Wydział Podstawowych Problemów Techniki
Politechnika Wrocławska

Implementacja abstrakcyjnej maszyny Warrena

Kajetan Bilski
Nr indeksu: 244942

Praca inżynierska napisana
pod kierunkiem
dr. Przemysława Kobylańskiego

Politechnika
Wrocławska
Wrocław 2020

Spis treści

Wstęp

Celem pracy było zaimplementowanie abstrakcyjnej maszyny Warrena dla czystego języka Prolog. Odpowiada on językowi trzeciego poziomu z opisu maszyny [1]. Praca zawiera, poza opisem implementacji, analizę efektywności jej działania na wybranych przykładach programów. Efektem pracy jest również kompilator służący do kompilacji czystego języka Prolog na instrukcje maszyny Warrena.
W wielu językach programowania, np. C/C++, kompilator zamienia kod języka na kod maszynowy, który może być wykonany przez procesor. Prolog różni się pod tym względem, ponieważ jego skompilowany kod nie jest wykonywany przez procesor, a przez maszynę Warrena (WAM). Prolog jest językiem programowania logicznego i jego działanie opiera się na udowodnieniu zapytania zadanego przez programistę za pomocą faktów i reguł (wbudowanych lub również danych przez programistę) i zwraca wynik oznaczający prawdziwość zapytania (czy da się je udowodnić). Prolog próbuje udowodnić zapytanie na wszystkie dostępne sposoby, które umożliwiają dane predykaty. Implementacja abstrakcyjnej maszyny Warrena musi zawierać całą tą funkcjonalność. Zaproponowana implementacja dla większej łatwości użytkowania i lepszego porównania z innymi implementacjami zawiera wbudowany kompilator, który w pojedynczym uruchomieniu może być używany razem z maszyną, ale może być też uruchomiony osobno. Oprócz tego sama maszyna zawiera wbudowane predykaty i instrukcje, które nie są częścią czystego Prologa, ale dają lepszy wgląd w jej działanie.

W drugim rozdziale opisana jest maszyna Warrena, jej działanie, struktura jej pamięci i jej instrukcje.
W trzecim rozdziale opisany jest projekt implementacji, sposób działania aplikacji i gramatyka kompilatora.
W czwartym rozdziale opisane są szczegóły implementacji, użyte technologie i narzędzia.
W piątym rodziale przedstawiona jest analiza efektywności zaproponowanej implementacji, instrukcja obsługi i porównanie z innymi implementacjami.

WAM - Abstrakcyjna maszyna Warrena

Abstrakcyjna maszyna Warrena (WAM) to abstrakcyjna maszyna zaprojektowana w 1983 przez Dawida H. D. Warrena i jest standardem dla implementacji kompilatorów Prologa. WAM zawiera swoją architekurę pamięci i zestaw instrukcji.

2.1 Prolog

Żeby zrozumieć WAM trzeba najpierw zrozumieć język Prolog. Kompilator Prologa przetwarza programy i zapytania. Program składa się z predykatów stanowi bazę wiedzy za pomocą której WAM próbuje wykonać zapytanie. Dla porównania z językami imperatywnymi można rozumieć zapytanie jako funkcję `main`, a predykaty jako inne funkcje, do których może odwoływać się zapytanie i one same. Zapytania i predykaty składają się z ciągu termów.

Termem może być zmienna, formuła atomowa lub term złożony. Wartości zmiennych w Prologu są przypisywane w czasie wykonywania zapytania i podobnie jak w językach funkcyjnych, po przypisaniu nie mogą być zmienione (wyjątki pojawiają się w czasie nawrotów, kiedy wartość zmienniej może być od niej odpięta). Wartością zmiennej może być jedynie inny term (w przypadku przypisania do innej zmiennej, staje się jej referencją). Formuły atomowe to proste symbole zwane czasem też stałymi, które charakteryzują się jedynie swoją nazwą (np. `a`, `1`, `moj_atom`). Termy złożone różnią się od atomów tym, że zawierają w sobie ciąg innych termów zwanych podtermami. Podtermy też mogą być termami złożonymi, co powoduje że term można przedstawić jako drzewo, gdzie najbardziej zewnętrzny term jest korzeniem, dziećmi każdego termu są jego podtermy, a liśćmi są zmienne i atomy.

Wykonywanie zapytania polega na udowodnieniu po kolei każdego z jego termów. WAM zwraca wynik *true* lub $false$ w zależności od tego czy zapytanie udaje się udowodnić czy nie. Każdy term można udowodnić jedynie za pomocą predykatu. Są dwa typy przedykatów: fakty i reguły. W predykacie pierwszy term nazywa się "głową ", a pozostałe ćiałem ". Każdy predykat musi mieć głowę, a od tego czy ma ciało zależy czy jest faktem czy regułą (reguły mają ciało, a fakty nie). Żeby udowodnić term trzeba najpierw znaleźć predykat, dla którego nazwa i liczba podtermów głowy zgadzają się z rozpatrywanym termem. Następnie udowadniana jest tożsamość obu termów (unifikacja). Jeżeli unifikacja się powiedzie, to jeśli predykat jest faktem to term jest udowodniony i WAM przechodzi do udowadniania następnego termu, a jeśli jest regułą, to musi jeszcze udowodnić wszystkie termy z ciała w taki sam sposób jak termy zapytania. Dlatego zapytanie można traktować jak regułę bez głowy. W przypadku niepowodzenia w unifikacji z głową WAM wycofuje się i jeśli to możliwe próbuje udowodnić term korzystając z innego predykatu.

2.2 Architektura pamięci

Pamięć WAM jest pojedynczym blokiem składającym się z komórek. Każda komórka pamięci zawiera tag i wartość, która jest adresem komórki w pamięci, który może być reprezentowany przez liczbę naturalną. W pamięci wydzielone są strefy o zmiennej wielkości, a także przechowywane są globalne rejestry.

2.2.1 Strefy pamięci

Strefa kodu (`code area`) przechowuje listę instrukcji i ich argumenty.
Sterta (`HEAP`) przechowuje wszystkie struktury i zmienne utworzone w czasie wykonywania zapytania.
Stos (`STACK`) przechowuje środowiska i punkty wyboru. Środowiska przechowują rejestry trwałe dla obecnie rozpatrywanego predykatu, a także wartości rejestrów do przywrócenia przed swoją dealokacją. Punkty wyboru przechowują informacje potrzebne do odtworzenia stanu maszyny po nawrocie i są tworzone kiedy maszyna może próbować udowadniać term na wiele sposobów.
Ślad (`TRAIL`) pamięta adresy zmiennych, do których zostały przypisane warotści w kolejności ich przypisania. Używany do odpinania przypisanych zmiennych w przypadku nawrotu.
`PDL` jest innym stosem używanym tylko w czasie unifikacji.

2.2.2 Rejestry

`P` wskaźnik na obecnie wykonywaną instrukcję
`CP` wskaźnik na instrukcję do której WAM będzie musiał wrócić po wyjściu z obecnego predykatu
`H` wskaźnik na koniec sterty
`S` używany do pamiętania adresów podtermów rozpatrywanej struktury
`E` wskaźnik na obecnie aktywne środowisko
`B` wskaźnik na ostatni punkt wyboru
`TR` wskaźnik na koniec śladu
`HB` wskaźnik na koniec sterty w momencie utworzenia ostatniego punktu wyboru
`X1, X2,...` rejestry tymczasowe
`A1, A2,...` rejestry argumentów

Rejestry tymczasowe, argumentów i trwałe służą do pamiętania adresów termów na stercie. Rejestry tymczasowe pamiętają termy zawarte w pojedynczym rozpatrywanym termie. Rejestry argumentów służą do przekazywania argumentów w momencie wywołania predykatu. Argumentami są podtermy termu, który WAM będzie unifikować z głową predykatu. Rejestry trwałe przechowują adresy zmiennych pojawiających się w więcej niż jednym termie w danym zapytaniu lub regule.

2.3 Zestaw instrukcji

Żeby WAM mogło wykonywać kod Prologa, ten kod musi być najpierw skompilowany do listy instrukcji WAM. Instrukcje te są wykonywane po kolei aż WAM nie dojdzie do końca kodu i zakończy wykonywanie sukcesem lub dojdzie do niepowodzenia w wykonywaniu i nie ma punktu wyboru do którego może wrócić wykonywanie kończy się porażką. Lista instrukcji akceptowanych przez WAM[1]:

`put_structure f/n,Xi`
Dodaje do sterty komórki reprezentujące nową strukturę `f/n` i kopiuje do rejestru `Xi` komórkę pokazującą na tę strukturę.

`set_variable Xi`
Dodaje do sterty komórkę reprezentującą nową zmienną i umieszcza w rejestrze `Xi` wskaźnik na nią.

`set_value Xi`
Kopiuje komórkę z rejestru `Xi` na koniec sterty.

`get_structure f/n,Xi`
Próbuje pobrać strukturę `f/n` z rejestru `Xi`. Jeżeli trafi na nieprzypisaną zmienną to tworzy tę strukturę na stercie i ustawia maszynę w tryb zapisywania na stercie podtermów tej struktury. Jeżeli trafi na komórkę pokazującą na odpowiednią strukturę to ustawia maszynę w tryb unifikowania podtermów (do pamiętania

ich adresów używa rejestru `S`). Jeżeli trafi na inną strukturę to zwraca *false*.

