UNIVERZITA KOMENSKÉHO V BRATISLAVE
FAKULTA MATEMATIKY FYZIKY A INFORMATIKY

80e687db-77df-4b93-992e-b291c6457462

Využitie SAT–solverov pri riešení ťažkých úloh

2011

Matúš Kukan

UNIVERZITA KOMENSKÉHO V BRATISLAVE
FAKULTA MATEMATIKY FYZIKY A INFORMATIKY

Využitie SAT–solverov pri riešení ťažkých úloh
Bakalárska práca

Študijný program: Informatika
Študijný odbor: 9.2.1. Informatika
Školiace pracovisko: Katedra informatiky FMFI
Školiteľ: Mgr. Richard Štefanec

Bratislava, 2011

Matúš Kukan

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Matúš Kukan
Študijný program: informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: 9.2.1. informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: slovenský

Názov: Využitie SAT-solverov pri riešení tažkých úloh

Cieľ: Vývoj heuristík použitých v SAT-solveroch a rastúca výpočtová sila počítačov umožnili riešenie problému SAT pre mnohé dostatočne veľké inštancie. Cieľom predkladanej bakalárskej práce je skúmať možnosti využitia týchto programov pri riešení iných NP-úplných problémov.

Vedúci: Mgr. Richard Štefanec
Katedra: FMFI.KI - Katedra informatiky

Dátum zadania: 26.10.2010

Dátum schválenia: 26.10.2010

doc. RNDr. Daniel Olejár, PhD.
garant študijného programu

student

Vedúci

Abstrakt

Vývoj najmodernejších SAT–solverov v posledných rokoch výrazne pokročil. V práci sa zaoberáme možnosťami využitia heuristík a technológií, ktoré sú v nich implementované a neustále zdokonaľované, na rýchle riešenie niektorých NP-úplných grafových problémov. Na dosiahnutie cieľa sa snažíme nájsť čo najvhodnejšie zakódovanie vstupného grafového problému do inštancie SATu. Zároveň jednotlivé kódovania porovnávame aj vzájomne a aj s backtrackovým algoritmom.

Kľúčové slová: SAT–solver, redukcia na SAT, NP-úplný problém.

Abstract

The power of the state-of-the-art SAT solvers has increased dramatically in the last decade. The thesis deals with the possibilities of using techniques and heuristics implemented in solvers, which are constantly being improved, to solve NP-complete graph problems efficiently. To achieve our objective we are trying to use different SAT encodings and comparing them with each other and also with backtrack algorithms.

Keywords: SAT–solver, translation to SAT, NP-complete problem.

Obsah

Zoznam obrázkov

Zoznam tabuliek

Úvod

SAT, z anglického Boolean satisfiability problem, je množina všetkých splniteľných booleovských výrazov. Tento problém je z hľadiska teórie zložitosti významný tým, že v roku 1971 Cook a Levin dokázali, že je NP-úplný. To znamená, že ľubovolný jazyk z triedy NP je polynomiálne redukovateľný na SAT. Dovtedy nebol známy žiaden taký jazyk. Následne Karp uviedol ďalších 21 NP-úplných problémov, pričom dôkazy robil tak, že redukoval na ne SAT [Kar10].

V posledných rokoch pokročil vývoj SAT–solverov, ktoré sa neustále zlepšujú a sú schopné vyriešiť stále väčšie inštancie SAT formúl. Od roku 2002 sa každoročne uskutočňujú súťaže najmodernejších solverov. Univerzálna povaha SAT problému ho umožňuje využiť aj na riešenie iných problémov z triedy NP. Cieľom tejto práce je skúmať možnosti riešenia NP-úplných problémov pomocou redukcie na SAT. Konkrétne sa zameriavame na grafové problémy. Preskúmať chceme rôzne spôsoby redukcie na SAT formule a ich vplyv na rýchlosť nájdenia riešenia SAT–solverom. Porovnávať budeme aj optimalizované preberanie všetkých možností riešenia problému. Riešenie problémov pomocou redukcie na SAT by mohlo mať väčšiu úspešnosť vďaka dlhodobému vývoju SAT–solverov a preto by sme chceli spraviť zoznam NP problémov, ktoré už vieme celkom efektívne riešiť, aby sa mohli pridávať aj ďalšie problémy nie len redukciou na SAT, ale aj na tieto problémy.

Kapitola 1

SAT a jeho aplikácie

V tejto kapitole si zadefinujeme problém SAT a povieme niečo o jeho histórii a prvých algoritmoch, takzvaných SAT–solveroch. Neformálne je SAT množina všetkých splniteľných booleovských výrazov, formúl. My v našej definícii budeme uvažovať iba formule v konjunktívnej normálovej forme, pričom je to ekvivalentný problém z hľadiska zložitosti lebo obidva problémy sú NP-úplné. Formule budeme kódovať tak, že premenné zoradíme a nahradíme číslami v binárnom tvare.

Definícia 1.0.1 Formula je v konjunktívnej normálovej forme (KNF), ak je to konjunkcia klauzúl, kde každá klauzula je disjunkcia literálov a literál je buď premenná alebo jej negácia.

Definícia 1.0.2 Formula je splniteľná, ak existuje ohodnotenie premenných také, že je pravdivá.

Definícia 1.0.3 SAT je jazyk nad abecedou $\{\wedge, \vee, \neg, (,), 0, 1\}$, pričom slovo patrí do jazyka, ak zodpovedajúca formula v KNF je splniteľná.

Na SAT sa teda môžeme pozerať ako na rozhodovací problém, kde treba rozhodnúť, či daná formula je splniteľná.

Veta 1.0.1 SAT je NP-úplný problém.

Dôkaz. Stephen Cook [Coo71] a Leonid Levin [Lev73] nezávisle dokázali v roku 1971. Dôkaz tu neuvádzame. □

Zaujíma nás KNF, pretože väčšina moderných SAT–solverov pracuje s výrazmi v KNF. Má tú výhodu, že je to celkom jednoduchý tvar, kde musia byť všetky klauzuly splnené, čo sa ľahšie kontroluje a prvé algoritmy, ktoré vznikli toto využívajú. Môžeme sa na výraz pozerať ako na množinu klauzúl a na klauzulu zase ako množinu literálov, preto budeme používať množinové značenie.

1.1 Algoritmy

V tejto časti si v krátkosti popíšeme niektoré používané algoritmy. Na vstupe dostanú množinu klauzúl a výstup hovorí, či je daná formula splniteľná.

Úplným základom prvých algoritmov, ako aj terajších najmodernejších je pravidlo odvodenia:

$$\frac{\{C, v\}\{D, \neg v\}}{\{C, D\}}$$

Hovorí, že keď máme v klauzule premennú a v nejakej inej jej negáciu, môžeme ich zjednotiť a ponechať iba ostatné premenné.

1.1.1 DP

Základ algoritmu navrhnutého Davisom a Putnamom v roku 1960 [DP60]:

1. ak nájdeš prázdnu klauzulu, skonči s výsledkom **nesplniteľný**
2. nájdi literály, ktoré sa vyskytujú iba v jednom stave (iba ako premenná alebo jej negácia)
3. odstráň všetky klauzuly, ktoré taký literál obsahujú (priradením vhodnej hodnoty premennej sú všetky splnené)
4. ak neostala žiadna klauzula, skonči s výsledkom **splniteľný**
5. vyber si literál x, ktorý sa vyskytuje v oboch stavoch
6. odvoď pomocou neho všetky možné klauzuly
7. odstráň všetky klauzuly v ktorých sa x vyskytuje
8. choď na krok 1

Tento postup má však nevýhodu, lebo môže kvadraticky narastať počet klauzúl. Ak sa v n klauzulách vyskytuje x a v m klauzulách $\neg x$, počet odvodených klauzúl je $n \cdot m$, ktoré sú navyše väčšie. V mnohých praktických prípadoch sa ukazuje, že pamäťovo algoritmus zlyháva.

1.1.2 DPLL

Preto bolo treba niečo zmeniť a druhá verzia algoritmu [DLL62] je založená na odvodení z literálu:

$$\frac{\{C, \neg v\}\{v\}}{\{C\}}$$

1. kým existuje klauzula obsahujúca iba literál x, opakovane vykonávaj odvodenie z x
2. ak nájdeš prázdnu klauzulu, skonči s výsledkom **nesplniteľný**
3. nájdi literály, ktoré sa vyskytujú iba v jednom stave (iba ako premenná alebo jej negácia)
4. odstráň všetky klauzuly, ktoré taký literál obsahujú (priradením vhodnej hodnoty premennej sú všetky splnené)

5. ak neostala žiadna klauzula, skonči s výsledkom **splniteľný**
6. vyber si literál x, ktorý sa vyskytuje v oboch stavoch
7. rekurzívne vykonaj proces s pridanou klauzulou $\{x\}$
8. rekurzívne vykonaj proces s pridanou klauzulou $\{\neg x\}$
9. ak aspoň jedna rekurzia uspela, skonči s výsledkom **splniteľný**
10. inak skonči s výsledkom **nesplniteľný**

Problematický je 6. bod algoritmu, v ktorom treba vybrať literál x. Nesprávny výber môže viesť k vyskúšaniu všetkých možností a preto existujú rôzne heuristiky na výber x.

Moderné DPLL SAT–solvery sa delia na:

- conflict-driven, ktoré sú úspešnejšie
- look ahead

Opísanie rozdielu týchto dvoch spôsobov prevzaté z [BHvMW09], kapitola 5.1: Predstavte si, že ste prvý krát v New Yorku. Vystúpili ste z taxíka a chcete si pozrieť najkrajšiu časť mesta. Zvažujete confict-driven a look ahead stratégie.

Pri prvej, keď prídete na križovatku, jednoducho sa vyberiete tam, kde to vyzerá na prvý pohľad najkrajšie. Ak prídete na miesto kde to vyzerá zle, vrátite sa na križovatku, kde ste pravdepodobne zle zabočili.

Pri look ahead stratégii sa vyberá smer zložitejšie. Najprv sa vydáte kúsok každým smerom, preskúmate to tam, vrátite sa na križovatku a až potom vyhodnotíte, ktorým smerom sa vydať. Aj keď táto stratégia vyzerá zložitejšie, niekedy býva úspešnejšia.

SAT–solvery, ktoré som používal sú conflict-driven DPLL algoritmy s mnohými vylepšeniami. Sú to minisat [N. 05], picosat a precosat [Bie10].

1.1.3 Local search

Existujú aj lokálne prehľadávacie (angl. local search) algoritmy pre SAT [SKC94]. Začínajú tak, že náhodne ohodnotia premenné. Ak nie sú splnené všetky klauzuly, skúsia zmeniť ohodnotenie jednej premennej a takto pokračujú v hľadaní riešenia. Odtiaľ názov lokálne prehľadávacie algoritmy. Je to ale nesystematický postup. Sú rôzne možnosti pre rozhodovanie ktorej premennej zmeniť ohodnotenie aby sa dosiahlo celkové zlepšenie. A takisto treba niekedy začať hľadať z iného miesta, lebo sa môže stať, že už nie je čo zlepšovať v danom okolí. Príkladmi takých algoritmov sú GSAT a WalkSAT [HS00].

1.2 Využitie

Hoci SAT–solvery majú v zásade exponenciálnu časovú zložitosť, v mnohých prípadoch sa vďaka pokročilým technikám úspešne používajú. Aspoň niektoré aplikácie si teraz uvedieme.

