Lista zadań na pracownię nr 5

Uwaga: zadania na pracownię obowiązują do wtorku po świętach, 3 kwietnia 2018 r., do godziny 0600. Przepraszam za opóźnienie w przygotowaniu zadań, ale mam nadzieję że te są ciekawe! — FS

Spełnialność formuł logicznych w CNFie: rezolucja.

Na ćwiczeniach i repetytorium w ubiegłym tygodniu zobaczyliśmy jak łatwo sprawdzić czy formuła w koniunkcyjnej postaci normalnej jest tautologią (i znaleźć wartościowanie falsyfikujące ją jeśli tautologią nie jest). Znacznie trudniejszym problemem, którym zajmiemy się w zadaniu na pracownię, jest sprawdzenie czy formuła w CNFie jest *spełnialna*, tj. czy istnieje wartościowanie spełniające taką formułę.

Problem spełnialności formuł rachunku zdań jest bardzo istotny, gdyż wiele praktycznych problemów występujących w informatyce (od reprezentacji i poprawności układów logicznych po zagadnienia związane z automatyczną weryfikacją programów) można przedstawić jako instancje tego problemu. Jest on jednak trudny obliczeniowo: jest sztandarowym przedstawicielem rodziny problemów dla których potrafimy efektywnie *sprawdzić* rozwiązanie (mając dane wartościowanie umiemy sprawdzić czy spełnia ono formułę), ale nie umiemy go efektywnie *znaleźć* (co więcej, większość teoretyków uważa że *nie da się* tego zrobić efektywnie).[1]

Nie oznacza to jednak że nie możemy próbować: wymyślono wiele technik pozwalających na relatywnie efektywne poszukiwanie wartościowania spełniającego daną formułę (lub stwierdzenie że jest ona sprzeczna). Dziś zajmiemy się implementacją jednej z ciekawszych z nich: rezolucji.

[1] Oczywiście, trywialnie można sprawdzić spełnialność formuły w DNFie: problemem jest sprowadzenie formuły do DNFu. Translacje do równoważnych formuł w CNFie lub DNFie mogą wykładniczo zwiększyć rozmiar formuły, ale do CNFu łatwo przetłumaczyć formułę tak żeby zachować jej *spełnialność* (używając świeżych zmiennych) — podczas gdy dla DNFu jest to równie trudne jak problem spełnialności CNFu.

Rezolucja w rachunku zdań.

Rezolucję dla rachunku zdań możecie pamiętać z poprzedniego semestru, z logiki dla informatyków. Dla tych którzy nie pamiętają, poniżej zamieszczamy krótkie przypomnienie.

Aby zdefiniować potrzebne pojęcia, przyjmijmy że przedstawiamy formułę logiczną w CNFie jako zbiór klauzul, z których każda jest zbiorem literałów (konkretną reprezentacją w Rackecie zajmiemy się później, tę przyjmiemy po stronie matematycznej). Kluczową obserwacją jest że zbiór klauzul $\{c_1 \cup \{p\}, c_2 \cup \{\neg p\}\}$, gdzie p nie występuje w c_1 ani c_2 jest spełnialny wtedy i tylko wtedy, gdy spełnialna jest klauzula $c_1 \cup c_2$ — pamiętając przy tym że zbiory c_i interpretujemy jako *koniunkcje* swoich elementów! Klauzulę $c_1 \cup c_2$ nazywamy *rezolwentą* klauzul c_1 i c_2.