`unify_variable Xi`
Jeżeli maszyna jest w trybie zapisywania podtermów to dodaje na koniec sterty nową nieprzypisaną komórkę i kopiuje ją do rejestru `Xi`.
Jeżeli maszyna jest w trybie unifikowania podtermów to kopiuje komórkę z pod adresu `S` do rejestru `Xi` (przestawiając potem `S` na następną komórkę).

`unify_value Xi`
Jeżeli maszyna jest w trybie zpisywania podtermów to kopiuje komórkę z rejestru `Xi` na koniec sterty.
Jeżeli maszyna jest w trybie unifikowania podtermów to unifikuje komórkę z adresu `S` i `Xi`.

`put_variable Xi,Aj`
Dodaje na koniec sterty nieprzypisaną komórkę, a następnie kopiuje ją do rejestrów `Xi` i `Aj`.

`put_value Xi,Aj`
Kopiuje komórkę z rejestru tymczasowego `Xi` do rejestru argumentu `Aj`.

`get_variable Xi,Aj`
Kopiuje komórkę z rejestru argumentu `Aj` do rejestru tymczasowego `Xi`.

`get_value Xi,Aj`
Unifikuje komórki w rejestrach `Xi` i `Aj`.

`call label`
Powoduje że WAM przechodzi do miejsca w kodzie oznaczonego etykietą `label` (ustawia odpowiednio rejestr `P` i rejestr powrotu `CP` oznaczający miejsce w kodzie z którego zostało wykonane ostatnie wywołanie). Jeśli nigdzie w kodzie nie ma etykiety `label` to zwraca *false*.

`proceed`
Wraca do ostatniego punktu wywołania (kopiuje `CP` do `P`).

`allocate N`
Tworzy nowe środowisko które przechowuje `N` rejestrów trwałych. Staje się ono aktywnym środowiskiem.

`deallocate`
Dealokuje obecnie aktywne środowisko i wraca do poprzedniego.

`try_me_else label`
Tworzy nowy punkt wyboru, którego punktem kontynuacji (miejscem w kodzie do którego przechodzi maszyna w przypadku powrotu do tego punktu wyboru) jest instrukcja z etykietą `label`.

`retry_me_else label`
Przywraca stan maszyny do stanu bezpośrednio po utworzeniu ostatniego punktu wyboru. Także zamienia punkt kontynuacji pamiętany przez aktywny punkt wyboru na miejce z etykietą `label`.

`trust_me`
Przywraca stan maszyny do stanu z przed utworzenia ostatniego punktu wyboru, usuwając go.

Opis projektu

Sposób działania opisanej implementacji jest oparty na sposobie działania SWI-Prologa z dodatkową możliwością używania kompilatora i maszyny WAM oddzielnie. Założeniem było zaimplementowanie podstawowej funkcjonalności SWI-Prologa, dając także lepszy wgląd w działanie WAM poprzez danie użytkownikowi możliwości uzyskania tekstowej reprezentacji kodu Prologa skompilowanego do listy instrukcji WAM, a następnie uruchomienia WAM dla tych instrukcji (lub napisanych ręcznie przez użytkownika). Do tego celu zawarte w aplikacji implementacja WAM i kompilator są dwiema łatwo rozróżnialnymi częściami uruchamianymi z głównej pętli aplikacji i możliwymi do uruchomienia osobno. W celu dalszej rozbudowy funkcjonalności i umożliwieniu analizy efektywności w projekcie implementacji znalazły się też dodatkowe instrukcje nie znajdujące się w zestawie instrukcji WAM dla czystego języka Prolog, wbudowane predykaty i wbudowana opcja pomiaru czasu i liczby inferencji.

3.1 Dodatkowa funkcjonalność

Zaproponowana implementacja zawiera wbudowane predykaty: `=/2`, `write/1`, `nl/0`, `./2` oraz `</2`. Ich kod zawsze jest dołączany na początek strefy kodu WAM przed kodem programu.
Predykat `=/2` unifikuje oba argumenty, tak jak w SWI-Prologu i jest zaimplementowany ze względu na wygodę użytkowania.
Predykat `write/1` wypisuje tekstową reprezentację termu z argumentu na standardowe wyjście. Działa rekurencyjnie dla termów złożonych i wypisuje też listy w opdowiedni sposób. Dla nieprzypisanych zmiennych wypisuje ich adres w pamięci.
Predykat `nl/0` wypisuje na standardowe wyjście nową linię.
Term `.(A,B)` reprzezentuje listę `B` z dołączonym pierwszym elementem `A`. Pustą listę oznacza term `[]`. Predykat `./2` sprawdza czy term jest poprawną reprezentacją listy, tzn. czy drugi argument jest poprawną reprezentacją listy (w tym może być pustą listą).
Predykat `</2` sprawdza czy oba argumenty są liczbami całkowitymi i czy pierwszy argument jest mniejjszy od drugiego.

Do implementacji części z tych predykatów wymagana była implementacja specjalnych instrukcji, które nie są nigdy generowane przez kompilator ale pojawiają się w kodzie wbudowanych predykatów, żeby umożliowić ich funkcjonalność nie dostępną dla czystego Prologa. Instrukcje `write`, `nl`, `less_than` uruchamiają funkcjonalność odpowiadających im predykatom dla odpowiednich rejestrów argumentów.

Przy uruchomieniu aplikacji dodatkowo jest dostępny tryb drukowania statystyk. W trakcie wykonywania każdego zapytania mierzony jest czas i liczba inferencji.

3.2 Kompilacja

Ważnym elementem implementacji jest kompilator. Aplikacja zawiera jeden kompilator kompilujący kod języka Prolog do tekstowej listy instrukcji WAM. Kompiluje on zarówno zapytania jak i programy. Metodą rozróżniania jest to, że zapytania w zaprzoponowanej implementacji zawsze muszą zaczynać się od `?-`.
Kompilator obsługuje też termy o składni nie zgodnej z czystym Prologiem:

`X = Y` jest kompilowane jak `=(X,Y)`,
podobnie `X < Y` i `X > Y`,
`[X]` oznacza listę i jest kompilowane jak `.(X,[])`, dozwolone są też listy o wielu elementach (`[X, Y, Z]`),
`[X | Y]` oznacza listę, której ogonem jest `Y`, także można użyć wielu elementów (`[X, Y | Z]`).

3.3 Gramatyka

Dwa mające w gramatyce tokeny to `VAR` i `STRUCT`. Oznaczają one odpowiednio nazwy zmiennych i struktur. Składają się z dowolnej kombinacji liter małych, wielkich, cyfr i podkreślników. Różnią się tym że nazwy zmiennych zaczynają się wielką literą, w przeciwieństwie do nazw struktur.

Algorithm 3.1: Gramatyka kompilatora

```
program
          predicates
          ?- terms .
predicates
          predicates predicate
          predicate
predicate
          term :- terms .
          term .
terms
          terms , term
          term
term
          STRUCT ( terms )
          STRUCT
          VAR
          term = term
          term < term
          term > term
          []
          [ terms ]
          [ terms | term ]
```

Opis implementacji

Do kompilacji zaproponowanej implementacji zostały użyte programy: g++ 7.5.0, flex 2.6.4, GNU Bison 3.0.4 i GNU Make 4.1. WAM został napisany w C++ w standardzie C++14 na system Linux (testowane na Ubuntu WSL).

4.1 Implementacja WAM

Implementacja została napisana tak, żeby zawierać pełną funkcjonalność abstrakcyjnej maszyny Warrena dla czystego Prologa i tą opisaną w poprzednim rozdziale. Mimo tego pojawiają się małe różnice w reprezentacji pamięci maszyny.

