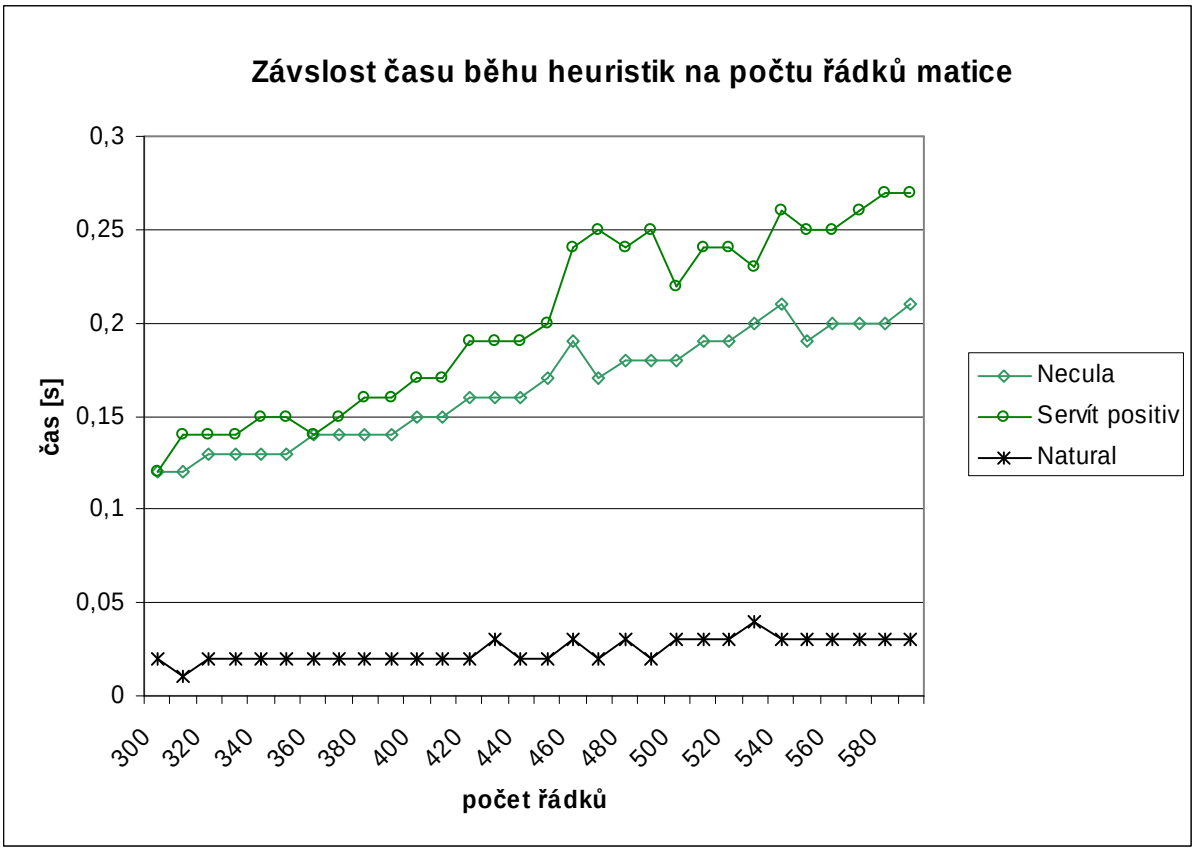
Tabulka 6.3.1 – Hustota reálných matic

Nyní musíme otestovat heuristiky v závislosti na změně rozměrů matice při zachování konstantní hustoty. Začneme s nízkými hodnotami rozměrů matic a budeme sledovat, jak roste výpočetní čas a kvalita nalezeného řešení UCP jednotlivých heuristik. Nejprve otestujeme čas pro nárůst počtu řádků matice. Grafy 6.3.4, 6.3.5, 6.3.6 a 6.3.7 zobrazují průběh pro generované matice, zvolený počet sloupců n=300, počet řádků m=300, hustota matic 5%, počet matic 30 s odstupem počtu řádků 10.

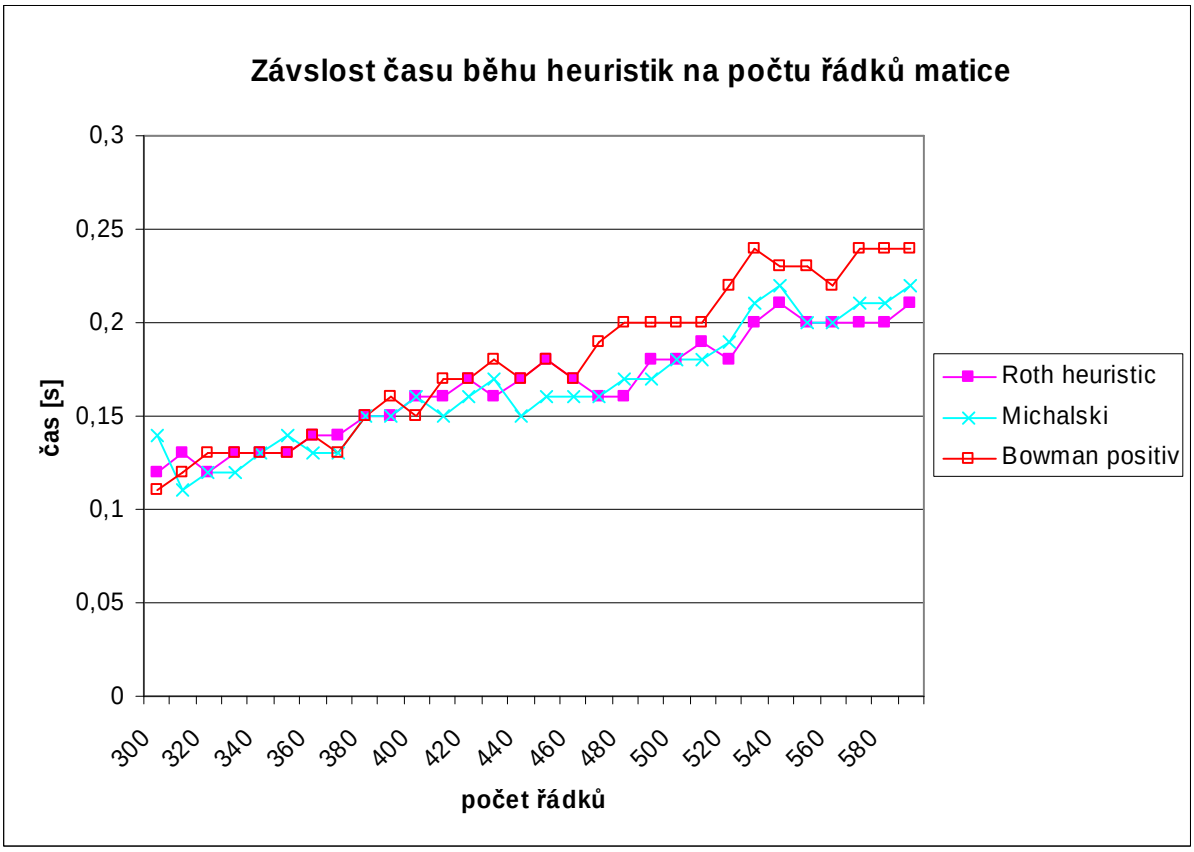
Grafy 6.3.7, 6.3.8, 6.3.9 a 6.3.10 zobrazují závislost výpočetního času heuristik na změně počtu sloupců matice, při konstantním počtu řádků o hodnotě m=300, hustota matic 5%. Z grafů vidíme, že závislost času pro měnící se počet sloupců při konstantních ostatních parametrech matice je mnohem vyšší, než pro měnící se počet řádků při zachování ostatních parametrů. Budeme se tedy dále věnovat měření závislosti heuristik na počtu sloupců, měření budeme provádět v oblastech, které nám pomohou získat odhad použitelnosti heuristik s ohledem na kvalitu nalezeného řešení.



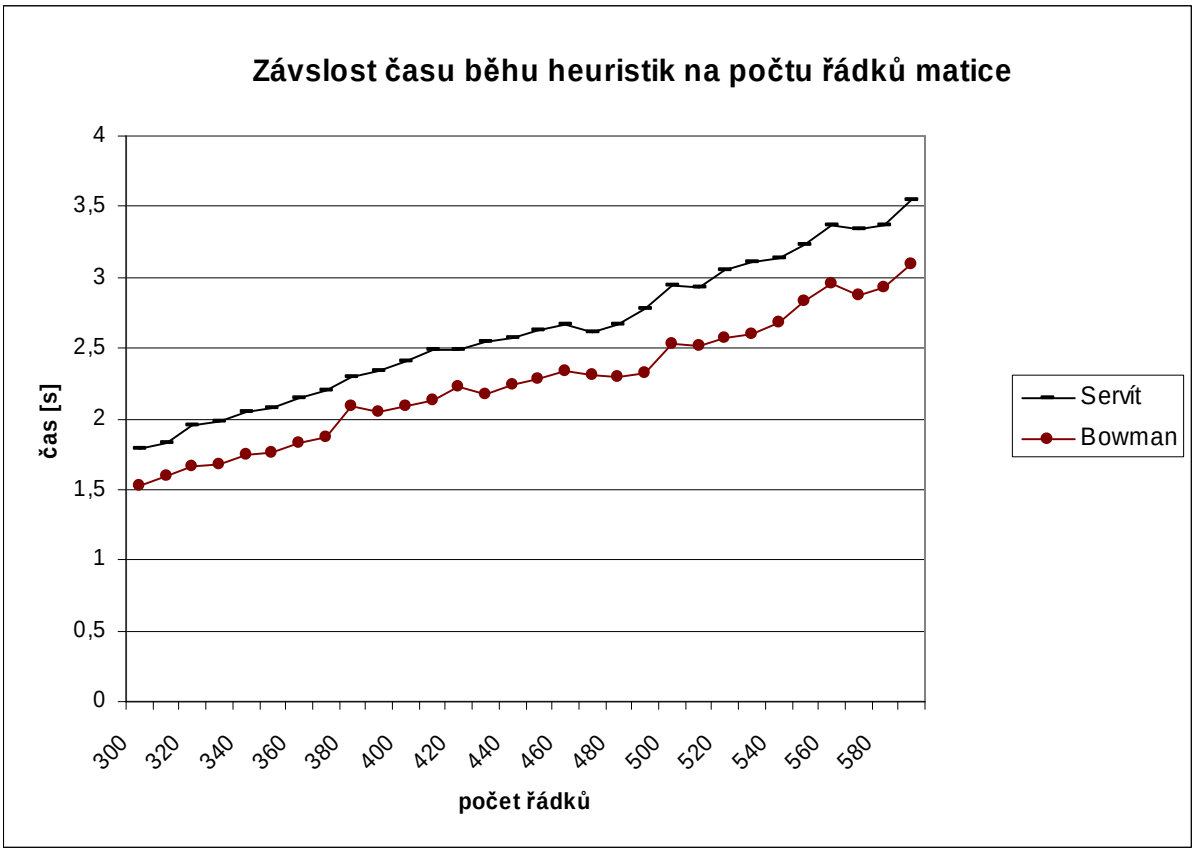
Graf 6.3.4 – Závislost času běhu heuristik na počtu řádků matice – hustota 5%



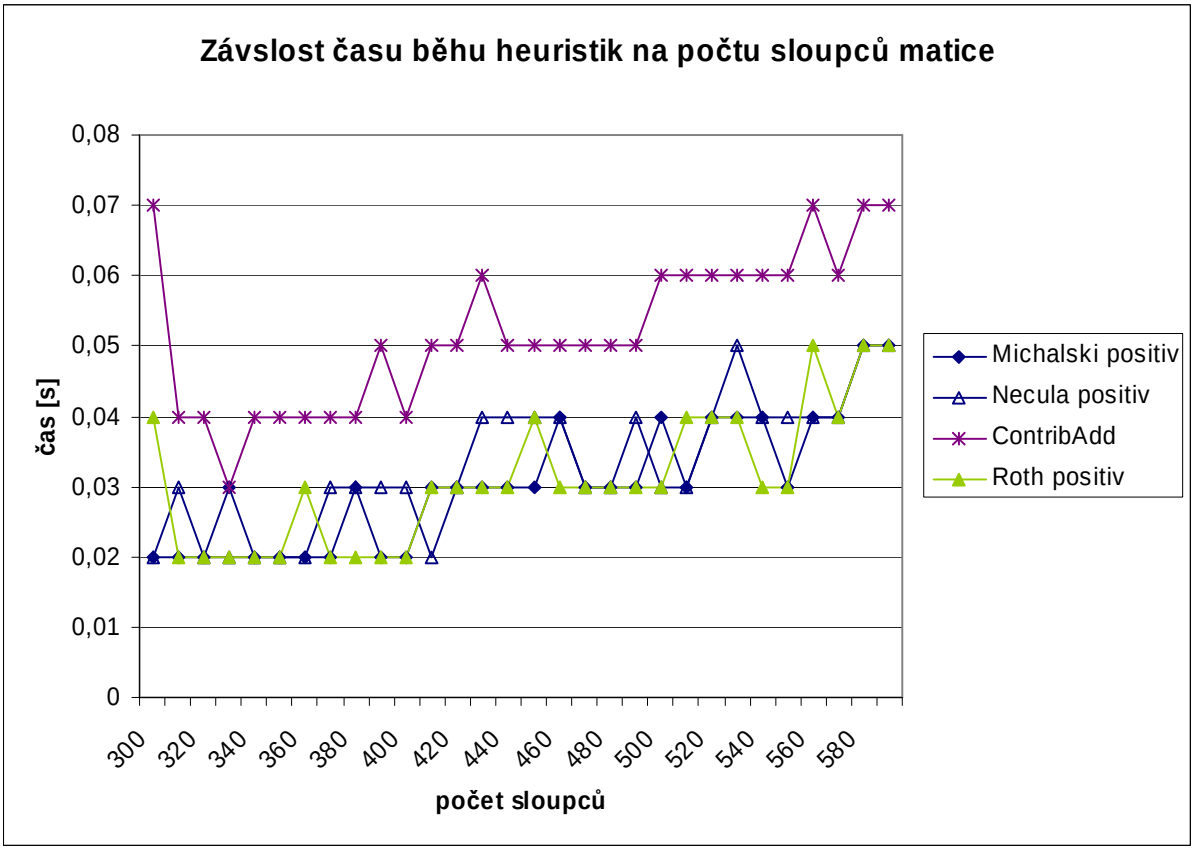
Graf 6.3.5 – Závislost času běhu heuristik na počtu řádků matice – hustota 5%



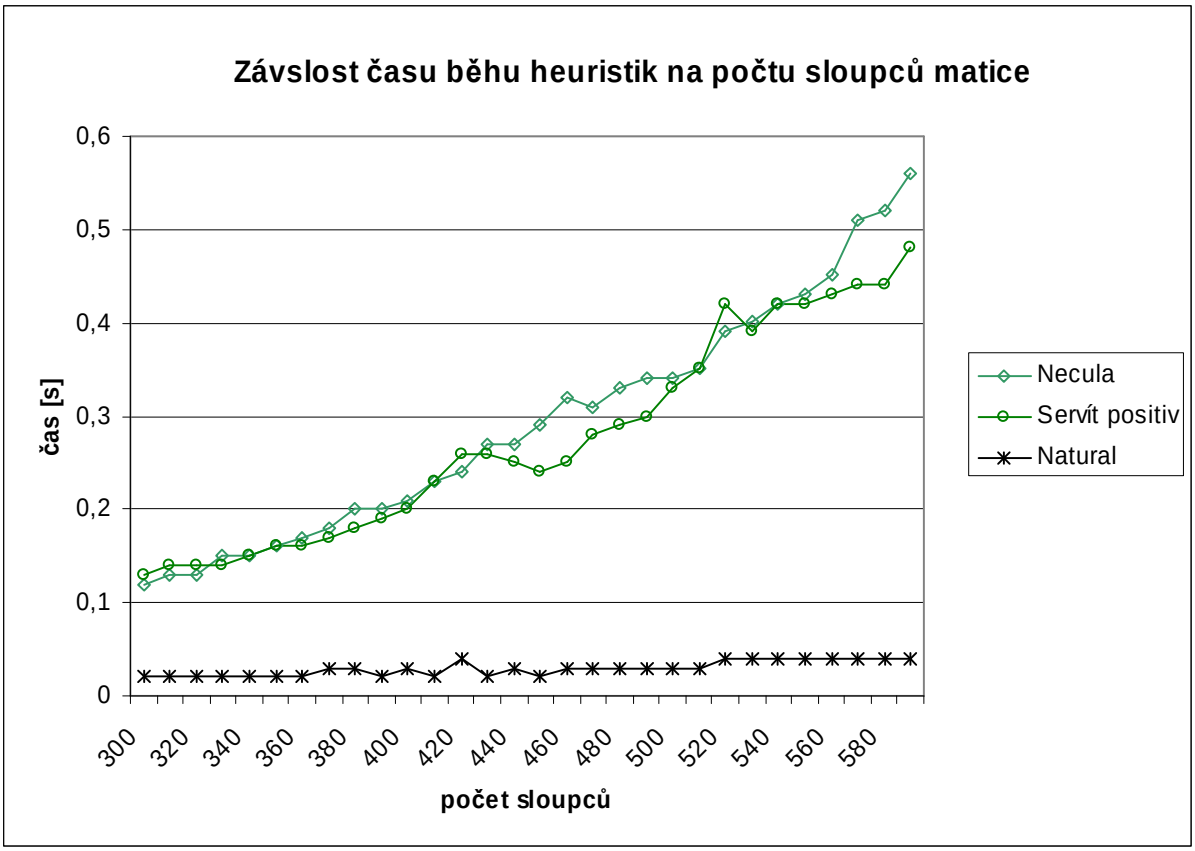
Graf 6.3.6 – Závislost času běhu heuristik na počtu řádků matice – hustota 5%



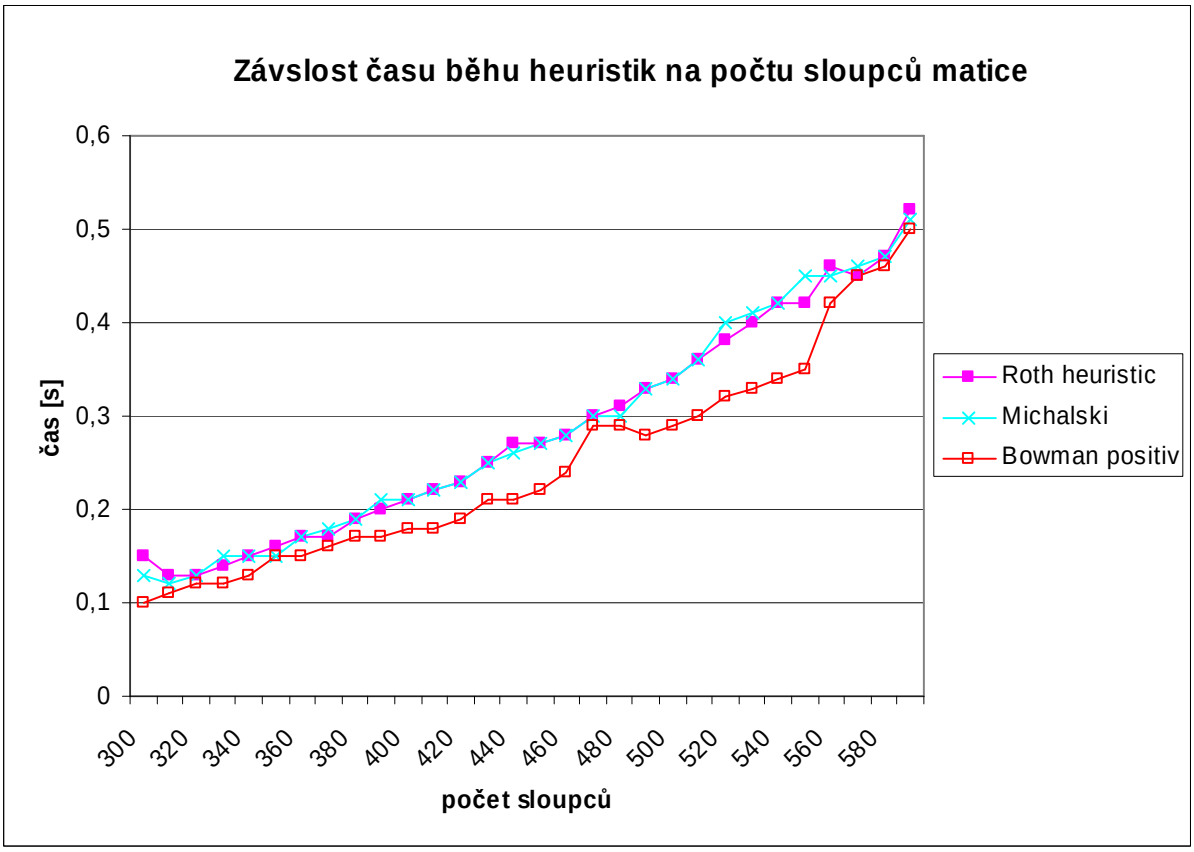
Graf 6.3.7 – Závislost času běhu heuristik na počtu řádků matice – hustota 5%



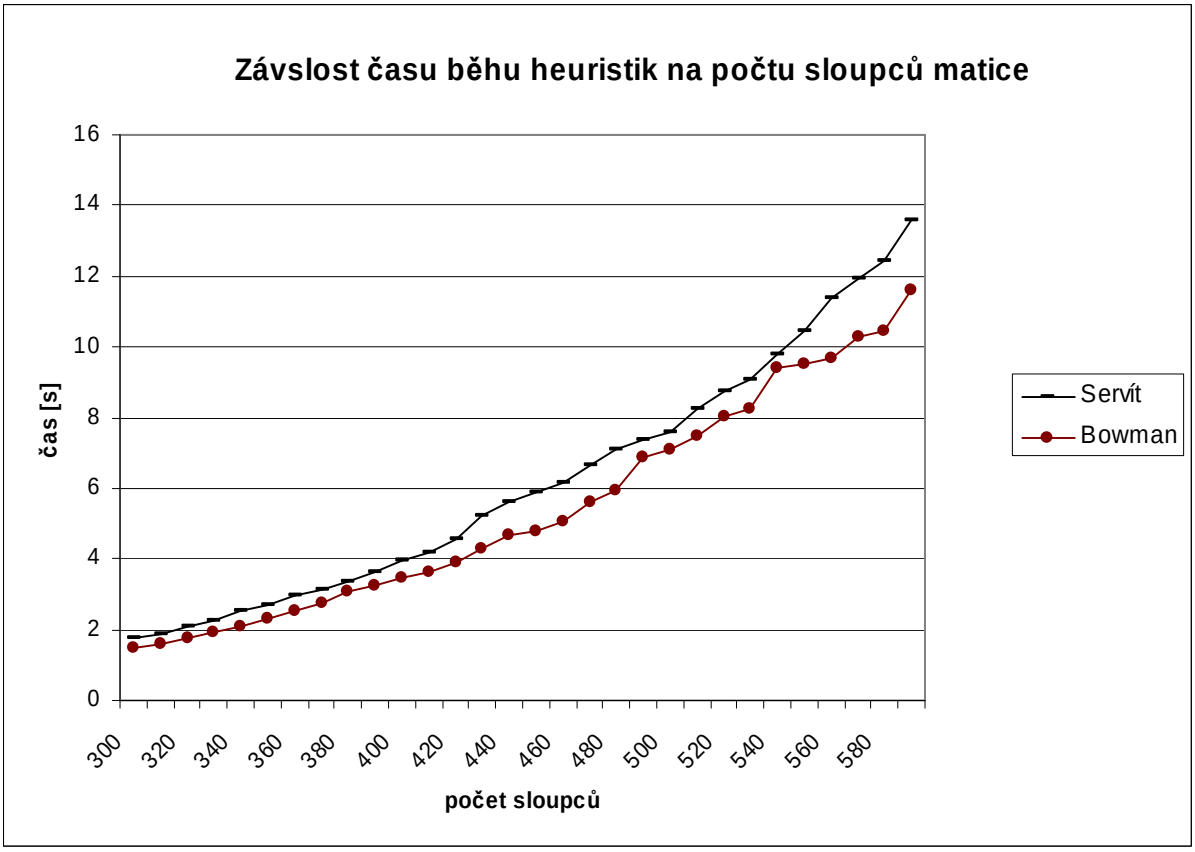
Graf 6.3.8 – Závislost času běhu heuristik na počtu sloupců matice – hustota 5%