Z naszej obserwacji wynika, że jeśli weźmiemy zbiór klauzul X (reprezentujący pewną formułę), to X jest spełnialny wtedy i tylko wtedy, gdy $X \cup \{\text{resolve}(c_1, c_2)\}$ jest spełnialny, gdzie $\text{resolve}(c_1, c_2)$ oznacza wzięcie rezolwenty pewnych klauzul $c_1, c_2 \in X$ dla których rezolwenta istnieje, tj. istnieje literał który występuje pozytywnie w jednej z nich, a negatywnie w drugiej. Możemy teraz oznaczyć przez $\overline{X}$ zbiór Y taki, że $X \subseteq Y$ i Y jest domknięty ze względu na rezolucję, tj. wszystkie możliwe rezolwenty klauzul z Y już należą do Y. Łatwo wtedy pokazać że X jest spełnialny wtedy i tylko wtedy gdy nie można przez rezolucję otrzymać klauzuli pustej, tj. gdy $\emptyset \notin \overline{X}$. Jeśli zaś klauzula pusta należy do $\overline{X}$, to powstała przez rezolucję z jakichś innych klauzul — ten ślad stanowi *rezolucyjny dowód sprzeczności* dla X.

Oczywistym problemem jest, że zbiór $\overline{X}$ może być duży — ale okazuje się że nie potrzebujemy go obliczać w całości: zauważmy że jeśli mamy zbiór klauzul Y i pewne $c_1, c_2 \in Y$ takie że $c_1 \subseteq c_2$, to Y jest spełnialny wtedy i tylko wtedy gdy $Y \setminus \{c_2\}$ jest spełnialny (dowolne wartościowanie spełniające c_1 spełnia również c_2). Mówimy wówczas że c_1 *subsumuje* c_2, lub że c_2 jest *łatwiejsza* od c_1. Ta obserwacja pozwala nam na wyliczenie mniejszego (choć potencjalnie nadal dużego) zbioru jako domknięcia wejściowego zbioru X — w szczególności dla formuł sprzecznych ostatecznym zbiorem będzie zawsze $\{\emptyset\}$.

Możemy też zauważyć że jeśli do zbioru klauzul Y należy klauzula o jednym literale $\{l\}$, to z Y można usunąć wszystkie wystąpienia zmiennej występującej w l. Wszystkie klauzule w których występuje l są łatwiejsze od $\{l\}$, a wszystkie klauzule w których występuje negacja l (nazwijmy ją l'), a więc klauzule postaci $c \cup \{l'\}$ rezolwują się z $\{l\}$ do c — a więc do klauzul trudniejszych od klauzul wyjściowych. Powyższe dwie obserwacje pozwalają nam zaimplementować proste, ale skuteczne optymalizacje pozwalające rozwiązać nawet relatywnie duże instancje problemu — choć oczywiście praktyczne SAT-solvery stosują

znacznie więcej, znacznie bardziej wymyślnych optymalizacji.

Implementacja rezolucji w Rackecie.

Większość implementacji algorytmu naszkicowanego powyżej jest przedstawiona w pliku `resolution.rkt`. Pozostało w nim jednak kilka miejsc które należy uzupełnić — zarówno żeby móc w ogóle rozwiązać problem, jak i żeby go usprawnić. Poniżej omówimy kluczowe aspekty kodu i sformułujemy zadania.

Reprezentacja *danych wejściowych* (formuł w CNFie) jest jedną z tych które pojawiły się na ćwiczeniach: formuła jest tagowaną listą klauzul — tagowanych list literałów. Jest to dobra reprezentacja wejściowa, do której można łatwo skonwertować bardziej czytelne reprezentacje przy użyciu procedur z ćwiczeń, ale nie jest wygodna przy rezolucji. Wprowadzamy więc dodatkową, wewnętrzną reprezentację klauzul którą będzie się posługiwał nasz algorytm. Ta dana jest przez poniższy predykat `res-clause?`.

```
(define (res-clause? x)
  (and (tagged-tuple? 'res-int 5 x)
       (sorted-varlist? (second x))
       (sorted-varlist? (third x))
       (= (fourth x) (+ (length (second x)) (length (third x))))
       (proof? (fifth x))))
```

Klauzula jest w tym ujęciu czwórką składającą się z listy zmiennych występujących w niej pozytywnie, listy zmiennych występujących negatywnie (przy czym obie listy są posortowane i nie zawierają powtórzeń), rozmiaru klauzuli (czyli sumy długości obu list) i jej *wyprowadzenia*, które ostatecznie da nam dowód sprzeczności, jeśli osiągniemy klauzulę pustą.