Pamięć maszyny zamiast być w jednym bloku w którym zawarte są wszystkie strefy[2], jest podzielona na bloki z których każdy odpowiada jednej strefie. Zaletą tego jest to, że aplikacji nie skończy się miejsce (zakładając że system operacyjny przydzieli jej wystarczająco miejsce), ponieważ używane struktury danych są dynamiczne i nie zaczną się na wzajem nadpisywać, jak to może się stać kiedy jest tylko jeden blok. Zmiany są też w komórkach pamięci. Kiedy istnieje więcej niż jeden blok pamięci adres nie może być już liczbą naturalną i musi oprócz tego zawierać wskaźnik na blok na który pokazuje. Do tego celu zaimplementowana jest klasa `Address`.
Na skutek tego klasa reprezentująca komórkę danych `DataCell` ma dwa pola: `std::string tag` i Address addr.
Używane adresy prawie zawsze pokazują na stertę, ale umożliwianie im pokazywania na inne bloki pozwala uniknąć skomplikowanych błędów. Niektóre rejestry wciąż mogą pokazywać na tylko jeden blok i nie zawierają taga, więc zamiast komórki danych mogą reprzezentowane przez liczbę naturalną (np. `H`, `S`, `P`). Jako że wiele elementów pamięci WAM używa tylko jednego pola z komórki (`tag` lub `addr`) dla znaczącej części z nich można zmienić zupełnie ich typ tak, żeby lepiej odpowiadał ich funkcji. Bloki pamięci reprezentowane są przez różne struktury danych:
Strefa kodu to `std::vector<std::vector<std::string>>`, gdzie każdy element zewnętrzego wektora zawiera w kolejności nazwę instrukcji i jej argumenty, wszystkie zapisane jako ciągi znaków. W trakcie ładowania kodu wszystkie etykiety są z niego usuwane i są zamiast tego pamiętane mapie gdzie kluczem jest nazwa etykiety, a wartością numer linii, której odpowiada, przyspiesza to wywołania predykatów.
Dla sterty została zaimplementowana klasa `MemoryBloc` która działa jak wektor komórek pamięci, z tą różnicą, że jego operatory dostępu są zmienione i w przypadku próby dostępu do elementu z poza granic wektora, automatycznie się on rozszerza, na nowych miejscach dodając nieprzypisane komórki (komórki z tagiem ŻEF "i swoim własnym adresem). Symuluje to zachowanie pamięci w oryginalnym projekcie Warrena, gdzie cała pamięć od początku wypełniona jest nieprzypisanymi komórkami.
Główny stos (`STACK`) jest bardziej zmodyfikowany. W przypadku użycia pojedynczego stosu, przechowuje on zarówno środowiska jak i punkty wyboru[1]. Zaletą tego rozwiązania jest to, że każdy punkt wyboru chroni środowiska pod nim na stosie przed nadpisaniem, ponieważ w przypadku nawrotu może isnieć konieczność powrotu do zdealokowanego środowiska. W zaproponawanej implementacji zamiast stosu `STACK` istnieją dwa stosy: `AND_STACK` przechowujący środowiska i `OR_STACK` przechowujący punkty wyboru. W ten sposób każdy z tych stosów jest zaimplementowany jako wektor jednego typu (środowiska i punkty wyboru są zaimplementowane jako swoje własne klasy z różnymi polami i bez metod). W tym przypadku zmienia się funkcja rejestru `B` (typu `int`), który dotąd pokazywał na ostatni punkt wyboru, ale w proponowanej implementacji

ostatni punkt wyboru jest zawsze ostatnim elementem `OR_STACK` i nie wymaga pamiętania jego lokalizacji. Rejestr `B` przejmuje zadanie chronienia zdealokowanych środowisk, pokazując wielkość stosu `AND_STACK` w momencie utworzenia ostatniego punktu wyboru i zabezpieczając przed nadpisaniem środowiska starszego od tego punktu wyboru.
Ślad jest typu `std::vector<Address>` żeby pamiętać adresy przypisywanych zmiennych.
Stos `PDL` jest typu `std::stack<Address>` i jest zmienną lokalną wewnątrz funkcji `unify`, ponieważ jest wykorzystywany tylko tam i powinnien być zerowany po każdym wywołaniu unifikacji.
Rejestry tymczasowe, argumentów i trwałe są zaimplementowane za pomocą tego samego typu co sterta, czyli `MemoryBloc`, ponieważ tych rejestrów może być potrzebna niewiadoma ilość i nie można ich reprezentować za pomocą statycznej struktury danych.

W podstawowym przypadku użycia (kompilator + WAM) aplikacja najpierw ładuje do strefy kodu wbudowane predykaty, następnie kompiluje i ładuje program (jeżeli podany). W tym momencie obecna długość kodu zostaje zapamiętana. Następnie aplikacja wyświetla użytkownikowi polecenie wpisania zapytania. Kiedy użytkownik je wpisze, zapytanie jest kompilowane i dodawane do strefy kodu. Wykonywanie zaczyna się od początku instrukcji zapytania i kończy gdy któraś instrukcja (lub `unify`) zwróci błąd albo maszyna dojdzie do końca kodu. W pierwszym wypadku ustawiana jest flaga `fail` i zaczyna się nawrót, czyli próba przywrócenia maszyny do stanu z ostatniego punktu wyboru. Jeśli taki punkt wyboru istnieje to wykonywanie wznawia się w miejscu w kodzie wskazanym przez dany punkt. W przeciwnym wypadku wykonywanie kończy się porażką i aplikacja wyświetla wartość wyjściową $fasz$. Jeżeli maszyna dojdzie do końca kodu, to wykonanie kończy się sukcesem i aplikacja pyta użytkownika czy kontynuować wykonywanie. Jeżeli użytkownik odpowie ”tak”to maszyna zachowuje się tak, jakby właśnie trafiła na błąd i zaczyna nawrót. W przeciwnym wypadku kończy wykonywanie.
Po zakończeniu wykonywania aplikacja usuwa ze strefy kodu kod zapytania zostawiając tylko kod wbudowanych predykatów i programu, a następnie prosi o wpisanie następnego zapytania. Aplikacja działa w pętli aż nie zostanie manualnie zatrzymana (ctrl+c).

Przykładowy fragment kodu odpowiedzialny za funkcję `unify` służącą do udowadniania tożsamości dwóch poddrzew unifikowanych termów:

```
bool unify(Address a1, Address a2){
    std::stack<Address> pdl;
    pdl.push(a1);
    pdl.push(a2);
    while(!pdl.empty()){
        Address d1 = deref(pdl.top());
        pdl.pop();
        Address d2 = deref(pdl.top());
        pdl.pop();
        if(d1 != d2){
            if(d1.getCell().tag == "REF" || d2.getCell().tag == "REF"){
                bind(d1,d2);
            } else{
                functor f1 = get_functor(d1.getCell().getAddr().getCell().tag);
                functor f2 = get_functor(d2.getCell().getAddr().getCell().tag);
                if(f1.first == f2.first && f1.second == f2.second){
                    for(unsigned int i = 1; i <= f1.second; i++){
                        pdl.push(d1.getCell().getAddr()+i);
                        pdl.push(d2.getCell().getAddr()+i);
                    }
                } else{
                    return true;
                }
            }
        }
    }
    return false;
}
```

4.2 Implementacja kompilatora

Kompilator składa się z leksera napisanego przy pomocy Flexa i parsera napisanego przy pomocy GNU Bisona. Gramatyka kompilatora jest opisana w poprzednim rozdziale. Kompilator pobiera kod Prologa jako `std::string` i zwraca listę instrukcji jako `std::string`. Lekser przetwarza cały wejściowy ciąg znaków na tokeny, usuwając przy tym białe znaki i komentarze (wszystko od znaku % do końca linii). Tokeny reprezentują statyczne znaki lub krótkie ciągi znaków używane w kodzie Prologa nie licząc tokenów `VAR` i `STRUCT`, które odpowiednio oznaczają nazwy zmiennych i struktur.
Parser składając termy predykatów lub zapytania konstruuje termy w postaci drzewa i zapamiętuje je aż do momentu złożenia ich w predykat lub zapytanie, kiedy generuje instrukcje. Generowanie instrukcji jest możliwe dopiero w tym momencie, ponieważ poznane zostają wszystkie zmienne i jest możliwe wyodrębnienie zmiennych, które muszą być przechowywane w rejestrach trwałych. W przypadku programu w momencie złożenia predykatów są one grupowane i generowane są instrukcje zarządzające punktami wyboru, tam gdzie jest więcej niż jeden predykat o takiej samej nazwie. Rejestry są indeksowane od 0. Instrukcje generowane dla danego termu różnią się znacząco w zależności gdzie jest ten term. Jeśli term jest w zapytaniu lub w ciele reguły to generowane są dla niego instrukcje konstruujące reprezentacje termu w maszynie i umieszczające ją w argumentach (np. `put_structure`, `set_variable` i `set_value`). Instrukcje dla struktur w takim termie są generowane od liści do korzenia. Każdy term jest konstruowany ze skonstruowanych już podtermów. Za to instrukcje generowane dla termów z faktów lub głowy reguły służą do zczytywania istniejących termów i sprawdzania ich zgodności z obecnie rozpatrywanym termem (np. `get_structure`, `unify_variable` i `unify_value`). W tym wypadku instrukcje dla struktur są generowane w kolejności od korzenia do liści.

Argumenty tymczasowe są przydzielane do termów w danym drzewie w kolejności zgodnej z przeszukaniem w szerz. Rejestry argumentów są przydzielane zgodnie z kolejnością argumentów, a rejestry trwałe z kojenością drugich wystąpień zmiennych w predykacie lub zapytaniu.

Przykładowy fragment kodu odpowiedzialny za składanie predykatów w program:

```
{
std::unordered_map<std::string, std::vector<unsigned int>> clauses;
for(unsigned int i=0; i<$1.ts.size(); i++){
       std::string name = $1.ts[i]->name + "/" + std::to_string($1.ts[i]->no_subterms);
       if(clauses.find(name) == clauses.end()){
             clauses[name] = std::vector<unsigned int>();
       }
       clauses[name].push_back(i);
}
int labels = 0;
std::string code;
for(std::pair<std::string, std::vector<unsigned int>> clause : clauses){
       code += clause.first + " :\n";
       if(clause.second.size() == 1){
              code += $1.codes[clause.second[0]];
       } else{
              for(unsigned int i=0; i<clause.second.size(); i++){
                     if(i == 0){
                            code += "try_me_else L" + std::to_string(labels) + "\n";
                     } else{
                            code += "L" + std::to_string(labels) + " :\n";
                            labels++;
                            if(i == clause.second.size()-1){
                                   code += "trust_me\n";
                            } else{
                                   code += "retry_me_else L" + std::to_string(labels) + "\n";
                            }
                     }
                     code += $1.codes[clause.second[i]];
              }
       }
}
ostring = code;
YYACCEPT;
}
```

Analiza efektywności

Program działa w terminalu na linuksie (był testowany na Ubuntu w WSL). Do jego kompilacji zostały użyte: g++ 7.5.0, flex 2.6.4, GNU Bison 3.0.4 i GNU Make 4.1. WAM został napisany w C++ w standardzie C++14.

