Moderní metody řešení problému pokrytí

Lukáš Krejčík

Prohlášení

Prohlašuji,že jsem svou bakalářskou práci vypracoval samostatně a použil jsem pouze podklady (literaturu,projekty,SW atd.) 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).

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

V Praze dne 26.5.2005 Lukáš Krejčík

Anotace
Cílem této bakalářské práce je naprogramovat metody zabývající se řešením minimálního pokrytí. Problém pokrytí se objevuje v mnoha oblastech počítačového výzkumu, syntéze logických obvodů a analýze spolehlivosti. Řešení problému pokrytí je obecně velmi zdlouhavá a nákladná záležitost, založená na prohledávání celého vyhledávacího prostoru. Cílem je tedy naprogramovat nové metody, které pro zadanou matici naleznou její minimální pokrytí, důraz je kladen na rychlost. K tomuto účelu mi byly poskytnuty na prostudování materiály uvedené v seznamu literatury. Nakonec jsem zde uvedl výsledky porovnání času běhu těchto algoritmů oproti standardním metodám.

Abstract
The aim of this work is to program methods solving the covering 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. The aim is to program new methods solving the covering problem with accent to solving speed. I have been given materials mentioned in the References. Finally, I mentioned experimental results and compared with the results of standard methods.

Obsah

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. V [1],[2],[6] jsou navrženy úpravy pro ořezání stavového prostoru pro urychlení výpočtu.

Při hledání minimálního řešení máme však ještě jiné možnosti, např. použití heuristiky popsané v [1],[6], 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.

V této práci jsou uvedeny techniky vylepšení rekurzivního algoritmu, které významným způsobem urychlí výpočet. První úprava popsaná v [1] se zabývá ořezáváním stavového prostoru, kde řešení přesahuje již nalezené řešení. Ostatní techniky, popsané v [2],[6], se zabývají výpočtem spodní hranice řešení, která poslouží k ořezání prostoru o místa,kde se řešení nenachází.

Dále je zde v kapitole 3 uveden popis algoritmů implementovaných pro řešení problému pokrytí, které byly testovány na uvedených maticích. Výsledky testů jsou uvedeny v kapitole 5.

2. Problém pokrytí

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 , cena je funkce definovaná na Y | 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 *i* je pokrytý sloupcem *j* pokud *Aij = 1*. Sloupce jsou ohodnoceny kladným reálným číslem, zvaném cena sloupce. Hledání řešení spočívá ve výběru sloupců do řešení, dokud nejsou pokryty všechny řádky. Cena řešení je tvořena součtem cen sloupců vybraných do řešení. Pokrytí matice může být nalezeno buď přesně, s použitím rekurzivního algoritmu, nebo použitím heuristiky. Přesné řešení problému pokrytí je ve většině případů velmi časově náročné, v některých případech je použití heuristiky jedinou možností.

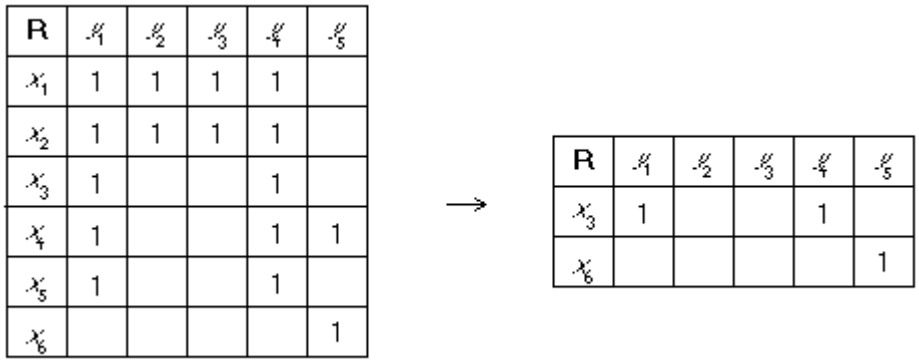
2.1 Dominance

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 *x2* 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 pokryto *x'*, bude pokryto i *x*. Řádek *x* tedy může být vyškrtnut z matice. Funkce *R_Dominance (struct CoveringMatrix *)* vždy vezme jeden řádek *x'* z matice a porovná ho s ostatními. Pokud nalezne řádek x obsahující 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 $O(n^2)$, kde *n* je počet řádků matice. Na obrázku 2.1.1 je znázorněna redukce matice před a po odstranění dominance řádků.

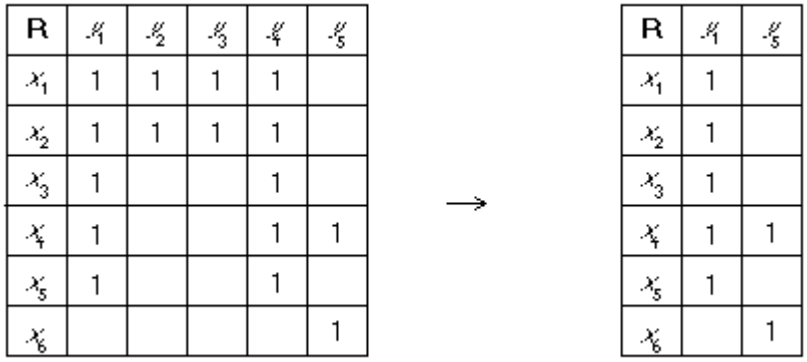


Obrázek 2.1.1 – Odstranění dominancí na množině řádků

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*. Funkce *S_Dominance (CoveringMatrix *)* vezme sloupec *y'* z matice a porovná ho s ostatními sloupci. Pokud sloupec *y* obsahuje nuly 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 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.

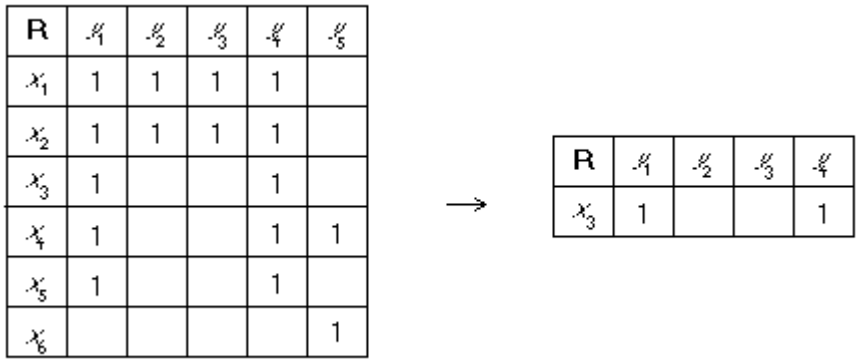


Obrázek 2.1.2 – Odstranění dominancí na množině sloupců

2.1.3 Nezbytné sloupce

Sloupec *y* množiny sloupců *Y* je nezbytný, pokud je jediný, který pokrývá řádek *x* množiny řádků *X*. Pokud *Y* je množina přímých implikantů, znamená to nesporný přímý implikant. 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. Funkce *SolveDominance (CoveringMatrix *, Reseni *)* volá funkce popsané výše tak dlouho,dokud je matice měněna řešením dominancí. Druhý parametr má strukturu ukazující na spojový seznam množiny nezbytných sloupců, které byly funkcí vybrány do řešení.

3 Řešení problému pokrytí

Poté co matici zredukujeme pomocí procedur popsaných výše (dominance), zbude matice, kterou tyto procesy už více nezmění. Tato matice je nazývána *cyklické jádro <X,Y,cena>*. Pokud je prázdné, množina všech nezbytných prvků byla nalezena během procesů popsaných výše.

Pokud není prázdné, minimalizace může být zakončena pomocí rekurzivního algoritmu. Princip algoritmu spočívá ve výběru prvku *y* množiny *Y* a generování dvou podproblémů, jeden spočívá ve výběru prvku *y* do minimálního řešení, druhý ve vyškrtnutí *y* z minimálního řešení. Tento algoritmus se rekurzivně aplikuje na cyklická jádra vzniklých podproblémů, minimální řešení problému je minimum z obou řešení podproblémů. Jak již bylo řečeno, tento algoritmus je velice časově náročný, s operační složitostí O (2^n), kde *n* je počet sloupců matice.

V algoritmu můžeme provést několik úprav, popsané v [1], které nám urychlí výpočet. Při nalezení minimálního pokrytí *Nejlepší* můžeme ořezat při prohledávání všechny větve, kde cena cesty do uzlu je vetší nebo rovna ceně již nalezeného pokrytí *Nejlepší*. Tím se urychlí výpočet, protože se nebude procházet prostor, kde se řešení určitě nenachází. Cena minimálního nalezeného řešení se nazývá horní hranice řešeni (*upper bound*).

3.1 Metoda větví a hranic

Tato úprava rekurzivního algoritmu spočívá ve výše uvedeném ořezávání vyhledávacího prostoru o místa, kde cena částečného pokrytí převyšuje cenu nejlepšího nalezeného pokrytí.

Vyhledávací prostor je reprezentován binárním vyhledávacím stromem, kde kořen představuje vstupní problém, hrany znamenají volání rekurze, každá je ohodnocena 0 nebo 1, podle toho, zda sloupec uvažovaný v rodičovském uzlu je do řešení přidán či ne.Vnitřní uzel reprezentuje částečné řešení problému s *cenou* rovnou součtu cen sloupců na cestě z kořene do tohoto uzlu, listu je dosaženo pokud je nalezeno pokrytí.

Algoritmus se rekurzivně aplikuje na cyklická jádra vzniklých podproblémů, minimální řešení problému je minimum z obou řešení podproblémů. Procedura *Branch_And_Bound (CoveringMatrix *)* , jejíž parametrem je ukazatel na strukturu, vybere tento sloupec y_1 do řešení a vyškrtne tento sloupec společně s řádky, které pokrývá, z matice. Jelikož odebráním sloupce z matice mohou vzniknout dominantní řádky, opět se rekurzivně zavolají funkce řešící dominance a pokračuje se ve větvení. Pokud je matice prázdná, je nalezeno nové řešení, cena tohoto řešení je rovna součtu cen všech sloupců tvořících řešení. Toto řešení je označeno *Best* s cenou *cena(Best)*, jako zatím nejlepší nalezené řešení.

Algoritmus pokračuje rekurzivně vyškrtnutím sloupce y_1 z matice a pokračuje ve větvení. V případě výběru sloupce y_n se pokračuje ve větvení v případě, že cena (*cesta* (y_n))+ cena (y_n) < cena (*Best)*. V případě vyškrtnutí sloupce se testuje, zda podmínka cena (*cesta* (y_n)) < cena (*Best)* je splněna (cena může přesáhnout *cenu (Best)* díky výběru nezbytných sloupců při řešení dominancí). Pokud není podmínka splněna, je větev oříznuta, protože v této části vyhledávacího prostoru se řešení s menší cenou, než je cena(*Best*) určitě nenachází.