Medzi prvé pokusy patrí využitie pri plánovaní v umelej inteligencii. Je to proces hľadania sekvencie úloh, ktoré splnia určitý cieľ. V roku 1992 Kautz a Selman navrhli redukovať plánovanie na SAT [KS92]. Spočiatku o to nebol veľký záujem ale neskôr sa ukázalo, že tento spôsob riešenia je porovnateľne rýchly a dajú sa dosiahnuť touto cestou zaujímave výsledky. Tento úspech podnietil rovnaký prístup pri riešení podobných problémov, ako je testovanie modelov systémov s konečným počtom stavov.

Prakticky sa využívajú SAT–solvery aj pri formálnej verifikácii a testovaní softvéru, v bioinformatike a mnohých iných oblastiach.

Takisto je možné zaujímavé využitie pri generovaní zadaní logických úloh ako je napríklad sudoku [Lab10].

A nakoniec spomeňme rozsiahlu prácu, ktorá sa zaoberá spôsobmi riešenia sytémov rovníc, za účelom vytvorenia nových nástrojov, využiteľných pri algebraickej kryptoanalýze [Bar07].

Viac sa o histórii, algoritmoch a využití dá nájsť napríklad v [BHvMW09].

Kapitola 2

Problémy a ich jednoduché riešenia

V tejto kapitole popíšeme rozhodovacie problémy, ktorým sme sa venovali. Ide o grafové problémy. Rozhodovacie znamená, že na vstupe dostaneme nejaký graf a máme rozhodnúť, či má danú vlastnosť. Vychádzame zo štandardnej definície grafu ako je uvedené v [Die00]:

Definícia 2.0.1 *Graf* je dvojica $G = (V, E)$, taká, že $E \subseteq V^2$. Prvky z V nazývame vrcholy a prvky z E hrany grafu G. Hrany sú teda dvojice vrcholov a hovoríme, že vrchol u *susedí* s vrcholom v, ak $(u, v) \in E$.

Používame označenie $n = |V|$ pre počet vrcholov grafu a $m = |E|$ pre počet hrán.

Ku každému problému takisto uvádzame čo najlepšie, ale zároveň jednoduché riešenie, ktoré sme implementovali. Spočíva v optimalizovanom preberaní všetkých možností, založenom na rekurzii. Tieto algoritmy budeme v ďalšom texte označovať pojmom backtrack.

2.1 Klika

Úloha: Nájsť v grafe k vrcholov takých, že každé dva z nich sú spojené hranou.

Definícia 2.1.1 Hovoríme, že graf $G = (V, E)$ má k-kliku, ak existuje $S \subseteq V$ taká, že $|S| = k$ a $\forall u, v \in S : (u, v) \in E$.

Riešenie: Budú nás zaujímať iba vrcholy, ktoré majú stupeň aspoň $k - 1$, lebo iba tie môžu byť súčasťou k-kliky. Toto obmedzenie je veľká výhoda, pretože v riedkych grafoch ich bude málo a teda rýchlo zistíme, že tam k-klika nie je a v hustých grafov je zase vysoká pravdepodobnosť, že sa nám relatívne rýchlo podarí nejakú k-kliku nájsť. Začneme s prázdnou klikou a

postupne budeme pridávať vrcholy do aktuálnej kliky v prípade, že je to možné. To jest ak sú spojené hranou so všetkými vrcholmi, ktoré aktuálne patria do kliky. Ak už sa nedá žiaden vrchol pridať a je ich málo, naposledy pridaný odoberieme a skúšame ďalej. Takto pokračujeme až raz nájdeme k-kliku alebo zistíme, že tam nie je.

Pseudokód:

```
Klika(G,Kl,k) {
    if (Kl.size() == k) return true;
    if (Kl.size() + G.size() < k) return false;
    vyber vrchol v so stupňom aspoň k-1;
    if (Klika(G–v,Kl,k)) return true;
    if (v susedí so všetkými vrcholmi v Kl)
        if (Klika(G–v,Kl+v)) return true;
    return false;
}
```

2.2 Vrcholové pokrytie

Úloha: Nájsť v grafe k vrcholov takých, že každá hrana susedí s nejakým z nich.

Definícia 2.2.1 Hovoríme, že graf $G = (V, E)$ má vrcholové pokrytie veľkosti k, ak $\exists S \subseteq V$ taká, že $|S| = k$ a $\forall x, y \in V : (x, y) \in E \Rightarrow x \in S \vee y \in S$.

Riešenie: Platí, že graf má vrcholové pokrytie veľkosti k práve vtedy ak v ňom existuje nezávislá množina veľkosti $n - k$ a to je práve vtedy keď v inverznom grafe existuje klika veľkosti $n - k$. Preto riešenie spočíva v invertovaní hrán a následnom hľadaní kliky veľkosti $n - k$.

2.3 Farbenie

Úloha: V tomto prípade treba ofarbiť vrcholy grafu k farbami tak, aby žiadne dva susediace vrcholy nemali rovnakú farbu.

Definícia 2.3.1 Hovoríme, že graf $G = (V, E)$ je k-ofarbiteľný, ak existuje funkcia $f : V \to \{1 \dots k\}$ pričom platí: $\forall u, v \in V : (u, v) \in E \Rightarrow f(u) \neq f(v)$

Riešenie: Postupne skúšame ofarbiť vrchol nejakou farbou a presunúť sa na ďalší vrchol. Ak nie je možné vrchol danou farbou ofarbiť, skúsime inú farbu a tak pokračujeme až do vyčerpania všetkých možností. Ak žiadna farba nevyhovuje, vrátime sa na predchádzajúci vrchol a zmeníme mu farbu. Takto sa pokračuje, kým sa nevyčerpajú všetky možnosti. Tých je ale veľmi

veľa a nevieme zistiť, či aktuálne čiastočné ofarbenie vedie k výsledku alebo nie.

Pseudokód:

```
Farbenie(G,k) {
    if (G.empty()) return true;
    vyber z G vrchol v;
    for (f = 0; f < k; f++) {
        if (v nesusedí s vrcholom farby f)  {
            ofarbi v farbou f;
            if (Farbenie(G–v,k)) return true;
        }
    }
    return false;
}
```

2.4 Monochromatický trojuholník

Úloha: Zistiť, či je možné rozdeliť hrany grafu na dve časti, pričom v žiadnej neexistuje trojuholník.

Definícia 2.4.1 Hovoríme, že graf $G = (V, E)$ neobsahuje monochromatický trojuholník, ak $\exists E_1, E_2 \subseteq E$ také, že: $E_1 \cap E_2 = \emptyset \wedge E_1 \cup E_2 = E \wedge \forall i \in \{1, 2\}$:

$$\forall u, v, w \in V : (u, v) \notin E_i \vee (v, w) \notin E_i \vee (u, w) \notin E_i.$$

Riešenie: Skúsime pridať hranu do množiny E_1 a ak tam nevznikne trojuholník, rekurzívne sa zavoláme na zvyšok grafu. Ak sa nám zvyšok hrán nepodarilo rozdeliť, vyskúšame pridať hranu do E_2. Ak ani to nevyjde, riešenie neexistuje.

Algoritmus teda skúša postupne vytvárať množiny E_1 a E_2 tak, aby neobsahovali trojuholník a ak sa raz minú všetky hrany, úspešne skončí. Na to aby efektívne zistil, či po pridaní aktuálnej hrany do množiny E_i v nej vznikne trojuholník, si pre každú hranu pamätáme zoznam trojuholníkov v pôvodnom grafe, ktorých je súčasťou a pre každý z nich sa pozrieme, či zvyšné dve hrany sú už v množine alebo nie. Tieto zoznamy trojuholníkov si pre každú hranu vytvoríme na začiatku behu programu.

Pseudokód:

```
Monochromatic(E,E1,E2) {
    if (E.empty()) return true;
    vyber z E hranu e;
    for (t ∈ Patri[e]) {
        if(zvyšné dve hrany z t sú v Ei)
            zamietni Ei;
    }
    if (E1 nie je zamietnuté)
        if (Monochromatic(E–e,E1+e,E2)) return true;
    if (E2 nie je zamietnuté)
        if (Monochromatic(E–e,E1,E2+e)) return true;
    return false;
}
Main() {
    načítaj graf G = (V,E);
    for (u,v,w ∈ V) {
        if (u,v,w tvoria trojuholník t) {
            Patri[(u,v)].push(t);
            Patri[(v,w)].push(t);
            Patri[(u,w)].push(t);
        }
    }
    return Monochromatic(E,∅,∅);
}
```

2.5 Rez grafu

Úloha: Rozdeliť vrcholy grafu na dve časti tak, aby k hrán spájalo vrcholy z rôznych častí.

Definícia 2.5.1 Hovoríme, že v grafe $G = (V, E)$ existuje k-rez, ak $\exists S_1, S_2 \subseteq V$ také, že:

$$S_1 \cap S_2 = \emptyset \wedge S_1 \cup S_2 = V \wedge |\{(u, v) \in E \mid u \in S_1, v \in S_2\}| = k.$$

Riešenie: Vyskúšame všetky možnosti rozdelenia vrcholov na dve množiny a zrátame, či je dostatok hrán tvoriacich rez daného rozdelenia. Pri rozdelovaní vrcholov do množín priebežne počítame veľkosť rezu. Ak zistíme, že aj keby všetky hrany nezaradených vrcholov tvorili rez, bolo by to menej ako k, nepokračujeme.

Pseudokód:

```
Rez(G,S1,S2,k) {
    if (G.empty())
        return (hrán tvoriacich rez == k);
    moznost=0;
    for (v ∈ V) moznost += deg(v);
    if (hrán tvoriacich rez + moznost < k) return false;
    if (Rez(G–v,S1+v,S2,k)) return true;
    if (Rez(G–v,S1,S2+v,k)) return true;
    return false;
}
```

2.6 Hamiltonovská kružnica

Úloha: Nájsť v grafe kružnicu, ktorá obsahuje všetky vrcholy.

Definícia 2.6.1 Hovoríme, že graf $G = (V, E)$ obsahuje Hamiltonovskú kružnicu, ak existuje bijektívna funkcia $f : V \to \{1 \dots n\}$ pričom platí:

$$\forall u, v \in V : (u, v) \notin E \Rightarrow (f(u) \neq f(v) + 1) \wedge (f(u) = 1 \Rightarrow f(v) \neq n)$$

Riešenie: Prehľadávanie grafu do hĺbky, pričom potrebujeme skúšať všetky možné cesty prehľadávania. Ak sa raz ocitneme v hĺbke n a z daného vrcholu existuje hrana do koreňa, graf obsahuje Hamiltonovskú kružnicu.

Pseudokód:

```
Hamilton(G,root,v) {
    Navstiveny[v]=true;
    if (všetky vrcholy sú navšívené)
        return (v,root) ∈ E;
    for (u ∈ susedia[v]) {
        if (\neg Navstiveny[v])
            if (Hamilton(G,root,u))
                return true;
    Navstiveny[v]=false;
    return false;
}
```

Kapitola 3

SAT kódovania

V tejto kapitole si popíšeme ako sa dajú niektoré NP-úplné problémy riešiť pomocou redukcie na SAT. Zostrojíme výraz, ktorý bude splniteľný práve vtedy ak má problém riešenie. Tento výraz nazývame kódovanie. Spýtame sa SAT–solvera na splniteľnosť a ak sa to dá, povie nám ako treba ohodnotiť premenné, z čoho sa dá spätne zostrojiť riešenie problému[1]. Vieme, že každý problém je polynomiálne redukovateľný na SAT, avšak nie vždy je jednoduché nejaké kódovanie nájsť. A takisto sa môže stať, že dané kódovanie je veľmi neefektívne a treba nájsť lepšie. Avšak existuje veľa problémov, ktoré sa dajú priamočiaro a efektívne redukovať na SAT. Na niektoré z nich sa teraz pozrieme.