5.1 Instrukcja obsługi

Sposoby użycia:

```
wam [--stats] [<program>]
wam -c <input> [<output>]
wam [--stats] -e <program> <query>
```

W pierwszym przypadku użycia `program` jest ścieżką do pliku z programem napisanym w języku Prolog. Program jest ładowany do pamięci, następnie WAM wyświtla "?-"i czeka na użytkownika, żeby wpisał zapytanie. Zapytanie musi być w jednej linii i kończyć się kropką. Po wykonaniu zapytania WAM resetuje swoją pamięć i oczekuje następnego zapytania od użytkownika i tak dalej w pętli aż użytkownik nie wyjdzię manualnie (ctrl+C).
W drugim przypadku użycia pobiera z pliku tektowego `input` program lub zapytanie i kompiluje go do listy instrukcji maszyny Warrena, które umieszcze w pliku tekstowym `output` jeśli został podany lub w `a.out` w przeciwnym wypadku. Kompilator rozróżnia program i zapytanie Prologa po tym, że zapytanie musi rozpoczynać się od "?-".
W trzecim przypadku użycia `program` i `query` są plikami tekstowymi zawierającymi instrukcje maszyny Warrena, takimi jakie można wygenerować używając opcji `-c`. WAM ładuje wykonuje te instrukcje pomijając kompilator.
Opcjonalny argument `--stats` włącza pomiar czasu i inferencji (wykonań instrukcji `call`). Pomiary wykonywane są osobno dla każdego zapytania w pierwszym przypadku użycia. Czas spędzony przez aplikację na oczekiwaniu na akcję użytkownika po zapytaniu o kontynuowanie ewaluacji nie jest wliczany do pomiaru. Wyniki każdego pomiaru są wyświetlane na standardowym wyjściu po zakończeniu wykonywania zapytania.

5.2 Przykłady programów

Do testowania szybkości działania zaproponowanej implementacji i porównania jej z SWI-Prologiem zostały użyte 4 różne programy napisane w Prologu.

Pierwszy program sprawdza jak szybko implementacja jest w stanie znaleźć wszystkie możliwe rozcięcia listy o 50, 100, 200, 400 i 800 elementach.

```
con50 :-
    list50(L),
    concat(X,Y,L),
```

```
        write(X),nl,
        write(Y),nl,nl,
        fail.

con100 :-
    list100(L),
    concat(X,Y,L),
        write(X),nl,
        write(Y),nl,nl,
        fail.

con200 :-
    list200(L),
    concat(X,Y,L),
        write(X),nl,
        write(Y),nl,nl,
        fail.

con400 :-
    list400(L),
    concat(X,Y,L),
        write(X),nl,
        write(Y),nl,nl,
        fail.

con800 :-
    list800(L),
    concat(X,Y,L),
        write(X),nl,
        write(Y),nl,nl,
        fail.

concat([],L,L).
concat([X|L1],L2,[X|L3]) :- concat(L1,L2,L3).
```

Drugi program sprawdza odwracanie listy w naiwny sposób.

```
nrev50 :-
        list50(L),
        nreverse(L,X),
        write(X), nl.

nrev100 :-
        list100(L),
        nreverse(L,X),
        write(X), nl.

nrev200 :-
        list200(L),
        nreverse(L,X),
        write(X), nl.

nrev400 :-
        list400(L),
```

```
        nreverse(L,X),
        write(X), nl.

nrev800 :-
        list800(L),
        nreverse(L,X),
        write(X), nl.

nreverse([X|L0],L) :- nreverse(L0,L1), concatenate(L1,[X],L).
nreverse([],[]).

concatenate([X|L1],L2,[X|L3]) :- concatenate(L1,L2,L3).
concatenate([],L,L).
```

Trzeci program sprawdza implementacje na algorytmie quciksort.

```
qs50 :-
        list50(L),
        qsort(L,X,[]),
        write(X), nl.

qs100 :-
        list100(L),
        qsort(L,X,[]),
        write(X), nl.

qs200 :-
        list200(L),
        qsort(L,X,[]),
        write(X), nl.

qs400 :-
        list400(L),
        qsort(L,X,[]),
        write(X), nl.

qs800 :-
        list800(L),
        qsort(L,X,[]),
        write(X), nl.

qsort([X|L],R,R0) :-
        partition(L,X,L1,L2),
        qsort(L2,R1,R0),
        qsort(L1,R,[X|R1]).
qsort([],R,R).

partition([X|L],Y,[X|L1],L2) :-
        X < Y,
        partition(L,Y,L1,L2).
partition([X|L],Y,L1,[X|L2]) :-
    X > Y,
        partition(L,Y,L1,L2).
partition([X|L],Y,L1,[X|L2]) :-
```

```
    X = Y,
        partition(L,Y,L1,L2).
partition([],X,[],[]).
```

Czwarty program sprawdza znajdywanie wszystkich podzbiorów na listach o 10, 11, 12, 13 i 14 elementach.

```
sub10 :-
    list10(L),
    subset(L,R),
    write(R),
    nl,
    fail.

sub11 :-
    list11(L),
    subset(L,R),
    write(R),
    nl,
    fail.

sub12 :-
    list12(L),
    subset(L,R),
    write(R),
    nl,
    fail.

sub13 :-
    list13(L),
    subset(L,R),
    write(R),
    nl,
    fail.

sub14 :-
    list14(L),
    subset(L,R),
    write(R),
    nl,
    fail.

sub15 :-
    list15(L),
    subset(L,R),
    write(R),
    nl,
    fail.

subset([X|L1],L2) :- subset(L1,L2).
subset([X|L1],[X|L2]) :- subset(L1,L2).
subset([],[]).

list10([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]).
list11([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]).
```

```
list12([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]).
list13([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]).
list14([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]).
list15([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]).
```

Same listy z programów 1, 2 i 3 zostały wycięte ze względu na ich długość.

5.3 Skompilowane przykłady

Poniżej przedstawione są fragmenty skompilowanych programów przykładowych.

Predykat `concat/3`:

```
concat/3 :
try_me_else L0
get_structure []/0,A0
get_variable X0,A1
get_value X0,A2
proceed
L0 :
trust_me
allocate 3
get_structure ./2,A0
unify_variable X0
unify_variable Y0
get_variable Y1,A1
get_structure ./2,A2
unify_value X0
unify_variable Y2
put_value Y0,A0
put_value Y1,A1
put_value Y2,A2
call concat/3
deallocate
```

Predykat `nreverse/2`:

```
nreverse/2 :
try_me_else L0
allocate 4
get_structure ./2,A0
unify_variable Y3
unify_variable Y0
get_variable Y2,A1
put_value Y0,A0
put_variable Y1,A1
call nreverse/2
put_value Y1,A0
put_structure []/0,X0
put_structure ./2,A1
set_value Y3
set_value X0
put_value Y2,A2
```

```
call concatenate/3
deallocate
L0 :
trust_me
get_structure []/0,A0
get_structure []/0,A1
proceed
concatenate/3 :
try_me_else L1
allocate 3
get_structure ./2,A0
unify_variable X0
unify_variable Y0
get_variable Y1,A1
get_structure ./2,A2
unify_value X0
unify_variable Y2
put_value Y0,A0
put_value Y1,A1
put_value Y2,A2
call concatenate/3
deallocate
L1 :
trust_me
get_structure []/0,A0
get_variable X0,A1
get_value X0,A2
proceed
nrev800/0 :
allocate 2
put_variable Y0,A0
call list800/1
put_value Y0,A0
put_variable Y1,A1
call nreverse/2
put_value Y1,A0
call write/1
call nl/0
deallocate
```

Predykaty `qsort/3` i `partition/4`:

```
partition/4 :
try_me_else L1
allocate 5
get_structure ./2,A0
unify_variable Y0
unify_variable Y2
get_variable Y1,A1
get_structure ./2,A2
unify_value Y0
unify_variable Y3
get_variable Y4,A3
put_value Y0,A0
put_value Y1,A1
```

```
call </2
put_value Y2,A0
put_value Y1,A1
put_value Y3,A2
put_value Y4,A3
call partition/4
deallocate
L1 :
retry_me_else L2
allocate 5
get_structure ./2,A0
unify_variable Y1
unify_variable Y2
get_variable Y0,A1
get_variable Y3,A2
get_structure ./2,A3
unify_value Y1
unify_variable Y4
put_value Y0,A0
put_value Y1,A1
call </2
put_value Y2,A0
put_value Y0,A1
put_value Y3,A2
put_value Y4,A3
call partition/4
deallocate
L2 :
retry_me_else L3
allocate 5
get_structure ./2,A0
unify_variable Y0
unify_variable Y2
get_variable Y1,A1
get_variable Y3,A2
get_structure ./2,A3
unify_value Y0
unify_variable Y4
put_value Y0,A0
put_value Y1,A1
call =/2
put_value Y2,A0
put_value Y1,A1
put_value Y3,A2
put_value Y4,A3
call partition/4
deallocate
L3 :
trust_me
get_structure []/0,A0
get_variable X0,A1
get_structure []/0,A2
get_structure []/0,A3
proceed
```