Graf 6.3.9 – Závislost času běhu heuristik na počtu sloupců matice – hustota 5%



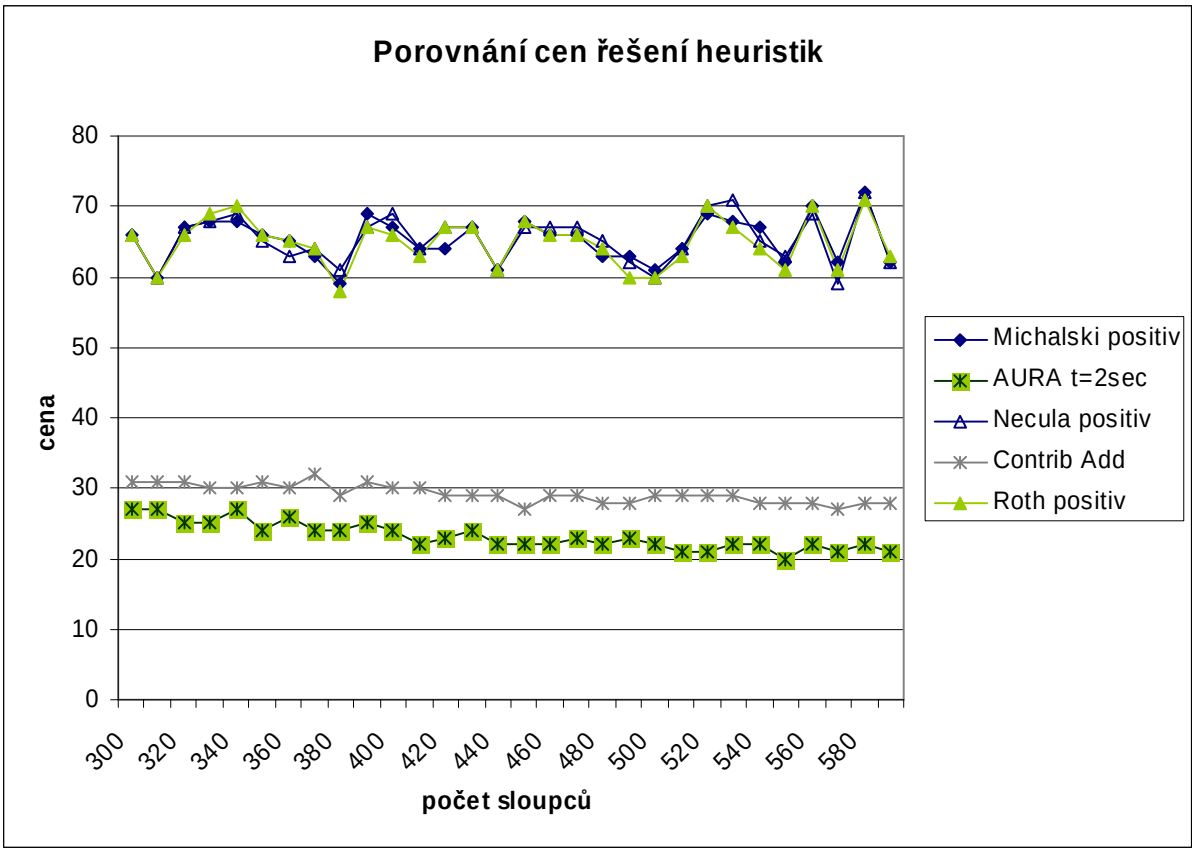
Graf 6.3.10 – Závislost času běhu heuristik na počtu sloupců matice – hustota 5%



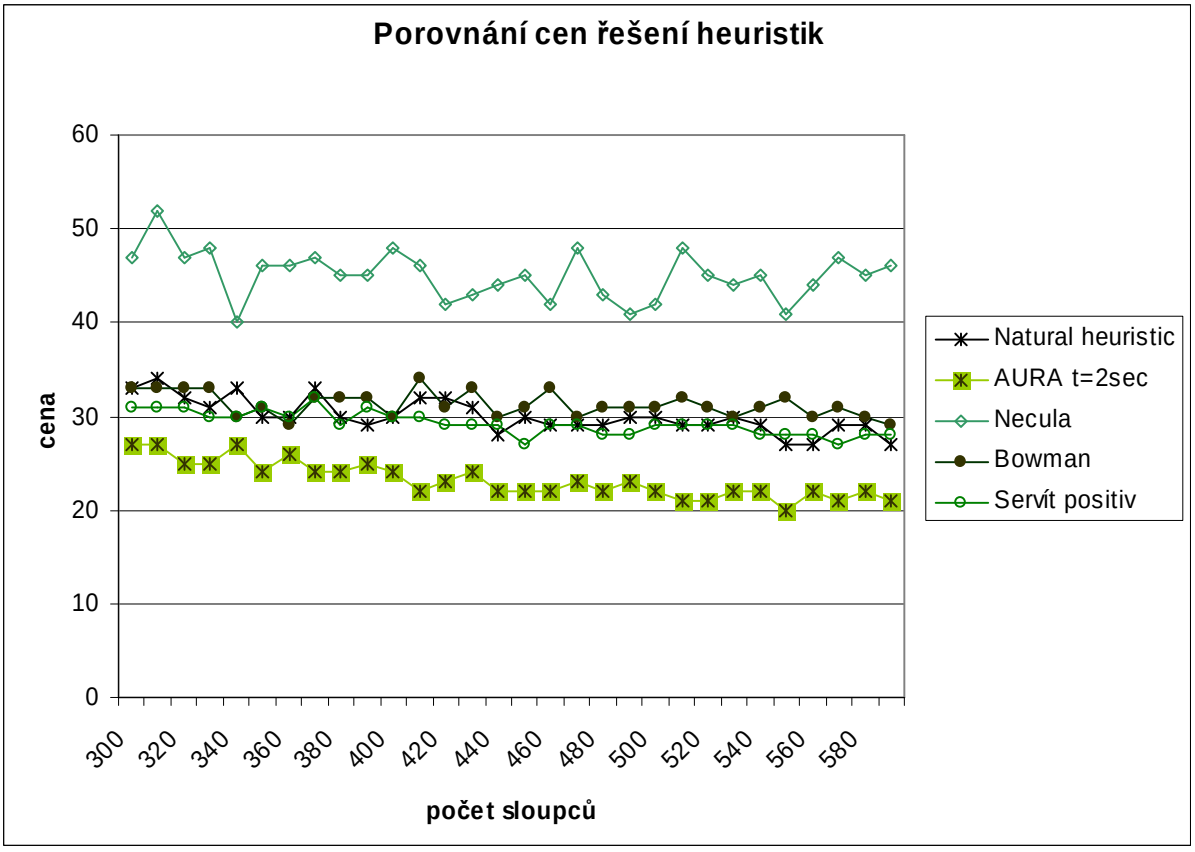
Graf 6.3.11 – Závislost času běhu heuristik na počtu sloupců matice – hustota 5%

Z grafů uvedených výše vidíme, že nejméně časově náročné jsou heuristiky Natural, ContribAdd, Roth positiv, Michalski positiv a Necula positiv. Heuristiky Roth positiv, Michalski positiv a Necula positiv zároveň vykazují i přibližně stejnou závislost výpočetního času při měnící se hustotě matic, jak je patrné z grafů 6.3.1 a 6.3.2. Bowmanova a Servítova heuristika přestává být použitelná již při počtu sloupců matice n=350, neboť jak vidíme, výpočetní čas je již příliš vysoký a nalezené řešení UCP je stejně kvalitní, jako řešení získané například heuristikou ContribAdd. Ostatní heuristiky jsou v tomto rozsahu z časového hlediska použitelné, záleží nám však ještě na kvalitě nalezeného řešení, jehož hodnoty zobrazují grafy 6.3.12, 6.3.13, 6.3.14 a 6.3.15. Jsou zde zobrazena nalezená řešení pro matice uvedené v grafech 6.3.8, 6.3.9, 6.3.10 a 6.3.11. Pro doplnění jsem zde uvedl výsledky získané algoritmem Aura, který byl spuštěn pro maximální čas $t=2$ sekundy.

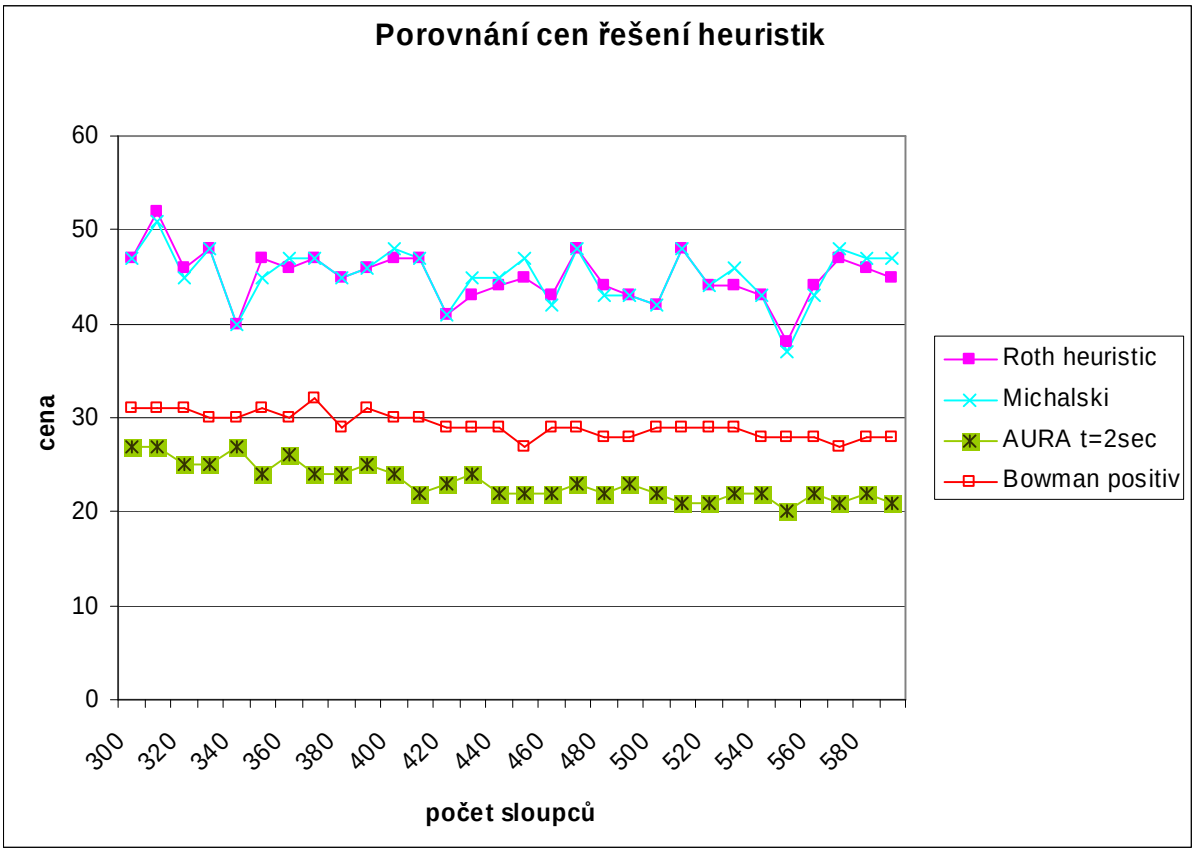
Z těchto grafů vidíme, že Servítova heuristika nalezne méně kvalitní řešení než heuristiky Servít positiv, Natural, Bowman nebo Bowman positiv, a není ani pro časovou náročnost pro tyto rozměry matice již použitelná. Heuristiky Necula, Roth a Michalski jsou pro tyto rozměry matice již na hranici použitelnosti, avšak řešení UCP získané pro tyto matice jsou rovněž velmi nekvalitní.



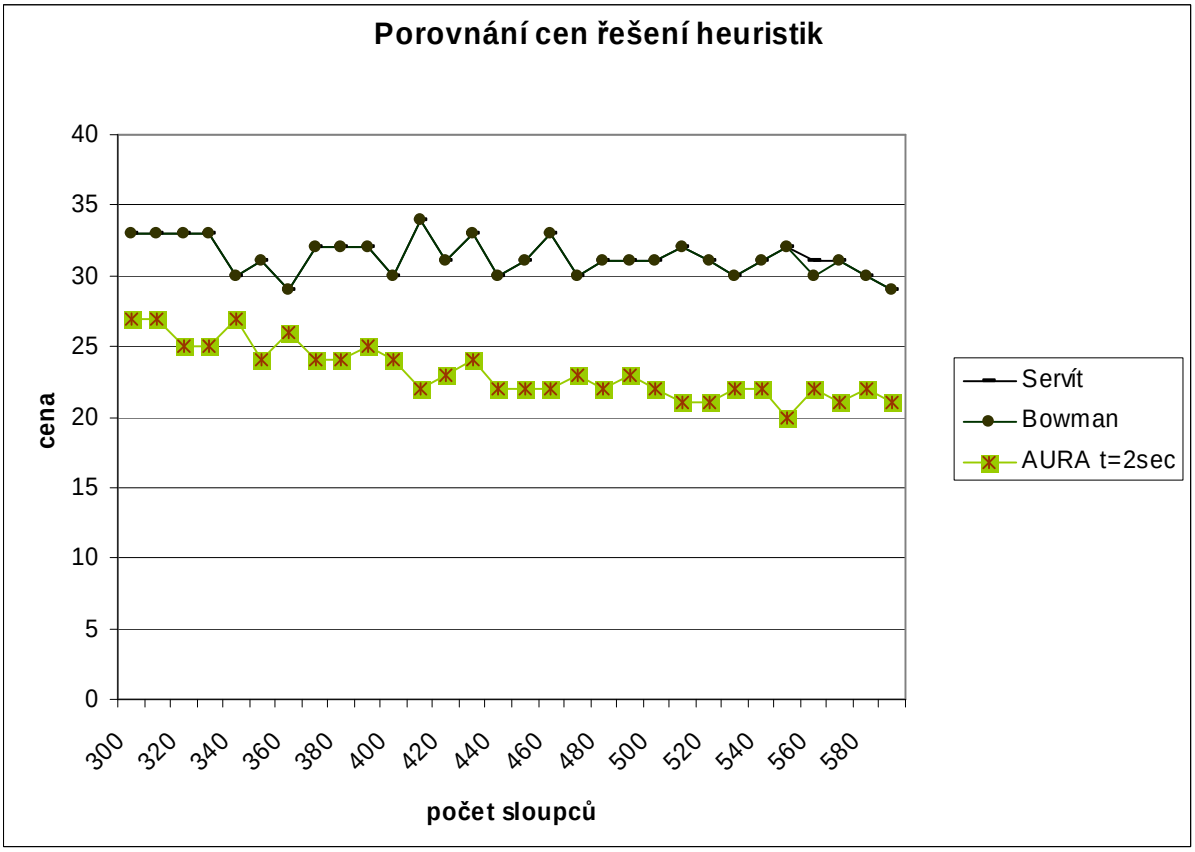
Graf 6.3.12 – Porovnání cen řešení získaných heuristikami



Graf 6.3.13 – Porovnání cen řešení získaných heuristikami

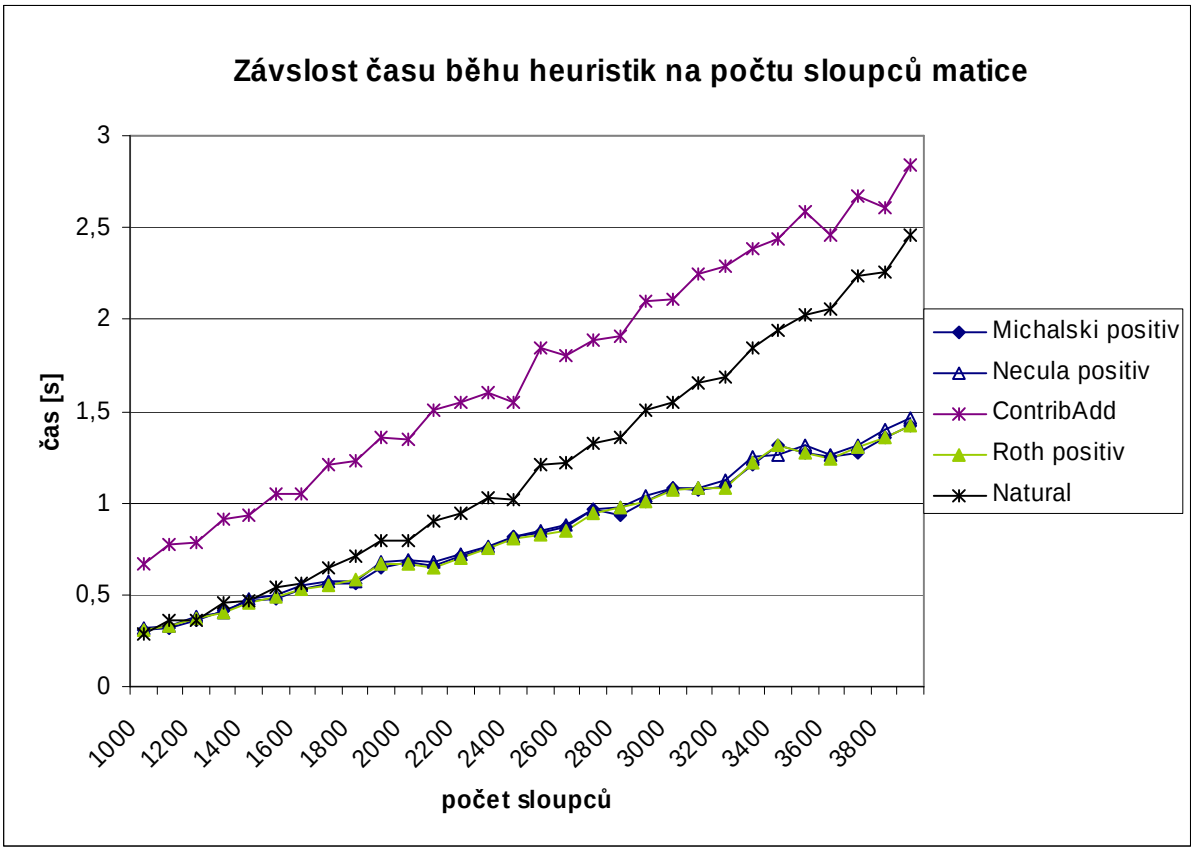


Graf 6.3.14 – Porovnání cen řešení získaných heuristikami



Graf 6.3.15 – Porovnání cen řešení získaných heuristikami

Časově nejméně náročné jsou heuristiky Necula positiv, ContribAdd, Michalski positiv, Roth positiv, Natural, Bowman positiv a Servít Positiv, které jsou v této oblasti z časového hlediska použitelné. Je nutné provést měření pro větší matice, abychom mohli získat odhad použitelnosti těchto heuristik s ohledem na výpočetní čas a kvalitu nalezeného řešení. Pro tyto účely byly vygenerovány matice s počtem řádků 1000 a počtem sloupců 1000 – 3900 s odstupem 100 sloupců, hustota 5%. Průběh času znázorňuje graf 6.3.16.



Graf 6.3.16 – Závislost času běhu heuristik na počtu sloupců matice

Jak vidíme z naměřených průběhů, přibližně stejně nízký časový průběh mají heuristiky Roth positiv, Michalski positiv a Necula positiv, kvalita řešení nalezená těmito heuristikami je však velmi nízká. Heuristika Natural a ContribAdd vykazují znatelně vyšší časové nároky avšak kvalita řešení nalezená těmito heuristikami je znatelně lepší, jak ukazuje graf 6.3.17, v mnoha případech je cena nalezeného řešení stejná u obou heuristik. V případech, kdy jsou nalezená řešení přibližně stejně minimální a časové závislosti jsou také velmi podobné, je dobré provést výběr heuristiky ještě na základě rozptylu hodnot nalezených řešení pro více použití jedné heuristiky na stejnou matici. V tabulce 6.3.2 a 6.3.3 jsou znázorněny rozptyly hodnot nalezených řešení, pro měření byly použity reálné matice získané programem BOOM, heuristiky byly použity v počtu 10 000 opakování.