3.1 Klika

Nasledujúce kódovanie je inšpirované článkom [Len10]:

Myšlienka spočíva v rozdelení vrcholov na tie, ktoré patria klike a tie, čo nepatria. Potom použijeme premenné, ktoré umožnia spočítať, či je k vrcholov v klike. Budeme používať počítadlo, ktoré inicializujeme na 0 a postupne pre všetky $v \in V$ ak x_v platí (vrchol v patrí klike), zväčšíme ho o 1.

Lineárne kódovanie

Formálne, zavedieme premenné x_v na určenie, či vrchol v patrí klike. Ďalej počítadlo c^i, pre $i \in \{0 \dots n\}$ bude pozostávať z $k+1$ premenných, pričom j-ta bude pravdivá práve vtedy ak $c^i = j$.

Kódovanie tvoria výrazy:
ϕ_1 na zabezpečenie toho aby to bola klika. Ak vyberieme vrcholy u a v, musí

[1] Výstižne to opisuje názov: "There and back again"[Tol97]

existovať hrana (u, v).

$$\phi_1 = \bigwedge_{(u,v)\in E} (\neg x_u \vee \neg x_v)$$

ϕ_2 inicializuje počítadlo na 0.

$$\phi_2 = c_0^0 \wedge \bigwedge_{j=1}^{k} \neg c_j^0$$

ϕ_3 skopíruje počítadlo c^{i-1} ak x_i neplatí. Čiže $\neg x_i \Rightarrow (c_j^i \Leftrightarrow c_j^{i-1})$

$$\phi_3 = \bigwedge_{i=1}^{n} \bigwedge_{j=0}^{k} (\neg c_j^{i-1} \vee c_j^i \vee x_i) \wedge (\neg c_j^i \vee c_j^{i-1} \vee x_i)$$

ϕ_4 zväčší počítadlo c^{i-1} o 1 ak x_i platí. $x_i \Rightarrow ((c_j^i \Leftrightarrow c_{j-1}^{i-1}) \wedge c_0^i)$

$$\phi_4 = \bigwedge_{i=1}^{n} ((\neg c_0^i \vee \neg x_i) \wedge \bigwedge_{j=1}^{k} (\neg c_j^{i-1} \vee c_{j-1}^i \vee \neg x_i) \wedge (\neg c_j^i \vee c_{j-1}^{i-1} \vee \neg x_i))$$

Výsledné kódovanie je $\phi_{klika} = \phi_1 \wedge \phi_2 \wedge \phi_3 \wedge \phi_4 \wedge c_k^n$.

Binárne kódovanie

Pre porovnanie sme skúsili implementovať aj druhé kódovanie, založené na rovnakej myšlienke s rozdielom, že počítadlo nebude lineárne, ale bude to priamo binárne číslo. Premenné sú $x_1^v \dots x_{\lceil log_2 k \rceil}^v$ Avšak napriek tomu, že kódovanie používa menej premenných, výpočet pomocou SAT–solvera trvá približne rovnako rýchlo.

Tvoria ho výrazy:
ϕ_1 ako v predchádzajúcom prípade. ϕ_2' a ϕ_3' veľmi podobné. Ostatné sú komplikovanejšie.

$$\phi_2' = \bigwedge_{j=1}^{\lceil log_2 k \rceil} \neg c_j^0$$

$$\phi_3' = \bigwedge_{i=1}^{n} \bigwedge_{j=1}^{\lceil log_2 k \rceil} (\neg c_j^{i-1} \vee c_j^i \vee x_i) \wedge (\neg c_j^i \vee c_j^{i-1} \vee x_i)$$

ϕ_{4a}' má za úlohu pripočítať ku c_j^i zvyšok (alebo pôvodnú jednotku), ak $c_1^{i-1}, \dots, c_{j-1}^{i-1} = 1, \dots, 1$. V tom prípade $\neg(c_j^i \Leftrightarrow c_j^{i-1})$.

$$\phi_{4a_1}' = \bigwedge_{i=1}^{n} \bigwedge_{j=1}^{\lceil log_2 k \rceil} (\bigvee_{p=1}^{j-1} \neg c_p^{i-1} \vee c_j^{i-1} \vee c_j^i \vee \neg x_i)$$

$$\phi'_{4a_2} = \bigwedge_{i=1}^{n} \bigwedge_{j=1}^{\lceil log_2 k \rceil} (\bigvee_{p=1}^{j-1} \neg c_p^{i-1} \vee \neg c_j^{i-1} \vee \neg c_j^i \vee \neg x_i)$$

Ak niekde medzi $c_1^{i-1} \dots c_{j-1}^{i-1}$ je 0, tak $(c_j^i \Leftrightarrow c_j^{i-1})$.

$$\phi'_{4b_1} = \bigwedge_{i=1}^{n} \bigwedge_{j=1}^{\lceil log_2 k \rceil} \bigvee_{p=1}^{j-1} (c_p^{i-1} \vee c_j^{i-1} \vee \neg c_j^i \vee \neg x_i)$$

$$\phi'_{4b_2} = \bigwedge_{i=1}^{n} \bigwedge_{j=1}^{\lceil log_2 k \rceil} \bigvee_{p=1}^{j-1} (c_p^{i-1} \vee \neg c_j^{i-1} \vee c_j^i \vee \neg x_i)$$

ϕ'_5 ešte zaručí, že $c^n = k$

$$\phi'_5 = \bigwedge_{j=1}^{\lceil log_2 k \rceil} (c_j^n \Leftrightarrow j - \text{ty bit čísla } k)$$

Výsledné kódovanie $\phi'_{klika} = \phi_1 \wedge \phi'_2 \wedge \phi'_3 \wedge \phi'_{4a_1} \wedge \phi'_{4a_2} \wedge \phi'_{4b_1} \wedge \phi'_{4b_2} \wedge \phi'_5$.

3.2 Vrcholové pokrytie

Pre tento problém sme použili rovnakú myšlienku, líši sa to iba jemne v pravidlách ϕ_1 aby vrcholy tvorili pokrytie.

$$\phi'_1 = \bigwedge_{(u,v) \in E} (x_u \vee x_v)$$

Výsledné kódovanie je $\phi_{pokrytie} = \phi'_1 \wedge \phi_2 \wedge \phi_3 \wedge \phi_4 \wedge c_k^n$.

3.3 Farbenie

Tento problém ma veľa rozličných kódovaní, ktoré opísal M.N. Velev [Vel07]. Tri z nich sme implementovali a v časti 4.3 porovnali.

Priame kódovanie

Každému vrcholu v prislúcha k premenných $x_1^v \dots x_k^v$, ktoré určujú farbu vrcholu.
ϕ_1 vynúti každému vrcholu nejakú farbu.

$$\phi_1 = \bigwedge_{v \in V} (\bigvee_{f=1}^{k} x_f^v)$$

ϕ_2 nepovolí viac farieb ako jednu.

$$\phi_2 = \bigwedge_{v \in V} \bigwedge_{\substack{f_1, f_2 \in \{1 \ldots k\} \\ f_1 \neq f_2}} (\neg x_{f_1}^v \vee \neg x_{f_2}^v)$$

ϕ_3 zabezpečí požadované správanie, aby susedné vrcholy mali rôznu farbu.

$$\phi_3 = \bigwedge_{(u,v) \in E} \bigwedge_{f=1}^{k} (\neg x_f^u \vee \neg x_f^v)$$

Výsledné kódovanie je $\phi_{farbenie} = \phi_1 \wedge \phi_2 \wedge \phi_3$.

Násobné kódovanie

Avšak existuje aj druhá možnosť, povoliť viac farieb jednému vrcholu, vynechaním klauzúl ϕ_2. Potom sa nám môže stať, že riešenie SAT-u bude obsahovať pravdivé premenné x_i^v a x_j^v pre $i \neq j$. V tom prípade si môžeme pre vrchol v vybrať farbu i alebo j. V oboch prípadoch dostaneme platné farbenie.

Násobné kódovanie $\phi'_{farbenie} = \phi_1 \wedge \phi_3$.

Logaritmické kódovanie

Tak ako v prípade k-kliky, aj tu je možnosť kódovať farby binárne. Používame premenné $x_1^v \ldots x_{\lceil log_2 k \rceil}^v$, ktoré určujú farbu vrcholu v. Tento spôsob sa ukazuje ako najefektívnejší [Vel07]. Je však zložitejšie implementovať kódovanie. Ukážeme si aspoň klauzuly pre $k \neq 3$:

Niečo podobné ako ϕ_1 a ϕ_2 pri priamom kódovaní nepotrebujeme, lebo každé ohodnotenie určuje nejaké číslo. Namiesto toho potrebujeme klauzule ϕ_4, aby sme nepoužili číslo väčšie ako k ak $k = 2^i, i \in N$.
Pre $k = 3$:

$$\phi_4 = \bigwedge_{v \in V} (\neg x_1^v \vee \neg x_2^v)$$

$$\phi'_3 = \bigwedge_{(u,v) \in E} ((x_1^u \vee x_2^u \vee x_1^v \vee x_2^v) \wedge (\neg x_1^u \vee x_2^u \vee \neg x_1^v \vee x_2^v) \wedge (x_1^u \vee \neg x_2^u \vee x_1^v \vee \neg x_2^v))$$

Výsledné kódovanie $\phi''_{farbenie} = \phi'_3 \wedge \phi_4$.

3.4 Monochromatický trojuholník

Kódovanie tohto problému je veľmi jednoduché. Postačuje n premenných x_v, na rozlíšenie do ktorej množiny patrí vrchol v, a pravidlá aby neboli spolu vrcholy, ktoré tvoria trojuholník.

$$\phi_{mono} = \bigwedge_{\substack{u,v,w \in V \\ (u,v) \in E \\ (v,w) \in E \\ (u,w) \in E}} (x_u \vee x_v \vee x_w) \wedge (\neg x_u \vee \neg x_v \vee \neg x_w)$$

3.5 Rez grafu

Opäť budeme používať premenné x_v na určenie, do ktorej množiny patrí v, a premenné pre počítadlá. Tentokrát ale treba spočítať hrany, ktoré majú konce v odlišných množinách, a netreba pridať žiadne iné klauzuly. Rez je to vždy. Preto si hrany v grafe zoradíme a hranu budeme netradične značiť písmenkom i.