Algoritmus má jako druhý parametr strukturu ukazující na spojový seznam množiny sloupců, které byly při větvení vybrány do řešení. Z obou volání algoritmu se vybere to, které má nižší cenu. Princip funkce *Branch_And_Bound* (*CoveringMatrix * , Řešení* *) je popsán diagramem 4.1.

Funkce *Branch_And_Bound* (*CoveringMatrix * CM, Řešení *Sol*)

Vstup: Pokrývací matice
Výstup: Množina sloupců tvořící řešení, nebo ∅ pokud řešení nebylo nalezeno

```
1. Vyřeš_Dominance (CM , Sol);
2. if (CM) == ∅ {
                if  cena ( Sol ) < cena ( Best ) return newBest;
                        else return ∅
        }
        else {
                Vyber do řešení sloupec j;
                if cena ( cesta (j)) + cena (j) < cena ( Best )
                        Branch_and_Bound ( CM, Sol1 );
                Vškrtni sloupec j z řešení;
                if cena ( cesta (j)) < cena ( Best )
                        Branch_and_Bound ( CM, Sol2 );
```

```
        return min (Sol1,Sol2);
}
```

Diagram 4.1. Algoritmus Branch and Bound

Tato úprava rekurzivního algoritmu urychlí výpočet, avšak pro větší matice je výpočetní doba stále neúnosně velká. V tomto textu si dále popíšeme metody ořezávání vyhledávacího prostoru.

3.2 Spodní hranice

Při řešení problému pokrytí metodou větví a hranic popsanou výše vždy při výběru sloupce *y* do řešení před rekurzivním voláním na takto vzniklý podproblém testujeme podmínku, zda *cena (cesta (y)) + cena (y) < cena (Best)* Tato podmínka nám zaručuje, že budou ořezány větve vyhledávacího stromu, kde cena řešení přesahuje cenu již nalezeného řešení. Vyhledávací prostor se nám tedy zmenší. Hodnota *cena (cesta (y)) + cena (y)* se nazývá *spodní hranice řešení*. Pokud tedy známe spodní hranici podproblému, můžeme při platnosti podmínky ukončovat rekurzi.

Předpokládejme nyní, že víme, že minimální řešení problému pokrytí matice *A* je rovno číslu, označme ho K. Potom cena částečného řešení odpovídající uzlu *U* je rovna *cesta (U) + K*. Pokud tedy známe cenu řešení *Best*, můžeme zastavit rekurzi, pokud *cesta (U) + K* $\geq$ *cena (Best)*.

Definice 3.2.1 *Nechť <X,Y,cena> je problém pokrytí a nechť X' je podmnožina X taková, že pro libovolné dva různé prvky* x_1 *a* x_2 *množiny X', všechny prvky y které pokrývají* x_1 *nepokrývají* x_2 *a opačně. Množina X', kterou budeme nazývat nezávislá podmnožina X, poskytuje nám spodní hranici.*

Jinými slovy, předpokládejme, že v matici *A* se vyskytují množiny řádků tak, že žádný sloupec pokrývající řádek z jedné množiny nepokrývá žádný řádek z jiné množiny.

Z definice 3.2.1 vyplývá, že pokud máme K nezávislých podmnožin řádků, tzn., žádné dva řádky z různých podmnožin není možné pokrýt jedním sloupcem, musíme pro pokrytí vybrat minimálně |K| sloupců. Pokud se nám podaří nalézt maximální počet nezávislých podmnožin řádků,označme tento počet MSIR (Maximal Set of Independent Rows), z definice 3.2.1 je to mohutnost množiny X'. Navíc musíme počítat s tím, že pokud chceme pokrýt řádek $x_1 \in$ X', musíme pro jeho pokrytí vybrat sloupec *y* ohodnocený hodnotou *cena (y)*. Vybereme tedy „nejlevnější" možnost pokrytí tohoto řádku a přičteme k celkovému MSIR. MSIR nám tedy dává spodní hranici pro řešení problému pokrytí. Při větvení funkce *Branch_and_Bound* můžeme tedy ořezat větve s cenou kde platí *cena částečného řešení + |MSIR|* $\geq$ *cena (Best)*. Nový algoritmus reprezentovaný funkcí Branch_and_Bound je popsán diagramem 3.2.1.

Funkce *Branch_And_Bound* (*CoveringMatrix *CM , Řešení * Sol*)

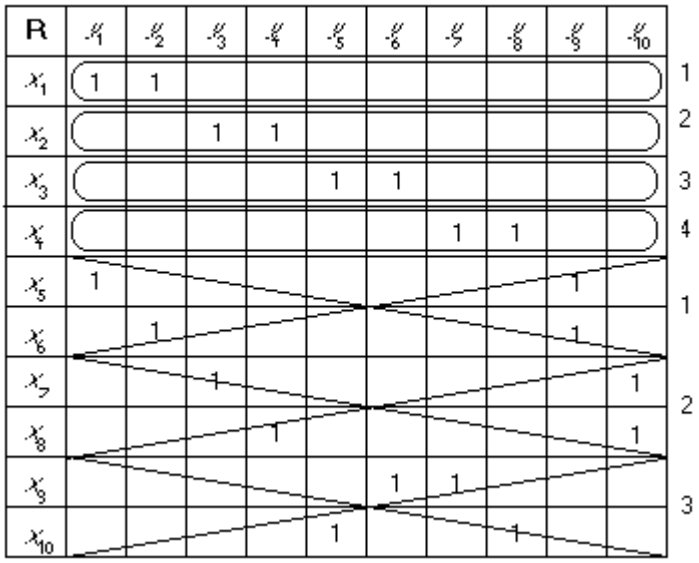
Vstup: Pokrývací matice
Výstup: Množina sloupců tvořící řešení, nebo ∅ pokud řešení nebylo nalezeno

```
1. Vyřeš_Dominance (CM , Sol);
2. if (CM) == ∅ {
            if  cena ( Sol ) < cena ( Best ) return newBest;
                else return ∅
    }
    else {
            Vyber do řešení sloupec j;
            Najdi MSIR pro vzniklý podproblém
            if cena ( cesta (j)) + cena (j) + |MSIR| < cena ( Best )
                Branch_and_Bound ( CM, Sol1 );
            Vškrtni sloupec j z řešení;
            Najdi MSIR pro vzniklý podproblém
            if cena ( cesta (j)) + |MSIR| < cena ( Best )
                Branch_and_Bound ( CM, Sol2 );
            return min (Sol1,Sol2);
    }
```

Diagram 3.2.1 Algoritmus Branch and Bound rozšířený o MSIR

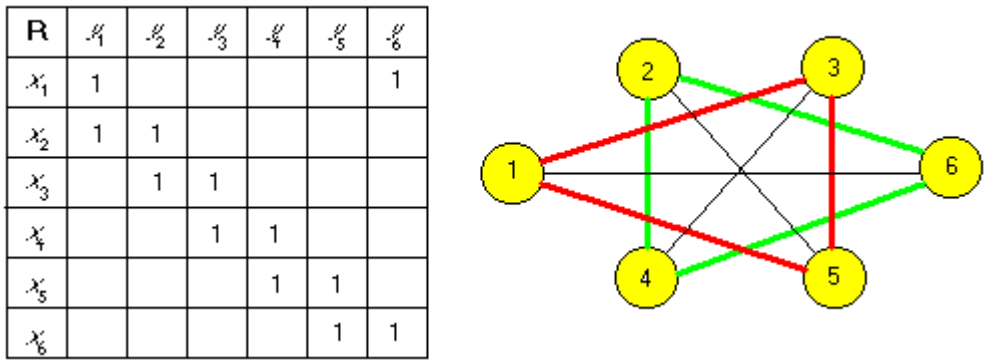
3.2.1 Maximální Množina Nezávislých Řádků

Na obrázku 3.2.1 tvoří MSIR řádky x_1,x_2,x_3 a x_4. Jsou to množiny řádků, pro které platí, že žádný sloupec v matici nepokrývá více řádků z MSIR. Jsou zde číselně znázorněny jednotlivé řádky tvořící MSIR, řádky škrtnuté s odpovídajícím číslem jsou závislé s řádkem stejného čísla, proto jsou v jednotlivých krocích z matice vyškrtnuty.



Obrázek 3.2.1 Maximální množina nezávislých řádků

Složitost nalezení MSIR je rovna složitosti problému nalezení největší kliky v grafu. Na obrázku 3.2.2 je znázorněn graf závislosti řádků, uzly jsou tvořeny čísly odpovídající číslům řádků, hrany v grafu znázorňují, že řádky x_i, x_j, odpovídající uzlům *i* a *j*, nejsou pokryty jedním sloupcem, tzn. Neexistuje sloupec k tak, že by platilo *A[i,k] = A[j,k] = 1*. Musíme tedy najít největší kliku v grafu. Vidíme, že musíme vybrat minimálně tři sloupce pro pokrytí [1,3,5] nebo [2,4,6].



Obrázek 3.2.2 Složitost nalezení MSIR

Složitost nalezení maximální množiny nezávislých řádků je rovna složitosti nalezení problému pokrytí. Hledáním přesné množiny nezávislých řádků bychom si časově nijak nepomohli.

Při hledání *MSIR* máme však ještě jiné možnosti, např. použití heuristiky, která vyhledá nějaké řešení v rozumném čase, avšak za cenu horšího řešení. Jelikož k nalezení minimálního řešení matice *A* nepotřebujeme přesně znát |*MSIR*|, stačí nám k jeho nalezení použití heuristiky. Nalezení menšího *MSIR* v tomto případě znamená pouze více větvení funkce *Branch_and_Bound*, neboť se podaří ořezat méně větví, kde se řešení nenachází. Nemá to tedy vliv na minimalitu řešení, prostor, kde se řešení nachází, se prohledá celý.

Pro výpočet jsem tedy zvolil heuristiku, kde výpočet probíhá v lineárním čase, za cenu neminimálního řešení. Diagram 3.2.2 znázorňuje funkci hledající *MSIR*.