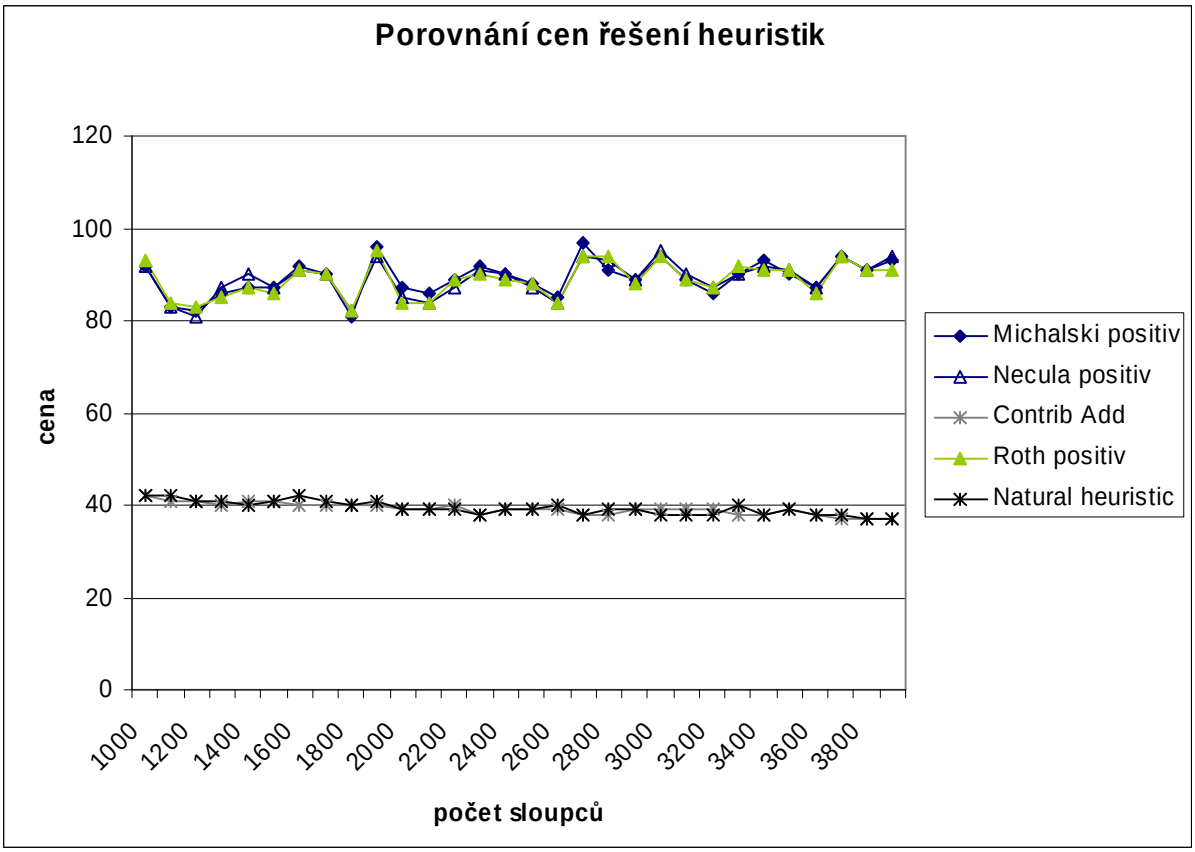
Počet sloupců	Počet řádků	Natural	Contrib Add	Roth	Roth positiv	Michalski	Michalski positiv
38	77	0,0	0,0	57,1	100,0	42,9	85,7
54	62	0,0	0,0	26,7	29,4	18,8	22,2
62	62	20,0	0,0	18,2	50,0	18,2	50,0
77	42	25,0	0,0	20,0	70,0	20,0	70,0
80	94	10,7	0,0	20,0	18,8	17,1	18,8
85	82	20,0	0,0	16,7	58,3	16,7	58,3
91	90	13,6	0,0	22,7	33,3	22,7	29,6
100	80	33,3	0,0	19,0	27,8	19,0	27,8
104	90	12,5	0,0	36,4	56,3	36,4	41,2
157	102	18,5	0,0	4,8	20,0	7,3	23,5
173	98	15,4	1,9	11,4	13,2	9,9	16,0

Tabulka 6.3.2 –Rozptyl heuristik [%]

Počet sloupců	Počet řádků	Necula	Necula positiv	Bowman	Bowman positiv	Servit	Servit positiv
38	77	57,1	100,0	0,0	0,0	0,0	0,0
54	62	26,7	22,2	0,0	0,0	0,0	0,0
62	62	18,2	50,0	0,0	0,0	0,0	0,0
77	42	20,0	70,0	0,0	0,0	0,0	0,0
80	94	20,0	18,8	3,8	0,0	3,8	0,0
85	82	16,7	66,7	0,0	0,0	0,0	0,0
91	90	22,7	33,3	0,0	0,0	0,0	0,0
100	80	25,0	27,8	0,0	0,0	0,0	0,0
104	90	36,4	56,3	0,0	0,0	0,0	0,0
157	102	7,3	20,0	3,8	0,0	3,8	0,0
173	98	9,9	14,5	8,0	1,9	8,0	1,9

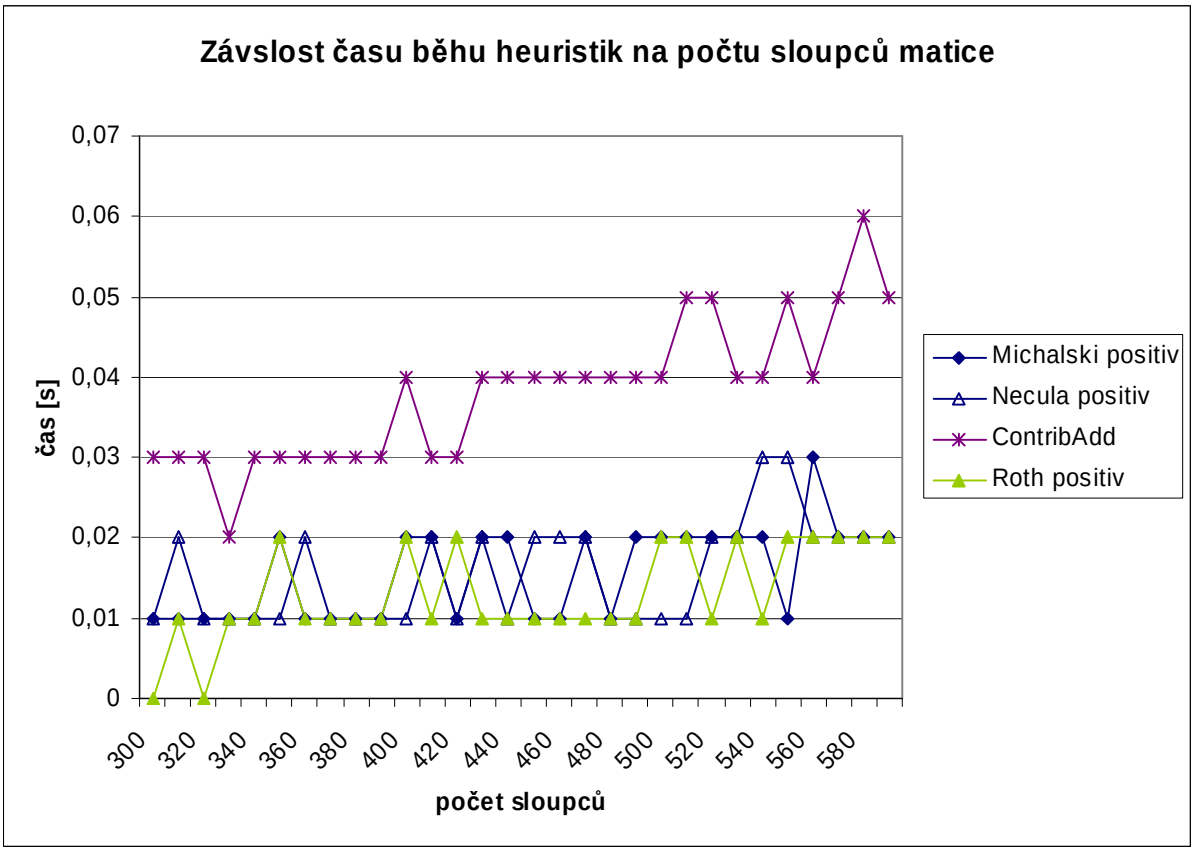
Tabulka 6.3.3 – Rozptyl heuristik [%]

Jak je vidět z naměřených dat v tabulkách uvedených výše, tak heuristika ContribAdd vykazuje minimální rozptyl, narozdíl od heuristiky Natural, která dosahuje rozptylu až 33%. Heuristiky pro nalezení řešení UCP v programu BOOM jsou však spouštěny buď po každé iteraci, nebo máme možnost nalézt řešení UCP na konci minimalizace. S ohledem na rozptyl a časovou závislost heuristik je dobré volit v prvním případě heuristiku Natural a v druhém případě, při hledání řešení UCP po poslední iteraci, heuristiku ContribAdd.

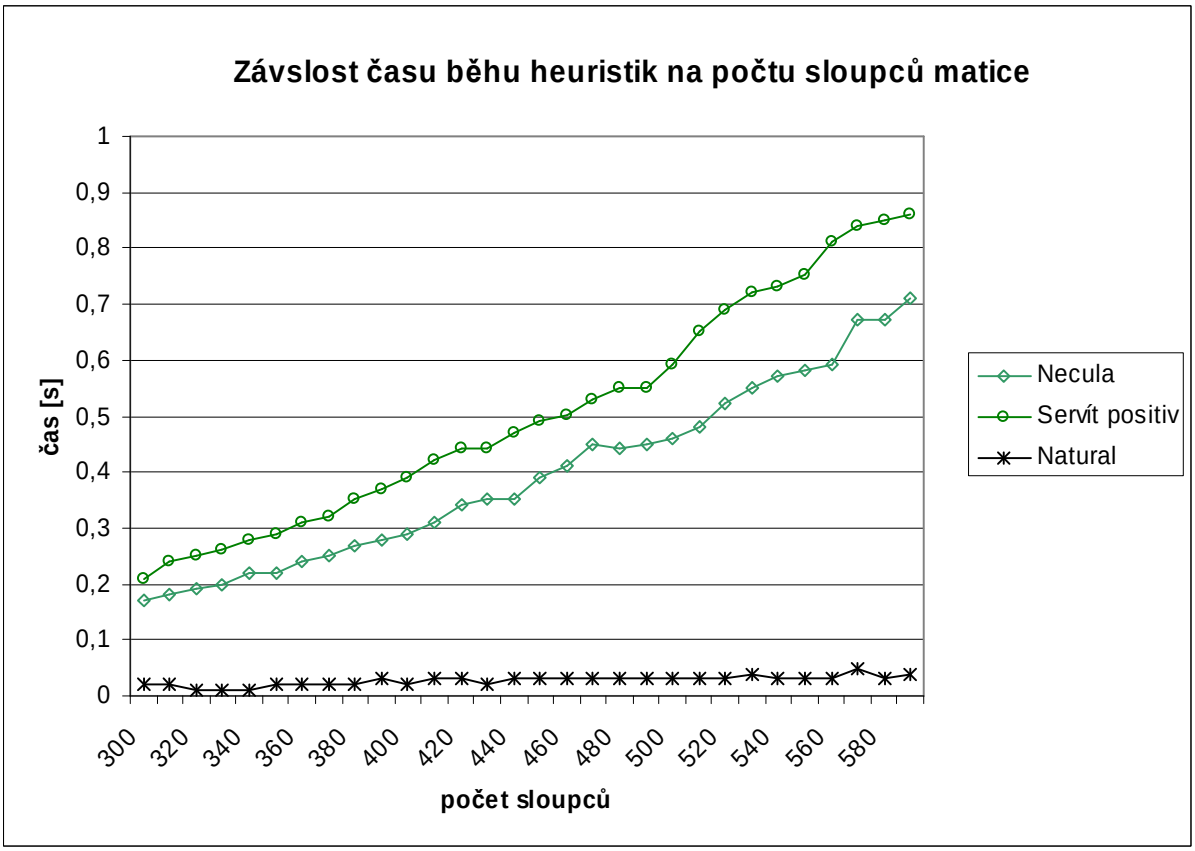


Graf 6.3.17 – Porovnání cen řešení získaných heuristikami

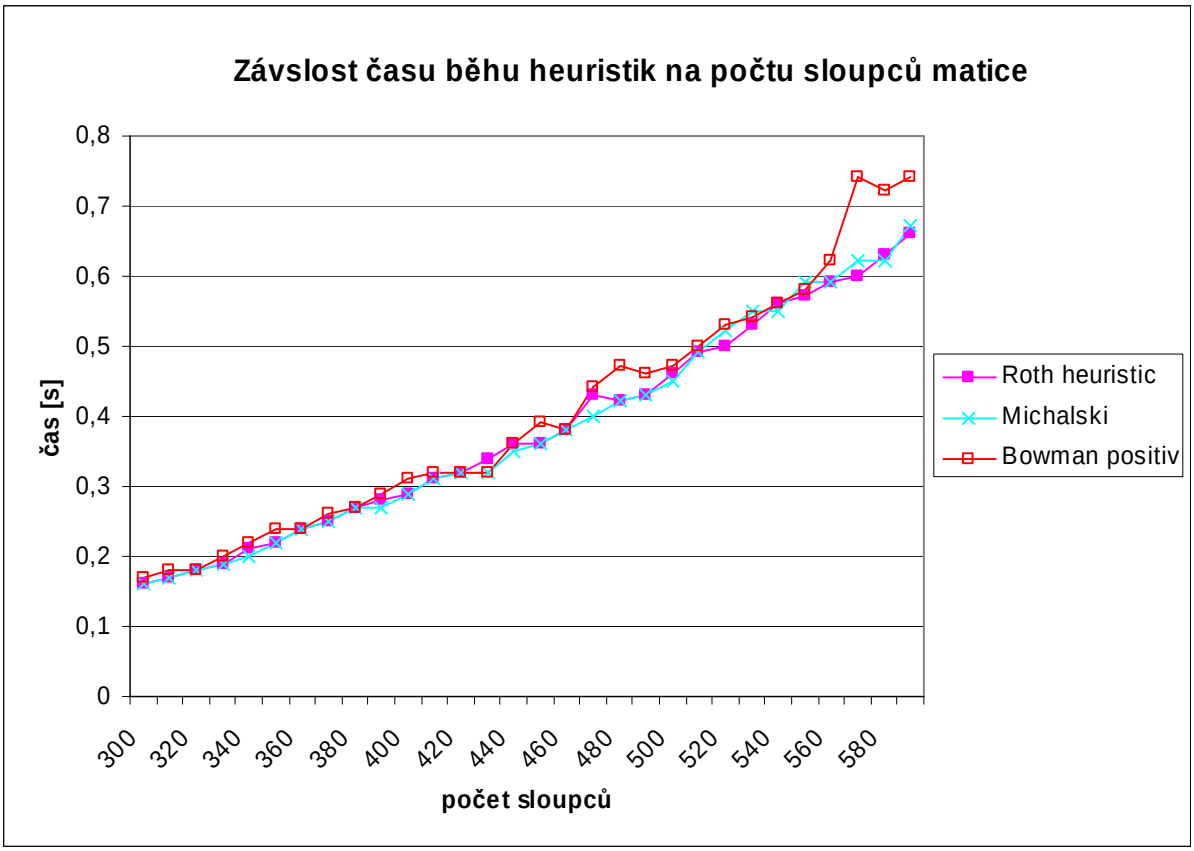
Dále jelikož hustota reálných matic se pohybuje ojediněle i k hranici 15%, ještě uvádím výsledky naměřené pro stejné parametry matic a hustotu 15%. V grafech 6.3.18 – 6.3.21 jsou uvedeny závislosti výpočetního času heuristik na počtu sloupců pro konstantní počet řádků m=300 a hustotu 15%, dále v grafech 6.3.22 – 6.3.25 je porovnání výše cen řešení nalezených těmito heuristikami. Je vidět, že u heuristik Bowman, Servít, Servít positiv, Roth, Michalski a Bowman positiv je zřetelně vidět nárůst výpočetního času, což se dá očekávat podle závislostí uvedených v grafech 6.3.1 – 6.3.3. Použitelnost heuristik však zůstává stejná, neboť heuristiky Natural a ContribAdd pro tuto hustotu matice mají naopak nižší hodnotu výpočetního času. Dále v grafu 6.3.26 je porovnání závislosti výpočetního času heuristik pro počet řádků matice m=1000, hustotu 15% a počet sloupců 1000 – 3900. Z naměřených závislostí vyplývá, že heuristika ContribAdd má od počtu sloupců 3000 při této hustotě nižší časové nároky, než heuristika Natural, neboť jak napovídá průběh v grafech 6.3.1 a 6.3.2, křivka časové závislosti heuristiky ContribAdd na hustotě matice klesá rychleji než u heuristiky Natural.



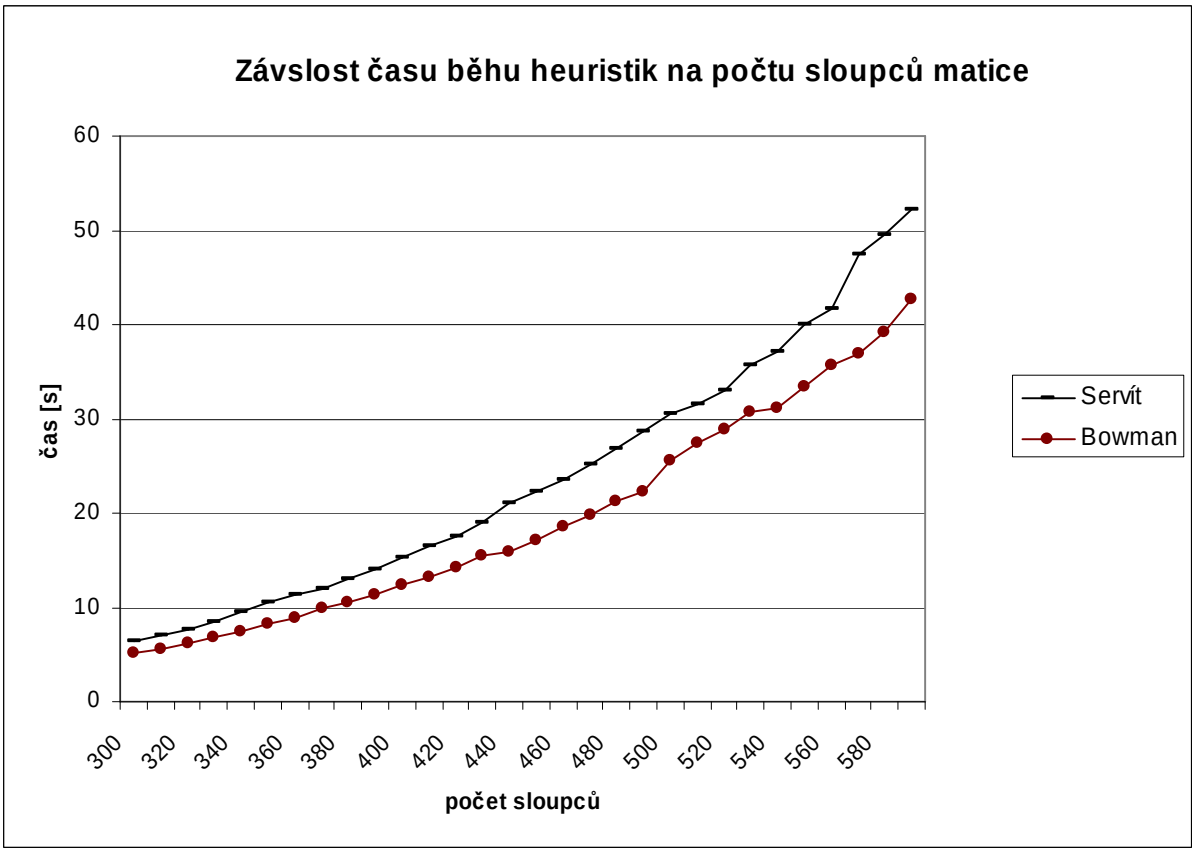
Graf 6.3.18 – Závislost času běhu heuristik na počtu sloupců matice – hustota 15%



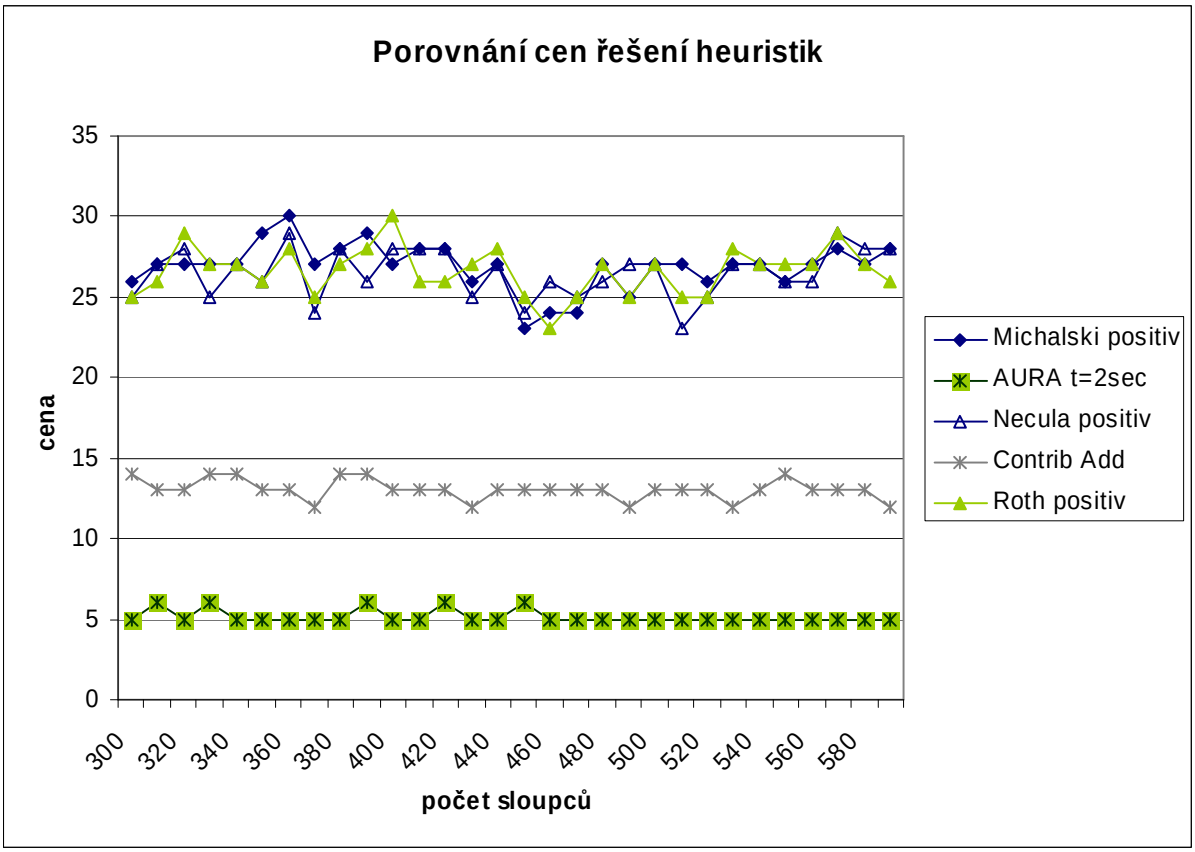
Graf 6.3.19 – Závislost času běhu heuristik na počtu sloupců matice – hustota 15%



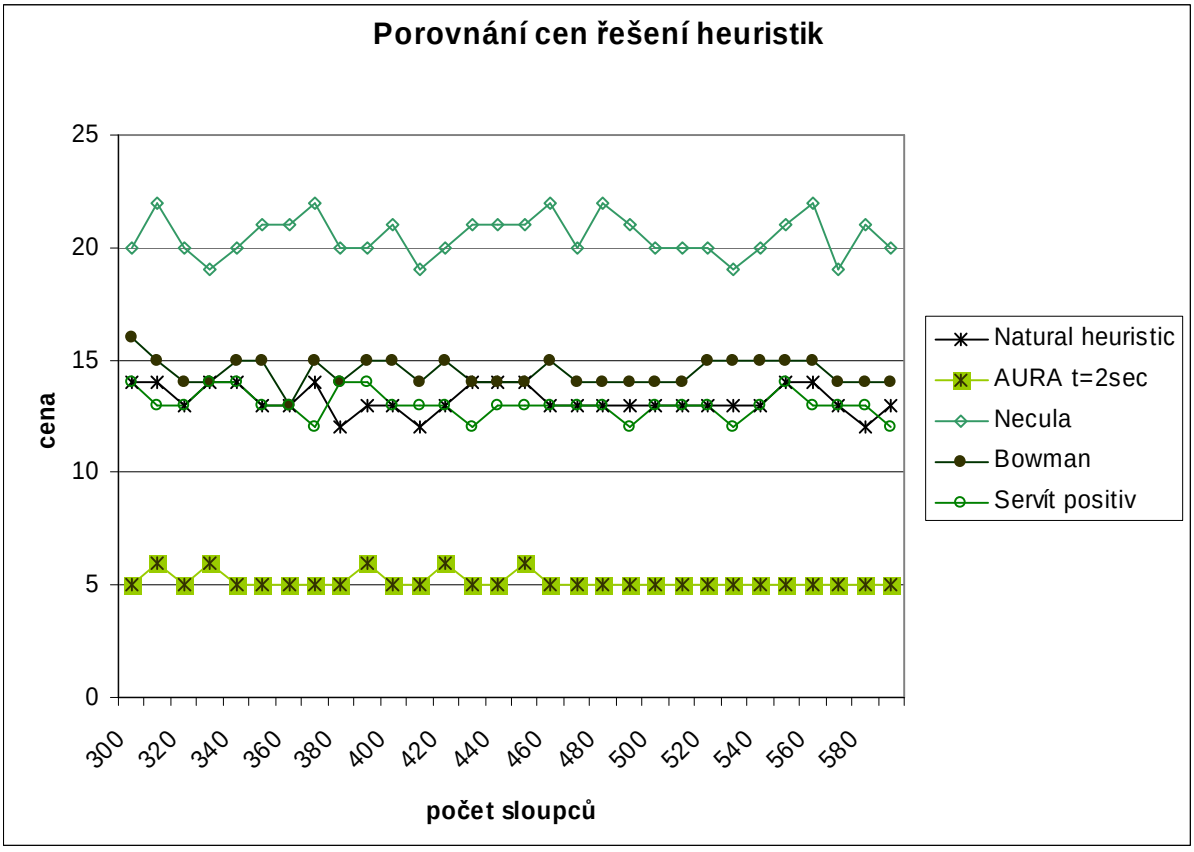
Graf 6.3.20 – Závislost času běhu heuristik na počtu sloupců matice – hustota 15%