ϕ_1 inicializuje počítadlo na 0.

$$\phi_1 = c_0^0 \wedge \bigwedge_{j=1}^{k} \neg c_j^0$$

Ďalej sú 4 možnosti:

ϕ_2 skopíruje počítadlo c^{i-1} ak $x_u \wedge x_v$, kde $i = (u, v)$.

$$\phi_2 = \bigwedge_{i=1}^{m} \bigwedge_{j=0}^{k} (\neg c_j^{i-1} \vee c_j^i \vee \neg x_u \vee \neg x_v) \wedge (\neg c_j^i \vee c_j^{i-1} \vee \neg x_u \vee \neg x_v)$$

ϕ_3 skopíruje počítadlo c^{i-1} ak $\neg x_u \wedge \neg x_v$, kde $i = (u, v)$.

$$\phi_3 = \bigwedge_{i=1}^{m} \bigwedge_{j=0}^{k} (\neg c_j^{i-1} \vee c_j^i \vee x_u \vee x_v) \wedge (\neg c_j^i \vee c_j^{i-1} \vee x_u \vee x_v)$$

ϕ_4 zväčší počítadlo c^{i-1} o 1 ak $x_u \wedge \neg x_v$, kde $i = (u, v)$.

$$\phi_4 = \bigwedge_{i=1}^{m} ((\neg c_0^i \vee \neg x_u \vee x_v) \wedge \bigwedge_{j=1}^{k} (\neg c_j^{i-1} \vee c_{j-1}^i \vee \neg x_u \vee x_v) \wedge (\neg c_j^i \vee c_{j-1}^{i-1} \vee \neg x_u \vee x_v))$$

ϕ_5 zväčší počítadlo c^{i-1} o 1 ak $\neg x_u \wedge x_v$, kde $i = (u, v)$.

$$\phi_5 = \bigwedge_{i=1}^{m} ((\neg c_0^i \vee x_u \vee \neg x_v) \wedge \bigwedge_{j=1}^{k} (\neg c_j^{i-1} \vee c_{j-1}^i \vee x_u \vee \neg x_v) \wedge (\neg c_j^i \vee c_{j-1}^{i-1} \vee x_u \vee \neg x_v))$$

Výsledné kódovanie je $\phi_{rez} = \phi_1 \wedge \phi_2 \wedge \phi_3 \wedge \phi_4 \wedge \phi_5 \wedge c_k^m$.

Avšak toto kódovanie je veľmi neefektívne. Domnievame sa, že to môže byť aj tým, že tu nie sú žiadne obmedzujúce klauzuly na premenné x_v. Alebo to je iba preto, že potrebujeme veľmi veľa premenných na sčítanie.

Vyskúšali sme teda aspoň zaviesť ďalšie premenné p_i, na odlíšenie kedy je treba pripočítavať jednotku a kedy nie. Pridať treba formulu $p_i \nLeftrightarrow (x_u \Leftrightarrow x_v)$ a potom je možné odstrániť ϕ_3 a ϕ_5 s tým, že vo ϕ_2 nahradíme premenné x literálom $\neg p_i$ a vo ϕ_4 literálom p_i.

Táto zmena výrazne urýchlila výpočet, keď sme použili ako SAT–solver precosat. V testoch ale používame minisat, ktorý je rýchly s oboma kódovaniami.

3.6 Hamiltonovská kružnica

Zvolili sme postup, ktorý očísluje vrcholy a tieto v danom poradí potom tvoria kružnicu ak existuje. Podobne ako pri počítadle v klike, máme $n.n$ premenných x_i^v, ktoré určujú bijekciu medzi V a $\{1 \dots n\}$. x_i^v, platí práve vtedy ak vrcholu v je priradené číslo i.

ϕ_1 zabezpečí aby každému vrcholu bolo priradené nejaké číslo.

$$\phi_1 = \bigwedge_{v \in V} (\bigvee_{i=1}^{n} x_i^v)$$

ϕ_2 zabezpečí aby každé číslo bolo priradené nejakému vrcholu.

$$\phi_2 = \bigwedge_{i=1}^{n} (\bigvee_{v \in V} x_i^v)$$

ϕ_3 zabezpečí aby každému vrcholu bolo priradené najviac jedno číslo.

$$\phi_3 = \bigwedge_{v \in V} (\bigwedge_{i \neq j} \neg x_i^v \vee \neg x_j^v)$$

ϕ_4 zabezpečí aby každé číslo bolo priradené najviac jednému vrcholu.

$$\phi_4 = \bigwedge_{i=1}^{n} (\bigwedge_{u \neq v} (\neg x_i^u \vee \neg x_i^v))$$

ϕ_5 zabezpečí aby to bola kružnica. Ak neexistuje hrana medzi u a v, tak nemôžu susediť v postupnosti.

$$\phi_5 = \bigwedge_{\substack{u,v \in V \\ (u,v) \notin E}} ((\neg x_1^u \vee \neg x_n^v) \wedge \bigwedge_{i=1}^{n-1} (\neg x_i^u \vee \neg x_{i+1}^v))$$

Na to aby premenné jednoznačne určovali bijekciu stačia podmienky $\phi_1 \wedge \phi_4$ alebo $\phi_2 \wedge \phi_3$. Keďže ale niekedy môžu aj nadbytočné klauzuly v kódovaní urýchliť výpočet, testovali sme viacero možností kódovania. Najpomalšie sa ukázali kódovania, ktoré neobsahovali ϕ_2. Najlepšie sú kódovania: $\phi_1 \wedge \phi_2 \wedge \phi_3 \wedge \phi_5$ a $\phi_2 \wedge \phi_3 \wedge \phi_4 \wedge \phi_5$. Rozhodli sme sa používať kódovanie $\phi_{ham} = \phi_2 \wedge \phi_3 \wedge \phi_4 \wedge \phi_5$. Zdá sa, že použiť všetky formule je už priveľa. Taký spôsob bol trochu pomalší.

Neskôr sme zistili, že týmto kódovaniam sa venovali v článku [HHU07]. Sú tam vysvetlené dôvody, prečo sa kódovania správaju odlišne.

V časti 4.2 sú porovnané kódovania ϕ_{ham}, $\phi_{ham1} = \phi_2 \wedge \phi_3 \wedge \phi_5$, $\phi_{ham2} = \phi_1 \wedge \phi_3 \wedge \phi_4 \wedge \phi_5$ a $\phi_{ham3} = \phi_1 \wedge \phi_4 \wedge \phi_5$.

Kapitola 4

Pozorovania

Generovali sme náhodné grafy s určitou veľkosťou n a danou pravdepodobnosťou p výskytu hrany. Počet vrcholov je n a každá možná hrana sa v grafe nachádza s pravdepodobnosťou p, očakávaný počet hrán je teda $p\binom{n}{2}$. Niekedy budeme používať označenie 'graf n p', čo znamená, že je to náhodný graf, ktorý má n vrcholov a pravdepodobnosť výskytu hrany je p.

Pri vytváraní grafov používame niekedy logaritmickú a niekedy lineárnu škálu, podľa toho, čo sa nám zdalo vhodnejšie. Logaritmická je použitá hlavne v prípadoch, keď sú veľmi velké rozdiely v časoch. Pre získanie lepšej predstavy o čase behu programu, sme ho spustili na jedenástich rôznych inštanciách grafu n p. Výsledná časová hodnota je ich priemer.

Programy sme implementovali v jazyku C++. Všetky testy bežali na procesore Intel® Core^{TM} i3-330M (3M Cache, 2.13 GHz) s 3 GB pamäte. Procesor je viacjadrový, avšak všetky použité programy sú jednovláknové a preto žiadny z nich nevyužil túto výhodu.

Ukážme si teraz niektoré zistené poznatky.

4.1 Ktorý SAT–solver použiť

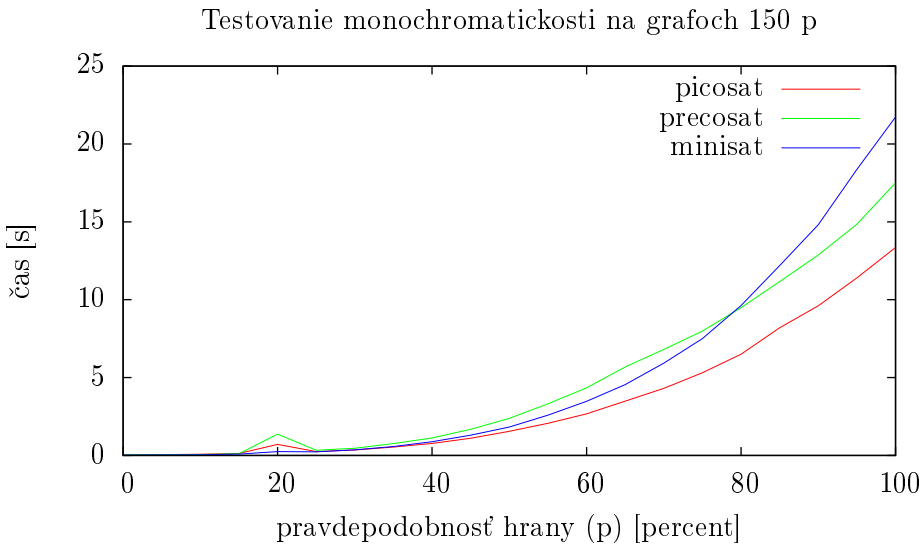
Samozrejme, chceme používať ten najlepší SAT–solver. Preto sme si ich niekoľko zaobstarali a testovali, ktorý bude najrýchlejší. Ukázalo sa však, že vo všeobecnosti najrýchlejší SAT–solver neexistuje. Ich úspešnosť je závislá od daného kódovania a každý môže byť v nejakom prípade dobrý. My sme používali picosat, precosat a minisat. Ukážeme si teraz na troch problémoch ich porovnanie.

Monochromatický trojuholník

Testovali sme grafy so 150 vrcholmi, menili pravdepodobnosť výskytu

hrany a sledovali ako rýchlo sa dá zistiť, či graf obsahuje monochromatický trojuholník. Merali sme iba čas a nezaujímalo nás, či je odpoveď áno alebo nie, lebo v tomto prípade čas od odpovede závisí minimálne. Už pri pravdepodobnosti 20% sa vyskytujú v grafoch so 150 vrcholmi monochromatické trojuholníky takmer vždy a to je čas behu ešte veľmi malý. A samozrejme, čím viac hrán, tým väčšia pravdepodobnosť výskytu.

Ako vidno na obrázku 4.1, v tomto prípade je najlepší picosat.

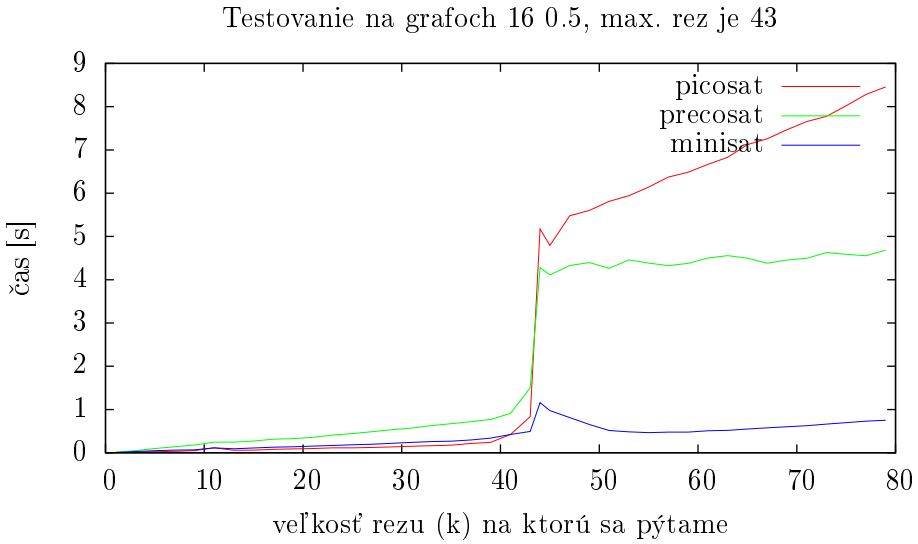


Obr. 4.1: Porovnanie SAT–solverov na probléme monochromatického trojuholníka

Rez grafu

V tomto prípade sme testovali SAT–solvery na grafoch so 16 vrcholmi s pravdepodobnosťou hrany 50%, ktoré mali maximálny rez veľkosti 43. V grafe na obrázku 4.2 je vidno, že teraz čas závisí od odpovede. Kým v grafe rez danej veľkosti exituje, sú to všetky solvery veľmi rýchlo schopné zistiť. Potom sa už správajú rôzne.