Funkce *int NajdiMSIR (CoveringMatrix *CM, int MSIR)*

Vstup: Pokrývací matice
Výstup: Velikost MSIR

while (CM != ∅)
{
1. Najdi řádek *i* s minimálním počtem jedniček
2. Vyjmi řádek *i* z matice společně s řádky, které mají jedničky ve stejném sloupci jako *i*.
3. MSIR+=min(cena sloupců,které řádek pokrývají);

}

Diagram 3.2.2 Heuristika pro nalezeni MSIR

3.3 Zvyšování spodní hranice

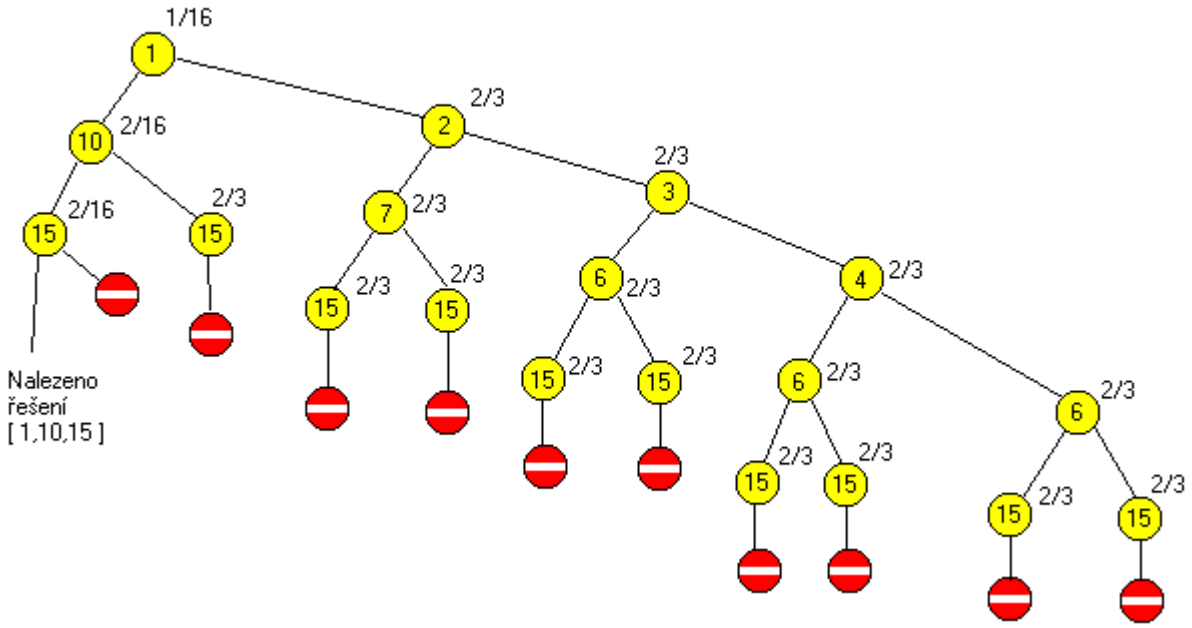
Uvažujme však matici na obrázku 3.3.1. Mohutnost MSIR je 1, zatímco velikost minimálního řešení je 3. MSIR tedy může být libovolně daleko od minimálního řešení.

R	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	s_9	s_{10}	s_{11}	s_{12}	s_{13}	s_{14}	s_{15}
x_1	1	1	1	1	1										
x_2	1					1	1	1	1						
x_3		1				1				1	1	1			
x_4			1				1			1			1	1	
x_5				1				1			1		1		1
x_6					1				1			1		1	1

Obrázek 3.3.1

Pro funkci *Branch_and_Bound* to znamená 29 volání, než se ujistíme,že se nikde nenalézá řešení s cenou menší než 3, které je nalezené už při třetím volání ! Na obrázku 3.3.2 je znázorněn graf volání *Branch_and_Bound*, hrany znamenají volání funkce, uzly jsou ohodnoceny hodnotou *(|cesta k uzlu| + spodní hranice / horní hranice)*. Vidíme, že po nalezení řešení po 3 voláních se musí algoritmus volat ještě 26krát, jen abychom se ujistili, že se nikde nenalézá lepší řešení, než již nalezené.

Podmínka zastavení rekurze je splněna, pokud *(horní hranice - |cesta k uzlu| - spodní hranice) ≤ 0*. Připomeňme, že *horní hranice* je pro nás cena nejlepšího zatím nalezeného pokrytí. Jediný způsob, jak zastavit rekurzi, pokud *(horní hranice - |cesta k uzlu| - spodní hranice) > 0*, je zlepšením spodní hranice.



Obrázek 3.3.2 Graf volání funkce Branch_and_Bound

3.4 Limitní spodní hranice

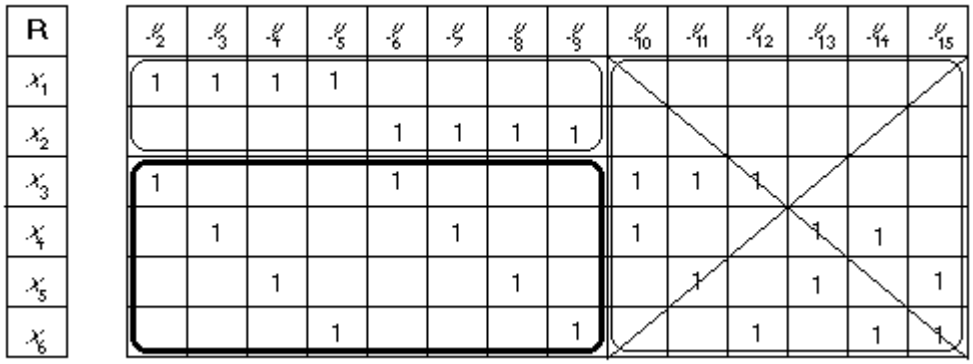
V tomto odstavci si ukážeme techniku, popsanou v [2], [6].

Definice 3.4.1: *Nechť <X,Y,cena> je problém pokrytí a nechť X' je nezávislá podmnožina X, popsaná v definici 3.2.1. Nechť lBound je spodní hranice nutná k pokrytí X, získaná díky X' a uBound je horní hranice. Označme Y' = { y ∈ Y | y ∩ X' = ∅ , cesta k uzlu + lBound + cena(y) ≥ uBound }. Pak A může být zmenšena na <X,Y-Y',cena>.*

Jinými slovy, *y* jsou všechny sloupce, které nepokrývají MSIR. Pokud je *(horní hranice – spodní hranice – (cesta k uzlu)) ≥ cena (y)*, všechny sloupce nepokrývající žádný řádek z MSIR můžou být z matice vyškrtnuty, a MSIR přepočítáno.

Zajímavá vlastnost, pokud aplikujeme limitní spodní hranici, zmenšením *<X,Y,cena>* na *<X,Y-Y',cena>* , je že rekurze je prakticky zastavena okamžitě, tzn., spodní hranice *<X,Y-Y',cena>* skoro pokaždé přesáhne horní hranici, nebo je nalezeno lepší řešení v *<X,Y-Y',cena>*.

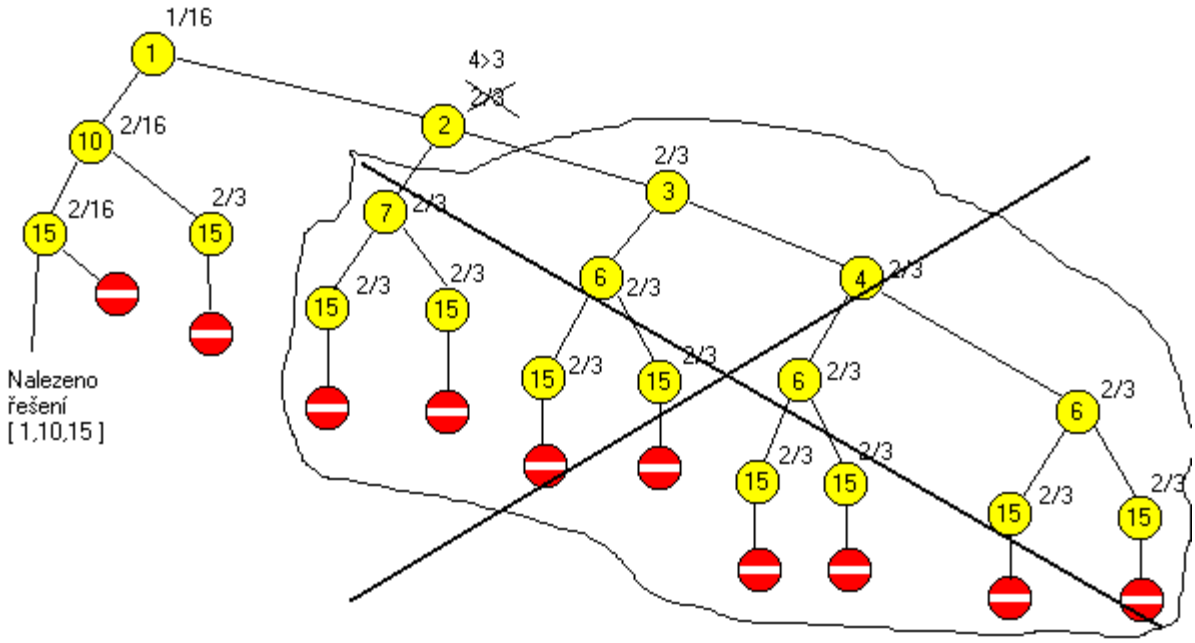
Pokud tedy pokračujeme v grafu 3.3.2 větvení nevybráním sloupce 1 do řešení, je splněna podmínka *(horní hranice – spodní hranice – (cesta k uzlu)) ≥ cena (y)*, můžeme vyškrtnout z matice všechny sloupce, které neprotínají MSIR a vypočteme nové MSIR. Obrázek 3.3.3 ukazuje redukovanou matici z obrázku 3.3.1.



Obrázek 3.3.3 – Získání nového MSIR

Tenkou čarou je vyznačeno původní MSIR, škrtnuty jsou sloupce, které neprotínají MSIR. Tlustou čarou pak nově vzniklé MSIR, vidíme, že z původních MSIR=2 máme MSIR = 4, což stačí na zastavení rekurze, protože lBound=4 > uBound = 3.

Na obrázku 3.3.4 vidíme, jak se sníží počet volání algoritmu. Škrtnutá ohraničená část grafu bude při prohledávání ořízntuta.



Obrázek 3.3.4 – Počet volání po vypočtení nového MSIR

3.5 Řešení pomocí heuristik

K řešení problému pokrytí můžeme použít heuristiku. Heuristiky k nalezení řešení jsou popsané v [3], [4], [5].

Přirozená heuristika spočívá ve výběru prvku *y* který pokrývá největší počet prvků X. Funkce *NaturalHeuristic (CoveringMatrix * , Reseni *)* vybere z matice sloupec který pokrývá nejvíce řádků, vyjme ho společně s řádky z matice a takto pokracuje dokud není matice prázdná, do struktury *Reseni* se zapíše odkaz na seznam sloupců vybraných do řešení.

Funkce *NaturalHeuristic* (*CoveringMatrix * CM, Řešení *Sol*)

Vstup: Pokrývací matice
Výstup: Množina sloupců tvořící řešení

while (CM == ∅) {
1. Z jisti počet jedniček ve všech sloupcích
2. Vyber sloupec s nejvíce jedničkami, přidej ho do řešení a vyjmi společne s řádky, které pokrývá, z matice

}

Diagram 3.5.1 Přirozená heuristika

Nicméně pokusné výsledky ukazují, že to není nejlepší heuristika. O moc lepší je heuristika navrhovaná v [5],[7], je založena na výpočtu příspěvku jednotlivých sloupců (pomocí cenové funkce), a na základě vypočtených hodnot se rozhoduje o jejich zahrnutí do řešení.