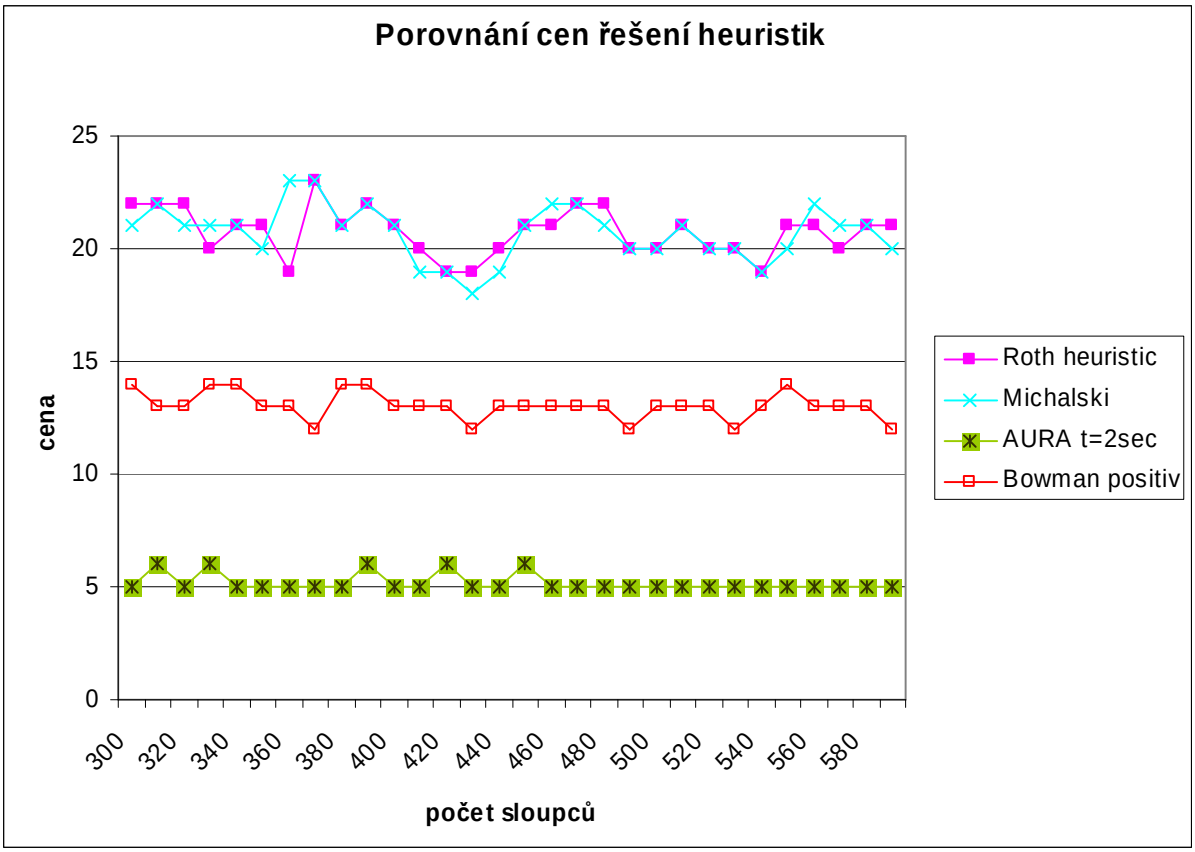
Graf 6.3.21 – Závislost času běhu heuristik na počtu sloupců matice – hustota 15%



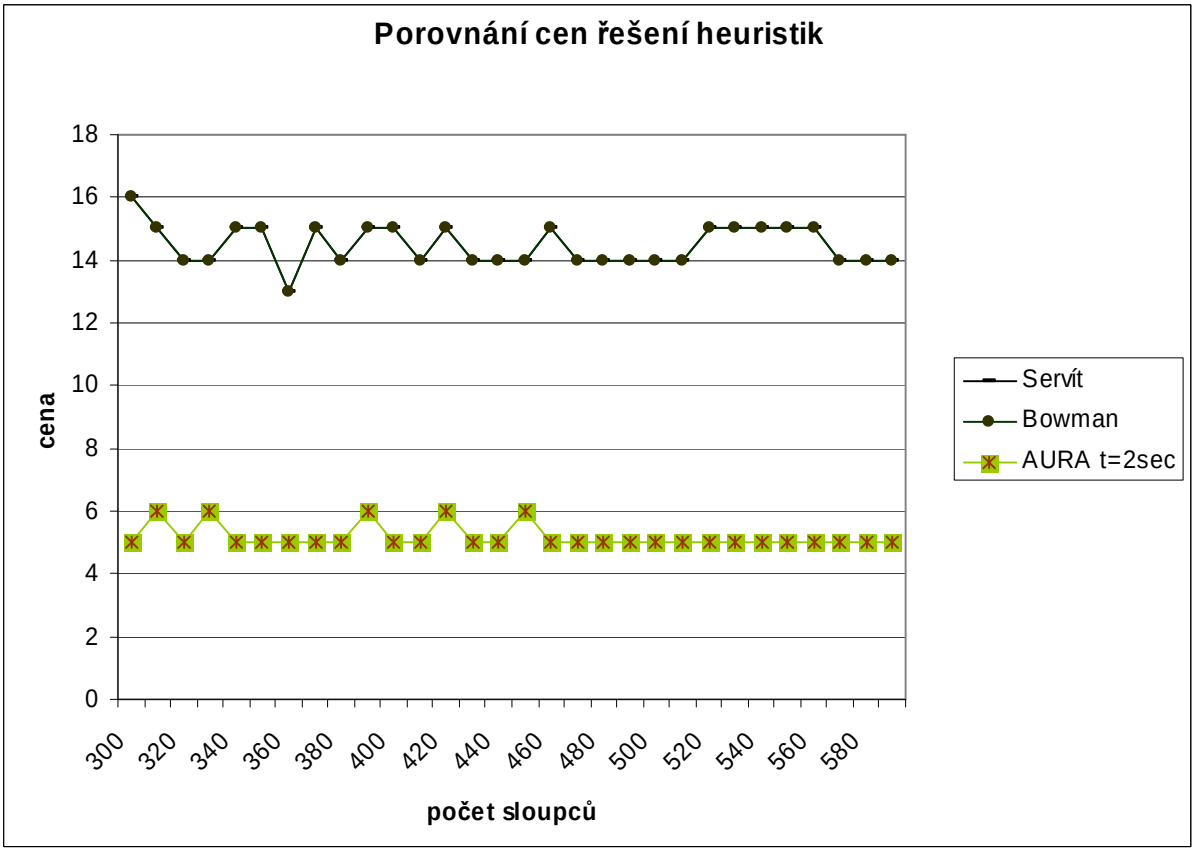
Graf 6.3.22 – Porovnání cen řešení získaných heuristikami



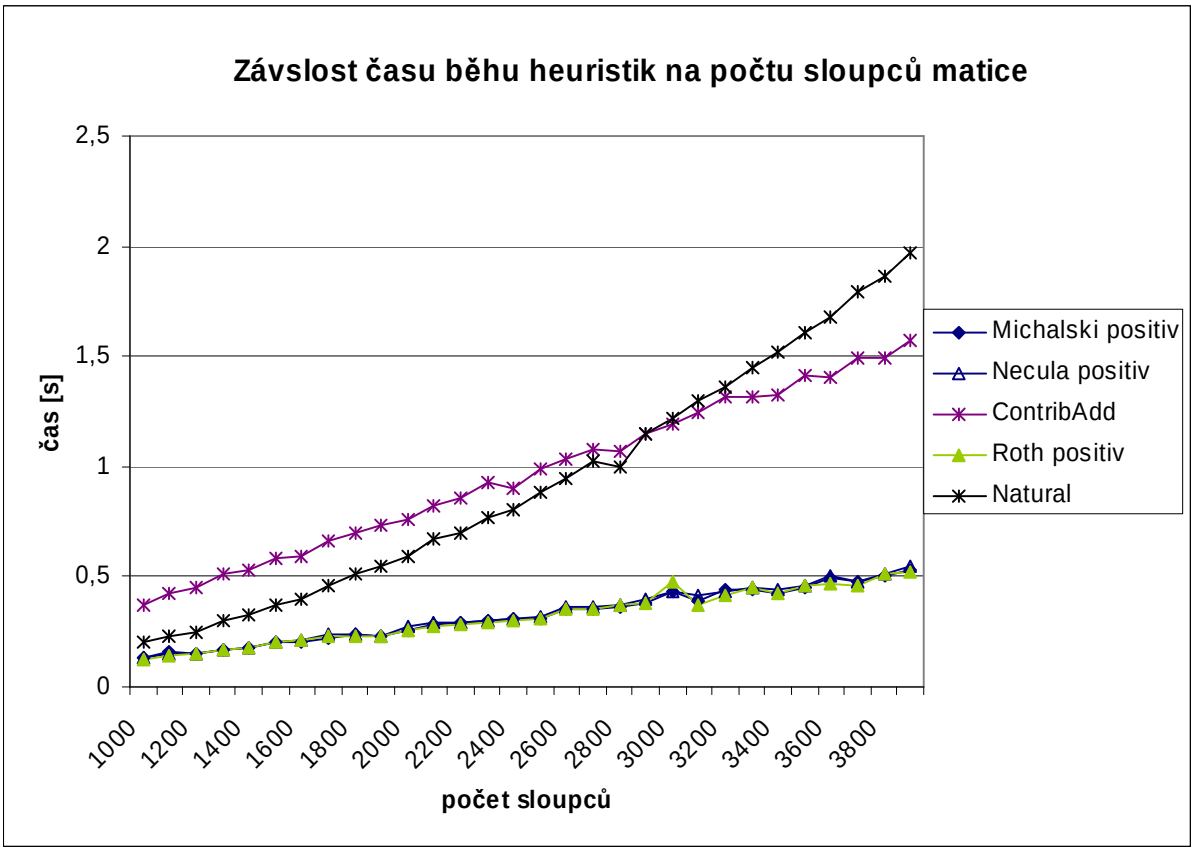
Graf 6.3.23 – Porovnání cen řešení získaných heuristikami



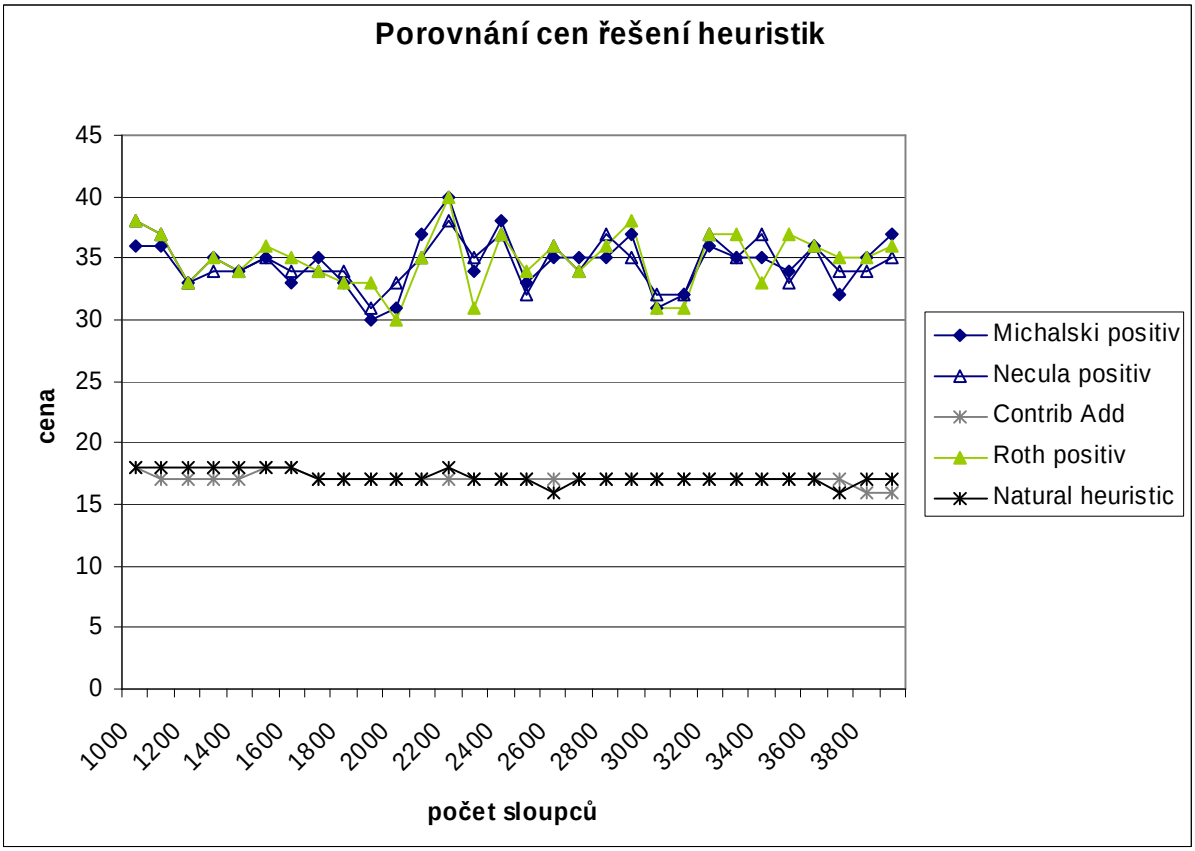
Graf 6.3.24 – Porovnání cen řešení získaných heuristikami



Graf 6.3.25 – Porovnání cen řešení získaných heuristikami



Graf 6.3.26 – Závislost času běhu heuristik na počtu sloupců matice – hustota 15%

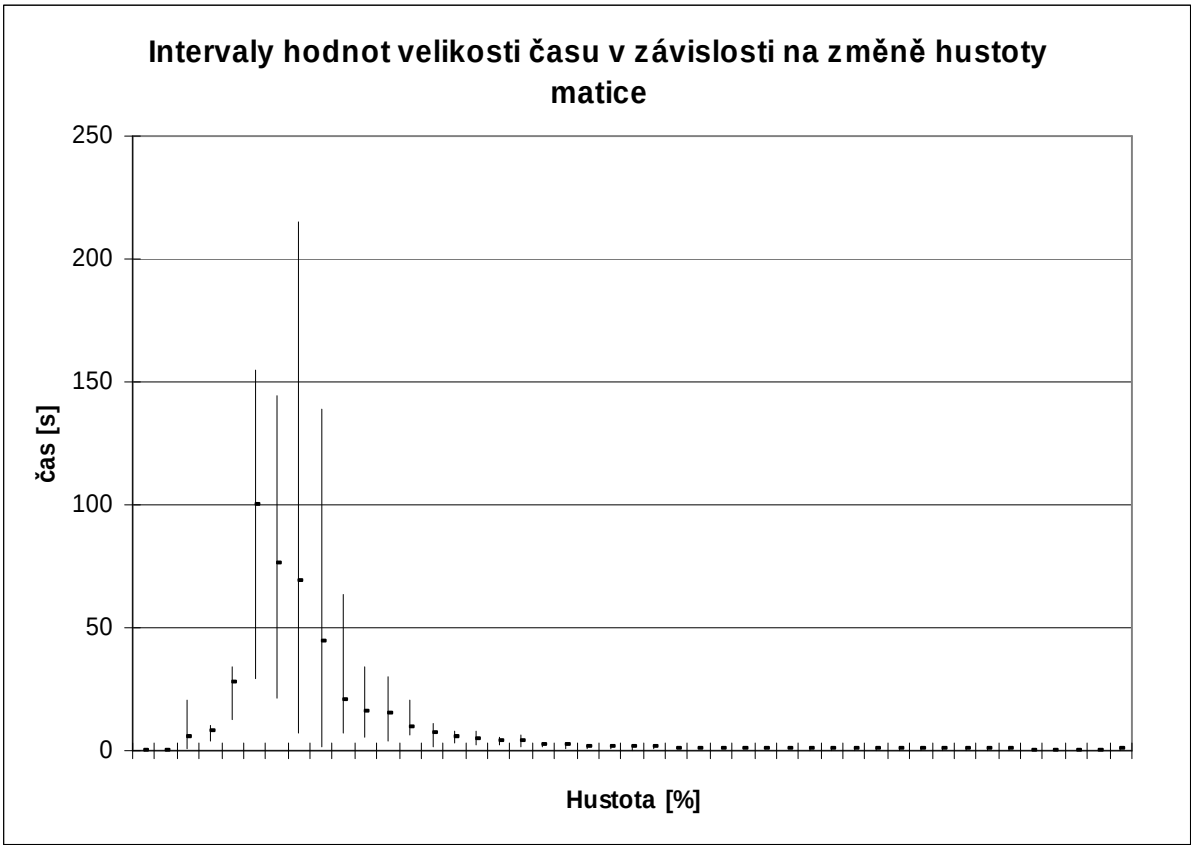


Graf 6.3.27 – Porovnání cen řešení získaných heuristikami

6.4 Měření výpočetního času Aury – nejednotkové ceny sloupců

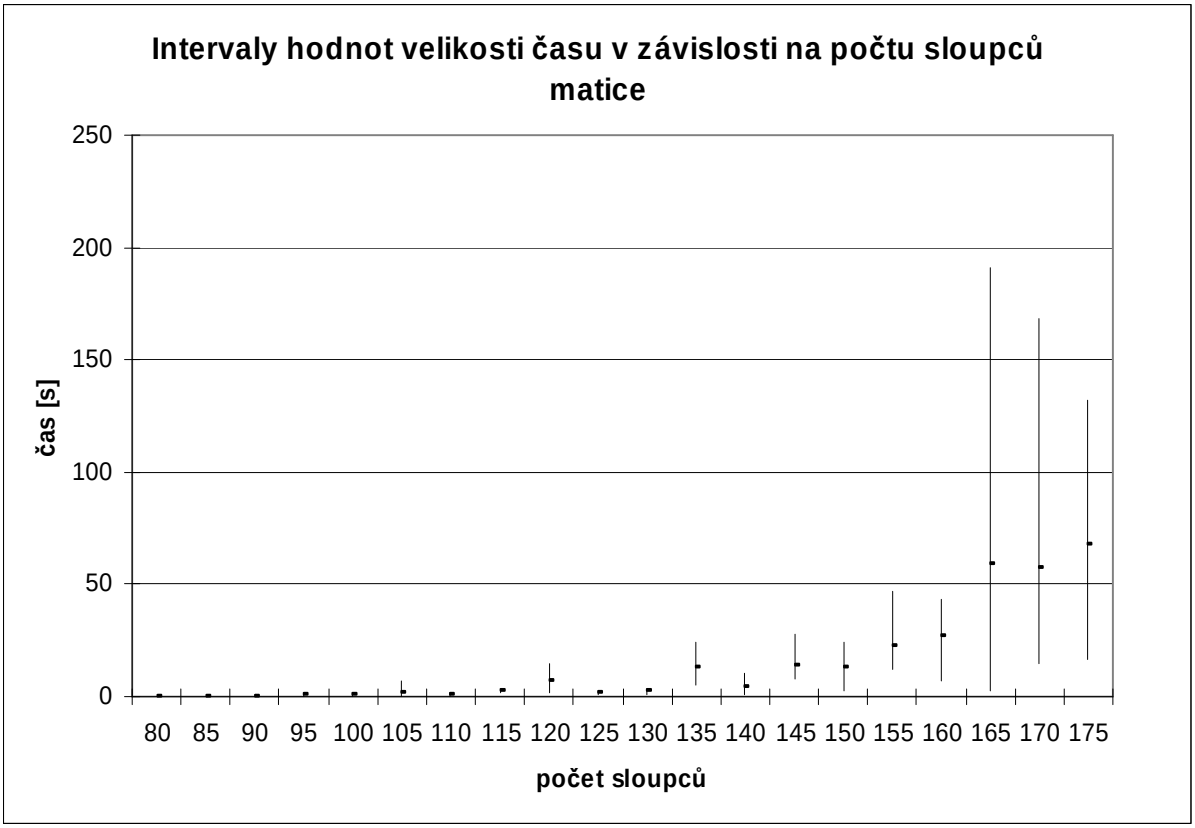
Ne vždycky je situace taková, že ceny všech sloupců v matici jsou rovny jedné. Je tedy ještě nutné otestovat algoritmy pro ceny sloupců různé od jedné a pro tuto situaci opět zvolit nejvýhodnější algoritmus pro nalezení řešení UCP. Generované matice pro stejné parametry jako v kapitole 6.2 vykazovaly příliš vysoký čas potřebný pro nalezení řešení UCP, a naměřené hodnoty byly tak nepoužitelné pro náš odhad. Byly proto vygenerovány matice s menším počtem řádků a sloupců, ceny sloupců byly generované v rozmezí 1–9. Pro nejednotkové ceny je očekáván nárůst výpočetního času, neboť se zmenší počet dominantních sloupců. Pokud má totiž dominující sloupec cenu vyšší než sloupec, kterému dominuje, nemůže být tento sloupec odstraněn z matice. To má za následek i menší pravděpodobnost vzniku dominantních řádků.

V grafu 6.4.1 jsou pro úplnost znázorněné intervaly maximálních a minimálních časů běhu Aury pro několik generovaných matic o stejných rozměrech, konkrétně 100 řádků, 100 sloupců a hustota v rozsahu 1 % – 89 %. Vidíme že při nejednotkových cenách sloupců se dá očekávat podobný průběh časové závislosti jako pro průběh při jednotkových cenách sloupců, který je znázorněn v kapitole 6.2.