Wyprowadzenia, czyli dowody że nasza klauzula powstała przez rezolucję z początkowej formuły, również są reprezentowane jako dane w Rackecie. Jeśli klauzula jest po prostu częścią początkowego zbioru, jest opatrywana tagiem `'axiom`. W przeciwnym wypadku musi być rezolwentą dwóch innych klauzul — a więc dowód reprezentujemy jako trójkę (tagowaną atomem `'resolve`) zawierającą zmienną względem której przeprowadziliśmy rezolucję, wyprowadzenie klauzuli w której zmienna występowała pozytywnie, i wyprowadzenie klauzuli w której zmienna występowała negatywnie. Procedura `check-proof` sprawdza czy dany dowód faktycznie dowodzi sprzeczności formuły, wyświetlając również informacje o poszczególnych krokach dowodu.

Główną procedurą algorytmu rezolucji jest `resolve-prove`. Jej dwa argumenty to listy klauzul: pierwsza z nich zawiera klauzule już przetworzone, druga — te które jeszcze przetworzyć należy. Zachowujemy przy tym pewne niezmienniki:

- w żadnej z list nie występuje klauzula sprzeczna (w momencie wygenerowania takiej po prostu kończymy dowód)
- jeśli dwie klauzule znajdują się w liście `checked` to ich rezolwenta albo sama jest w jednej z list, albo jest ona łatwiejsza od pewnej klauzuli znajdującej się w jednej z list.

Z niezmienników tych wynika że jeśli lista `pending` jest pusta, to zakończyliśmy poszukiwanie dowodu i formuła jest spełnialna: wystarczy wygenerować waluację na podstawie listy `checked`. W przeciwnym wypadku bierzemy klauzulę do przetworzenia i znajdujemy wszystkie jej rezolwenty z klauzulami już przetworzonymi. Te rezolwenty musimy następnie przetworzyć (w procedurze `subsume-add-prove`): sprawdzić czy wygenerowaliśmy klauzule puste lub trywialne (i wyrzucić te drugie), a następnie dodać je do listy klauzul do przetworzenia. Jest to też miejsce w którym możemy wykonać optymalizacje: przetworzyć w prostszy sposób klauzule jednoelementowe, a w przypadku przetwarzania rezolwent wyrzucić z naszych list klauzule łatwiejsze od innych. Procedura `prove` transformuje formułę w CNFie do opisanej wyżej reprezentacji i uruchamia proces znajdowania dowodu, zaś procedura `prove-and-check` dodatkowo sprawdza poprawność otrzymanego dowodu lub częściowej waluacji.

Kilka fragmentów powyższego programu jest pozostawionych jako zadania: albo nie zaimplementowanych (jak `resolve`), albo z trywialnymi implementacjami które należy poprawić. Poniżej szczegółowo opisujemy wymagania. Wszystkie zaimplementowane procedury, jak też całość programu dowodzącego, należy gruntownie przetestować! Wszystkie trzy zadania należy przesłać w jednym pliku, o nazwie `nazwisko-imie.rkt`, rozwijając szablon `resolution.rkt` tam gdzie to potrzebne. W rozwiązaniu można używać procedur bibliotecznych opisanych w sekcjach 4.1, 4.2, 4.6 i 4.9 dokumentacji, choć poza procedurami użytymi w szablonie nie ma tam chyba procedur istotnie ułatwiających implementację tego zadania.

Ćwiczenie 1. Podstawowa implementacja rezolucji.