Picosat je najpomalší a dokonca má stále väčší problém zistiť, že už väčší rez neexistuje. Je to preto, že kódovanie pre väčšie k obsahuje viac premenných. Precosatu to ale nevadí a odpovedá stále rovnako rýchlo. Minisat je niekoľkokrát rýchlejší a má najväčší problém dokázať, že neexituje rez veľkosti 44. Potom to už zvláda rýchlejšie, ale neskôr zase funkcia rastie kvôli veľkému počtu premenných.



Obr. 4.2: Porovnanie SAT–solverov na probléme rezu grafu

Hamiltonovská kružnica

Pri tomto probléme najlepšie výsledky dosahoval precosat. Je to vidno v tabuľkách 4.2 a 4.3, v časti 4.4.6. Pri grafoch, v ktorých sa kružnica nenachádza je síce podobne rýchly ako ostatné, avšak vie veľmi rýchlo aj kružnicu nájsť, s čím už majú picosat a minisat problém (tab. 4.3).

4.2 Ako vymyslieť kódovanie

Nie je jedno aké kódovanie sa použije, aj keby sa líšili iba minimálne. Tak tomu bolo pri kódovaní rezu v grafe. Pridali sme navyše premenné, aby sme mohli zjednodušiť klauzuly. Výsledkom je, že sme dosiahli lepšie časy, aj keď iba mierne, ako vidno na obrázku 4.10 v časti 4.4.4. Pri ťažkých inštanciách problému už je takmer jedno, ktoré kódovanie použijeme (tab. 4.1).

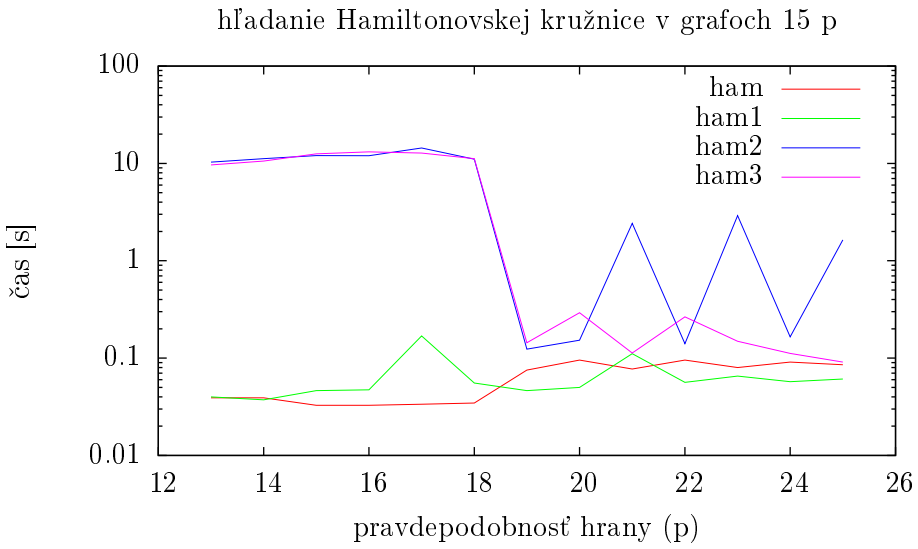
Najlepšie ale vidno rozdiely v kódovaní pri probléme Hamiltonovskej kružnice. Porovnávali sme štyri rôzne kódovania, ktoré sú popísané v časti 3.6 na malých grafoch, iba s 15 vrcholmi. Použili sme precosat, ktorý je pre daný problém najlepší.

Kódovania ϕ_{ham} a ϕ_{ham1} sú veľmi podobné, avšak v niektorých prípadoch, keď graf Hamiltonovskú kružnicu neobsahoval, bolo kódovanie ϕ_{ham1} aj 100 násobne pomalšie. ϕ_{ham} má oproti nemu iba nejaké klauzuly navyše, preto sa domnievame, že pomocou nich sa dá veľmi rýchlo vylúčiť splniteľnosť formule. Žiaľ, toto pozorovanie na obrázku nevidno. Museli by sme

testovať na väčších grafoch. Na týchto grafoch precosat odpovedal vždy takmer okamžite.

ϕ_{ham2} má takisto klauzuly navyše oproti ϕ_{ham3}, a sú tiež veľmi podobné. Sú podstatne pomalšie oproti predchádzajúcim dvom kódovaniam, hoci majú rádovo rovnaký počet premenných a takisto aj klauzúl. V tomto prípade sme nespozorovali, že jedno z nich by bolo niekedy výrazne lepšie.

Porovnanie vidno na nasledujúcom obrázku 4.3.



Obr. 4.3: Porovnanie rozličných kódovaní pri probléme Hamiltonovskej kružnice

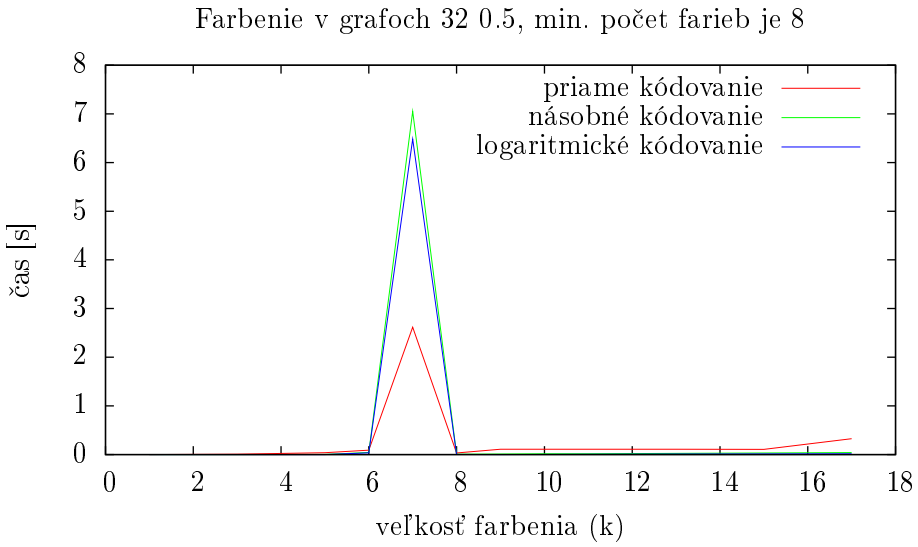
4.3 Správanie v hraničných situáciach

Čo je ľahšie? Dokazovať alebo vyvracať tvrdenie. V tejto časti si ukážeme, že ani to, či sa skôr podarí nájsť príklad alebo dokázať, že neexistuje, sa vo všeobecnosti nedá rozhodnúť. Väčšinou SAT–solver rýchlejšie odpovie, že je formula splniteľná, ako keby mal zisťovať, že nie je. Tak tomu bolo aj v predchádzajúcej časti, keď sme pozorovali správanie pri hľadaní rezu v grafe (obr. 4.2). Veľmi výrazne to je vidno aj pri probléme farbenia (obr. 4.4).

Farbenie

Farbenie je veľmi ťažký problém a zväčšenie počtu vrcholov iba o 1, môže spôsobiť niekoľkonásobný časový nárast. Ťažké je zistiť najväčší počet farieb, ktorý nestačí na ofarbenie (obr. 4.4). Preto sme museli zvoliť rozumne malý

graf.



Obr. 4.4: Pozorovanie závislosti času od odpovede pri farbení grafu

Hamiltonovská kružnica

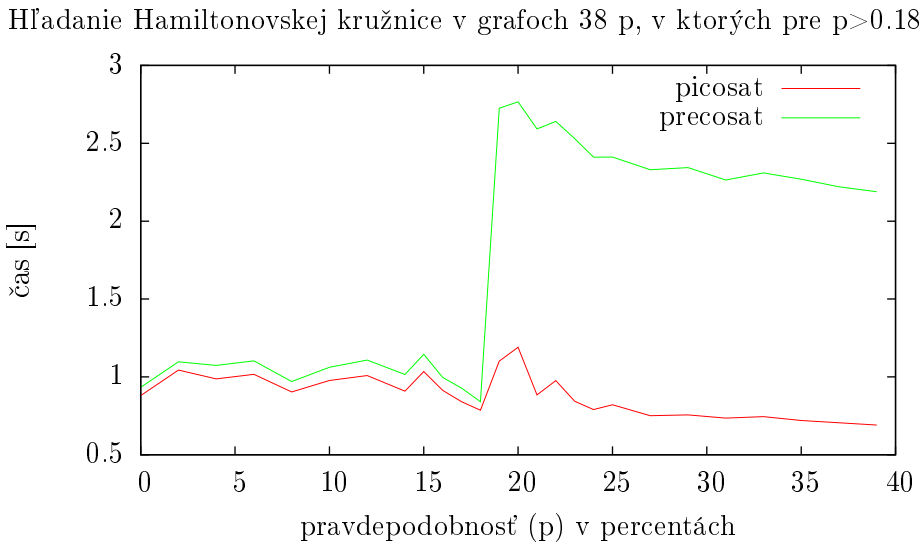
Avšak pri hľadaní Hamiltonovskej kružnice tomu je naopak (obr. 4.5). Testovali sme na grafoch 38 p. Vyberali sme grafy, ktoré pre $p > 0.18$ kružnicu obsahujú a pre $p \leq 0.18$ neobsahujú. Je to prirodzená hranica, aby sme nemuseli generovať zbytočne veľa grafov, ktoré nemajú požadovanú vlastnosť. Pri použití precosatu rozdiel veľmi jasne vidno.

Nevieme prečo v tomto prípade dlhšie trvá kružnicu nájsť ako zistiť, že žiadna v grafe nie je. Veľmi zreteľne to je vidno aj v tabuľkách 4.2 a 4.3. Avšak backtrack, samozrejme, nemôže skôr vylúčiť existenciu kružnice ako ju nájsť, lebo iba prehľadáva. SAT–solvery sa pravdepodobne pri prehľadávaní vedia niečo naučiť, na základe čoho potom môžu existenciu kružnice vylúčiť.

Klika

Domnievame sa, že správanie v okolí hranice medzi splniteľnosťou a nesplniteľnosťou je vlastnosť daného problému. Presnejšie, daného typu inštancií problému (porovnanie obr. 4.6 a obr. 4.12). V časti 4.5 sa totiž venujeme ťažkým inštanciám k-kliky.

Nemusí sa to samozrejme rovnako prejaviť pri všetkých kódovaniach alebo použitých SAT–solveroch (obr. 4.5). Potvrdzuje to aj nasledujúci prí-



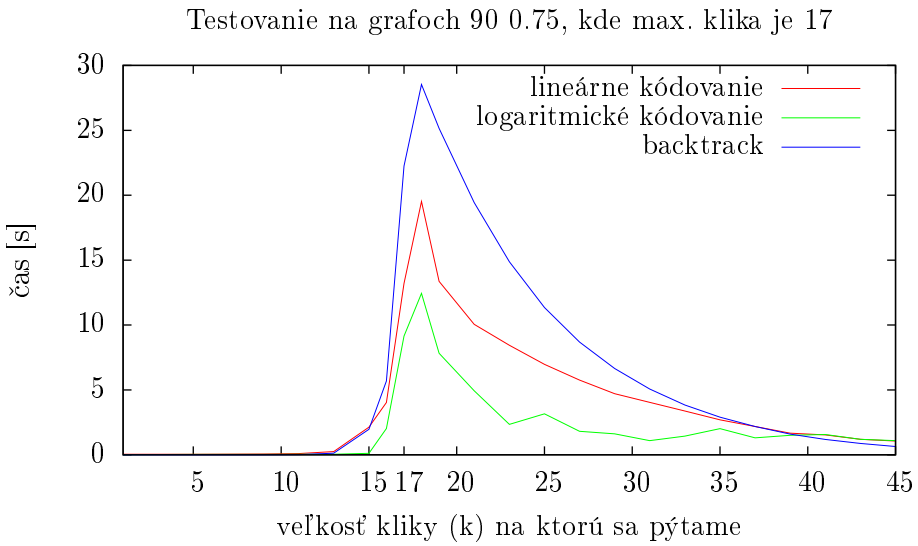
Obr. 4.5: Pozorovanie závislosti času od odpovede pri hľadaní Hamiltonovskej kružnice

klad.