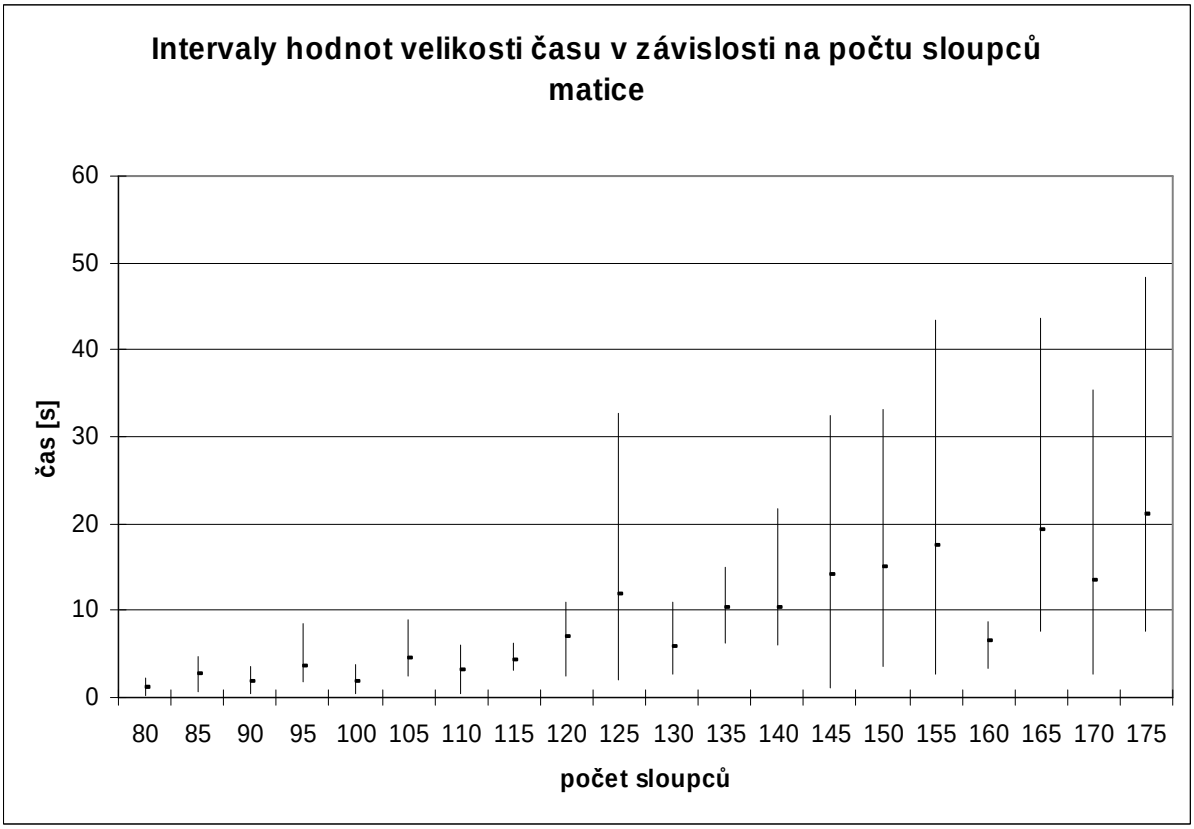


Graf 6.4.1 – Závislost času běhu Aury na změně hustoty matice

Dále se budeme spíše věnovat měření matic o takových rozměrech, z kterých bude možno vyčíst krajní meze použitelnosti Aury pro nejednotkové ceny sloupců. Je třeba zjistit, pro jaký počet sloupců matice je ještě možné pro nalezení řešení UCP použít Auru. Graf 6.4.2 znázorňuje intervaly maxima a minima časů běhu Aury pro několik generovaných matic o rozměrech 80 řádků a 80 – 170 sloupců a hustotě 5 %. Graf 6.4.3 Zobrazuje údaje naměřené pro stejný rozměr matice a hustotu 15 %.



Graf 6.4.2 – Závislost času běhu Aury na počtu sloupců matice – hustota 5%

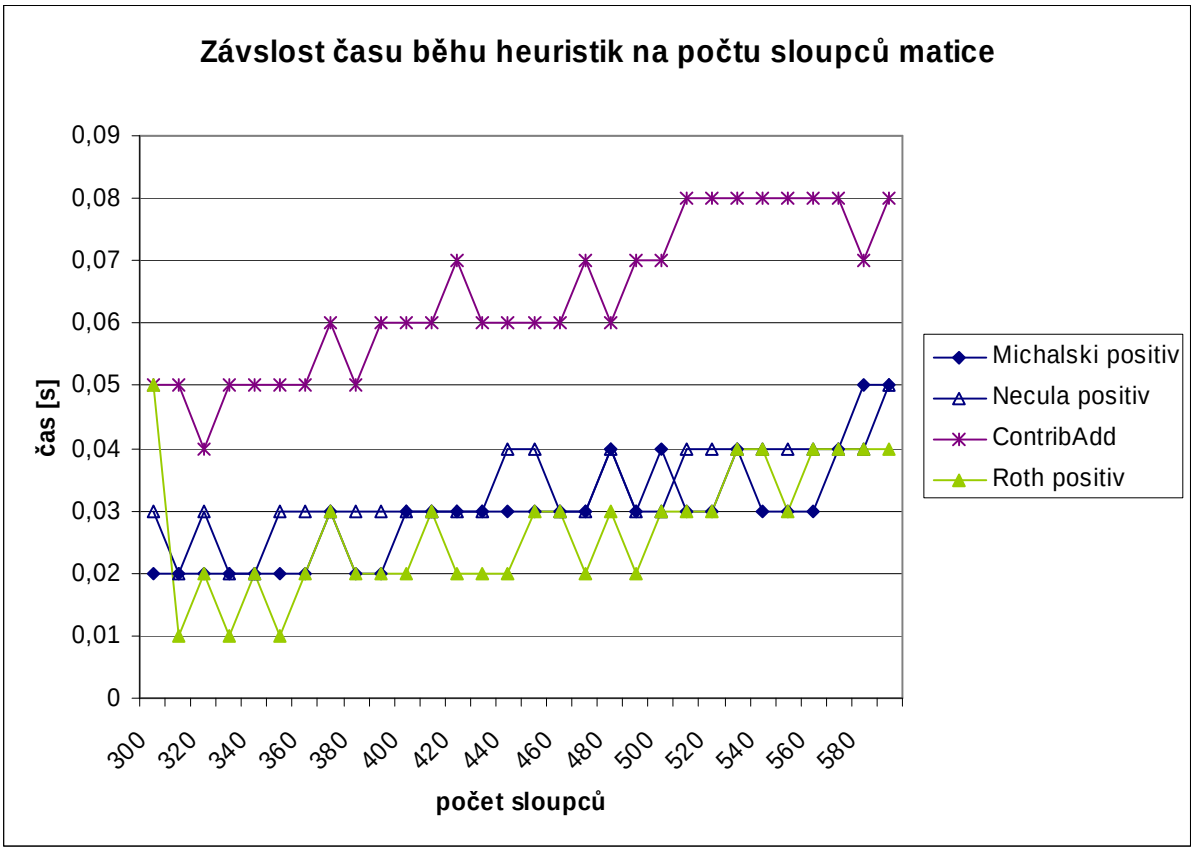


Graf 6.4.3 – Závislost času běhu Aury na počtu sloupců matice - hustota 15%

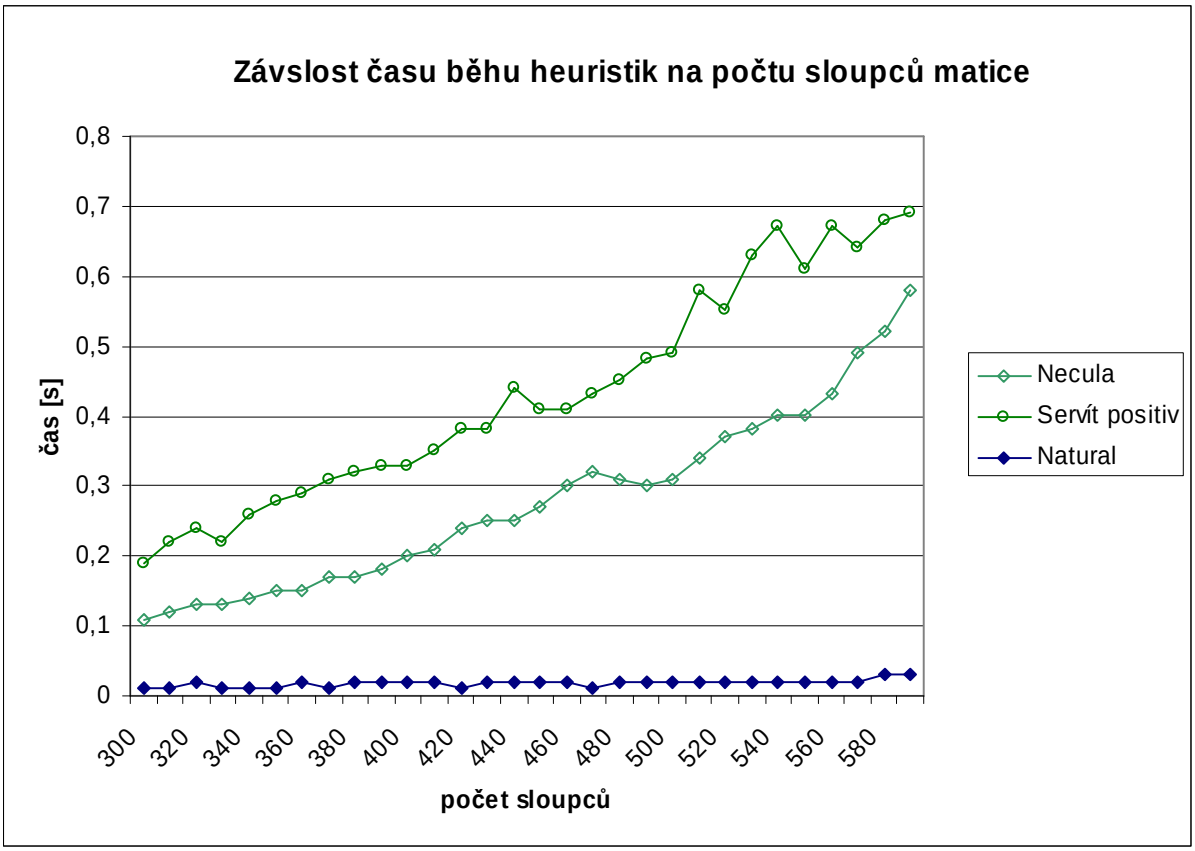
Z naměřených průběhů vyplývá, že pokud jsou ceny sloupců nejednotkové, lze očekávat nalezení řešení UCP pomocí Aury v přijatelném čase jen do rozměrů matice přibližně 150 řádků a 150 sloupců při hustotě matice 5% a při hustotě 15% je použitelnost Aury pro matice do rozměru 120 řádků a 120 sloupců.

6.4 Měření výpočetního času heuristik – nejednotkové ceny sloupců

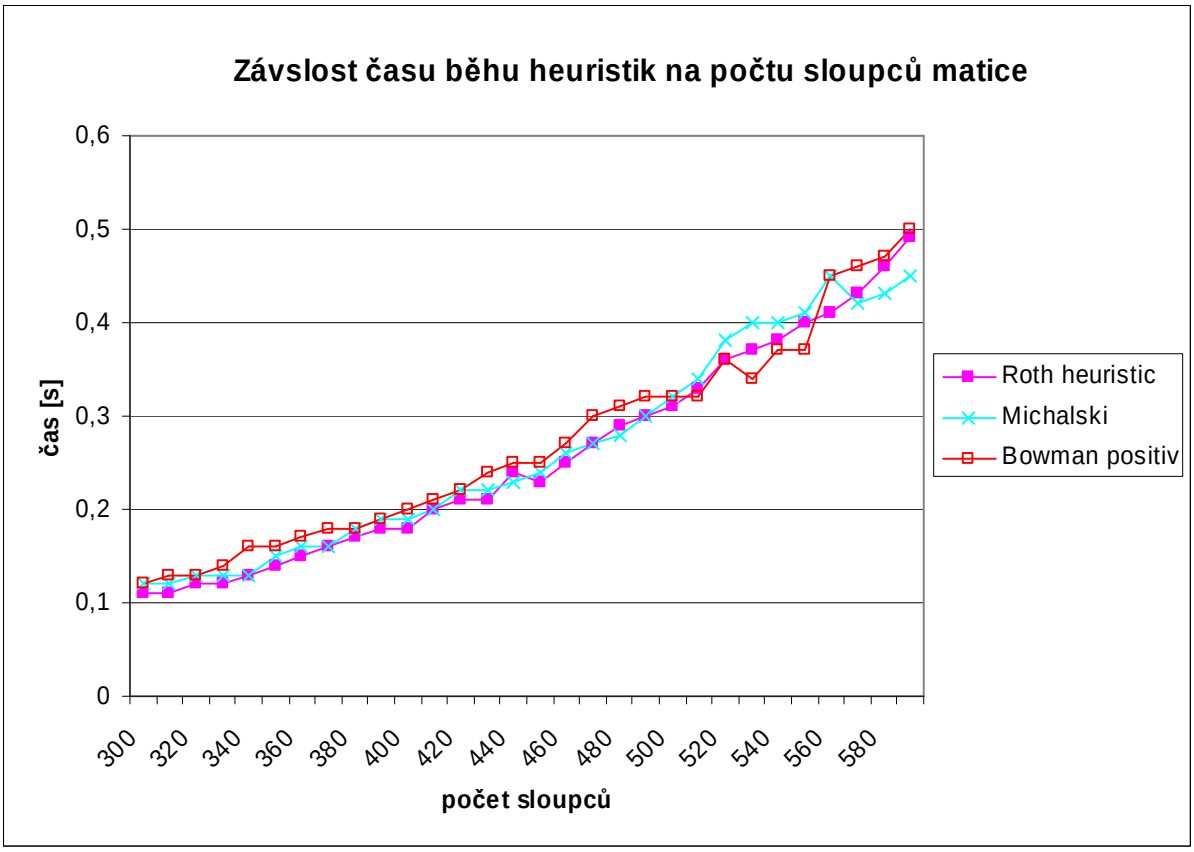
Měření výpočetního času pro nejednotkové ceny sloupců matice bylo provedeno na maticích měřených v kapitole 6.2, pouze byly uvažovány ceny sloupců v rozsahu 1 – 9.



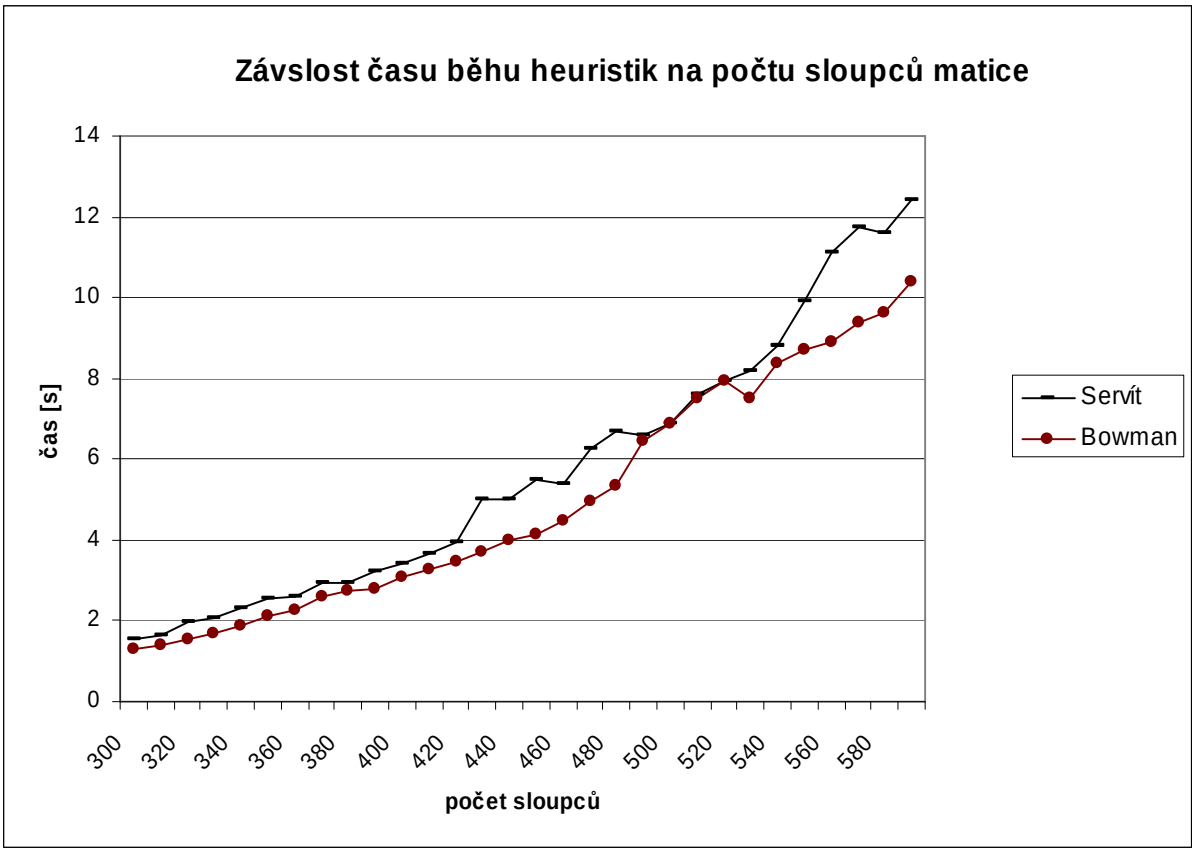
Graf 6.4.1 – Závislost času běhu heuristik na počtu sloupců matice – hustota 5%



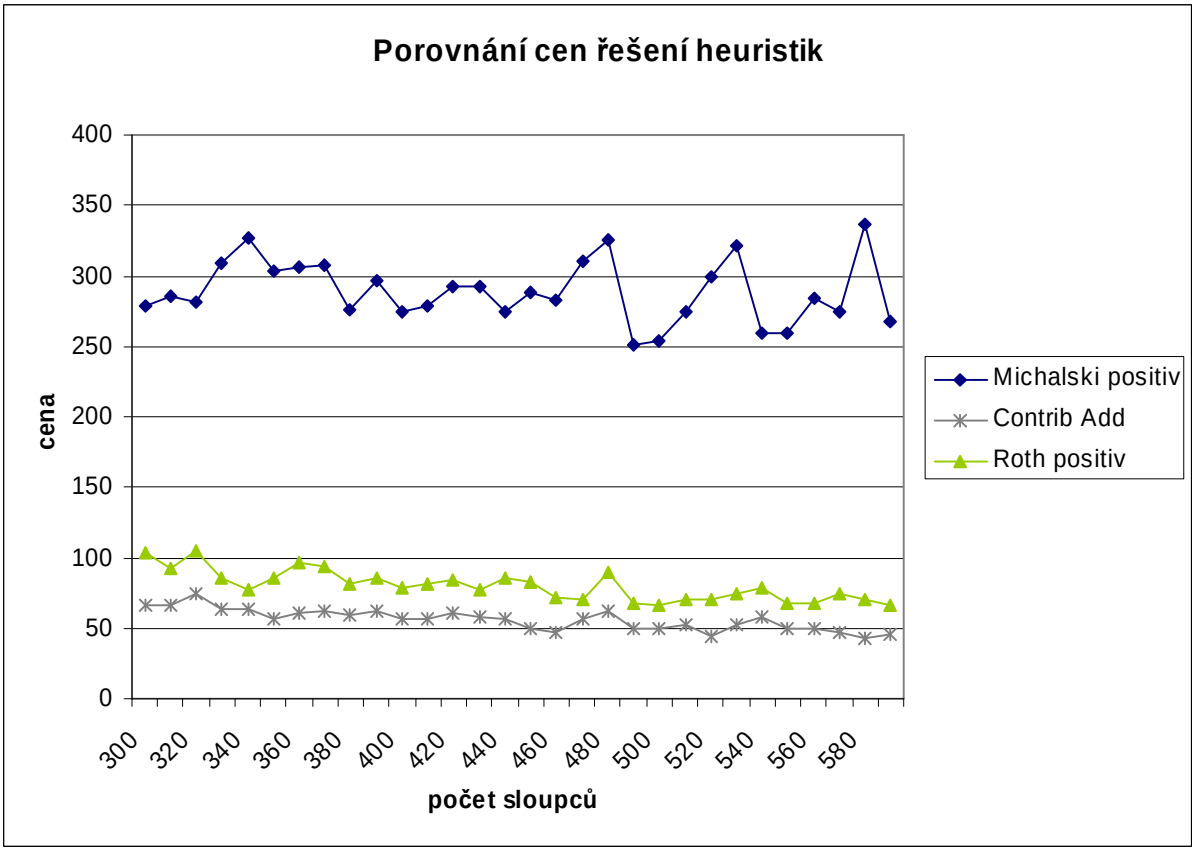
Graf 6.4.2 – Závislost času běhu heuristik na počtu sloupců matice – hustota 5%



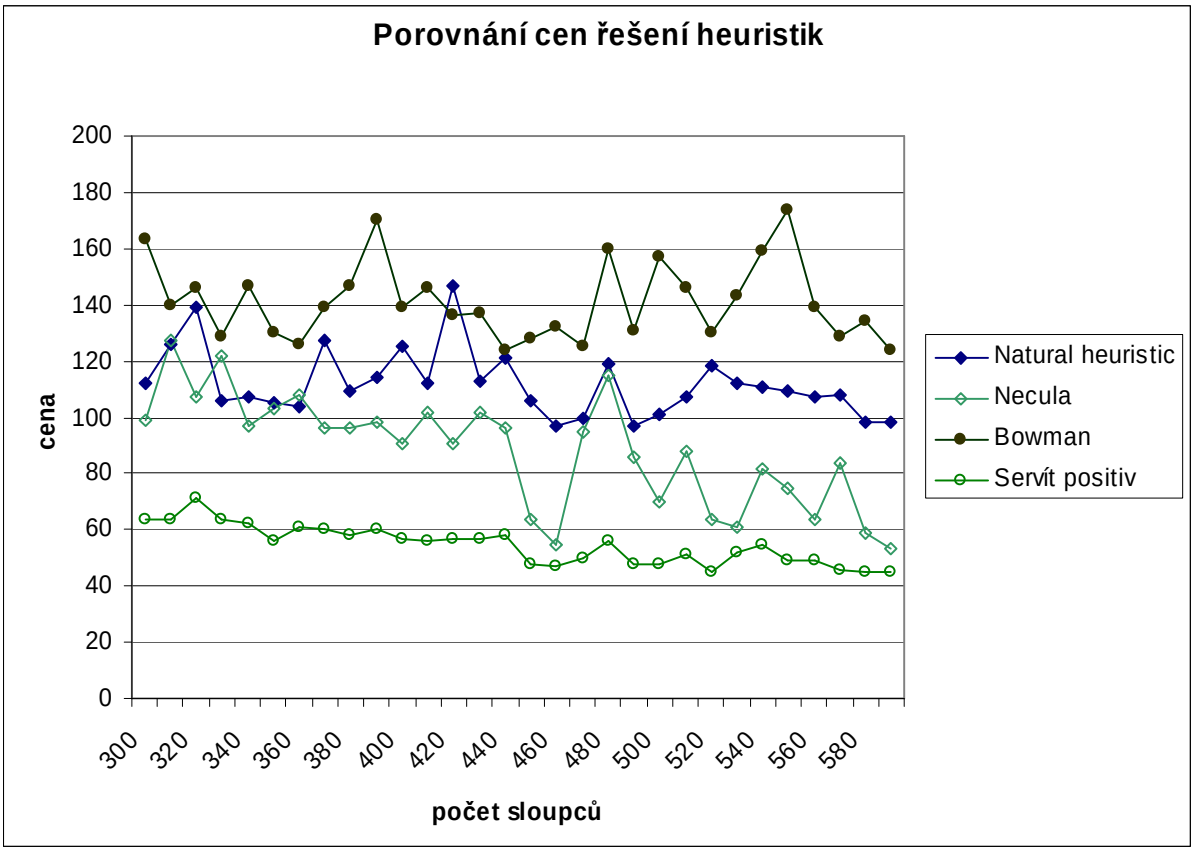
Graf 6.4.3 – Závislost času běhu heuristik na počtu sloupců matice – hustota 5%