Definice 3.5.1: *Síla pokrytí SP řádku x_i je definována jako*

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

Síla pokrytá SP řádku x_i je tím větší, čím méně sloupců řádek pokrývá. Pokud je pokrytý pouze jedním sloupcem, SP=1 a sloupec je vybrán do řešení jako nezbytný.

Definice 3.5.2: *Sloupcový příspěvek SP sloupce y_j je definován jako*

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

SP se tím více zvyšuje, čím více sloupec pokrývá řádků, které jsou pokryty nejmenším počtem sloupů. Ukázka výpočtu je znázorněna na obrázku 3.5.1, algoritmus je znázorněn diagramem 4.3.

R	s_1	s_2	s_3	s_4	s_5	SP(x_i)
x_1	1	1	1	1		1/4=0.25
x_2	1	1	1	1		1/4=0.25
x_3	1			1		1/2=0.50
x_4	1			1	1	1/3=0.33
x_5	1			1		1/2=0.50
x_6					1	1/1=1.00
SP(y_j)	1.83	0.50	0.50	1.83	1.33	

Obrázek 3.5.1 – Ukázka výpočtu sloupcového příspěvku

Funkce *ContribAddHeuristic* (*CoveringMatrix * CM, Řešení *Sol*)

Vstup: Pokrývací matice
Výstup: Množina sloupců tvořící řešení

while (CM == ∅) {
pro všechna $y_j \in Y$ Spočítej SP(y_j)
Vyber sloupec s největším SP, přidej ho do řešení a vyjmi společne s řádky, které pokrývá, z matice
}

Diagram 3.5.2 Heuristika založená na výběru sloupců s ohledem na „kvalitu“ pokrytí

Počet jedniček v řádku vypočítáme vždy před voláním ContribAddHeuristic, neboť se při běhu algoritmu nemění. Všechny řádky pokryté sloupcem jsou z matice vyškrtnuty. Musíme však přepočítat SP sloupců, které pokývají řádky pokryté sloupcem, který má být vyškrtnut z matice, protože se jejich příspěvek sníží právě o SP vyjmutých řádků. SP se tedy přepočítává maximálně *n*-krát, máme tedy lineární složitost tohoto algoritmu, *O(n)*, kde *n* je počet sloupců.

U obou heuristik nastává problém, pokud sloupec má stejné ohodnocení, jako sloupec, s kterým ho porovnáváme ve snaze najít sloupec s větším ohodnocením. Pak se musíme rozhodnout, který sloupec vybrat. Můj algoritmus se náhodně rozhoduje, jestli vezme první sloupec, nebo druhý. Proto je dobré volat heuristiku několikrát, a z těchto volání vybrat nejmenší řešení.

Zbývá ještě otázka,zda pomocí heuristiky řešit celou vstupní matici, nebo zkusit jejich aplikaci pouze na cyklické jádro. Každá z funkcí představující heuristiku má tedy jako poslední parametr možnost použití funkce řešící dominance, implicitní nastavení neřeší dominance. Hlavička funkce tedy vypadá takto

ContribAddHeuristic (*CoveringMatrix * CM, Řešení *Sol, int Dominance=0*)

Všechny testované matice jsou několikrát řešeny pomocí heuristik bez řešení dominancí, potom s odstraněním dominancí.

Přirozená heuristika je založena na výběru sloupce do řešení, následném vyškrtnutí z matice a pokračování, dokud není matice prázdná, složitost algoritmu je tedy *O(n)*, kde *n* je počet sloupců v matici, ovšem za cenu neminimálního řešení. Složitost Contrib Add heuristiky je rovněž založena na výběru sloupce do řešení a jeho následném vyškrtnutí z matice, před každým výběrem se však provádí výpočet příspěvku sloupce, s výše zmíněnou složitostí O(n). Výsledná složitost algoritmu je tedy $O(n^2)$.

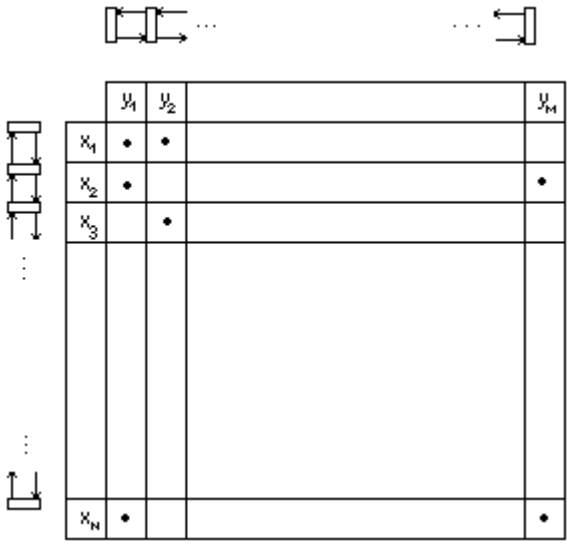
4 Implementace

4.1 Vnitřní paměťová reprezentace

Pro vnitřní reprezentaci potřebujeme strukturu, kde budeme mít uloženou matici *A* o rozměrech *MxN* s ohledem na definici 2.1.

Musíme počítat s tím, že budeme často vyškrtávat sloupce a řádky z matice, struktura by tedy mela umožňovat snadný přechod přes vynechané sloupce a řádky. Pokud bychom sloupci, resp. řádku přiřadili atribut zda je, či není v matici, museli bychom pokaždé testovat jeho hodnotu a to by bylo časově náročné. Potřebujeme znát offset, o kolik sloupců se máme v matici posunout od sloupce j ke sloupci k (přeskočit všechny vyškrtnuté sloupce).

Použijeme tedy následující strukturu. Matice *A* o rozměrech *MxN* je uložena v paměti jako jednorozměrné homogenní pole prvků typu *char*, vkládání prvků do pole se provádí po řádcích, tzn., adresa prvku V_{ij} = **(A+i*M+j)*. K této struktuře jsou vytvořeny dva obousměrně zřetězené spojové seznamy o *M*, resp., *N* prvcích,každý prvek seznamu nese informace o offsetu adresy, kde je uložen řádek, resp., sloupec matice, viz obrázek Tato reprezentace je vhodná, protože při redukování matice o řádek nebo sloupec jen přepíšeme ukazatele ve spojovém seznamu, toto má pozitivní vliv na rychlost vyhledávání prvků. Adresa prvku V_{ij} je tedy **(A + Seznam_Sloupcu->Offset + Seznam_Radku->Offset)*. Navíc při takovéto adresaci prvků pole pouze sčítáme, což má také značný vliv na urychlení.



Obrázek 4.1 – Vnitřní reprezentace matice pokrytí

4.2 Popis ovládání programu

Vstup programu je realizován vstupním souborem, kde na začátku je uveden rozměr matice, čísla oddělená mezerami, a pak následuje matice uložená po řádcích, první řádek obsahuje ceny sloupců. Čísla v řádcích jsou odděleny rovněž mezerami. Vstupní soubor se zadává z příkazové řádky.

Výstup se vypisuje na obrazovku a zároveň do souboru, který může být uvedený jako druhý parametr programu z příkazové řádky. Implicitní název výstupního souboru je „out.dat“.

Program vyřeší vstupní matici nejdříve pomocí heuristik a potom pomocí exaktních metod.

Názvy funkcí implementovaných v programu pro řešení problému pokrytí:

void NaturalHeuristic (struct CoveringMatrix *CM,struct HeaderSol *SolUCP,int Dominance=0)
Funkce přirozená heuristika, popsaná v odstavci 3.5.

void ContribAddHeuristic (struct CoveringMatrix *CM,struct HeaderSol *SolUCP,int Dominance=0)
Heuristika,která vybírá sloupce podle vypočteného sloupcového příspěvku,viz odstavec 3.5.

int BAB (struct CoveringMatrix *CM,struct HeaderSol *SolUCP)
Metoda větví a hranic,popsaná v odstavci 3.1.

int BABMSIR (struct CoveringMatrix *CM,struct HeaderSol *SolUCP)
Vylepšená metoda větví a hranic, popsaná v odstavci 3.2.

int BABLLB (struct CoveringMatrix *CM,struct HeaderSol *SolUCP)
Vylepšená metoda větví a hranic, popsaná v odstavci 3.4.

5 Experimentální výsledky

V této části textu je uvedeno vyhodnocení algoritmů řešících problém pokrytí. Bylo vyhodnoceno několik různých typů matic různých velikostí, každá matice je označena svým názvem a velikostí. Jsou zde srovnány obě heuristiky, a je porovnána kvalita při odstranění dominancí a bez odstranění dominancí. Některé matice byly řešeny exaktními metodami, výsledky jsou rovněž uvedeny, kvalita heuristik byla porovnána k hodnotě získané pomocí těchto exaktních metod. Hustota matice je počítána jako *1 / (velikost / počet jedniček)*. K testování byl použit počítač AMD Duron 850Mhz s 256Mb Ram.

Matice byly získány programem BOOM, popsaného v [7], minimalizací PLA souborů uvedených ve sloupci *Název*. Minimalizace programem BOOM je do jisté míry randomizovaná, počet generovaných sloupců záleží na počtu iterací minimalizačního algoritmu. Ve sloupci *Iterace* je uveden počet iterací, pro které byla minimalizace prováděna. Ve sloupci *Rozměr* je uveden rozměr matice ve formátu *Počet sloupců x Počet řádků*. Sloupec *Hustota* zobrazuje poměr počtu jedniček k velikosti matice, *Hustota=1/(Velikost matice/Počet jedniček)*. Sloupce v maticích byly při generování programem BOOM ohodnoceny cenou, která vyjadřuje počet literálů. Cena je uvedena v prvním řádku vygenerované matice. Cenu řešení tvoří součet těchto cen sloupců zahrnutých do řešení.

5.1 Heuristiky

Matice uvedené v tabulce byly řešeny nejprve heuristikami. Je zde uveden čas běhu algoritmu, formát [sec:set], kde tato položka chybí, byl čas neměřitelně malý.

Jelikož řešení heuristikami probíhá do jisté míry náhodně, každá heuristika byla spuštěna pro 500 průchodů.Byly vždy měřeny nejdříve celé matice, a pak pomocí odstranění dominancí byla testována aplikace heuristik pouze na cyklické jádro. Funkce reprezentující heuristiky tedy nejdříve zavolaly funkce pro odstranění dominantních řádků a sloupců, během těchto dominancí byly přidány do řešení nezbytné sloupce. Zbylé cyklické jádro bylo pak dále řešeno pomocí heuristiky.

Z takto získaných hodnot je uvedena minimální nalezená hodnota řešení, maximální hodnota a průměrná hodnota, získaná jako (součet všech cen / počet průchodů). Kvalita řešení je posuzována s ohledem na výsledek získaný pomocí exaktní metody, jako relativní odchylka od této hodnoty, tzn., ((prům. cena heuristiky – cena exakt.) / cena exakt.). Kde není hodnota uvedena, matice nebyla měřena exaktní metodou z důvodů časové náročnosti.