Merali sme čas hľadania k-kliky v pomerne veľkom grafe (obr. 4.6). Použili sme minisat a dve možné kódovania kliky a správanie je veľmi podobné pri oboch kódovaniach a takisto pri použití backtracku. Trvá trochu dlhšie povedať, že riešenie neexistuje ale takisto aj nájsť riešenie blízko hranice trvá podstatne dlhšie ako kúsok ďalej od hranice.

4.4 Porovnanie s backtrackmi

Na záver by sme chceli na problémoch, ktorým sme sa venovali ukázať, či je riešenie sprostredkované pomocou SAT–solvera rýchlejšie, ako pomocou špeciálne vytvorených backtrackov pre daný problém. Je to trochu problematické, lebo naše algoritmy, ktoré prehľadávajú mierne orezaný priestor riešení, sa pri niektorých problémoch správajú veľmi nepredvídateľne. Nedajú sa totiž dopredu odlíšiť ťažké inštancie problému od ľahkých. Na grafoch s rovnakými vlastnosťami preto niekedy môže trvať výpočet veľmi krátko a inokedy naopak podstatne dlhšie. Funkcie preto pri niektorých problémoch nie sú hladké, ale skôr zubaté.



Obr. 4.6: Pozorovanie závislosti času od odpovede pri hľadaní kliky

4.4.1 Monochromatický trojuholník

Tento problém je pre backtrack veľmi náročný. Už 11 vrcholové grafy pre $p > 50\%$ totiž obsahujú príliš veľa hrán na to aby sme ich stihli v krátkom čase rozdeliť na dve množiny všetkými možnými spôsobmi.

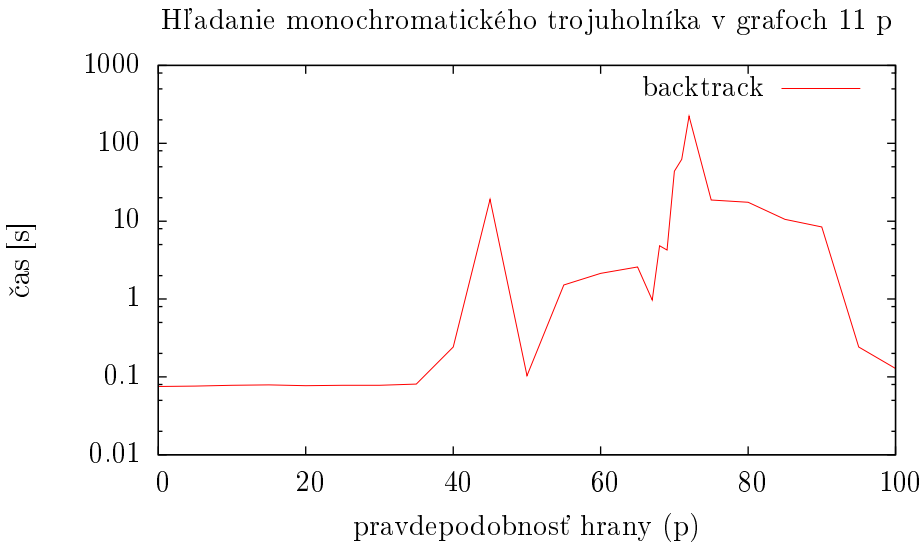
Na obrázku 4.7 je vidno čas, ktorý potreboval backtrack na riešenie problému. Je veľmi rôznorodý, a aj preto sme nespozorovali nejaký výrazný rozdiel medzi grafmi s monochromatickým trojuholníkom a bez neho. Testovali sme na grafoch s 11 vrcholmi. Vyberali sme grafy, ktoré pre $p < 70\%$ trojuholník neobsahujú a pre $p \geq 70\%$ obsahujú. Pre väčšie grafy by to už trvalo pridlho. Pre $p = 72\%$ narazil backtrack až na dve ťažké inštancie. Aj preto je tam veľký rozdiel a použili sme logaritmickú škálu.

Tento problém je teda extrémne výhodné redukovať na SAT. Tak je totiž možné hľadať trojuholníky aj na grafoch so 150 vrcholmi (4.1) v krátkom čase.

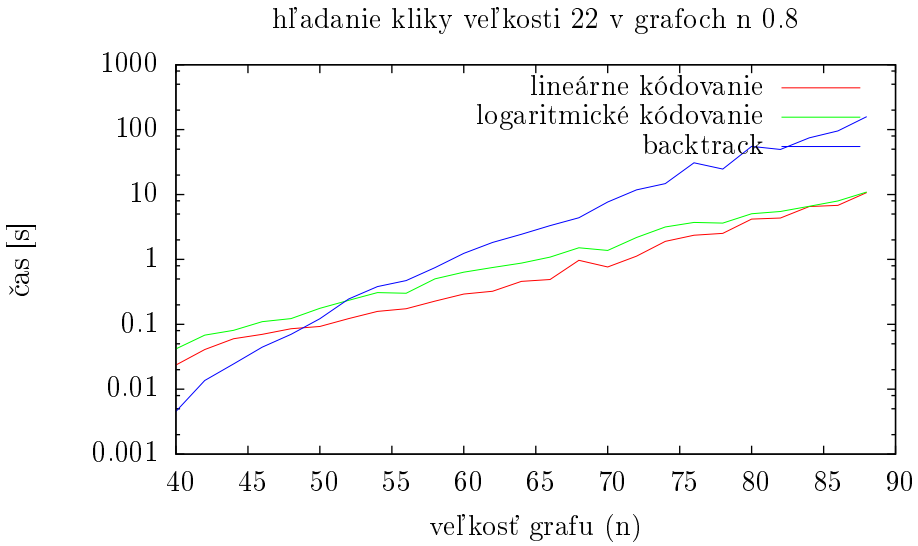
4.4.2 Klika

Dobre sa oba spôsoby porovnávajú na probléme k-kliky, pretože sú na menších grafoch približne rovnako rýchle. Avšak pre väčšie grafy už je backtrack rádovo pomalší, ako vidno na obrázku 4.8.

Hľadali sme kliku veľkosti 22 v rôzne veľkých grafoch s pravdepodobnosťou hrany 80%. Parametre boli nastavené tak, že sa tam klika danej veľkosti nikdy nenachádzala. Ako SAT–solver sme použili precosat.



Obr. 4.7: Pozorovanie času behu backtracku na probléme monochromatickosti

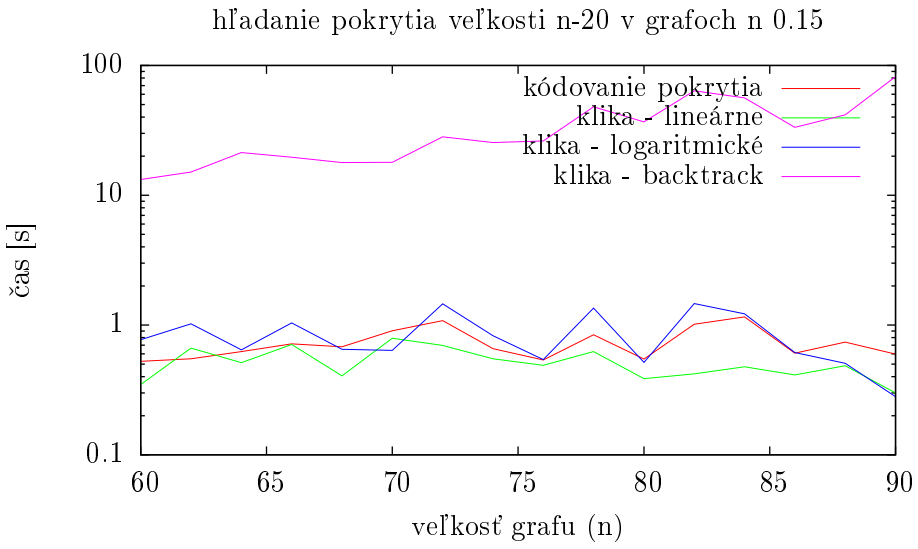


Obr. 4.8: Pozorovanie závislosti času od veľkosti na probléme kliky

4.4.3 Vrcholové pokrytie

Neimplementovali sme špeciálny backtrack pre riešenie vrcholového pokrytia, ale redukovali sme pokrytie na hľadanie kliky. Aj v tomto prípade je riešenie pomocou SAT–solvera rádovo rýchlejšie, či už riešime problém priamo pomocou kódovania pokrytia alebo najprv redukujeme problém na kliku a až potom zakódujeme (obr. 4.9).

Testovali sme grafy s pravdepodobnosťou hrany 15%, v ktorých existovalo pokrytie veľkosti $n - 20$. V tomto prípade, pre menšie n, bolo potrebné niekedy vygenerovať nový graf, lebo sa občas stalo, že v ňom pokrytie neexistuje. Avšak pre $n \geq 70$ už bola pravdepodobnosť taká malá, že z 11 pokusov všetky grafy pokrytie obsahovali. Ako SAT–solver sme použili precosat.



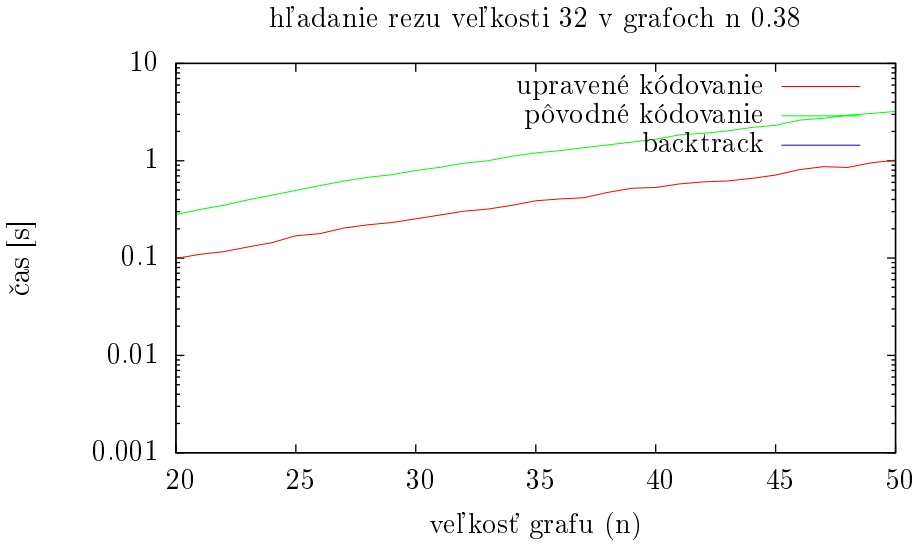
Obr. 4.9: Pozorovanie závislosti času od veľkosti na probléme pokrytia

4.4.4 Rez grafu

Toto je jediný prípad, v ktorom bol SAT–solver pomalší (obr. 4.10). Je to zrejme preto, že kódovanie používa rádovo m^2 premenných, čo je rádovo n^4. Celkový počet možných riešení je teda rádovo $2^{(n^4)} = 16^n$, kým backtrack skúša rádovo 2^n možností.