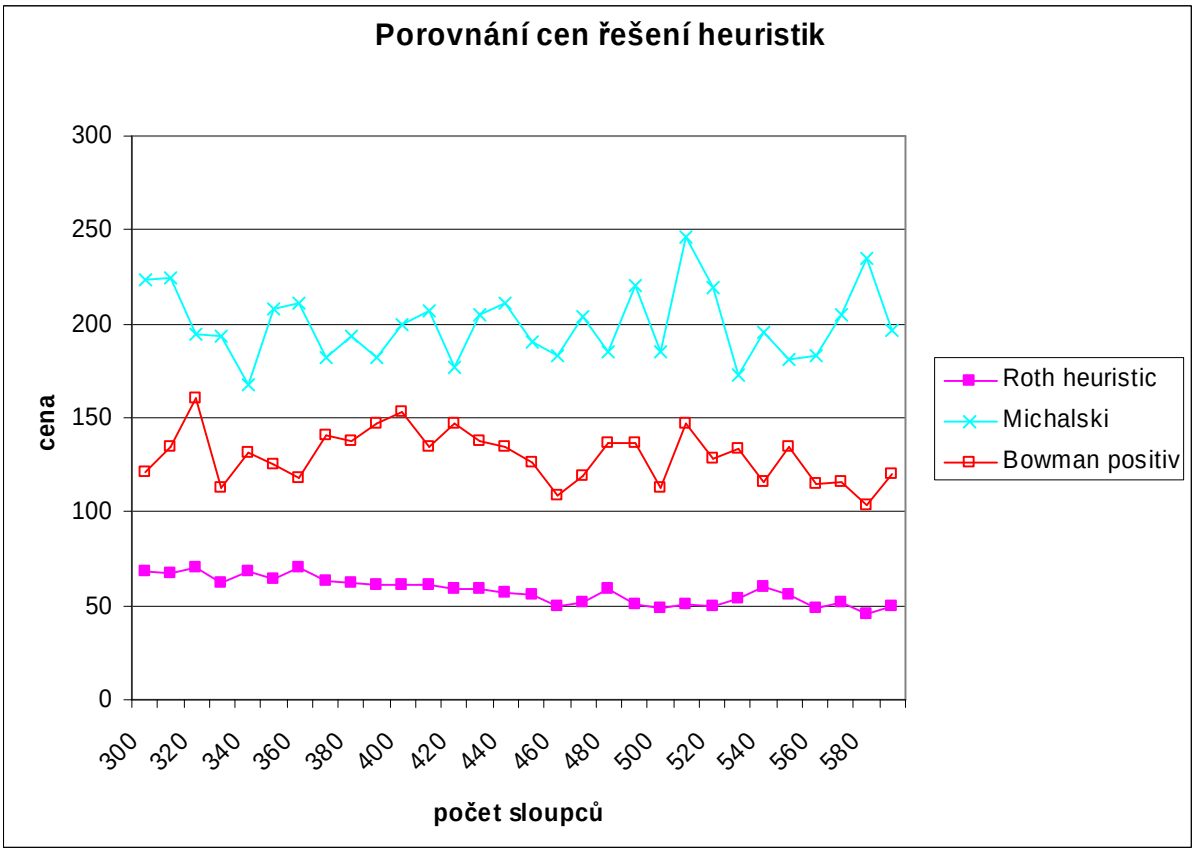
Graf 6.4.4 – Závislost času běhu heuristik na počtu sloupců matice – hustota 5%



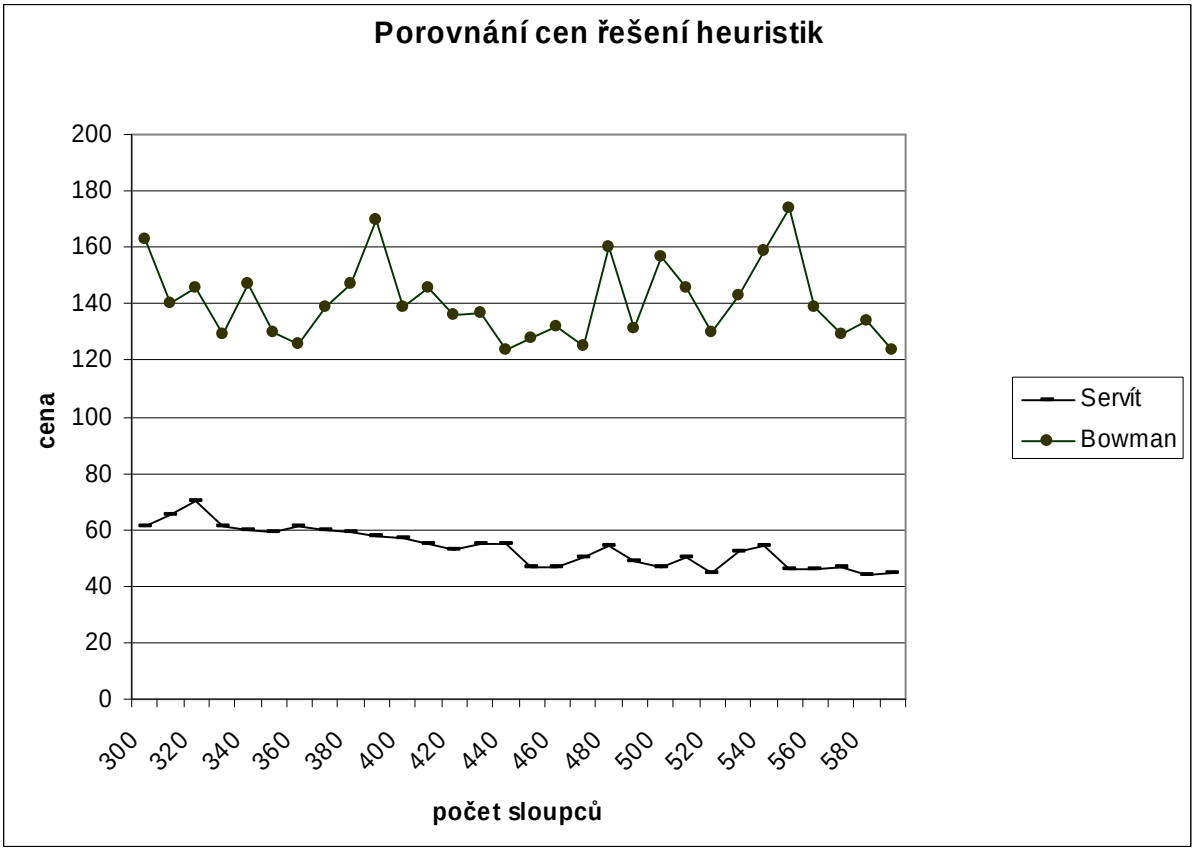
Graf 6.4.5 – Porovnání cen řešení získaných heuristikami



Graf 6.4.5 – Porovnání cen řešení získaných heuristikami

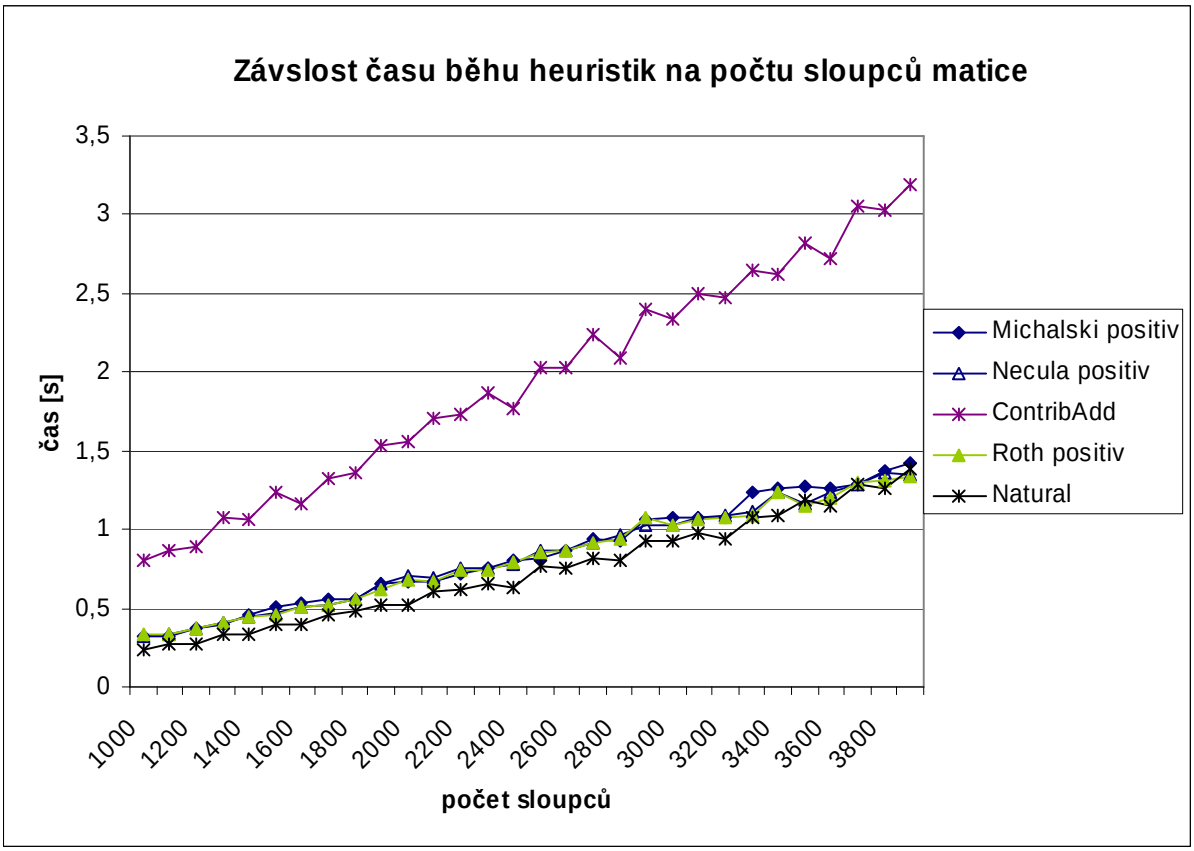


Graf 6.4.6 – Porovnání cen řešení získaných heuristikami

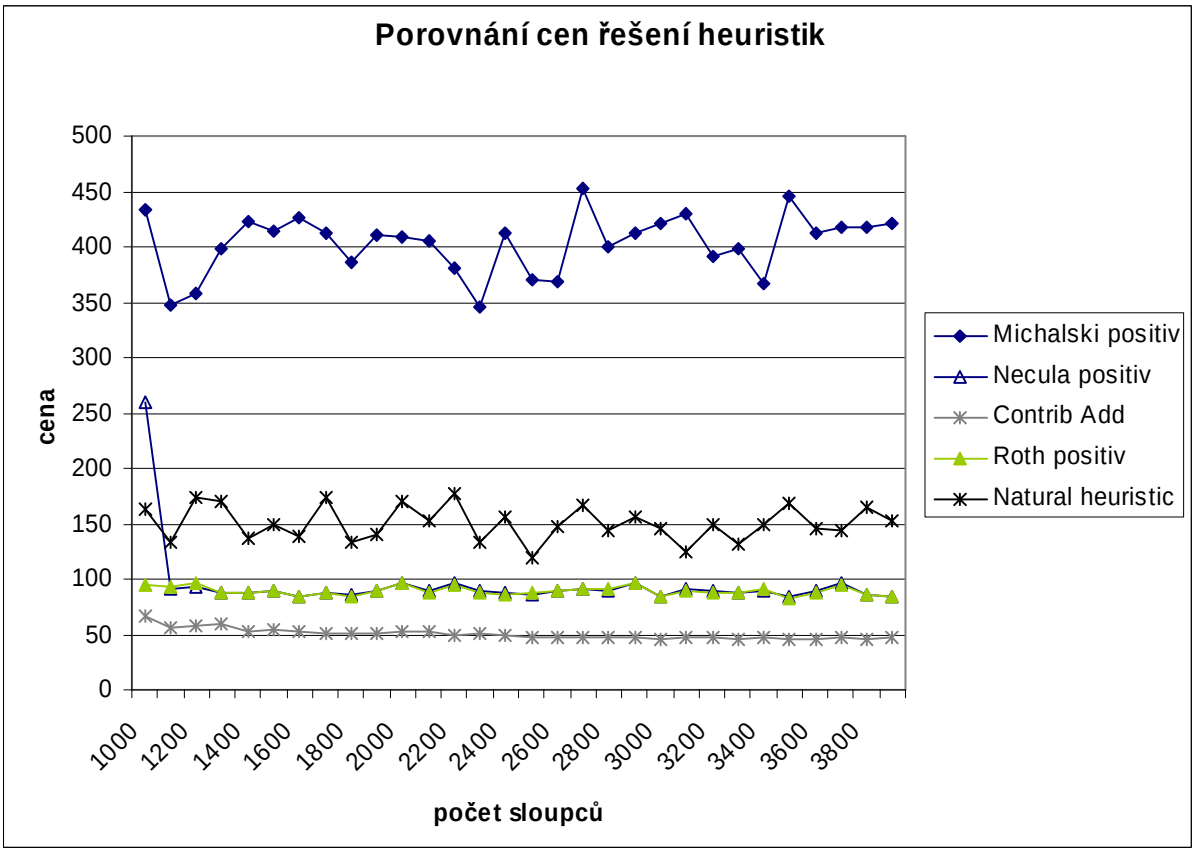


Graf 6.4.7 – Porovnání cen řešení získaných heuristikami

Z naměřených průběhů vidíme, že situace je naprosto odlišná než v případě jednotkových cen sloupců. V grafu není uvedeno porovnání cen řešení s cenami nalezenými heuristikou Necula positiv pro neúnosně vysoké ceny těchto řešení. Pro tyto rozměry matic heuristiky Roth, ContribAdd a Servít positiv dosahují velmi dobrých výsledků v akceptovatelném čase. Heuristika Servít dosahuje zřejmě nejnižších cen nalezených řešení pro nejednotkové ceny sloupců, avšak její použitelnost je omezena na velikost matice přibližně pro počet sloupců n=400 a počet řádků m= 400. Pro více sloupců se zdá být nejlepší heuristika Servít positiv a ContribAdd, avšak pro počet sloupců 400 a počet řádků 300 heuristika Servít positiv časovou náročností převyšuje naše požadavky. Pro ještě vetší matice pak připadají v úvahu heuristiky Roth positiv, Natural, ContribAdd, Michalski positiv a Necula positiv. Zbývá otestovat tyto heuristiky pro ještě vetší rozměry matice. Graf 6.4.8 znázorňuje závislost výpočetního času těchto heuristik na počtu sloupců matice, počet řádků m=1000, počet sloupců m=1000 – 3900 a hustotu 5%. V grafu 6.3.9 je znázorněno porovnání výše cen řešení nalezených těmito heuristikami. Z naměřených závislostí vyplývá použitelnost heuristiky ContribAdd do počtu sloupců 2500, při větším počtu sloupců je lepší zvolit rychlejší heuristiku Roth, která sále nachází přijatelné řešení.

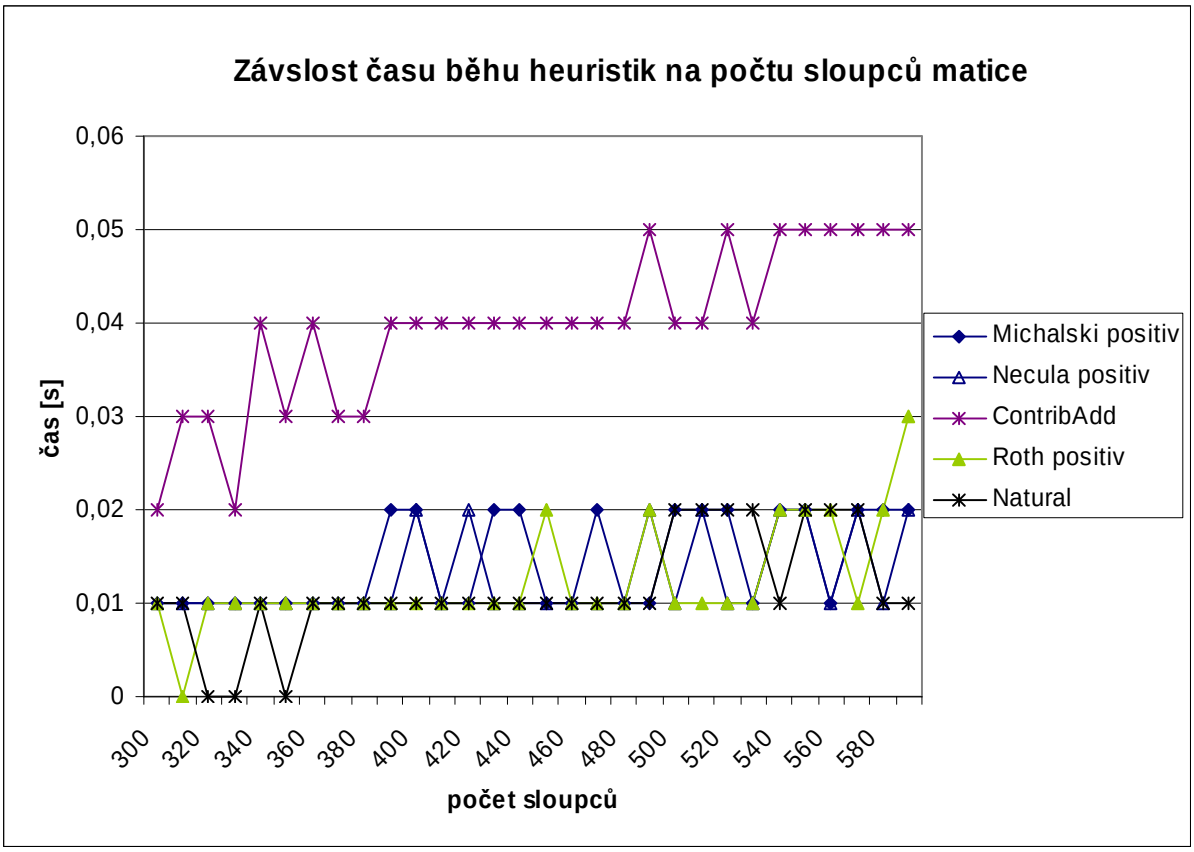


Graf 6.4.8 – Závislost času běhu heuristik na počtu sloupců matice – hustota 5%

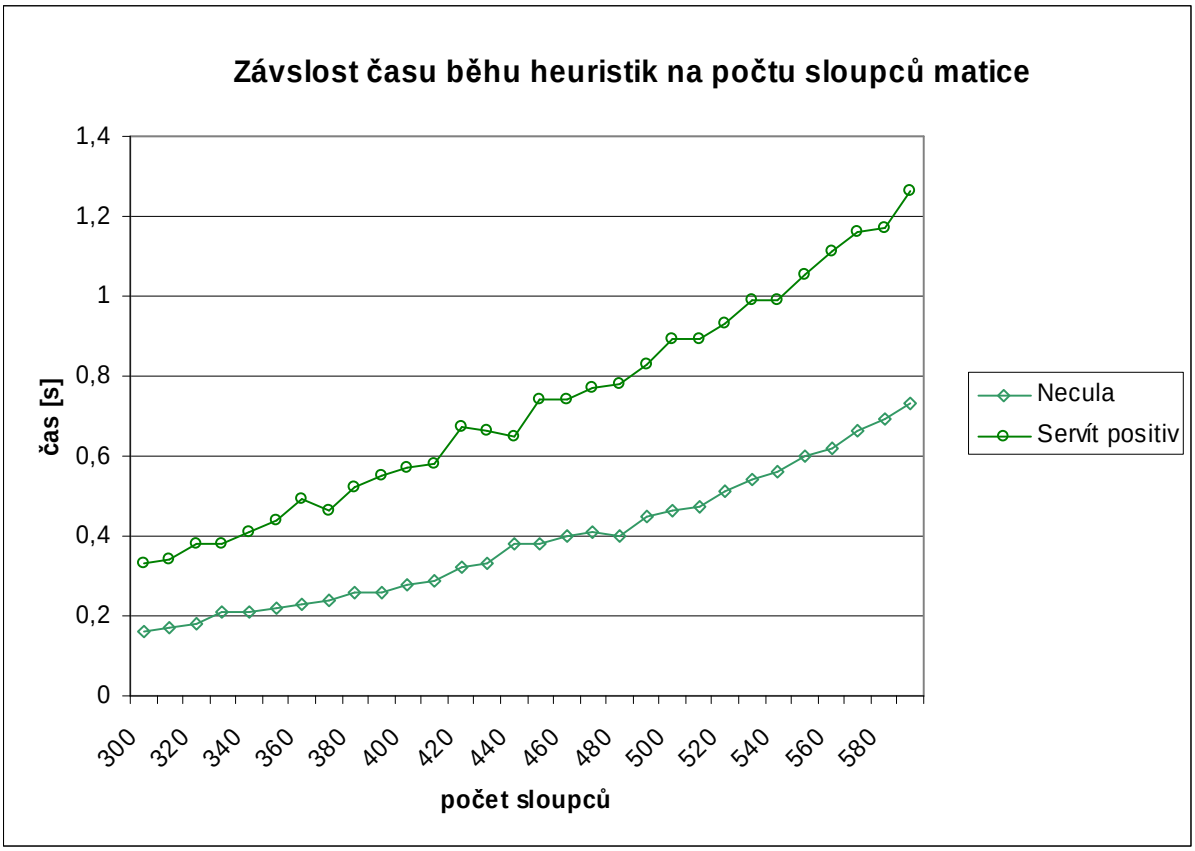


Graf 6.4.9 – Porovnání cen řešení získaných heuristikami

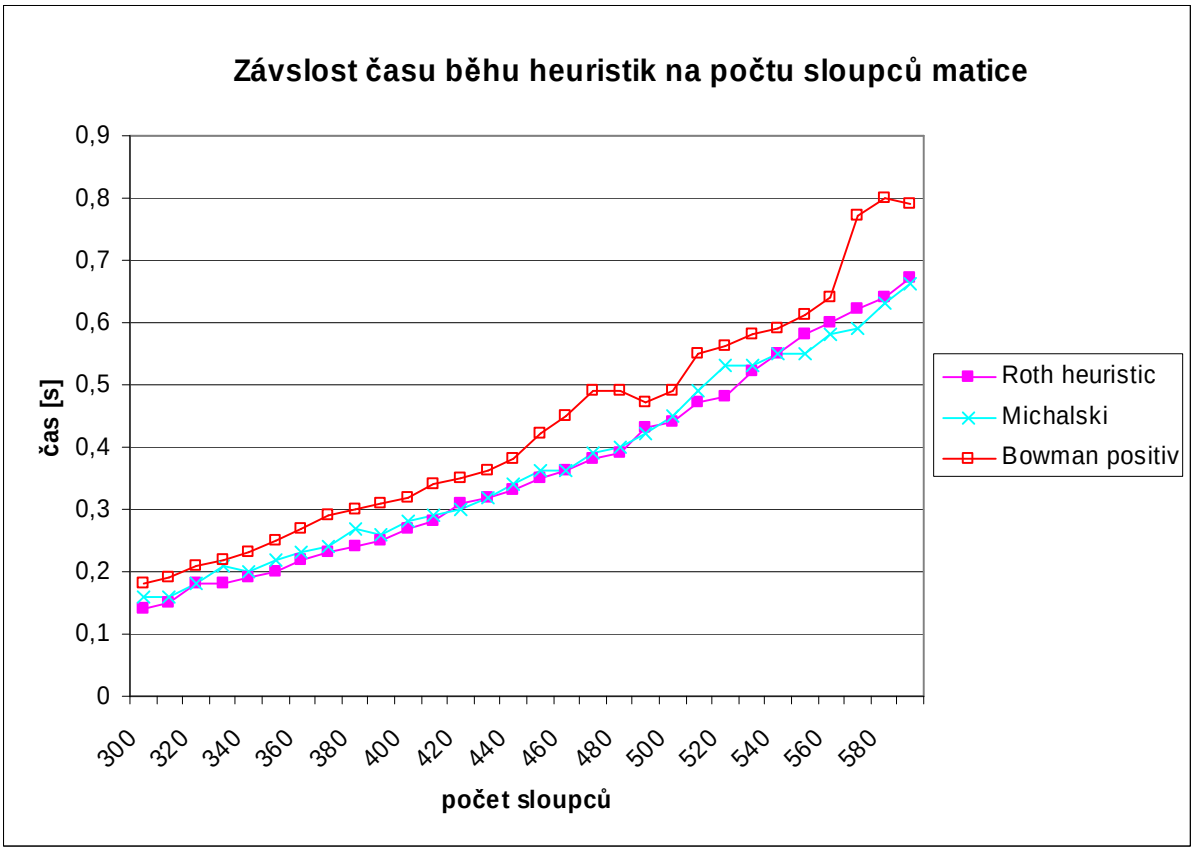
Dále ještě uvádím grafy závislosti výpočetního času pro rozměry matic m=300 řádků a n=300 – 590 sloupců a hustotě 15% při nejednotkových cenách sloupců. Časové závislosti heuristik pro tyto parametry matic jsou uvedeny v grafech 6.4.10 – 6.4.13. Při této hustotě matic je heuristika Servít nepoužitelná již při počtu sloupců n=300 a počtu řádků m=300. Řešení získané heuristikami Michalski positiv, Necula positiv a Natural je velmi nekvalitní pro tyto rozměry matice, jak uvádí grafy 6.4.14 – 6.4.17, avšak spolu s heuristikami ContribAdd a Roth positiv dosahují přijatelných časových hodnot a je tedy nutné je testovat pro větší rozměry matice.