W pierwszym zadaniu należy zaimplementować brakujące części podstawowego algorytmu rezolucji, a więc przede wszystkim uzupełnić implementację procedury `resolve`. Należy zatem sprawdzić czy zmienne pozytywne z jednej klauzuli przecinają się niepusto ze zmiennymi negatywnymi drugiej: jeśli tak, należy stworzyć odpowiednią klauzulę jako wynik (pamiętając o stworzeniu jej wyprowadzenia i zachowując istotne niezmienniki!), jeśli nie — zwrócić `false`. Należy też poprawić implementację procedury `clause-trivial?` sprawdzającej czy klauzula jest trywialna (a więc czy zawiera zarówno zmienną, jak i jej negację). W tym celu warto zdefiniować wspólne dla obu procedur procedury

pomocnicze.[2]

Uwaga! To zadanie jest proste, więc styl rozwiązania będzie miał istotny wpływ na ocenę!

Ćwiczenie 2. Odtwarzanie częściowej waluacji.

Jeśli znajdziemy dowód sprzeczności, umiemy go łatwo sprawdzić używając procedury `check-proof`. Nie umiemy jednak sprawdzić czy program nas nie oszukał mówiąc że formuła jest spełnialna kiedy taka nie jest. Aby tego uniknąć, chcemy żeby procedura `generate-valuation` zwracała nie tylko symbol `'sat`, ale też częściową waluację dla której nasza formuła będzie spełniona. Aby wygenerować wartościowanie, trzeba zauważyć że klauzule jednoelementowe wymuszają na nas wybór wartościowania dla ich jedynej zmiennej, ale też że wygenerowanie części wartościowania może sprawić że część pozostałych klauzul będzie trudniejsza do spełnienia — a więc po wygenerowaniu wartości dla każdej zmiennej należy uprościć pozostałe nam jeszcze klauzule. Jeśli znajdziemy się w sytuacji w której każda z klauzul ma co najmniej dwie niezwartościowane jeszcze zmienne — możemy wybrać dowolną. Zwróćcie uwagę że algorytm ten działa wyłącznie dlatego, że zbiór którego używamy jest zamknięty ze względu na rezolucję.

Ćwiczenie 3. Optymalizacje: subsumpcja i klauzule jednoelementowe.

Ostatnim zadaniem związanym z rezolucją jest zaimplementowanie optymalizacji omówionych powyżej. Sprowadza się to do rozszerzenia implementacji dwóch procedur: `subsume-add-prove` i `resolve-single-prove`. W pierwszej z nich wystarczy sprawdzić czy pierwsza z nowych klauzul jest łatwiejsza lub trudniejsza od klauzul znajdujących się na listach `checked` i `pending`. Jeśli nowa klauzula jest łatwiejsza od którejś z już obecnych, możemy ją zwyczajnie wyrzucić (podobnie jak klauzulę trywialną). W przeciwnym wypadku wyrzucamy z list wszystkie klauzule łatwiejsze od nowej, a do listy klauzul do przetworzenia wstawiamy nowo wygenerowaną — i kontynuujemy przetwarzanie.

W przypadku klauzul jednoelementowych sytuacja jest znacząco prostsza: jeśli zaimplementowaliśmy poprzednią optymalizację, wiemy że w żadnej z list nie ma klauzul które byłyby łatwiejsze od tej którą aktualnie przetwarzamy, ale też że każda klauzula którą możemy zrezolwować z `s-clause` jest trudniejsza od klauzuli z której powstała — a więc możemy ją *zastąpić* jej rezolwentą, również

[2] Ta prosta implementacja może się zapętlić ze względu na wielokrotne dodawanie tych samych klauzul. Zamiast rozwiązywać ten problem tu, rozwiążemy go przy okazji optymalizacji w Ćwiczeniu 3.

w zbiorze klauzul już przetworzonych! Jeśli klauzuli nie da się zrezolwować z `s-clause`, pozostaje bez zmian. **Uwaga!** Należy pamiętać o zachowaniu porządku na klauzulach, a także o dodaniu `s-clause` do zbioru klauzul przetworzonych!