Testovali sme rýchlosť nájdenia rezu veľkosti 32 v grafoch s pravdepodobnosťou hrany 38%. Nemuseli sme generovať viac grafov ako 11, lebo sa v nich rez vždy nachádzal. Použili sme minisat. Funkciu pre backtrack na obrázku 4.10 nevidno, lebo vždy odpovedal okamžite.

Ale aj v tomto prípade môže byť použitie SAT–solvera výhodnejšie. Rez



Obr. 4.10: Pozorovanie závislosti času od veľkosti na probléme rezu grafu

grafu, ako sa zdá, má totiž nerovnomerne rozmiestnené ťažké inštancie medzi grafmi, ktoré vieme generovať. Preto, hoci väčšinou backtrack skončí ihneď, niekedy je veľmi pomalý. Zatiaľ čo SAT–solveru trvá výpočet rovnomernejšie. Ukážeme si preto na niekoľkých typoch grafov porovnanie v tabuľke 4.1. Testovali sme grafy s 55, 60 a 65 vrcholmi. Pravdepodobnosť hrany bola 38%. Rez veľkosti 32 sa v grafe vždy nachádzal. Niektoré merania sme ukončili bez toho aby sme sa dozvedeli výsledok, ale veľmi pravdepodobne sa aj v nich rez nachádzal. V tých prípadoch sme do priemeru rátali čas, kedy sme program zastavili. Podľa priemeru v poslednom riadku vidíme, že je niekedy výhodné použiť SAT–solver. Má totiž väčšiu šancu na úspech pri ťažkých inštanciách. Je to veľmi pravdepodobne kvôli tomu, že využíva náhodné reštarty a preto sa nezasekne v nejakej vetve, ktorá nevedie k riešeniu, čo sa ľahko môže stať backtracku.

4.4.5 Farbenie

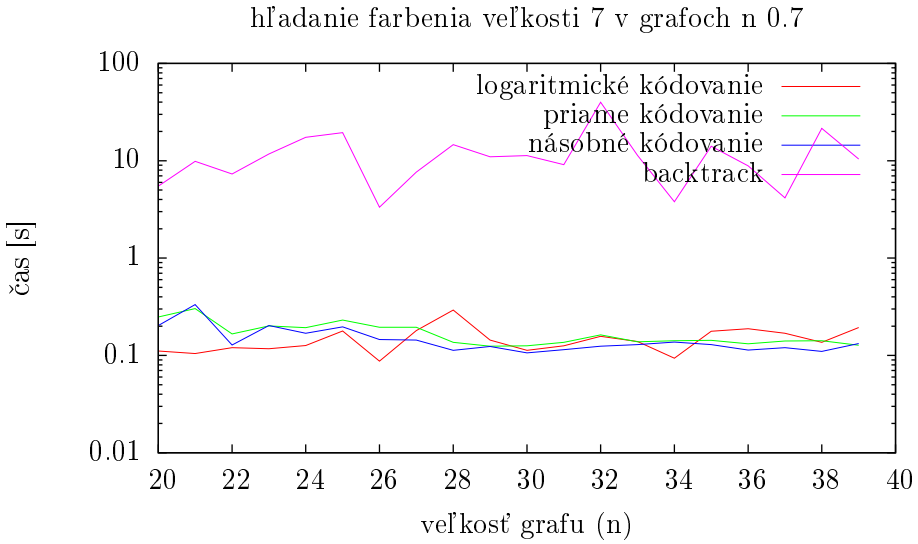
Tento problém je kombinatoricky veľmi náročný a aj preto sme museli testovať na malých grafoch. Pri väčších grafoch je už backtrack veľmi pomalý. Použili sme aspoň pomerne veľkú pravdepodobnosť výskytu hrany 70%. Vyberali sme grafy, ktoré sa siedmimi farbami ofarbiť nedajú. Na riešenie sme použili picosat. (obr. 4.11). Podľa obrázka to vyzerá tak, že je jedno, ktoré kódovanie použijeme. Je to kvôli tomu, že vzhľadom na backtrack sú všetky niekoľkonásobne rýchlejšie. V skutočnosti by ale logaritmické kódovanie malo byť vo väčšine prípadov rýchlejšie.

Tabuľka 4.1: Porovnanie backtracku a SAT–solverov na grafoch n 0.38 pri hľadaní rezu veľkosti 32

	grafy 55 0.38			grafy 60 0.38			grafy 65 0.38		
č.	bt.	pôv.	up.	bt.	pôv.	up.	bt.	pôv.	up.
1	0.00	6.24	1.70	0.02	18.26	4.40	0.38	15.48	6.06
2	0.00	6.24	2.66	0.02	7.65	2.36	0.00	11.60	3.84
3	0.00	6.79	1.91	1.64	8.13	15.58	27.37	11.55	3.35
4	0.00	5.74	1.77	0.00	38.09	84.81	>1800	10.51	5.78
5	0.16	6.60	1.82	0.77	9.68	2.29	0.00	17.19	12.36
6	0.00	6.24	1.70	>660	9.23	24.85	3.32	279.54	36.65
7	0.00	6.65	2.05	0.01	7.60	2.29	0.18	10.26	10.26
8	0.00	6.79	1.63	0.00	9.35	6.29	0.00	10.37	14.92
9	0.04	13.49	119.8	0.73	8.20	2.35	0.00	10.28	6.80
10	0.69	6.16	1.78	>660	>660	>660	>1800	15.34	17.55
11	0.00	6.46	1.61	>660	27.09	10.24	0.00	10.79	374.11
12	0.16	6.78	2.18	0.02	9.30	6.63	>1800	65.66	16.47
13	0.00	5.92	1.94	0.10	9.11	3.14	0.82	18.94	6.32
14	0.00	6.79	3.24	0.00	9.30	2.35	0.00	94.99	5.02
15	0.00	6.43	1.62	0.27	8.40	3.80	>1800	>1800	>1800
16	0.00	6.44	2.24	700	13.21	6.76	>1800	>1800	>1800
17	11.52	7.00	4.36	0.02	7.94	2.48	0.00	11.98	6.18
18	0.00	6.60	1.89	418.84	18.15	2.99	>1800	21.46	14.37
19	0.00	6.52	1.89	>660	7.84	2.73	0.00	11.16	50.51
20	0.00	6.84	1.89	0.37	8.40	2.39	0.06	537.61	>1800
pr.	0.63	6.84	7.98	186.14	44.75	42.44	541.61	238.24	299.53

4.4.6 Hamiltonovská kružnica

Aj pri tomto probléme sme neboli schopní vygenerovať rozumný obrázok, keďže nevieme oddeliť ťažké inštancie. Preto porovnávame v tabuľkách 4.2 a 4.3. Pri grafoch veľkosti 30 stihli programy v časovom limite ukončiť výpočet aj na grafoch s kružnicou, aj bez nej. Pri veľkosti 40 sa už backtracku podarilo do polhodiny nájsť iba 4 kružnice zo 14. V žiadnom prípade sa mu nepodarilo prehľadať celý priestor riešení, takže nemohol povedať, že sa tam kružnica nenachádza. SAT–solvery stihli odpovedať vždy a je zaujímavé, že keď sa v grafe kružnica nenachádza, trvá im to kratšie. Výnimkou je minisat pri grafoch veľkosti 30. Mal niekedy problém vylúčiť existenciu kružnice. Nevyzerá to na náhodu, ale nevieme, čím by to mohlo byť spôsobené. Pravdepodobne teda je možné nejakými heuristikami ľahko vylúčiť existenciu kružnice, čo sme si neuvedomili a nezohľadnili pri našom algoritme. Je ale veľmi povzbudivé, že keď kódujeme problém na SAT, nemusí nás to zají-



Obr. 4.11: Pozorovanie závislosti času od veľkosti na probléme farbenia

mať, pretože podstata problému ostane zakódovaná a SAT–solvery používajú rozličné heuristiky a technológie, ktoré sa neustále zdokonaľujú a sú schopné veľmi rýchlo kružnicu vylúčiť, aj keď pre nich sú to iba nejaké klauzuly.

Tabuľka 4.2: Porovnanie backtracku a SAT–solverov na grafoch 30 0.15 pri hľadaní Hamiltonovskej kružnice

grafy 30 0.15 s kružnicou					grafy 30 0.15 bez kružnice				
č.	bt	mini	pico	preco	č.	bt	mini	pico	preco
1	0.02	0.95	1.02	1.68	1	1198.07	0.77	0.44	0.54
2	524.08	4.53	11.03	2.90	2	36.17	0.87	0.48	0.54
3	0.06	4.35	1.04	2.19	3	33.95	0.76	0.51	0.56
4	75.80	4.72	10.47	1.81	4	6.96	178.01	0.52	0.53
5	0.01	18.97	34.53	11.02	5	832.23	91.94	0.45	0.52
6	36.36	1.94	11.15	6.64	6	6.71	392.74	0.50	0.49
7	61.25	1.17	8.26	5.09	7	0.54	0.75	0.51	0.58
8	0.03	1.02	0.94	1.60	8	5.00	338.20	0.49	0.55
9	2.91	24.30	25.01	20.37	9	34.75	0.78	0.47	0.54
10	3.93	2.86	10.42	3.41	10	1.57	0.77	0.46	0.53
11	0.00	6.11	0.74	5.66	11	124.65	0.77	0.46	0.54
pr.	64.04	6.45	10.42	5.67	pr.	207.33	91.49	0.48	0.54

Tabuľka 4.3: Porovnanie backtracku a SAT–solverov na grafoch 40 0.15 pri hľadaní Hamiltonovskej kružnice

grafy 40 0.15 s kružnicou					grafy 40 0.15 bez kružnice				
č.	bt	mini	pico	preco	č.	bt	mini	pico	preco
0	>2000	33.35	114.52	15.61	2	>2000	2.36	1.37	1.40
1	139.12	4.01	2.06	9.78	4	>2000	2.28	1.28	1.47
3	>2000	8.10	8.62	6.18	6	>2000	2.92	1.36	1.53
5	>2000	192.76	514.00	19.44	7	>2000	2.33	1.32	1.52
8	850.30	3.28	2.94	6.10	11	>2000	2.43	1.32	1.45
9	>2000	2.86	5.56	4.44	12	>2000	2.26	1.38	1.50
10	428.44	5.43	4.22	4.71	13	>2000	2.31	1.32	1.41
15	>2000	2.64	3.34	9.60	14	>2000	3.65	1.37	1.42
16	18.92	4.76	32.13	8.92	19	>2000	3.14	1.27	1.50
17	>2000	6.64	1.42	5.11	20	>2000	2.24	1.36	1.52
18	>2000	180.11	74.08	13.60	21	>2000	2.23	1.28	1.46
22	>2000	33.03	38.43	11.13	23	>2000	2.38	1.34	1.48
24	>2000	3.71	10.08	6.19	25	>2000	4.40	1.39	1.51
26	>2000	4.74	29.78	6.33	28	>2000	2.27	1.39	1.50
pr.	1531.2	34.67	60.08	9.08	pr.	2000	2.66	1.34	1.48

4.5 Ťažké inštancie

Nakoniec sme spravili na probléme k-kliky porovnanie na ťažkých inštanciách [Xu]. Sú to grafy získané transformáciou z porovnávacích testov pre SAT. Vytvorené sú s úmyslom aby v nich bolo ťažké nájsť kliku.