Při testování bez odstranění dominancí byla ve 23,07% úspěšnější přirozená heuristika, ve 46,15% byla úspěšnější Contrib Add heuristika. Řešení dominancí bylo u přirozené heuristiky úspěšnější ze 64,1%, horší ze 12,8%, u Contrib Add heuristiky bylo úspěšnější z 58,9%, horší nebylo nikdy. Průměrná relativní odchylka od hodnoty získané pomocí heuristik je v tabulce 5.1.1. Všechny naměřené hodnoty jsou uvedené v následujících tabulkách.

	Heuristika			
	Bez dominancí		S řešením dominancí	
	Přirozená	Contrib Add	Přirozená	Contrib Add
rel.odchylka	0.0673	0.0552	0.0272	0.0272

Tabulka 5.1.1 Průměrná relativní odchylka heuristik od exaktní metody

	Bez odstranění dominancí	S odstraněním dominancí	
	% úspěšnější	% úspěšnější	%neúspěchu
Přirozená heuristika	23,07	64,1	12,8
Contrib Add heuristika	46,15	58,9	nikdy

Tabulka 5.1.2 Procento úspěšnosti heuristik

Přirozená heuristika bez odstranění dominancí								
					Cena řešení			
Název	Iterace	Rozměr	Hustota	Čas	Min	Max	Průměr	Odchylka
10x100	1	21x56	0.081633		80	80	80	0
10x100	100	147x56	0.074101	00:001	71	74	72	0,241379
10x200	1	45x98	0.044898		166	171	166	0,006061
10x200	100	248x98	0.042092	00:004	151	153	152	0,125926
10x2x200	1	135x203	0.024667	00:002	336	342	339	0,069401
10x2x200	100	574x203	0.021524	00:051	290	294	293	
10x300	1	88x149	0.032032		266	266	266	0,055556
10x300	100	349x149	0.027596	00:021	246	246	246	
10x300_40%	1	28x297	0.115801		52	52	52	0,04
10x300_40%	100	32x297	0.113215		52	52	52	0,04
10x500	1	131x241	0.019955	00:002	501	514	510	0,069182
10x500	100	407x241	0.018932	00:045	480	480	480	
15x300	1	95x160	0.037895		243	248	244	0,065502
15x300	100	2201x160	0.032900	03:081	157	159	158	
200x200	1	15x108	0.099386		65	65	65	0
200x200	100	1581x108	0.102894	00:091	51	51	51	
20x100_30%	1	25x34	0.045882		137	137	137	0
20x100_30%	100	44x34	0.055481		129	129	129	0,131579
20x200	1	30x100	0.064		117	117	117	0
20x200	100	1718x100	0.053824	01:040	77	77	77	
20x200_20%	1	64x98	0.017857	00:001	421	427	423	0,004751
20x200_20%	100	402x98	0.022794	00:009	310	350	333	
20x200_30%	1	86x108	0.012274		610	610	610	0
20x200_30%	100	206x108	0.01735	00:002	485	503	494	0,053305
20x20x20	1	82x204	0.050753	00:001	91	92	91	0,123457
20x20x20	100	658x204	0.049251	00:079	73	73	73	
20x20x30_50%	1	122x153	0.021536		205	207	206	0,019802
20x20x30_50%	100	289x153	0.023679	00:007	153	168	158	
20x20x50_45%	1	243x282	0.010463	00:007	502	519	508	0,038855
20x20x50_45%	100	778x282	0.012133	02.000	329	354	333	
20x300	1	62x156	0.039702		220	225	223	0,037209
20x300	100	3699x156	0.036765	11.014	131	134	131	
50x10x50	1	80x246	0.029573		173	177	173	0,005814
50x10x50	100	2336x246	0.029304	09:014	106	123	116	
50x20x100_50%	1	463x514	0.004072	00:043	1914	1949	1933	
50x20x100_50%	100	2911x514	0.005839	49:017	810	871	844	
50x50	1	5x27	0.281481		16	16	16	0
50x50	100	213x27	0.226917		12	16	13	0,083333

Tabulka 5.1.3 Výsledky - Přirozená heuristika bez odstranění dominancí

Contrib Add heuristika bez odstranění dominancí								
					Cena řešení			
Název	Iterace	Rozměr	Hustota	Čas	Min	Max	Průměr	Odchylka
10x100	1	21x56	0.081633		80	80	80	0
10x100	100	147x56	0.074101	00:001	67	67	67	0,155172
10x200	1	45x98	0.044898		166	166	166	0,006061
10x200	100	248x98	0.042092	00:004	155	159	157	0,162963
10x2x200	1	135x203	0.024667	00:002	337	337	337	0,063091
10x2x200	100	574x203	0.021524	00:051	305	305	305	
10x300	1	88x149	0.032032		264	266	264	0,047619
10x300	100	349x149	0.027596	00:021	240	245	242	
10x300_40%	1	28x297	0.115801		52	52	52	0,04
10x300_40%	100	32x297	0.113215		52	52	52	0,04
10x500	1	131x241	0.019955	00:002	507	515	511	0,071279
10x500	100	407x241	0.018932	00:045	489	489	489	
15x300	1	95x160	0.037895		233	236	234	0,021834
15x300	100	2201x160	0.032900	03:081	155	159	156	
200x200	1	15x108	0.099386		65	65	65	0
200x200	100	1581x108	0.102894	00:091	47	47	47	
20x100_30%	1	25x34	0.045882		137	137	137	0
20x100_30%	100	44x34	0.055481		116	116	116	0,017544
20x200	1	30x100	0.064		117	117	117	0
20x200	100	1718x100	0.053824	01:040	74	74	74	
20x200_20%	1	64x98	0.017857	00:001	421	421	421	0
20x200_20%	100	402x98	0.022794	00:009	305	313	310	
20x200_30%	1	86x108	0.012274		610	610	610	0
20x200_30%	100	206x108	0.01735	00:002	490	502	495	0,055437
20x20x20	1	82x204	0.050753	00:001	89	89	89	0,098765
20x20x20	100	658x204	0.049251	00:079	82	82	82	
20x20x30_50%	1	122x153	0.021536		204	207	206	0,019802
20x20x30_50%	100	289x153	0.023679	00:007	151	162	155	
20x20x50_45%	1	243x282	0.010463	00:007	499	510	504	0,030675
20x20x50_45%	100	778x282	0.012133	02.000	349	349	349	
20x300	1	62x156	0.039702		226	226	226	0,051163
20x300	100	3699x156	0.036765	11.014	125	130	127	
50x10x50	1	80x246	0.029573		177	177	177	0,02907
50x10x50	100	2336x246	0.029304	09:014	112	121	115	
50x20x100_50%	1	463x514	0.004072	00:043	1903	1921	1912	
50x20x100_50%	100	2911x514	0.005839	49:017	798	873	837	
50x50	1	5x27	0.281481		16	16	16	0
50x50	100	213x27	0.226917		13	14	13	0,083333

Tabulka 5.1.4 Výsledky - Contrib Add heuristika bez odstranění dominancí

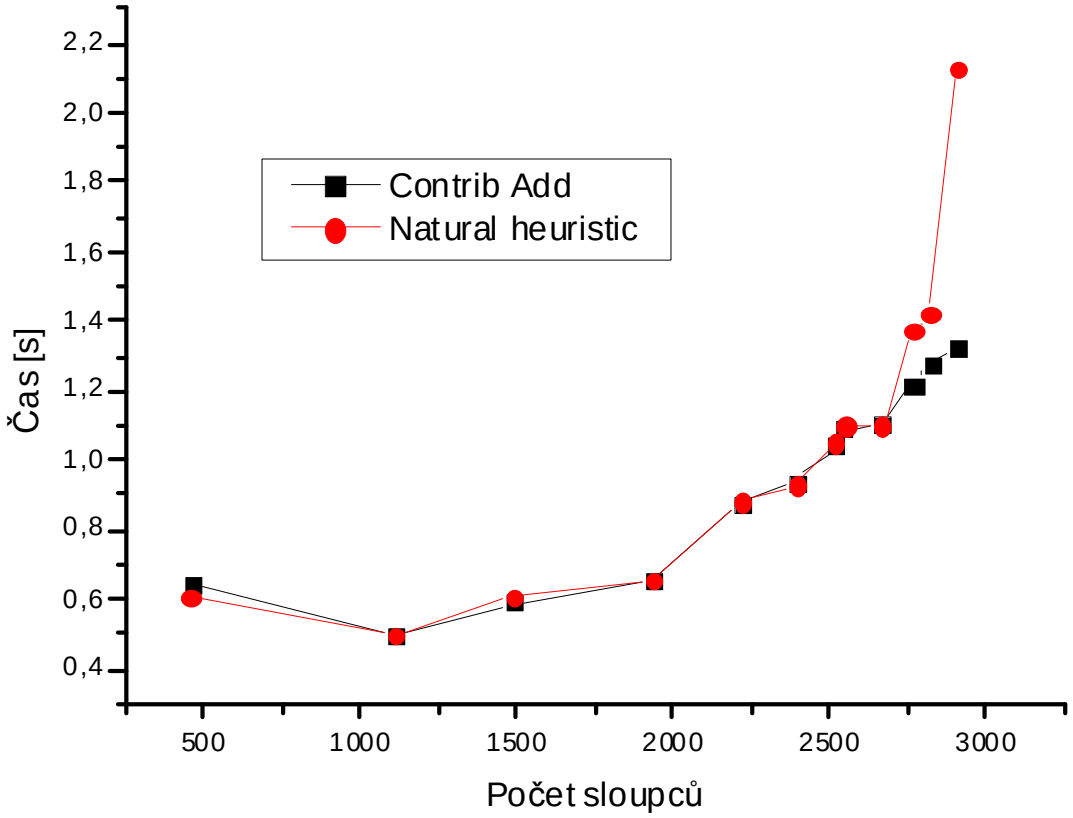
Přirozená heuristika s odstraněním dominancí								
					Cena řešení			
Název	Iterace	Rozměr	Hustota	Čas	Min	Max	Průměr	Odchylka
10x100	1	21x56	0.081633		80	80	80	0
10x100	100	147x56	0.074101	00:001	71	72	71	0,224138
10x200	1	45x98	0.044898		165	165	165	0
10x200	100	248x98	0.042092	00:004	150	154	152	0,125926
10x2x200	1	135x203	0.024667	00:002	324	328	325	0,025237
10x2x200	100	574x203	0.021524	00:051	302	304	302	
10x300	1	88x149	0.032032		252	252	252	0
10x300	100	349x149	0.027596	00:021	221	236	229	
10x300_40%	1	28x297	0.115801		50	50	50	0
10x300_40%	100	32x297	0.113215		50	50	50	0
10x500	1	131x241	0.019955	00:002	477	477	477	0
10x500	100	407x241	0.018932	00:045	464	478	472	
15x300	1	95x160	0.037895		229	229	229	0
15x300	100	2201x160	0.032900	03:081	151	165	163	
200x200	1	15x108	0.099386		65	65	65	0
200x200	100	1581x108	0.102894	00:091	49	51	49	
20x100_30%	1	25x34	0.045882		137	137	137	0
20x100_30%	100	44x34	0.055481		116	116	116	0,017544
20x200	1	30x100	0.064		117	117	117	0
20x200	100	1718x100	0.053824	01:040	77	77	77	
20x200_20%	1	64x98	0.017857	00:001	421	421	421	0
20x200_20%	100	402x98	0.022794	00:009	307	331	319	
20x200_30%	1	86x108	0.012274		610	610	610	0
20x200_30%	100	206x108	0.01735	00:002	492	503	497	0,059701
20x20x20	1	82x204	0.050753	00:001	81	84	83	0,024691
20x20x20	100	658x204	0.049251	00:079	66	72	69	
20x20x30_50%	1	122x153	0.021536		202	202	202	0
20x20x30_50%	100	289x153	0.023679	00:007	149	152	150	
20x20x50_45%	1	243x282	0.010463	00:007	492	499	495	0,01227
20x20x50_45%	100	778x282	0.012133	02.000	333	346	337	
20x300	1	62x156	0.039702		215	216	215	0
20x300	100	3699x156	0.036765	11.014	131	134	132	
50x10x50	1	80x246	0.029573		172	172	172	0
50x10x50	100	2336x246	0.029304	09:014	106	123	115	
50x20x100_50%	1	463x514	0.004072	00:043	1898	1907	1903	
50x20x100_50%	100	2911x514	0.005839	49:017	800	813	807	
50x50	1	5x27	0.281481		16	16	16	0
50x50	100	213x27	0.226917		12	12	12	0