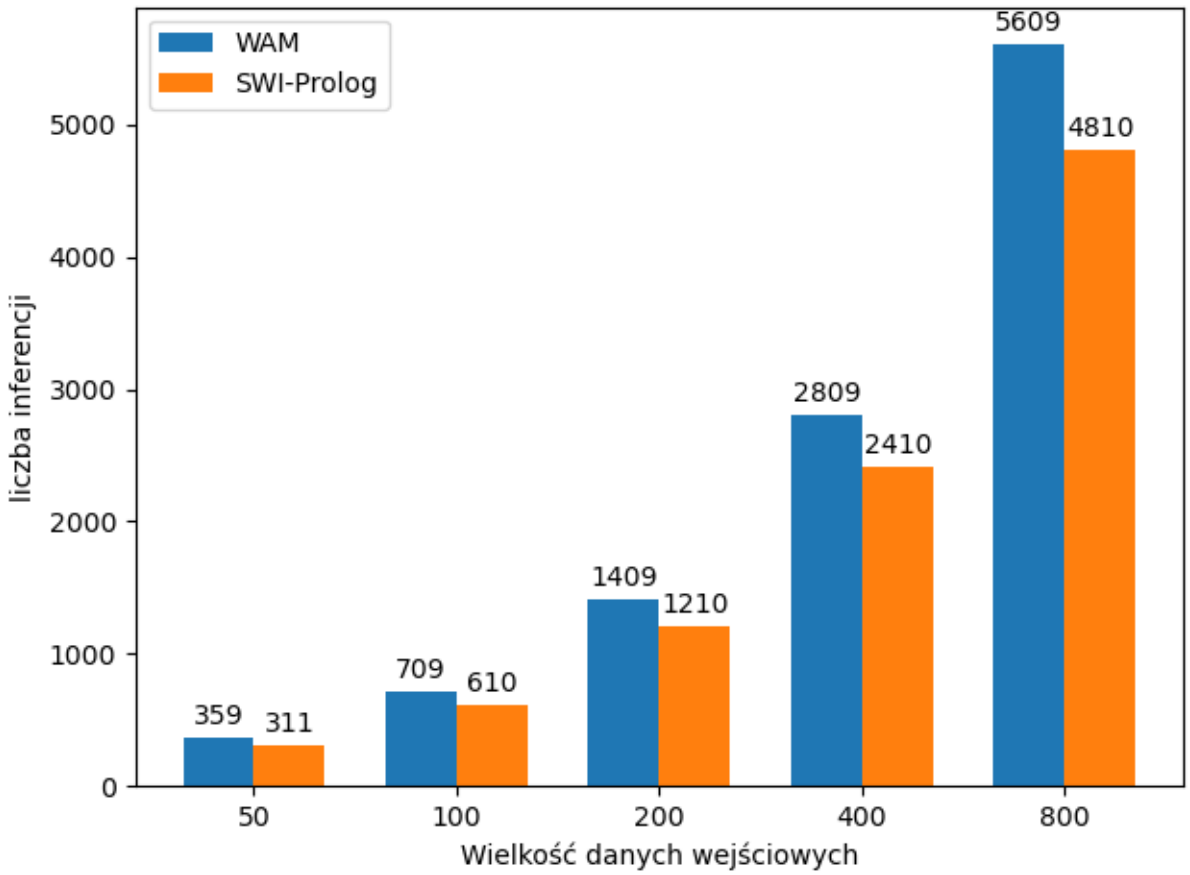
```
qsort/3 :
try_me_else L4
allocate 7
get_structure ./2,A0
unify_variable Y1
unify_variable Y0
get_variable Y5,A1
get_variable Y3,A2
put_value Y0,A0
put_value Y1,A1
put_variable Y4,A2
put_variable Y2,A3
call partition/4
put_value Y2,A0
put_variable Y6,A1
put_value Y3,A2
call qsort/3
put_value Y4,A0
put_value Y5,A1
put_structure ./2,A2
set_value Y1
set_value Y6
call qsort/3
deallocate
L4 :
trust_me
get_structure []/0,A0
get_variable X0,A1
get_value X0,A2
proceed
/end{lstlisting}

Predykat \texttt{subset/2}:\\
\begin{lstlisting}
subset/2 :
try_me_else L0
allocate 2
get_structure ./2,A0
unify_variable X0
unify_variable Y0
get_variable Y1,A1
put_value Y0,A0
put_value Y1,A1
call subset/2
deallocate
L0 :
retry_me_else L1
allocate 2
get_structure ./2,A0
unify_variable X0
unify_variable Y0
get_structure ./2,A1
unify_value X0
unify_variable Y1
```

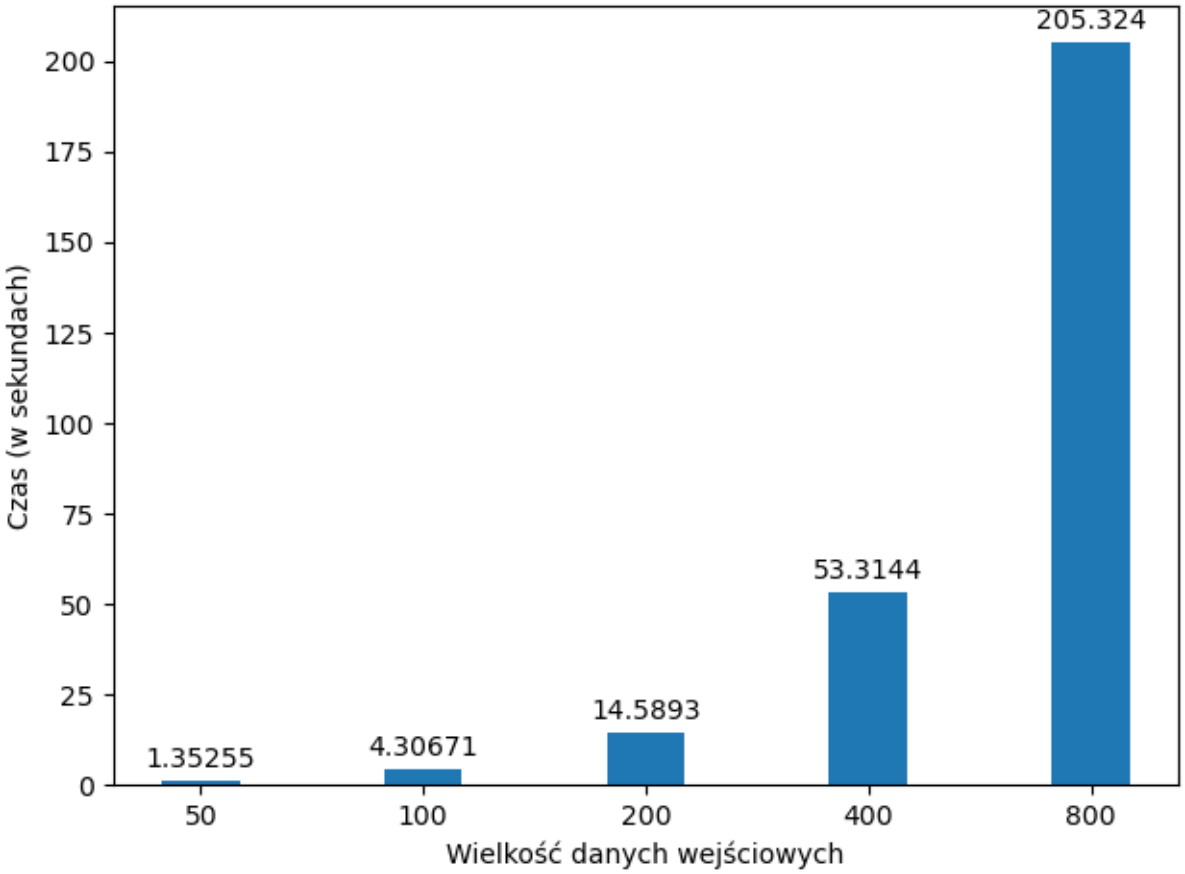
```
put_value Y0,A0
put_value Y1,A1
call subset/2
deallocate
L1 :
trust_me
get_structure []/0,A0
get_structure []/0,A1
proceed
```