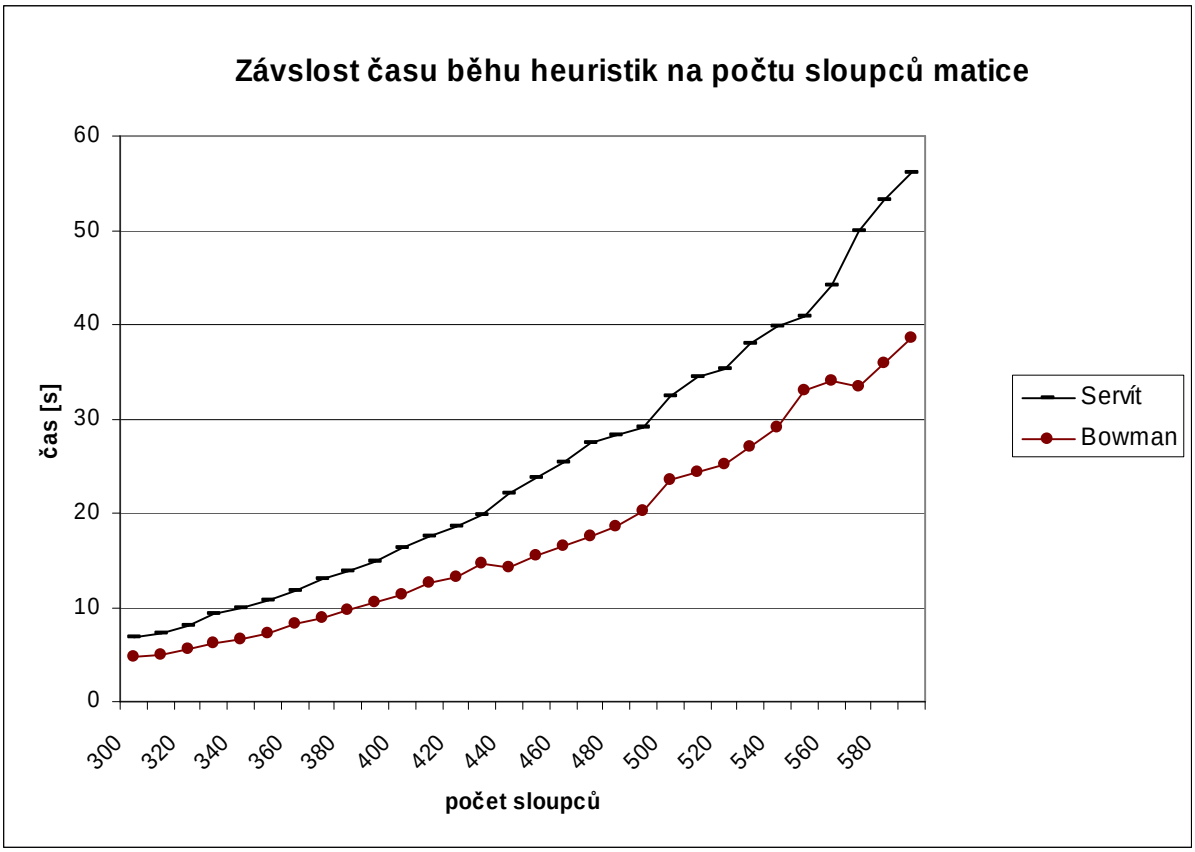
Graf 6.4.10 – Závislost času běhu heuristik na počtu sloupců matice – hustota 15%



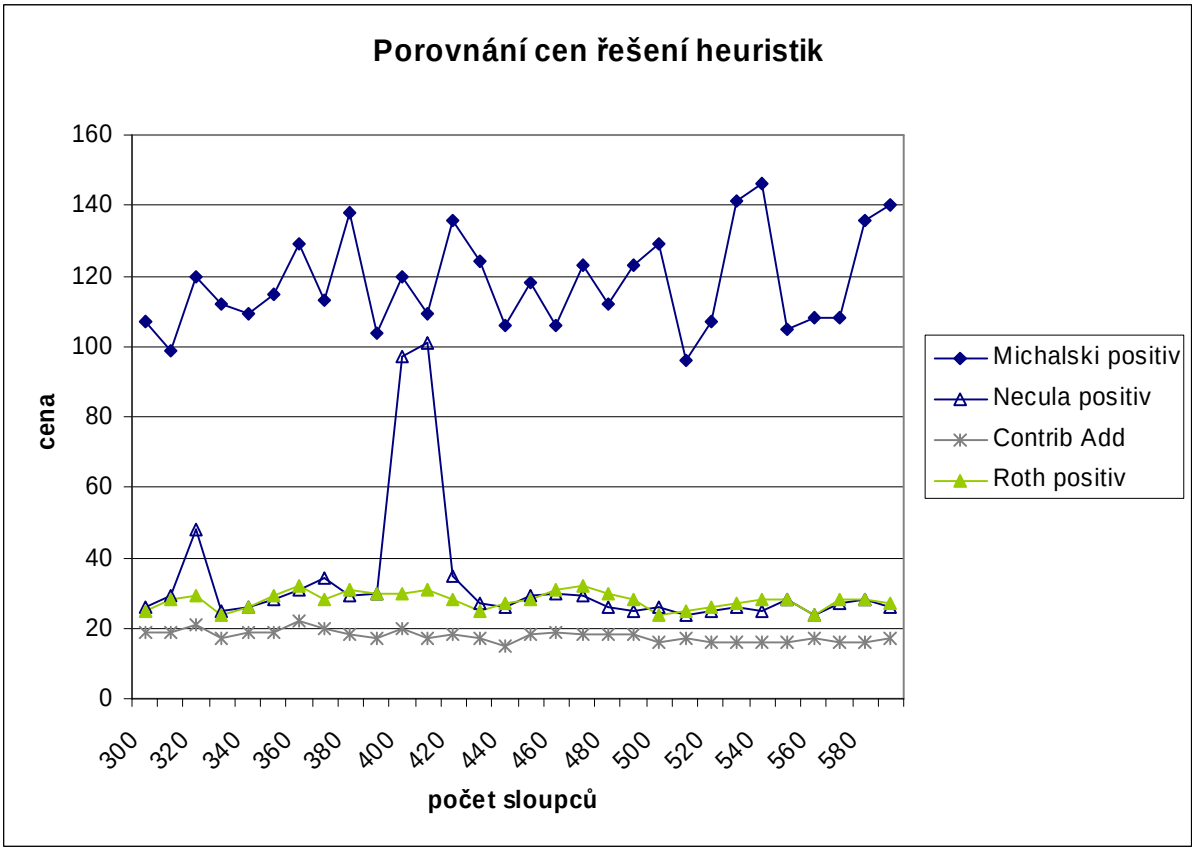
Graf 6.4.11 – Závislost času běhu heuristik na počtu sloupců matice – hustota 15%



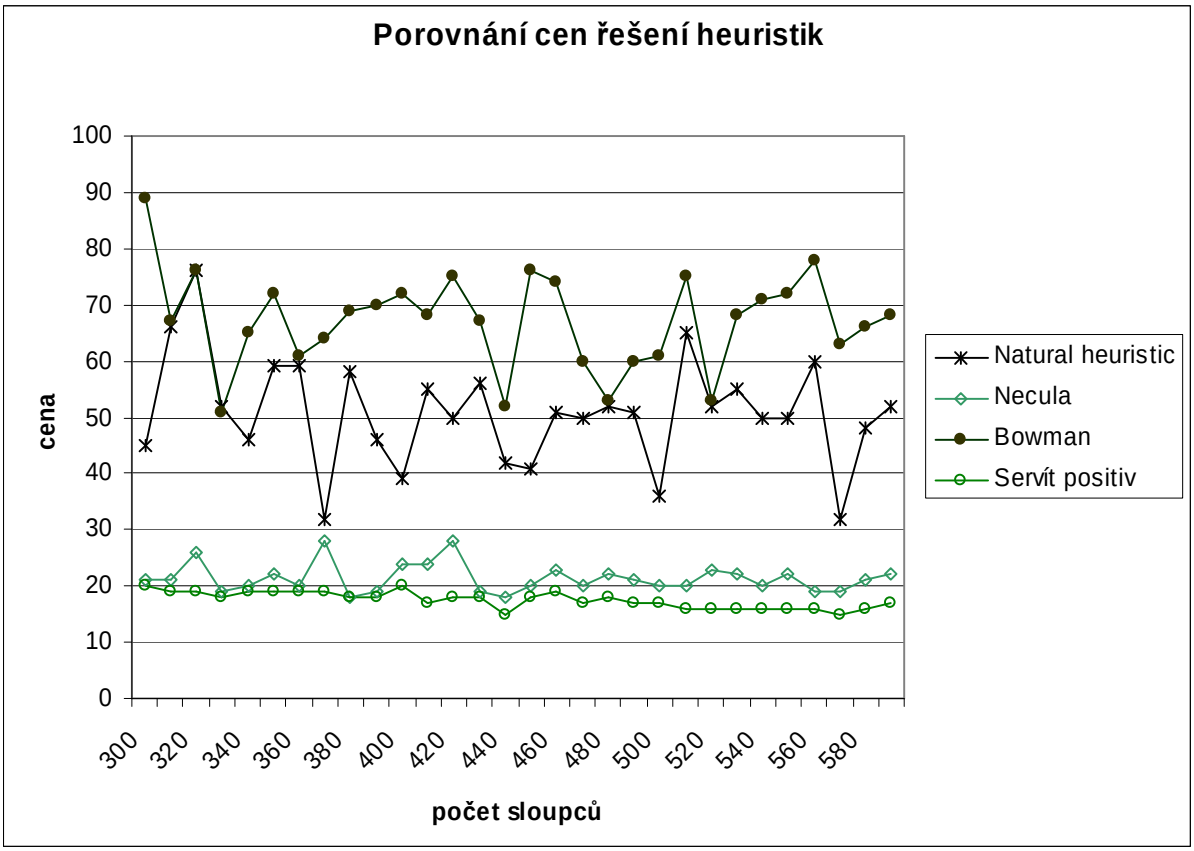
Graf 6.4.12 – Závislost času běhu heuristik na počtu sloupců matice – hustota 15%



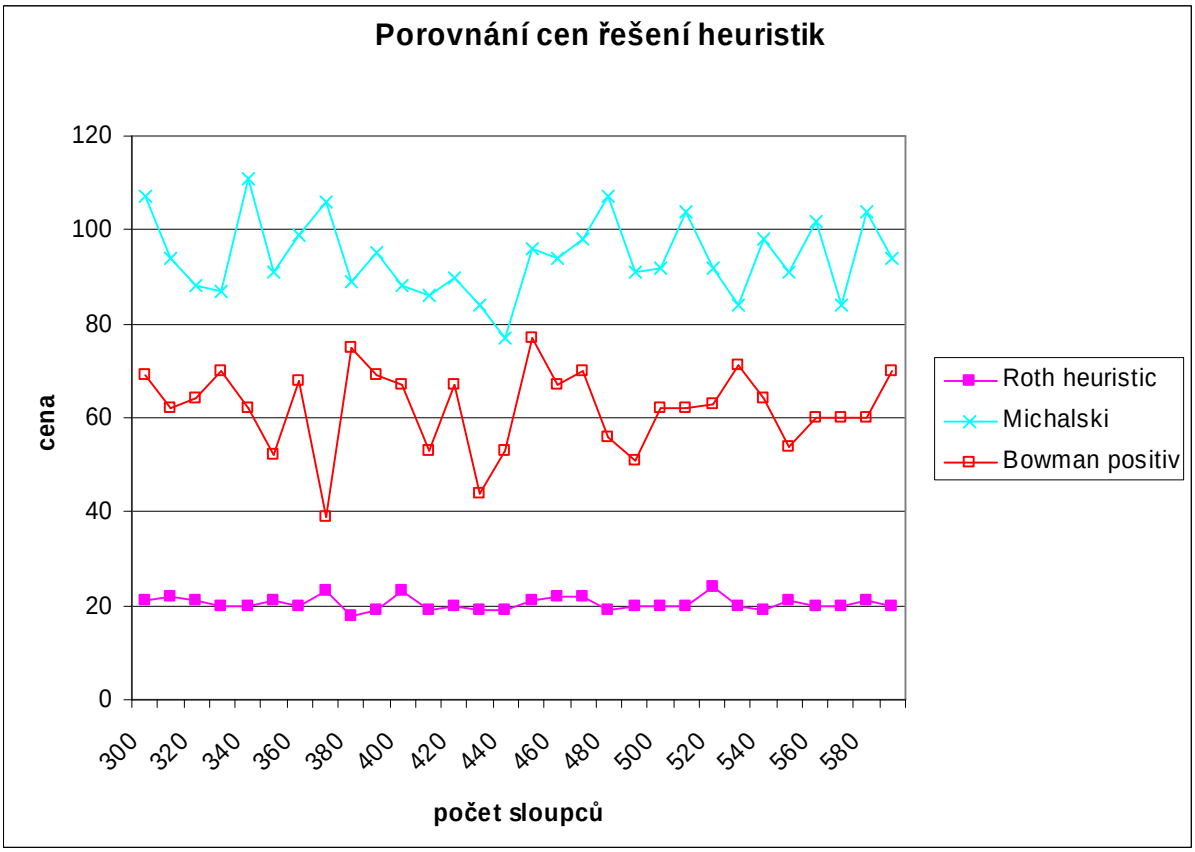
Graf 6.4.13 – Závislost času běhu heuristik na počtu sloupců matice – hustota 15%



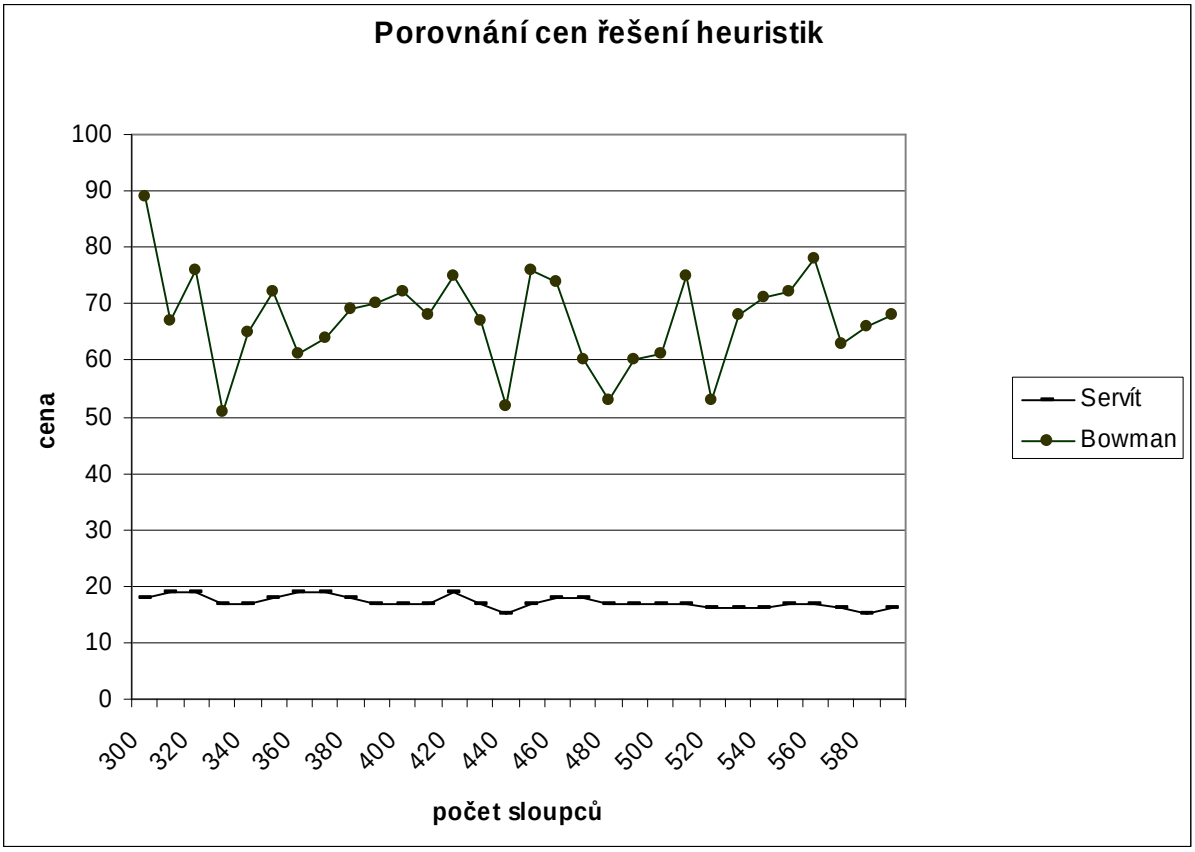
Graf 6.4.15 – Porovnání cen řešení získaných heuristikami



Graf 6.4.16 – Porovnání cen řešení získaných heuristikami

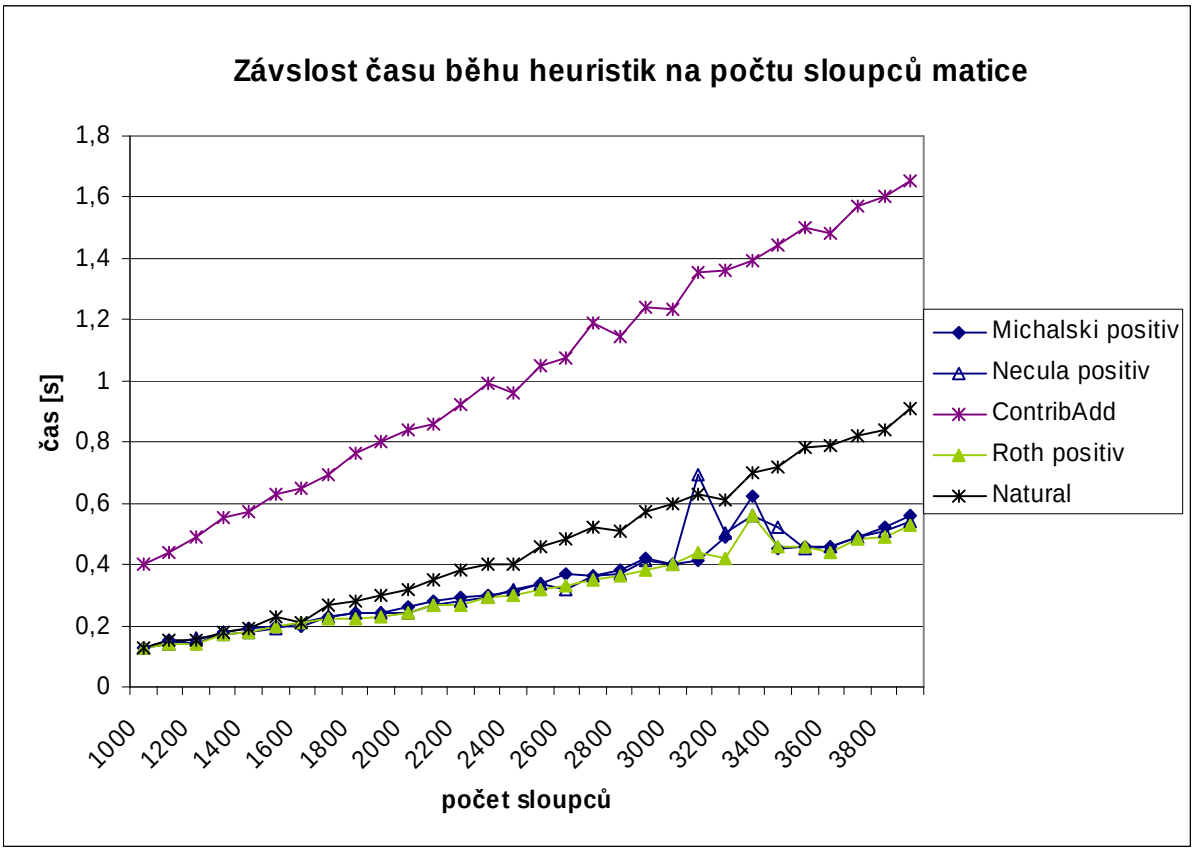


Graf 6.4.17 – Porovnání cen řešení získaných heuristikami

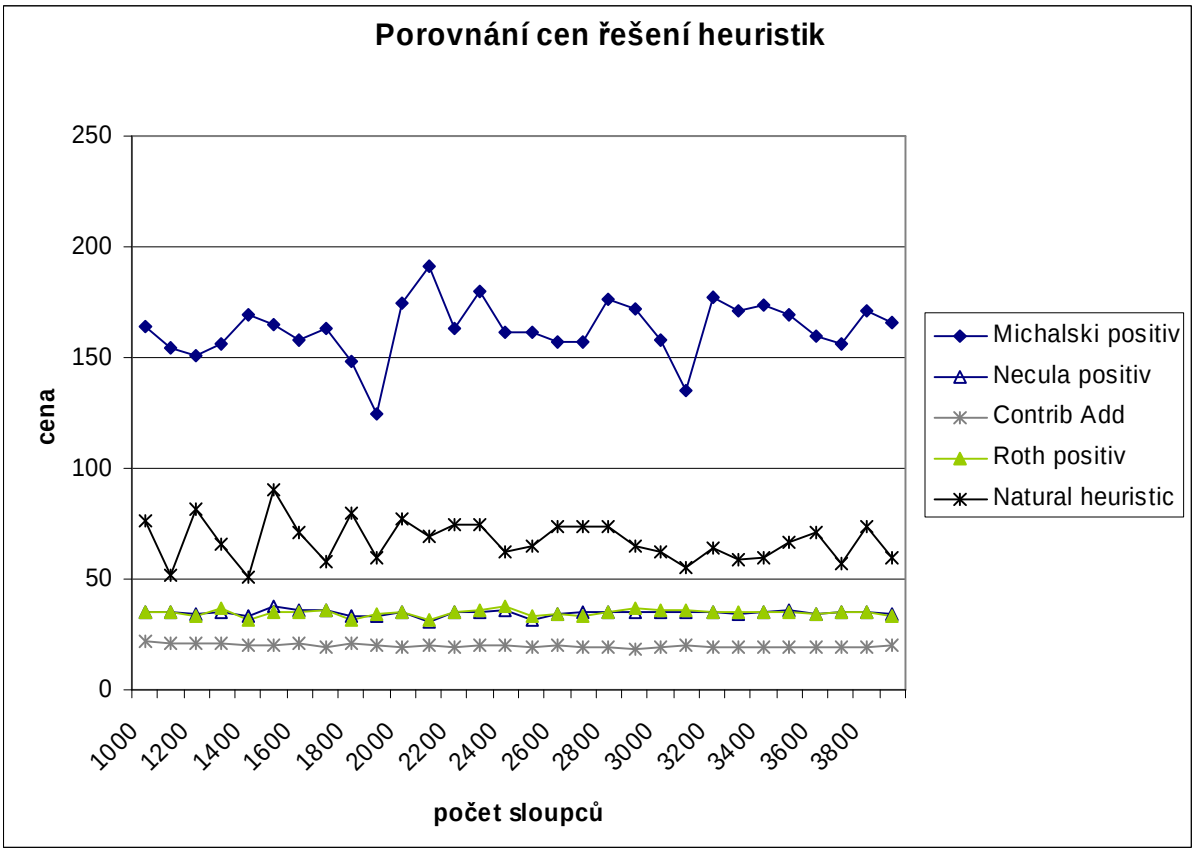


Graf 6.4.18 – Porovnání cen řešení získaných heuristikami

Vybrané heuristiky byly testovány na maticích o rozměrech m=1000 řádků, n=1000 – 3900 sloupců a hustotě 15% při nejednotkových cenách sloupců. Naměřené časové závislosti heuristik na měnícím se počtu sloupců jsou zobrazeny v grafu 6.4.19. Z naměřených průběhů vyplývá, že heuristika ContribAdd je použitelná do počtu sloupců 4000, dále je dobré použít heuristiku Roth positiv, která nalezne řešení rychleji s akceptovatelnou cenou. Tabulka 6.4.1 a 6.4.2 znázorňuje rozptyl cen řešení nalezených heuristikami pro reálné matice při získané programem BOOM, při nejednotkových cenách sloupců.