Tabulka 5.1.5 Výsledky - Přirozená heuristika s odstraněním dominancí

Contrib Add heuristika s odstraněním dominancí								
					Cena řešení			
Název	Iterace	Rozměr	Hustota	Čas	Min	Max	Průměr	Odchylka
10x100	1	21x56	0.081633		80	80	80	0
10x100	100	147x56	0.074101	00:001	66	68	67	0,155172
10x200	1	45x98	0.044898		165	165	165	0
10x200	100	248x98	0.042092	00:004	146	166	155	0,148148
10x2x200	1	135x203	0.024667	00:002	318	327	321	0,012618
10x2x200	100	574x203	0.021524	00:051	292	296	293	
10x300	1	88x149	0.032032		252	252	252	0
10x300	100	349x149	0.027596	00:021	229	239	233	
10x300_40%	1	28x297	0.115801		50	50	50	0
10x300_40%	100	32x297	0.113215		50	50	50	0
10x500	1	131x241	0.019955	00:002	477	477	477	0
10x500	100	407x241	0.018932	00:045	451	453	451	
15x300	1	95x160	0.037895		229	229	229	0
15x300	100	2201x160	0.032900	03:081	155	159	156	
200x200	1	15x108	0.099386		65	65	65	0
200x200	100	1581x108	0.102894	00:091	47	47	47	
20x100_30%	1	25x34	0.045882		137	137	137	0
20x100_30%	100	44x34	0.055481		116	116	116	0,017544
20x200	1	30x100	0.064		117	117	117	0
20x200	100	1718x100	0.053824	01:040	74	74	74	
20x200_20%	1	64x98	0.017857	00:001	421	421	421	0
20x200_20%	100	402x98	0.022794	00:009	299	323	310	
20x200_30%	1	86x108	0.012274		610	610	610	0
20x200_30%	100	206x108	0.01735	00:002	486	496	490	0,044776
20x20x20	1	82x204	0.050753	00:001	82	82	82	0,012346
20x20x20	100	658x204	0.049251	00:079	66	69	67	
20x20x30_50%	1	122x153	0.021536		202	202	202	0
20x20x30_50%	100	289x153	0.023679	00:007	147	147	147	
20x20x50_45%	1	243x282	0.010463	00:007	491	498	494	0,010225
20x20x50_45%	100	778x282	0.012133	02.000	323	335	328	
20x300	1	62x156	0.039702		216	216	216	0,004651
20x300	100	3699x156	0.036765	11.014	125	128	126	
50x10x50	1	80x246	0.029573		172	172	172	0
50x10x50	100	2336x246	0.029304	09:014	112	119	115	
50x20x100_50%	1	463x514	0.004072	00:043	1896	1905	1901	
50x20x100_50%	100	2911x514	0.005839	49:017	798	807	802	
50x50	1	5x27	0.281481		16	16	16	0
50x50	100	213x27	0.226917		13	14	13	0,083333

Tabulka 5.1.6 Výsledky - Contrib Add heuristika s odstraněním dominancí

Dále jsem zde uvedl závislost času běhu heuristik na počtu sloupců v matici. Minimalizací vstupního souboru 50x20x100_50% pomocí programu BOOM byly generovány matice s různým počtem sloupců, volbou různého počtu iterací. Výsledný graf je na obrázku 5.1.7



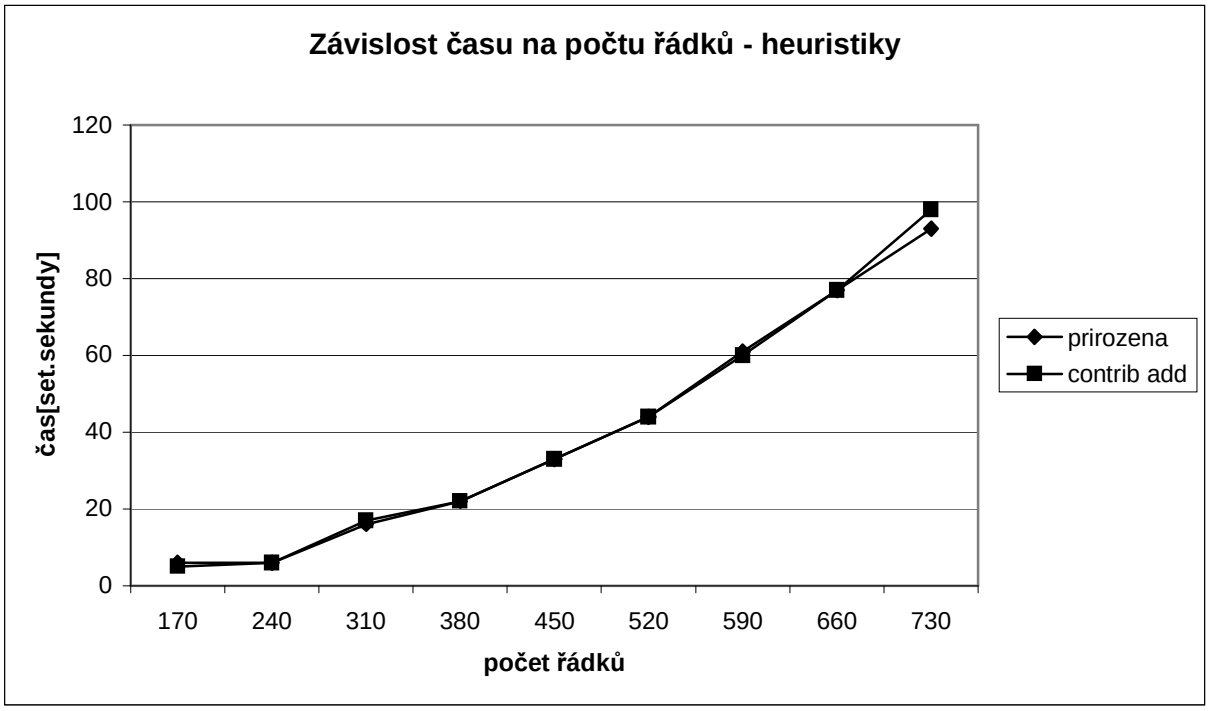
Obrázek 5.1.7 Graf závislosti času běhu algoritmu na počtu sloupců

V grafu vidíme lineární složitost přirozené heuristiky v závislosti na počtu sloupců, přirozená heuristika je založená na výběru sloupce do řešení a vyškrtnutí z matice společně s řádky,které pokrývá, takto se pokračuje,dokud není matice prázdná. Složitost Contrib Add heuristiky je rovněž založena na výběru sloupce do řešení a jeho následném vyškrtnutí z matice, před každým výběrem se však provádí výpočet příspěvku sloupce, se složitostí O(n). Výsledná složitost algoritmu, jak je patrné i z grafu, je tedy $O(n^2)$.

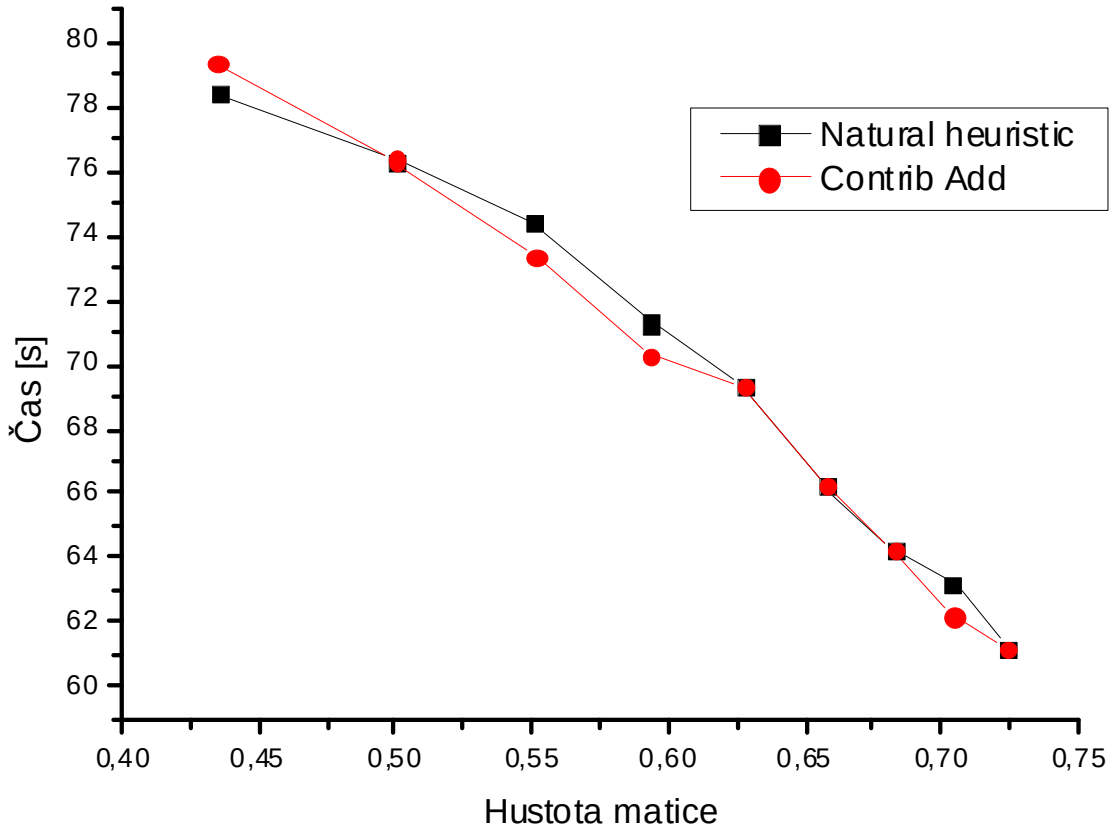
Contrib Add heuristika je tedy časově náročnější, než přirozená heuristika, ve většině případů však dává přesnější výsledky. Ale pro velké matice je z časových důvodů stále ještě použitelná. Při aplikaci heuristik pouze na cyklické jádro časová náročnost silně roste, jak je vidět v tabulkách 5.1.3 – 5.1.6, je to způsobeno časem potřebným k odstranění dominantních řádků a sloupců, jejichž operační složitost je $O(n^2)$, kde *n* je počet řádků, resp. sloupců.