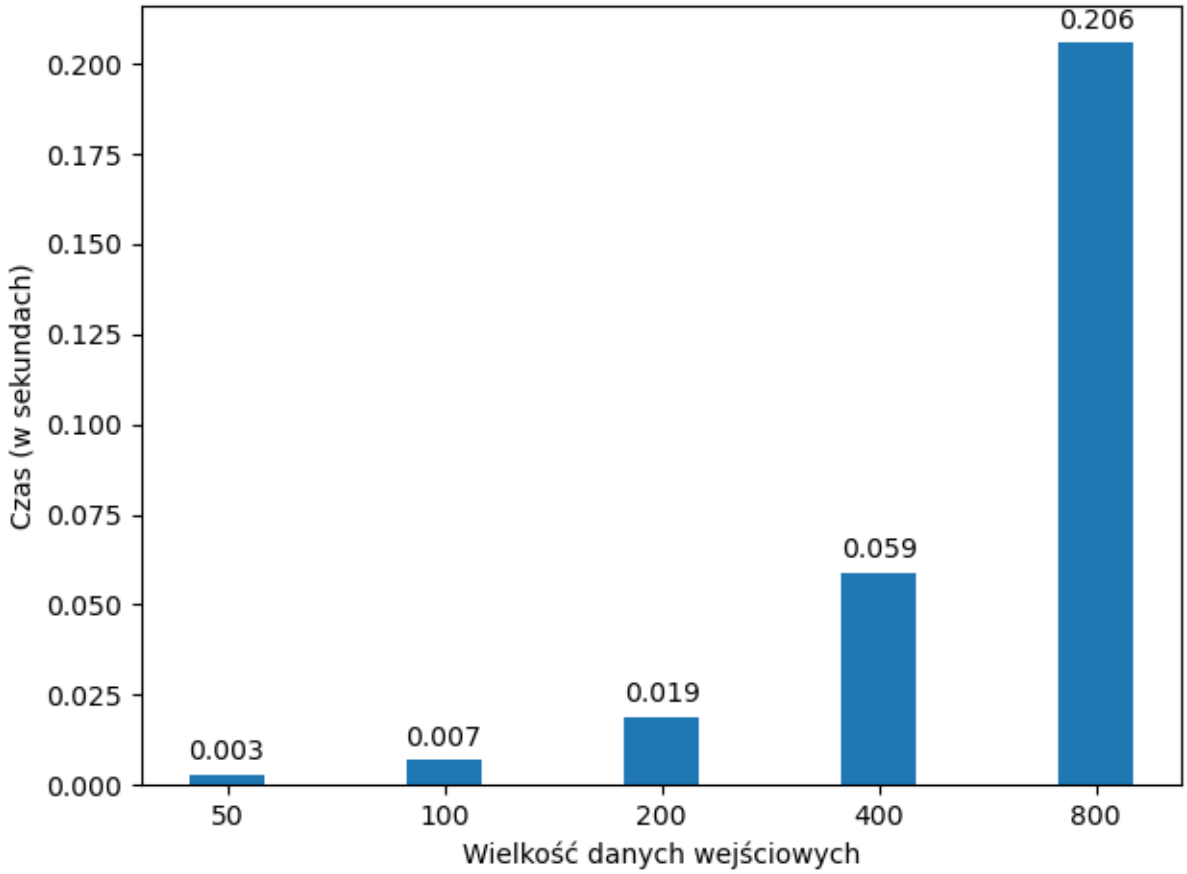
5.4 Porównanie z SWI-Prologiem

Zaproponowana implementacja została poddana testom na wyżej wymienionych programach razem z SWI-Prologiem. Poniżej są wyniki eksperymentów: ilość inferencji i czas wykonywania. Na wykresach inferencji proponowana implementacja jest oznaczona jako WAM. Czas mierzony jest dla WAM z dokładnością do 6 cyfr znaczących, a dla SWI-Prologa z dokładnością do milisekund

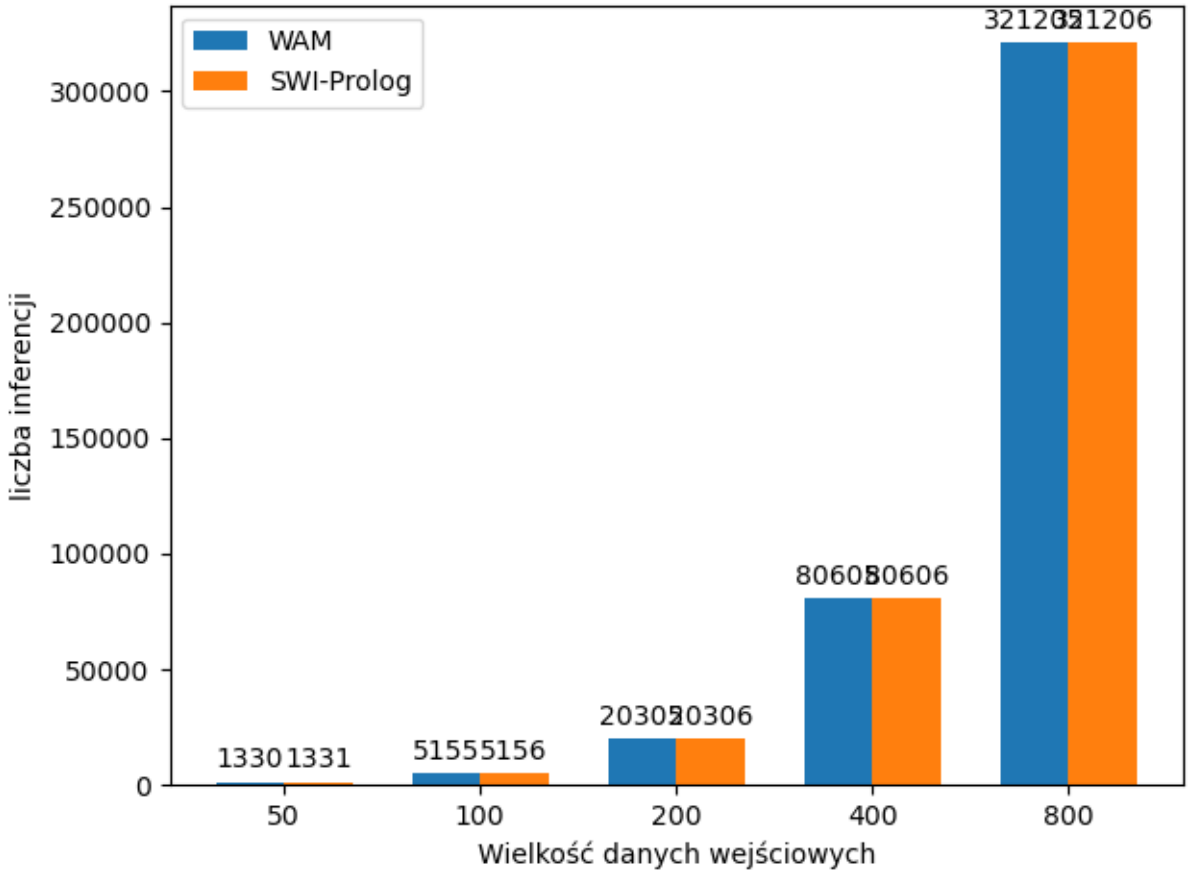


W przypadku rozcięć liczba inferencji dla proponowanej implementacji i dla SWI-Prologa rośnie liniowo z rozmiarem danych. Dla małych danych SWI-Prolog ma przewagę pod względem liczby inferencji. Przewaga ta też rośnie liniowo.

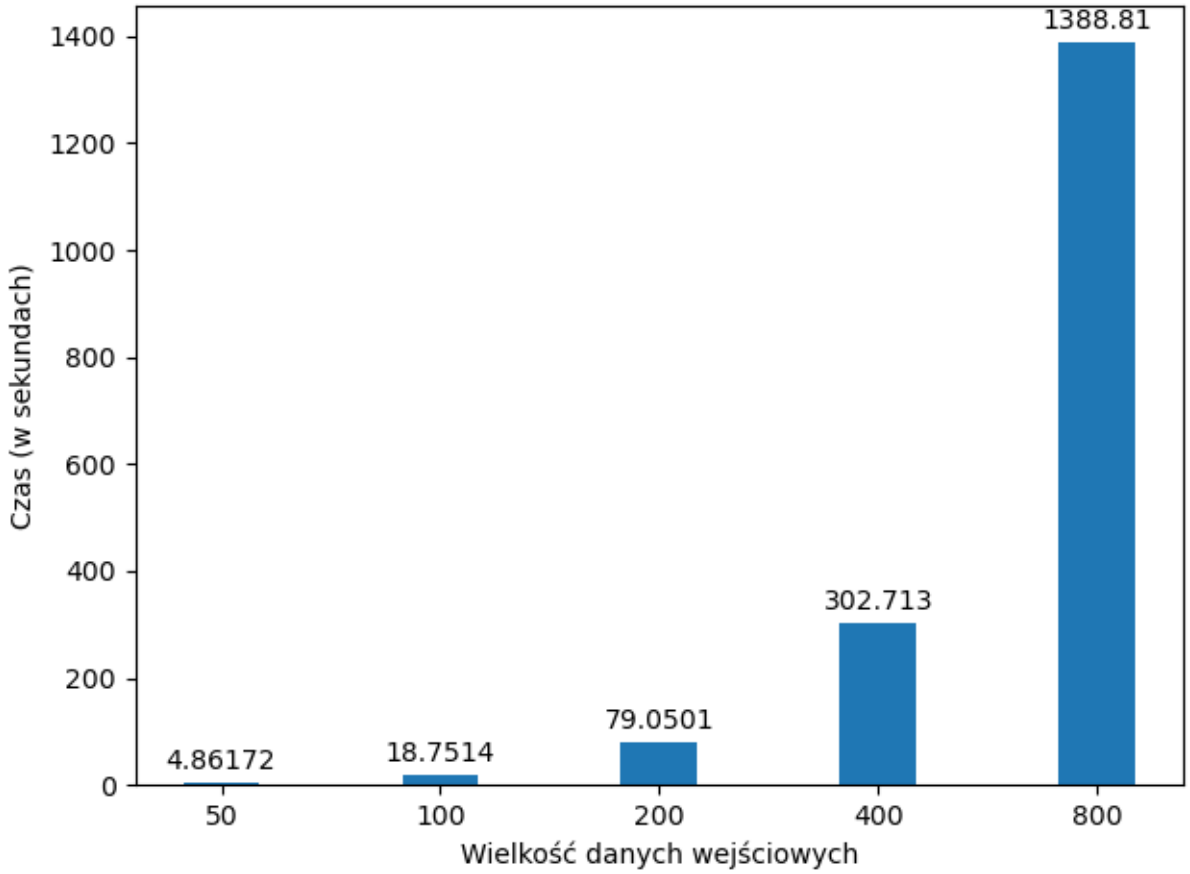

205.324
200
175
150
125
100
75
53.3144
50
25
14.5893
1.35255
4.30671
0
50
100
200
400
800
Czas (w sekundach)
Wielkość danych wejściowych