Na testovanie sme použili 5 grafov so 450 vrcholmi. Každý z nich má maximálnu kliku veľkosti 30. Obsahujú približne 82% hrán, preto ich porovnávame s grafmi 450 0.82.

Backtrack s nimi mal naozaj problém. Nepodarilo sa mu nájsť ani kliku veľkosti 20, čo pri náhodnom grafe trvá pomerne krátko. V jednom prípade sme ho nechali bežať podstatne dlhšie.

Z druhej strany sme testovali hľadanie kliky veľkosti 300. To sa na náhodných grafoch podarilo vyvrátiť. Na ťažkých inštanciách sme výpočet zastavili.

Namerané hodnoty zhŕňa tabuľka 4.4. Samozrejme, nemôžeme vedieť aká bola v skutočnosti najväčšia klika v náhodných grafoch. Vyskúšali sme na jednom z nich zistiť túto hodnotu pomocou SAT–solvera. Kliku veľkosti 29 našiel za 6m, 30 za 0.5h a veľkosti 31 za 13 hodín. Domnievame sa teda, že to je maximálna klika a dokázať to by trvalo ešte dlhšie. Vychádzame z pozorovaní v časti 4.3 (obr. 4.6).

Tabuľka 4.4: Porovnanie backtracku na ťažkých a náhodných inštanciách k-kliky

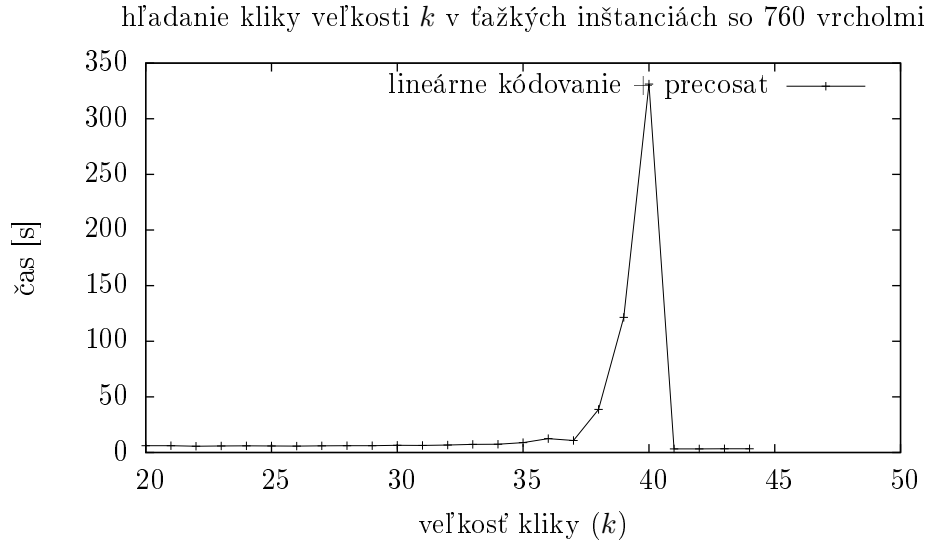
ťažké inštancie so 450 vrcholmi, max. klika je veľkosti 30					
graf č.	1	2	3	4	5
k=20	>24h	>9h	>9h	>9h	>9h
k=300	72m	314m	398m	14m	127m
náhodné grafy 450 0.82, max. klika je približne veľkosti 31					
graf č.	1	2	3	4	5
k=20	275s	120s	586s	1474s	43s
k=300	>10h	>10h	>10h	>10h	>10h

Zamerajme sa teraz na riešenie problému cez SAT. Na spomínanom obrázku 4.6 vidno ako sa správajú SAT–solvery na náhodných grafoch. Platí to tak všeobecne, aj pre väčšie grafy je tvar funkcie približne rovnaký. Rozdiel je iba v tom, že funkcia nadobúda pre väčšie n podstatne väčšie hodnoty a takisto by mala byť strmšia. Neboli sme preto schopní ani pomocou SAT–solverov zistiť maximálnu kliku v grafoch 450 0.82. Trvalo to priveľmi dlho.

Avšak tieto ťažké inštancie vieme pomocou SAT–solverov veľmi úspešne riešiť. Konkrétne, ak použijeme lineárne kódovanie a precosat. Aj niektoré iné kombinácie boli celkom rýchle (napr. minisat s binárnym kódovaním), ale táto bola výrazne najlepšia. Ťažké inštancie sme boli pomocou precosatu schopní vyriešiť pre všetky možné veľkosti kliky do 10 sekúnd. Preto na obrázku 4.12 sú inštancie až so 760 vrcholmi a maximálnou klikou 40. Graf pre inštancie so 450 vrcholmi by mal rovnaký tvar a maximálnu časovú hodnotu 10s.

Odlišnosťou je, že v tomto prípade vieme veľmi rýchlo zistiť, ak graf kliku neobsahuje. Ukázalo sa, že štruktúra v grafe, ktorá mala za cieľ spraviť inštanciu ťažkou, spôsobila, že s ňou mal SAT–solver omnoho menšie problémy ako s náhodnými grafmi. Malo by preto zmysel v ďalšej práci sa venovať aj iným takýmto problémom a zistiť nakoľko všeobecne to platí. Či existujú inštancie, s ktorými má SAT–solver väčšie problémy ako s náhodnými a nakoľko sa vyskytujú v praktických problémoch, ktoré potrebujeme riešiť. V mnohých prípadoch sa totiž ukazuje, ako sme okrajovo spomínali v časti 1.2, že SAT–solvery sú veľmi úspešné a používané. Vďaka tomu je o ne záujem a ich vývoj neustále napreduje.

Na záver ešte podotknime, že má zmysel sa venovať rôznym kódovaniam a rôznym SAT–solverom lebo niekedy sú medzi nimi veľké časové rozdiely. Hlavne to má zmysel, ak chceme vyriešiť naozaj ťažkú inštanciu a máme k dispozícii aj menšie s podobnou štruktúrou, na ktorých môžeme spraviť porovnanie. Z doterajších skúseností sa domnievame, že porovnanie by bolo s veľkou pravdepodobnosťou podobné aj na veľkej inštancii.



Obr. 4.12: Pozorovanie závislosti času od odpovede na ťažkých inštanciách k-kliky

Záver

V našej práci sme si dali za cieľ preskúmať možnosti využitia SAT–solverov pri riešení NP-úplných úloh.

V prvej kapitole sme sa venovali algoritmom, ktoré sú základom SAT–solverov a spomenuli sme oblasti, v ktorých sa už úspešne používajú.

V ďalšej časti sme sa zamerali na šesť rôznych grafových problémov a ku každému uviedli backtrack, ktorý sme implementovali a použili na riešenie. Takisto sme pre každý problém zostrojili aspoň jedno kódovanie, pomocou ktorého sme ho mohli previesť na SAT.

V poslednej kapitole prezentujeme viaceré testy a porovnania, ktoré sme uskutočnili. Ukázalo sa, že nie je jedno aký SAT–solver použijeme. V niektorých prípadoch sú rozdiely veľmi veľké a neexistuje SAT–solver, ktorý by bol najrýchlejší pre každý problém a kódovanie.

Pre niektoré problémy sme porovnávali viaceré kódovania. Môžu byť veľké rozdiely v čase zistenia splniteľnosti aj napriek tomu, že obsahujú približne rovnako veľa premenných a klauzúl.

Ďalej sme skúmali ako rýchlo SAT–solver odpovie na náhodných grafoch s rovnakými parametrami, z ktorých jeden má a druhý nemá danú vlastnosť. Takmer vždy trvalo kratšie nájsť ohodnotenie premenných splniteľnej formule. Iba pri Hamiltonovskej kružnici trvalo dlhšie ju nájsť, ako zistiť, že sa v podobnom náhodnom grafe nenachádza.

Pri porovnaní SAT–solverov s backtrackmi nastal iba jeden prípad keď bolo použitie backtracku výhodnejšie. Pri reze grafu malo totiž kódovanie priveľa premenných a SAT–solver nestačil. Avšak aj pri tom probléme ho pre veľké grafy mohlo byť výhodné použiť, ak sa rez v grafe nachádzal.

Na záver sme ešte spravili porovnanie na ťažkých inštanciách k-kliky. Pre backtrack boli naozaj omnoho ťažšie, najmä ak sme hľadali kliku, ktorá sa v grafe nachádzala. Kliky tam boli totiž 'schované'. Ak sa v grafe k-klika nenachádzala, trval výpočet kratšie ako na náhodných grafoch. Avšak zistili sme, že pre SAT–solver sú dané kódovania jednoduchšie. Najmä použitie lineárneho kódovania a precosatu bolo veľmi úspešné. Preto by bolo zaujímavé sa tomuto v ďalšej práci venovať a porovnať ťažké inštancie iných problémov.

Literatúra

[Bar07] G.V. Bard. Algorithms for solving linear and polynomial systems of equations over finite fields with applications to cryptanalysis. 2007. 5

[BHvMW09] A. Biere, M. Heule, H. van Maaren, and T. Walsh. Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185, 2009. 4, 5

[Bie10] A. Biere. Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT Race 2010. Technical report, Technical Report 10/1, Institute for Formal Models and Verification, Johannes Kepler University, 2010. 4

[Coo71] S.A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158. ACM, 1971. 2

[Die00] R. Diestel. Graph Theory. 2000. *Graduate Texts in Mathematics*, 2000. 6

[DLL62] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962. 3

[DP60] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM (JACM)*, 7(3):201–215, 1960. 3

[HHU07] A. Hertel, P. Hertel, and A. Urquhart. Formalizing dangerous sat encodings. *Theory and Applications of Satisfiability Testing–SAT 2007*, pages 159–172, 2007. 17

[HS00] H.H. Hoos and T. Stützle. Local search algorithms for SAT: An empirical evaluation. *Journal of Automated Reasoning*, 24(4):421–481, 2000. 4

[Kar10] R.M. Karp. Reducibility among combinatorial problems. *50 Years of Integer Programming 1958-2008*, pages 219–241, 2010. 1

[KS92] H. Kautz and B. Selman. Planning as satisfiability. In *Proceedings of the European Conference on Artificial Intelligence ECAI*, volume 54, pages 359–363. John Wiley & Sons, Inc., 1992. 5

[Lab10] I. Labáth. Automatické generovanie logických úloh s využitím SAT solvera : bakalárska práca, 2010. 5

[Len10] R. Lenhardt. Proof of concept: Fast solutions to np-problems by using sat and integer programming solvers. *Arxiv preprint arXiv:1011.5447*, 2010. 11

[Lev73] L.A. Levin. Universal search problems. *Problemy Peredachi Informatsii*, 9(3):265–266, 1973. 2

[N. 05] N. Een and N. Sörensson. MiniSat: A SAT solver with conflict-clause minimization. *Sat*, 5, 2005. 4

[SKC94] B. Selman, H.A. Kautz, and B. Cohen. Noise strategies for improving local search. In *Proceedings of the national conference on artificial intelligence*, pages 337–337. JOHN WILEY & SONS LTD, 1994. 4

[Tol97] J.R.R. Tolkien. *The hobbit, or, There and back again*. Ballantine, 1997. 11

[Vel07] M.N. Velev. Exploiting hierarchy and structure to efficiently solve graph coloring as SAT. In *Proceedings of the 2007 IEEE/ACM international conference on Computer-aided design*, pages 135–142. IEEE Press, 2007. 13, 14

[Xu] K. Xu. Bhoslib: Benchmarks with hidden optimum solutions for graph problems (maximum clique, maximum independent set, minimum vertex cover and vertex coloring)–hiding exact solutions in random graphs. web site. 30