Graf 6.4.19 – Závislost času běhu heuristik na počtu sloupců matice – hustota 15%



Graf 6.4.20 – Porovnání cen řešení získaných heuristikami

Počet sloupců	Počet řádků	Natural	Contrib Add	Roth	Roth positiv	Michalski	Michalski positiv
38	77	0,0	0,0	25,0	46,2	75,0	85,7
54	54	0,0	0,0	5,9	38,9	33,3	33,3
62	62	8,3	0,0	27,3	35,7	29,6	51,7
77	77	7,7	0,0	12,5	50,0	32,1	67,7
80	80	8,7	0,0	13,9	24,3	20,5	19,8
85	85	16,7	8,3	36,0	64,3	21,9	57,6
91	91	5,1	0,0	15,5	25,0	23,3	32,4
100	100	29,5	0,0	15,0	30,1	23,4	32,0
104	104	25,0	0,0	46,7	69,6	37,9	64,1
157	157	11,5	2,7	10,6	16,7	7,9	20,9

Tabulka 6.4.1 –Rozptyl heuristik [%], nejednotkové ceny sloupců

Počet sloupců	Počet řádků	Necula	Necula positiv	Bowman	Bowman positiv	Servit	Servit positiv
38	77	54,5	200,0	0,0	25,0	0,0	0,0
54	62	27,3	58,5	17,1	17,1	0,0	0,0
62	62	7,7	100,0	4,5	13,6	0,0	0,0
77	42	28,6	110,5	3,1	3,8	0,0	0,0
80	94	18,0	34,7	4,3	2,8	0,0	0,0
85	82	28,1	113,3	0,0	19,2	0,0	0,0
91	90	20,0	52,7	1,8	10,3	0,0	0,0
100	80	20,2	30,3	1,6	6,1	0,0	0,0
104	90	20,8	63,9	0,0	26,3	0,0	0,0
157	102	7,5	42,7	5,0	2,7	0,0	0,0

Tabulka 6.4.2 –Rozptyl heuristik [%],nejednotkové ceny sloupců

7 Začlenění do programu BOOM

V předchozí kapitole jsme otestovali použitelnost Aury a jednotlivých heuristik podle velikosti a hustoty matice, nyní můžeme vytvořit adaptační mechanismus schopný podle parametrů matice rozhodnout, jakou metodu pro nalezení řešení UCP zvolíme. Tento mechanismus bude začleněn do programu BOOM.

7.1 Adaptační mechanismus - implementace

Pro vytvoření adaptačního mechanismu bylo zapotřebí pro vybrané heuristiky popsané v [4] implementovaný kód přepsat do stejného formátu, pro stejné datové struktury a názvy funkcí, pro které byla implementována Aura. Jelikož datová struktura reprezentující matici použitá v implementaci Aury, popsaná v [1], se liší od datové struktury reprezentující matici použité v programu BOOM, vytvořil jsem interface převádějící tyto struktury. Adaptační mechanismus je prováděn funkcí *AutoSelectSolver*, která provede výběr nejlepšího algoritmu pro nalezení řešení UCP podle velikosti matice, hustotu matice a cen sloupců.

Nejprve si shrňme poznatky získané testováním metod řešení UCP v kapitole 6. Metody jsme testovali nejprve pro jednotkové ceny sloupců matice. Při měření závislosti

výpočetního času Aury na hustotě matice byl zjištěn veliký nárůst času v oblasti hustoty 10% - 40%. Hustota reálných matic se však pohubuje nejčastěji mezi hodnotami 1% - 5%, kde má tendenci slabě stoupat, výjimečně se vyskytují matice s hustotou do 15%. Pro další měření jsme tedy zvolili matice s hustotou 5% a 15%. Pro hustotu 5% bylo měřením zjištěno, že pro počet sloupců n=200 a počet řádků m=200 ještě lze nalézt řešení UCP v rozumném čase, tato šance je však velmi nízká. Pro hustotu matice 15% jsou tyto hodnoty nižší, n=180 a m=180. Pro měření při nejednotkových cenách sloupců a hustotě 5% se odhad počtu řádků a sloupců ještě zmenší, n=150 a m=150, pro hustotu 15% odhadujeme hodnoty n=120 a m=120. tyto odhady hodnot řádků a sloupců budou použity v adaptačním mechanizmu pro výběr metody řešení UCP.

Pokud matice svými rozměry tyto hodnoty přesáhne, použijeme pro řešení UCP heuristiky. Testy heuristik byly prováděny opět pro hustoty 5% a 15%. Pro jednotkové ceny sloupců a hustotu 5% je nejlépe použitelná heuristika ContribAdd, a to do rozměrů matice n=1000 a m=1000. Při větších rozměrech by byl výpočet již příliš pomalý, volíme proto heuristiku Natural. Pro větší hustotu matice časová náročnost heuristiky ContribAdd klesá rychleji než náročnost heuristiky Natural a je proto nejlépe použitelná pro rozměr matice, kde již nemůžeme z časových důvodů použít Auru. Ostatní heuristiky vykazují velmi nekvalitní řešení v horším čase.

Pro nejednotkové ceny sloupců a hustotu 5% vykazuje nejlepší výsledky Servítova heuristika, její časová náročnost však umožňuje použití do rozměrů n=400 a m=400. Pro větší rozměry volíme heuristiku ContribAdd upravenou pro nejednotkové ceny sloupců, její časová náročnost umožňuje použití pro matice o rozměrech maximálně n=2500 a m=2500, pro větší matice volíme heuristiku Roth positiv, která vrací pro tyto parametry matice přijatelné řešení v rozumném čase. Pro hustotu matice větší než 5% volíme heuristiku ContribAdd do rozměrů matice n=3000 a m=3000, jelikož tato heuristika má při větší hustotě matice nižší časové nároky. Pro ještě větší matice volíme heuristiku Roht positiv.

Do mechanismu je zabudována uživatelem volitelná možnost zvolit si snahu (*effort*) mechanismu nalézt kvalitnější řešení, effort je volen stupni 1 – 10, pro *effort = 1* bude řešení UCP nalezeno nejrychleji avšak s nižší kvalitou, pro *effort = 10* bude hledání řešení UCP trvat déle, avšak kvalita řešení bude vyšší. Funkci *AutoSelectSolver* zobrazuje algoritmus 7.1.1.

```
AutoSelectSolver (A,Sol,MaxAuraRunTime,effort)
{
        default MaxAuraRunTime=2
        if (NumRows <100*effort && NumCols<100*effort )
        SolveDominance(A)
        /*pokud je matice dostatečně malá*/
        /*vyřešením dominancí může být zmenšena tak, že bude možno řešit CP Aurou*/

        if (unified col costs)
        {
          if (Matrix density < 6%)
                if (NumRows <20*effort && NumCols<20*effort )
                        use Aura
                else if (NumRows <100*effort && NumCols<100*effort )
                        use ContribAdd heuristic
```

```
            else use Natural Heuristic
        else if (Matrix density > 6%)
            if (NumRows <18*effort && NumCols<18*effort )
                use Aura
            else if use ContribAdd heuristic
    }
    else if (! unified col costs )
    {
        if (Matrix density < 6%)
            if (NumRows <15*effort && NumCols<15*effort )
                use Aura
            else if (NumRows <40*effort && NumCols<40*effort )
                use Servit heuristic
            else if (NumRows <250*effort && NumCols<250*effort )
                use ContribAdd heuristic
            else use Roth positiv heuristic
        else if (Matrix density > 6% )
            if (NumRows <12*effort && NumCols<12*effort )
                use Aura
            if (NumRows <300*effort && NumCols<300*effort )
                use ContribAdd heuristic
            else use Roth positiv heuristic
    }

}
```

Algoritmus 7.1.1 – Funkce AutoSelectSolver

Funkčnost mechanismu jsem testoval na generovaných PLA, které byly pomocí nástroje BOOM minimalizovány a pomocí nástroje Espresso testovány, zda odpovídají původním neminimalizovaným PLA.

Uživatel má zároveň možnost zvolit si pro řešení heuristiky Natural, ContribAdd, Servit a Roth positiv, ostatní nebyly z důvodu nízké kvality použity. Aura byla do programu BOOM implementována namísto současného exaktního algoritmu. Do Aury je implementována možnost ukončit výpočet, pokud výpočetní čas přesáhne určitou hranici, zvolenou uživatelem. Pokud tato hranice je však tak nízká, že pomocí předčasně ukončené Aury není nalezeno žádné řešení, nalezne se řešení UCP pomocí heuristiky ContribAdd.

7.2 Adaptační mechanismus - experimentální výsledky

Implementovaný adaptační mechanismus byl otestován na CM_PLA benchmarcích, které mi byly poskytnuty za tímto účelem. Testování jsem prováděl pro několik hodnot parametru effort, pro nižší hodnoty parametru se očekává nalezení řešení v kratším čase ale s nižší kvalitou, pro vyšší hodnoty parametru nalezení řešení bude trvat déle, ale výsledná kvalita by měla být vyšší. Naměřené hodnoty jsou uvedeny v tabulce 7.2.1. Pro úplnost v tabulce ještě uvádím řešení, kde byla pro řešení UCP použita Aura. Počet iterací BOOMu je roven 10. Testy byly prováděny na počítači s procesorem Intel celeron 2.2 GHz, 512Mb RAM.

			Boom Auto solver,effort:			Boom Aura
benchmark	*i/o/p*		1	5	10	
		počet termů	9	9	9	8
		počet literálů	24	24	24	24
		output cost	11	11	11	12
b05_3	35/5/20	čas [s]	0,07	0,08	0,08	0,03
		počet termů	10	9	9	9
		počet literálů	25	23	23	23
		output cost	26	22	22	20
b04_1	77/9/37	čas [s]	0,2	0,28	0,29	0,13
		počet termů	13	13	13	13
		počet literálů	47	47	48	56
		output cost	13	13	13	13
s13207_1	700/5/55	čas [s]	1,1	2,2	2,1	1,4
		počet termů	21	20	20	18
		počet literálů	63	59	61	62
		output cost	26	27	26	28
b12_2	126/7/66	čas [s]	0,7	4,1	5,2	t>100
		počet termů	71	44	43	42
		počet literálů	241	159	153	171
		output cost	163	238	242	252
c7552_1	207/48/81	čas [s]	43	39	40	t>100
		počet termů	31	29	31	30
		počet literálů	134	129	143	138
		output cost	35	32	31	35
s13207.1_2	700/8/96	čas [s]	6,2	17,2	17	t>100
		počet termů	45	33	35	33
		počet literálů	176	130	139	136
		output cost	52	64	66	62
b12_1	126/11/128	čas [s]	4,49	22,44	103	t>100
		počet termů	71	54	52	51
		počet literálů	293	242	227	235
		output cost	129	159	165	150
s9234_3	1464/464/307	čas [s]	25	23	227	t>100
		počet termů	75	38	39	39
		počet literálů	262	156	149	165
		output cost	202	235	234	201
s38584.1_1	247/38/99	čas [s]	431	680	610	672
		počet termů	112	57	53	51
		počet literálů	393	219	194	192
		output cost	279	284	274	292
c15850.1_1	611/96/313	čas [s]	330	350	510	t>1106

Tabulka 7.2.1 – Porovnání výsledků - CM_PLA benchmarky

V tabulce je za názvem benchmarku uveden počet vstupních proměnných / počet výstupních proměnných / počet termů a dále srovnávány kvality nalezených řešení. Pro některé benchmarky jsou naměřené časy přibližně stejné jak pro všechny hodnoty effortu, tak pro použití Aury. Je to způsobeno tím, že pokud je matice pokrytí malá, tak automatický výběr metody řešení UCP vybere pokaždé Auru. Dále vidíme, že pro některé matice je při vyšší hodnotě effortu řešení nepatrně horší. Je to způsobeno vytvářením matice UCP v programu BOOM, jelikož tento proces je vytvářen randomizovanou heuristikou. Pro

změření rozptylu možných řešení jsem provedl měření na několika vybraných CM_PLA benchmarcích, pro řešení UCP byla použita Aura bez časového omezení, BOOM byl spuštěn 5x pro každý benchmark, při stejných parametrech. Naměřené výsledky jsou uvedeny v tabulce 7.2.2.

benchmark	minimalizace č.:	1	2	3	4	5
b05_1.pla	počet termů	16	17	17	16	16
	počet literálů	55	57	59	52	55
	output cost	31	28	31	30	28
	čas [s]	8	6	7,2	9	4,2
s1238_2.pla	počet termů	13	14	14	13	14
	počet literálů	47	49	53	42	49
	output cost	22	22	22	18	23
	čas [s]	3,7	10	4,5	4,4	6,7
s5378_2.pla	počet termů	7	7	7	8	8
	počet literálů	21	19	18	26	26
	output cost	10	11	10	10	10
	čas [s]	0,6	0,9	0,7	0,4	0,4
s9234_6.pla	počet termů	25	25	25	24	24
	počet literálů	96	97	89	92	93
	output cost	61	61	62	61	57
	čas [s]	55	61	51	15	14

Tabulka 7.2.2 – Rozptyl řešení získaných BOOMem.

Z tabulky je vidět, že hodnoty řešení se pro stejné parametry BOOMu mohou lišit, z tohoto důvodu se může stát, že pro vyšší hodnotu effortu je nalezené řešení horší, než při nižší hodnotě. Použití exaktního algoritmu pro řešení UCP může mít z těchto důvodů rovněž horší řešení.

Dále zde uvádím měření některých MCNC benchmarků. Výsledky jsou uvedeny v tabulce 7.2.3. Vliv hodnot effortu se projevuje až pro ty benchmarky, pro které je generována matice UCP velikých rozměrů, pak je dobře vidět rostoucí čas a kvalita řešení pro vyšší hodnotou effortu. U benchmarků, kde se výpočetní časy téměř rovnají, je tato skutečnost patrně způsobena stejnými nebo podobnými rozměry vzniklé matice UCP, a je vybrána automatickým výběrem metody řešení UCP stejná metoda. Rozměry matice jsou navíc tak malé, že čas vynaložený pro nalezení řešení UCP je zanedbatelný oproti minimalizaci PLA, výsledný čas je tvořen z největší části právě touto minimalizací. Odchylky v řešení u těchto benchmarků jsou způsobeny vytvářením matice UCP při minimalizaci programem BOOM.

			Boom Auto solver,effort:			Boom Aura
benchmark	*i/o/p*		1	5	10	
		počet termů	42	42	42	42
		počet literálů	177	177	177	177
		output cost	45	45	45	45
alcom	15/38/90	čas [s]	0,03	0,03	0,03	0,03
		počet termů	220	228	229	220
		počet literálů	1854	1907	1912	1854
		output cost	1079	1030	1025	1079
apex1	45/45/1440	čas [s]	33	35	31	33
		počet termů	1039	1039	1039	1039
		počet literálů	14477	14477	14477	14477
		output cost	1069	1069	1069	1069
apex2	39/3/1576	čas [s]	22	49	48	58
		počet termů	310	318	317	310
		počet literálů	2408	2453	2448	2414
		output cost	896	856	859	903
apex3	54/50/1036	čas [s]	25	44,4	58	20
		počet termů	502	497	498	496
		počet literálů	4076	4040	4048	4032
		output cost	1597	1578	1586	1587
apex4	9/19/1907	čas [s]	402	258	337	t>>100
		počet termů	1088	1088	1088	1088
		počet literálů	6089	6089	6089	6089
		output cost	1192	1192	1192	1192
apex5	117/88/2710	čas [s]	122	129	138	t>>100
		počet termů	59	58	58	58
		počet literálů	472	471	471	471
		output cost	96	103	103	103
b4	33/23/680	čas [s]	1,9	2	1,8	1,9

Tabulka 7.2.3 – Porovnání výsledků - MCNC benchmarky

8 Závěr

Současný nejlepší algoritmus AURAII pro nalezení řešení unátního pokrytí byl úspěšně implementován, v algoritmu byly v této práci navrženy úpravy značně urychlující výpočet a úpravy umožňující použít algoritmus na matice s cenami sloupců různé od jedné. Byla zde navržena heuristika pro přesnější nalezení množiny nezávislých řádků a heuristiky s různými strategiemi výběru sloupců do řešení, zrychlující výpočet. U heuristiky ContribAdd byla provedena drobná úprava pro nejednotkové ceny sloupců, zajišťující mnohem přesnější nalezení řešení problému pokrytí touto heuristikou. Dále byly přepsány kódy některých heuristik popsaných v [4].

Implementované algoritmy byly testovány na reálných a generovaných maticích, abychom byli schopni vytvořit adaptační mechanismus, schopný na základě cen sloupců, rozměrů a hustoty matice, vybrat nejlepší algoritmus pro nalezení řešení problému pokrytí. Z naměřených výsledků uvedených v kapitole 6 vyplývá, že exaktní metodou AURAII jsme

schopni nalézt řešení problému pokrytí v přijatelném čase pro matice o rozměrech do 200 řádků a 200 sloupců při jednotkových cenách sloupců a hustotě matice do 5%, při vyšší hustotě matice je maximální počet sloupců a řádků nižší. Pokud jsou ceny sloupců nejednotkové, jsou maximální hodnoty řádků a sloupců ještě nižší, hodnoty hustoty reálných matic jsou však většinou nízké, jak je uvedeno v kapitole 6.

Pro matice přesahující svými rozměry možnosti Aury adaptační mechanismus vybere příslušnou heuristiku. Z výsledků v kapitole 6 je zřejmé, že pro nejednotkové ceny sloupců a nízkou hustotu matic je nelepší heuristika ContribAdd, která je však pro velké matice časově náročná a je lepší zvolit heuristiku Natural. Pokud jsou ceny sloupců nejednotkové, nejlepších výsledků dosáhneme použitím heuristiky Servít positiv, která však vykazuje velké časové nároky a je použitelná pro matice o rozměrech do 400 sloupců a 400 řádků. Pro větší matice musíme zvolit upravenou heuristiku ContribAdd, která dosahuje velmi dobrých výsledků v přijatelném čase.

Kvalita řešení získaná Aurou byla pro matice s nižšími časovými nároky porovnána s řešeními získanými funkcí *Branch_and_Bound* s výpočtem spodní hranice pomocí *MSIR*, z důvodu ověření funkčnosti Aury. Ceny nalezených řešení byly vždy ekvivalentní. Implementovaný adaptační mechanismus byl začleněn do programu BOOM.

9 Literatura

[1] Lukáš Krejčík. Moderní metody řešení problému pokrytí, 2005

[2] Luca P. Carloni. Negative Thinking In Search Problems, 1997

[3] Soha Hassoun, Tsutomu Sasao. Logic Synthesis and Verification, 2002

[4] Kamil Kopejtko, Moderní metody řešení problému pokrytí, 2007

[5] E.I.Goldberg, L.P.Carloni, T.Villa, R.K.Brayton, A.L.Sangiovanni-Vincenrelli. Negative Thinking by Incremental Problem Solving: Application to Unate Covering

[6] M.Servít. A heuristic method for solving weighted set covering problems

[7] O.coudert. Two-level logic minimization: an overview. *Integration*, 17-2:97-140, October 1994

[8] Fišer, P. - Hlavička, J.: BOOM - a Boolean Minimizer. Research Report DC-2001-05, Prague, CTU Publishing House, June 2001, 37 pp

[9] R.K.Brayton, G.D. Hatchel, C.T. McMullen, A.L.Sangiovanni-Vincenrelli, *Logic Minimization Algorithms for VLSI Synthesis* (Kluwer Academic Publishers, Dordrecht,1984)

[10] E.L. Jr. McCluskey, Minimization of Boolean Functions, *Bell Systém Technical Journal*, Vol. 35, pp. 1417-1444, April 1959.

[11] R.L. Rudell, *Logic Synthesis for VLSI Design*, PhD Thesis, UCB/ERL M89/49, 1989.

A Seznam použitých zkratek

BSONIR Best Set Of Non-Intersecting Rows
MSIR Maximal Set of Independent Rows
UCP Unate Covering Problem

B Ovládání programu BOOM

Program BOOM se ovládá parametry z příkazové řádky. Stávající parametry jsou:

syntax: BOOM [options] source [destination] [report]
Minimization Options:
-FRn Define FC-Min:CD-search ratio n (0-100), 0 is default
-DFn Define the FC-Min Depth Factor, 10 is default
-DDx Define the FC-Min Depth Factor Direction
0 DF = 1:n (from -DFn)
1 DF = n:1 (from -DFn), default
-CMn Define CD-search mutations ratio n (0-100)
-RMn Define implicant reduction mutations ratio n (0-100)
-Ex Select implicant expansion type:
0 Sequential search
1 Distributed multiple IE
2 Distributed exhaustive IE
3 Multiple IE (default)
4 Exhaustive IE
-CPx Select the covering problem solution algorithm:
0 LCMC
1 Contributive selection (default)
2 Contributive removal
3 Exact cover
-Sxn Define stopping criterion x of value n:
t stop after n seconds
i stop after n iterations (default is Si1)
n stopping interval equal to n
q stop when quality of solution meets n
-Qx Define optimization criterion x:
t terms
l literals
o output cost
b literals+output cost (default)
g number of gates
G number of gate equivalents
-single Single-output minimization
-endcov Cover at the end only
-SD Substitute output don't cares
-Ox[n] Define output format:
P PLA (default)
V[n] VHDL. If n is specified, the function is split into max. n-input gates
W[n] VHDL - inputs and outputs as vectors. If n is specified, the function is split into max. n-input gates
v Verilog
B BLIF
E Equations
H HTML (table)
-nocheck Don't check the input function for consistence

-noIR Don't perform the implicant reduction
-namevars Implicitly name the variables

Special Functions (cannot be combined with minimization):
-info Prints the information about the PLA function (for fd type only)
-Tcheck Checks the input function for consistence
-Tsplit Splits intersecting terms
-Tminterm Splits function into minterms

Nově přidané parametry příkazové řádky pro volbu funkce hledání řešení CP jsou:

-CPx Select the covering problem solution algorithm:
3 Exact Aura cover
4 Auto select cover procedure
5 Natural Heuristic cover
6 ContribAdd Heuristic cover
7 Roth Positiv Heuristic cover
8 Servit Heuristic cover
-tx Stop Aura after x seconds
-fx Effort level, 1-10, 1 - fastest cover & low quality solution, 10 - slowest cover & high quality solution

Parametry CP3 – CP8 lze vybrat pro řešení CP požadovanou metodu. Parametrem –t*x* lze zastavit Auru po *x* sekundách. Parametrem –fx lze vybrat požadovaný effort, popsaný v kapitole 7.1.