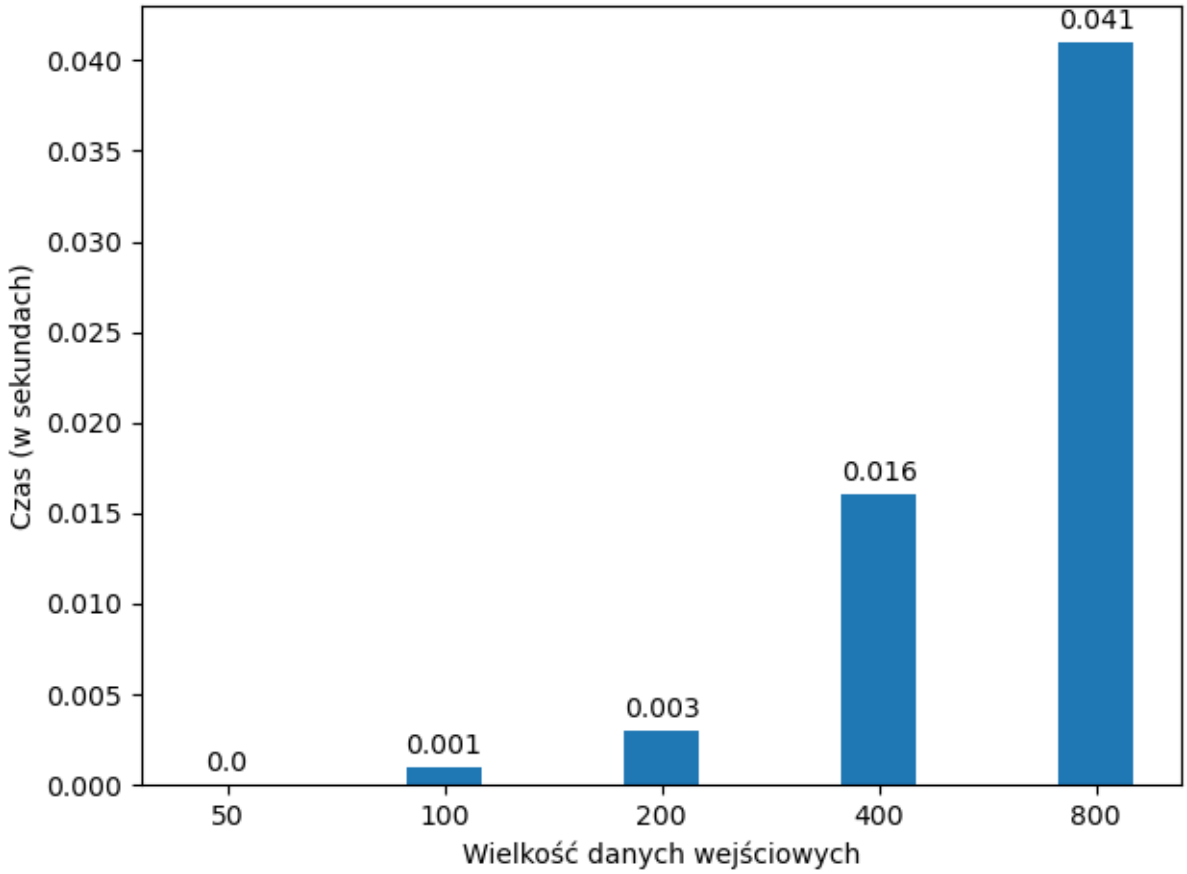


Dla pomiarów czasu widać wzrost bliższy kwadratowemu dla obu implementacji, przy czym SWI-Prolog jest około 1000 razy szybszy.

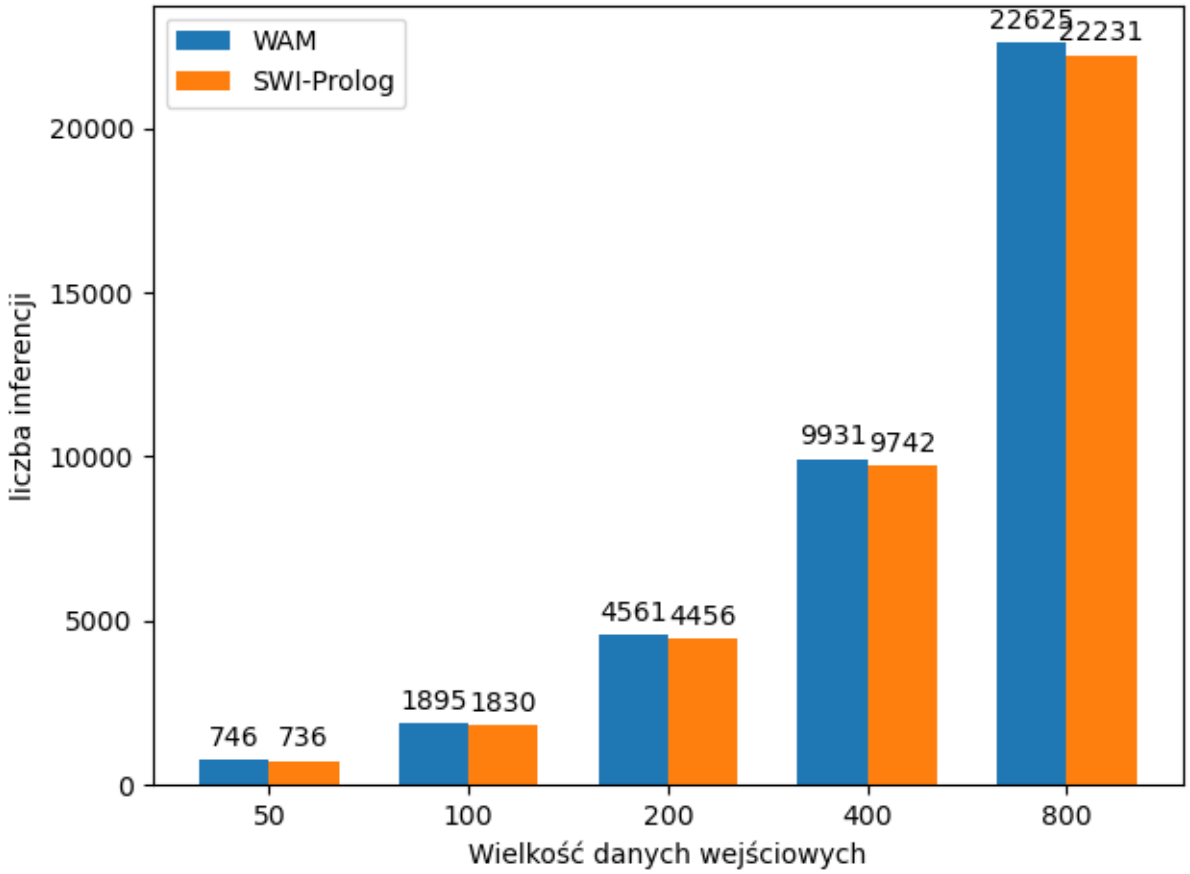


W przypadku naiwenego odwrócenia listy dla dowolnych danych zaproponowana implementacja potrzebuje dokładnie jedną inferencję mniej.