Na obrázku 5.1.8 je vidět časová závislost heuristik na počtu řádků, počet sloupců je 500. Obě heuristiky se v této závislosti příliš neliší, jejich cenové funkce přepočítávají pouze

sloupcový příspěvek. Zvyšující se počet řádků tedy znamená přibližně stejný nárůst času u obou heuristik. Měření probíhalo heuristikami bez odstranění dominantních sloupců a řádků.



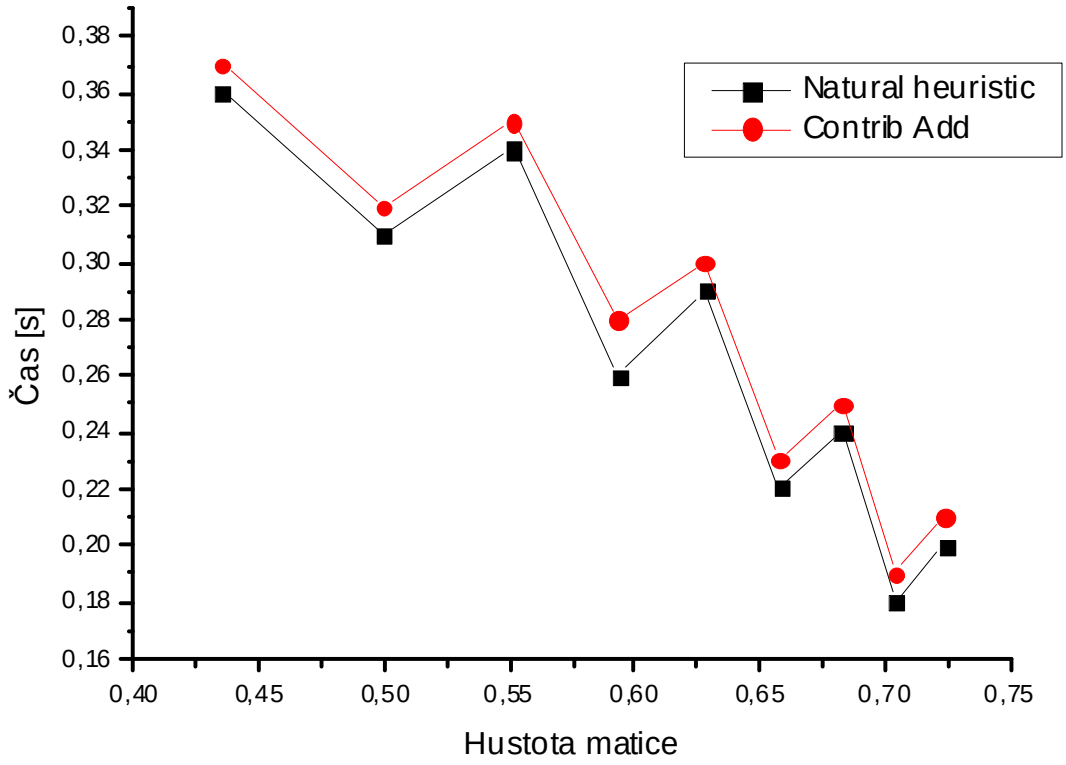
Obrázek 5.1.8 Graf závislosti času běhu algoritmu na počtu sloupců



Obrázek 5.1.9 Časová závislost heuristik na hustotě matice o velikosti 1000x1500 – s odstraněním dominancí.

Na obrázku 5.1.9 vidíme časovou závislost heuristik na hustotě matice o rozměrech 1000x1500, je vidět, že při zvyšující se hustotě výpočetní čas klesá, je to zřejmě způsobeno vyšší pravděpodobností výskytu dominantních řádků a sloupců, po jejichž odstranění je zbylá matice menší. Měření probíhalo heuristikami s odstraněním dominantních sloupců a řádků.

Na obrázku 5.1.10 vidíme časovou závislost na hustotě matice pro heuristiky s odstraněním dominantních řádků a sloupců. Výpočetní čas není ovlivněn výskytem dominancí, záleží pouze na kvalitě vybraného sloupce cenovou funkcí, matice se při každém kroku zmenší o počet řádků pokrytý vybraným sloupcem. Časová závislost tedy může být i nepatrně vyšší u matice s větší hustotou, jak je vidět z grafu.



Obrázek 5.1.10 Časová závislost heuristik na hustotě matice o velikosti 1000x1500 – bez odstranění dominancí.

5.2 Exaktní metody

Matice byly rovněž řešeny pomocí exaktních metod, pokud to dovolovala časová náročnost. V tabulce 5.2.1 jsou uvedené názvy matic a metody, které byly použity na jejich řešení. Je vidět, že časový rozdíl mezi výsledky je obrovský. To je způsobeno exponenciální složitostí problému. V některých případech však došlo k vyřešení problému pomocí dominancí (byly do řešení vybrány nezbytné sloupce), cyklické jádro bylo prázdné. U takových matic je ve sloupci počet volání je uvedena 1. Některé matice nebylo možné otestovat všemi metodami z důvodů časové náročnosti. Sloupec Limitní spodní hranice znázorňuje výsledky získané pomocí metody větví a hranic vylepšenou o vypočítávání limitní spodní hranice, sloupec MSIR znázorňuje výsledky získané pomocí metody větví a hranic vylepšenou o výpočet maximální množiny nezávislých řádků. Vidíme, že některé uvedené matice není nijak časově náročné vyřešit pomocí metody větví a hranic s vypočítáváním limitní spodní hranice, zatímco metoda s výpočtem MSIR je již časově náročnější. Při srovnání s obyčejnou metodou větví a hranic je rozdíl obrovský, metoda je již téměř nepoužitelná pro svou časovou náročnost. Čas uvedený v tabulce je ve formátu *hh:mm:ss.*

čas=hh:mm:ss									
				B&B s Limitní spodní hranicí			**B&B s MSIR**		
Matice	Iterace	Rozměr	Hustota	Čas	Cena	Volání	Čas	Cena	Volání
10x100	1	21x56	0.081633		80	1			
10x100	100	147x56	0.074101	0:51	58	41836	13:45	58	399256
*10x100	100	147x56	0.074101	0:33	14	4761	3:39	14	43028
10x200	1	45x98	0.044898		165	1			
10x200	100	248x98	0.042092	7:28:23	135	4304307			
10x2x200	1	135x203	0.024667		317	110	0:01	317	335
10x300	1	88x149	0.032032		252	1			
10x300_40%	1	28x297	0.115801		50	1		50	1
10x300_40%	100	32x297	0.113215		50	1		50	1
10x500	1	131x241	0.019955		477	36		477	51
15x300	1	95x160	0.037895		229	13		229	19
200x200	1	15x108	0.099386		65	1			
20x100_30%	1	25x34	0.045882		137	1			
20x100_30%	100	44x34	0.055481		114	30		114	68
20x200	1	30x100	0.064		117	1			
20x200_20%	1	64x98	0.017857		421	1		421	1
20x200_30%	1	86x108	0.012274		610	1		610	1
20x200_30%	100	206x108	0.01735	1:23:49	469	1461322			
20x20x20	1	82x204	0.050753		81	15		81	21
20x20x30_50%	1	122x153	0.021536		202	1517	0:02	202	16266
20x20x50_45%	1	243x282	0.010463	0:34	489	9475	2:53	489	78050
20x300	1	62x156	0.039702		215	3		215	2
50x10x50	1	80x246	0.029573		172	1		172	1
50x50	1	5x27	0.281481		16	1		16	1
50x50	100	213x27	0.226917	0:04	12	2134	0:48	12	86815

*Byla uvažována cena = 1 pro všechny sloupce v matici.

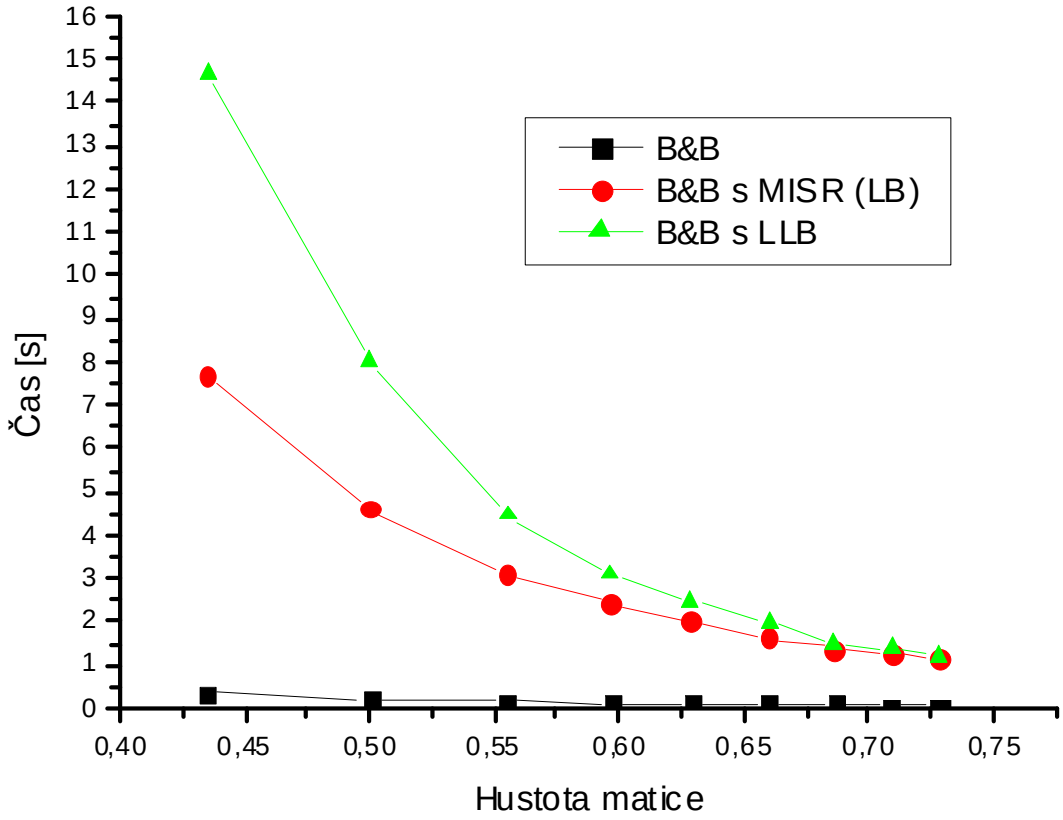
Tabulka 5.2.1 Výsledky vylepšených exaktních metod

V tabulce 5.2.2 jsem uvedl výsledky metody větví a hranic s výpočtem limitní spodní hranice v porovnání s metodou větví a hranic bez vylepšení. Nebyly testovány matice, které to neumožňovaly z důvodu časové náročnosti.

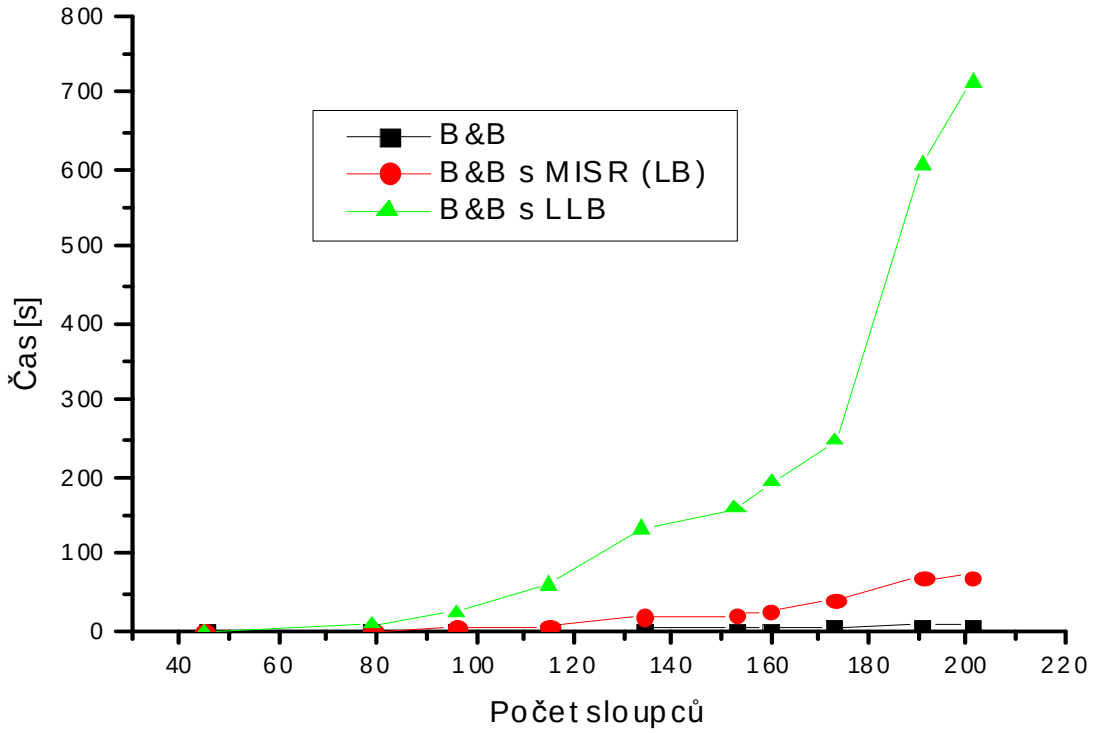
čas=hh:mm:ss									
				B&B s Limitní spodní hranicí			**B&B**		
Matice	Iterace	Rozměr	Hustota	Čas	Cena	Volání	Čas	Cena	Volání
*10x100	100	147x56	0.074101	0:33	14	4761	32:14:10	14	1E+08
10x2x200	1	135x203	0.024667		317	110	0:05	317	1576
10x500	1	131x241	0.019955		477	36	0:01	477	84
15x300	1	95x160	0.037895		229	13		229	28
20x100_30%	100	44x34	0.055481		114	30		114	408
20x20x20	1	82x204	0.050753		81	15		81	49
20x20x30_50%	1	122x153	0.021536		202	1517	0:42	202	509746
20x300	1	62x156	0.039702		215	3		215	3
50x50	100	213x27	0.226917	0:04	12	2134	8:50	12	4E+06

Tabulka 5.2.2 Výsledky metody větví a hranic

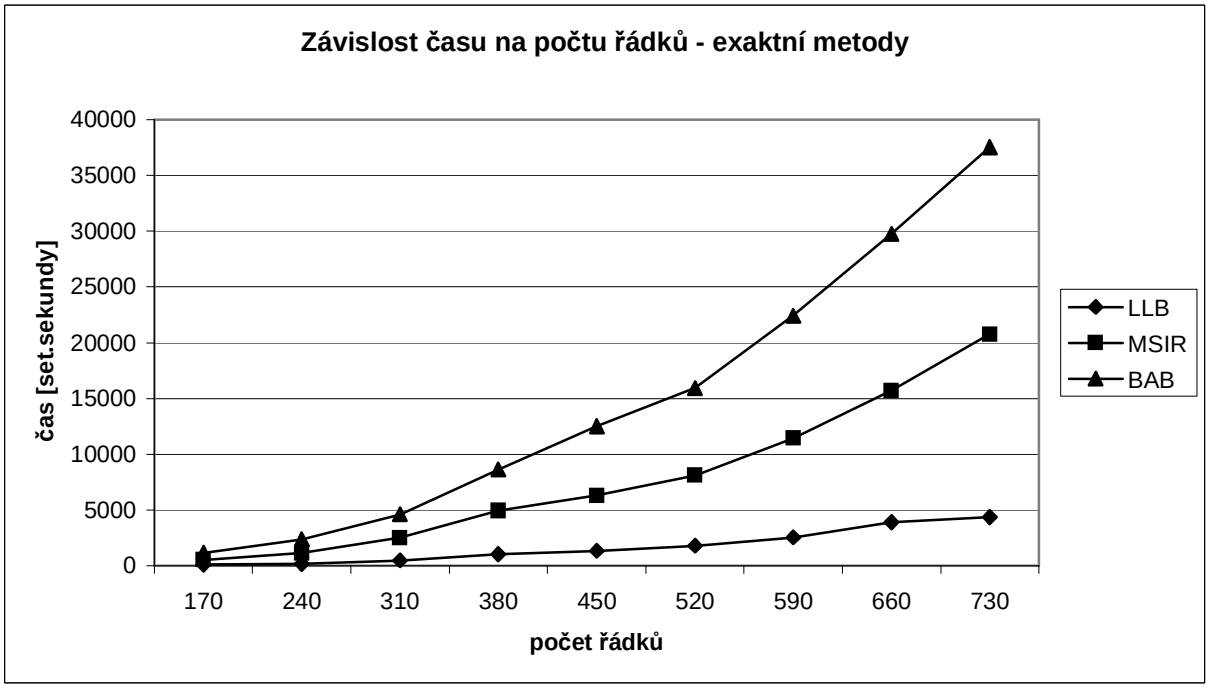
V grafu 5.2.1 je uvedena naměřená závislost výpočetního času exaktních metod na hustotě matice. Je vidět, že při narůstající hustotě klesá průměrná časová složitost, je to zřejmě způsobeno větším výskytem dominancí, tedy výsledné cyklické jádro je menší. Dále jsem měřil závislost všech tři exaktních metod na počtu sloupců, znázorněnou v grafu 5.2.2. Je vidět exponenciální nárůst počtu volání algoritmu při menším ořezávání vyhledávacího prostoru. Proto jsou velmi významné metody výpočtu spodní hranice, protože tím se redukuje počet volání algoritmu exponencielně. K testování jsem použil soubor 50x50.PLA, programem BOOM jsem vygeneroval 10 matic s různými počty sloupců a stejným počtem řádků. U této matice časová situace umožňovala testování všemi třemi exaktními metodami.



Graf 5.2.1 Časová závislost exaktních metod na hustotě matice



Graf 5.2.2 Časová závislost exaktních metod na počtu sloupců v matici



Graf 5.2.3 Časová závislost exaktních metod na počtu řádků v matici

Graf 5.2.3 nám ukazuje časovou závislost exaktních metod na počtu řádků. Asymptotická složitost na počtu řádků je v nejhorším případě $O(n^2)$, je dán složitostí hledání dominancí, která má složitost $O(n^2)$. Počet větvení algoritmu není závislý na počtu řádků.

6 Závěr

Proběhlo testování několika matic pomocí všech minimalizačních metod, z výsledků je patrné, že pro získání přesné minimální hodnoty je třeba použít exaktní metody, avšak z důvodů časové náročnosti je zvláště u velkých matic jediné přijatelné řešení pomocí heuristik, za cenu neminimálního řešení. Řešení pomocí algoritmu větví a hranic je časově velmi náročné, úpravy provedené v algoritmu, popsané v kapitole 3 značně urychlí výpočet.

Pokud vylepšíme metodu o výpočet maximální množiny nezávislých řádků, čas výpočtu se zlepší na již poměrně přijatelnou hodnotu pro matice jinak touto metodou z časových důvodů neřešitelné. Vylepšení o výpočet limitní spodní hranice ještě výrazněji sníží časové nároky. Časové rozdíly mezi třemi verzemi exaktního algoritmu jsou obrovské, kvůli exponenciální složitosti problému.

V některých případech však nebylo možné pomocí exaktních metod matice vyřešit, v praxi u velmi velkých problémů je jediným východiskem použití heuristiky. Z naměřených výsledků je patrné, že matice, které vyžadují pro nalezení minimálního pokrytí vylepšenou metodou větví a hranic velké množství času jsou snadno řešitelné pomocí heuristiky, avšak řešení není minimální, odchylky jsou uvedeny v tabulkách v kapitole 5.

V některých případech se jevila přesnější přirozená heuristika, ve většině případů Contrib Add heuristika, pokud jsme heuristikami řešili pouze cyklická jádra matic, Contrib Add heuristika nebyla nikdy horší než pokud jsme řešili celou vstupní matici.

Přesných výsledů tedy dosáhneme pouze s použitím exaktních metod, ovšem u hodně velkých problémů jsou časové nároky velmi velké. U velkých vstupních matic je tedy jediná přijatelná možnost řešení pomocí heuristik,za cenu neminimálního řešení.

Výsledkem této práce jsou naměřené hodnoty času řešení problému různými metodami, jejich porovnání a zhodnocení.

Odkazy

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

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

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

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

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

[6] O.Coudert, J.C.Madre, New Ideas for Solving Covering Problem

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