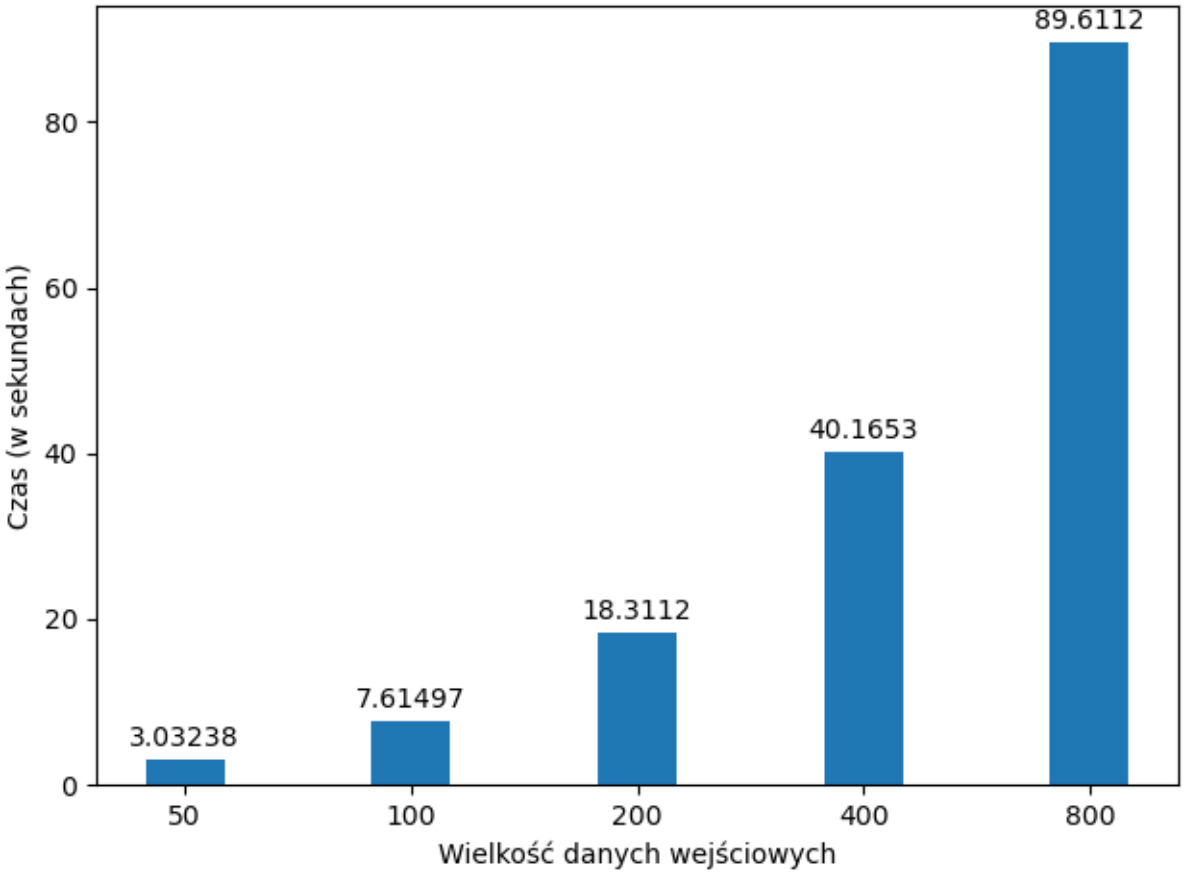

1388.81
1400
1200
1000
800
600
400
200
0
Czas (w sekundach)
302.713
79.0501
18.7514
4.86172
50
100
200
400
800
Wielkość danych wejściowych

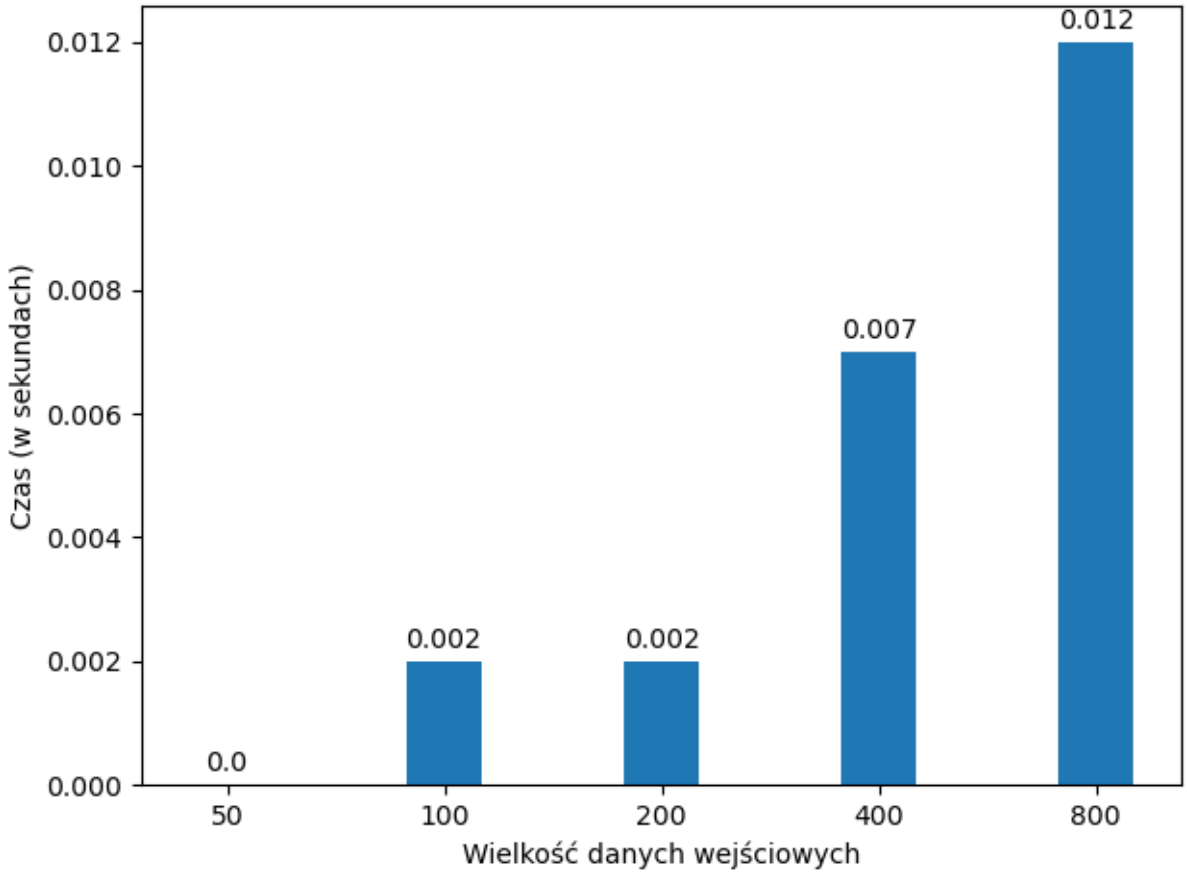


W kategorii czasu dla proponowanej implementacji widać kwadratowy wzrost, to dla SWI-Prologa tempo wzrostu jest trochę mniejsze. SWI-Prolog jest do 33000 razy szybszy.

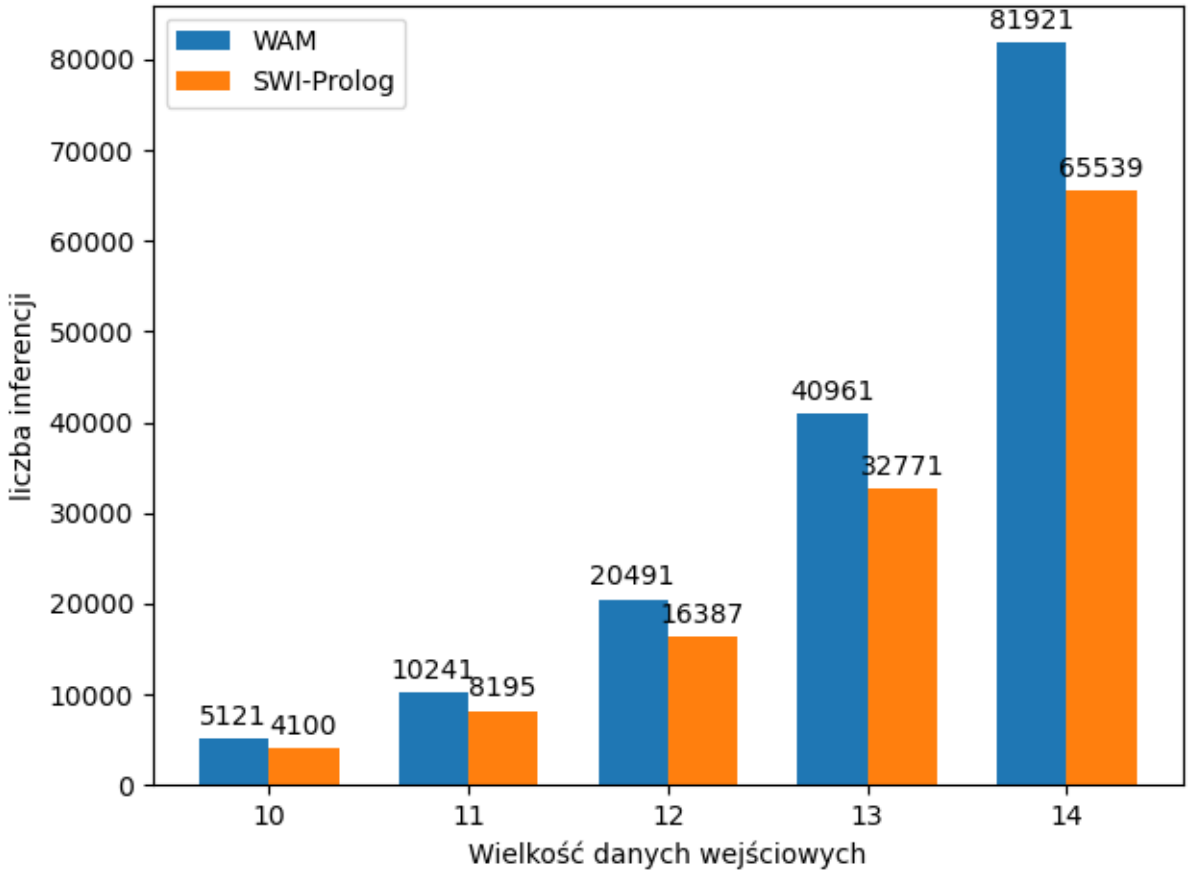


Dla quicksorta różnica w liczbie inferencji tak samo jak dla rozcięć jest na początku niewielka i rośnie ze wzrostem wielkości danych.

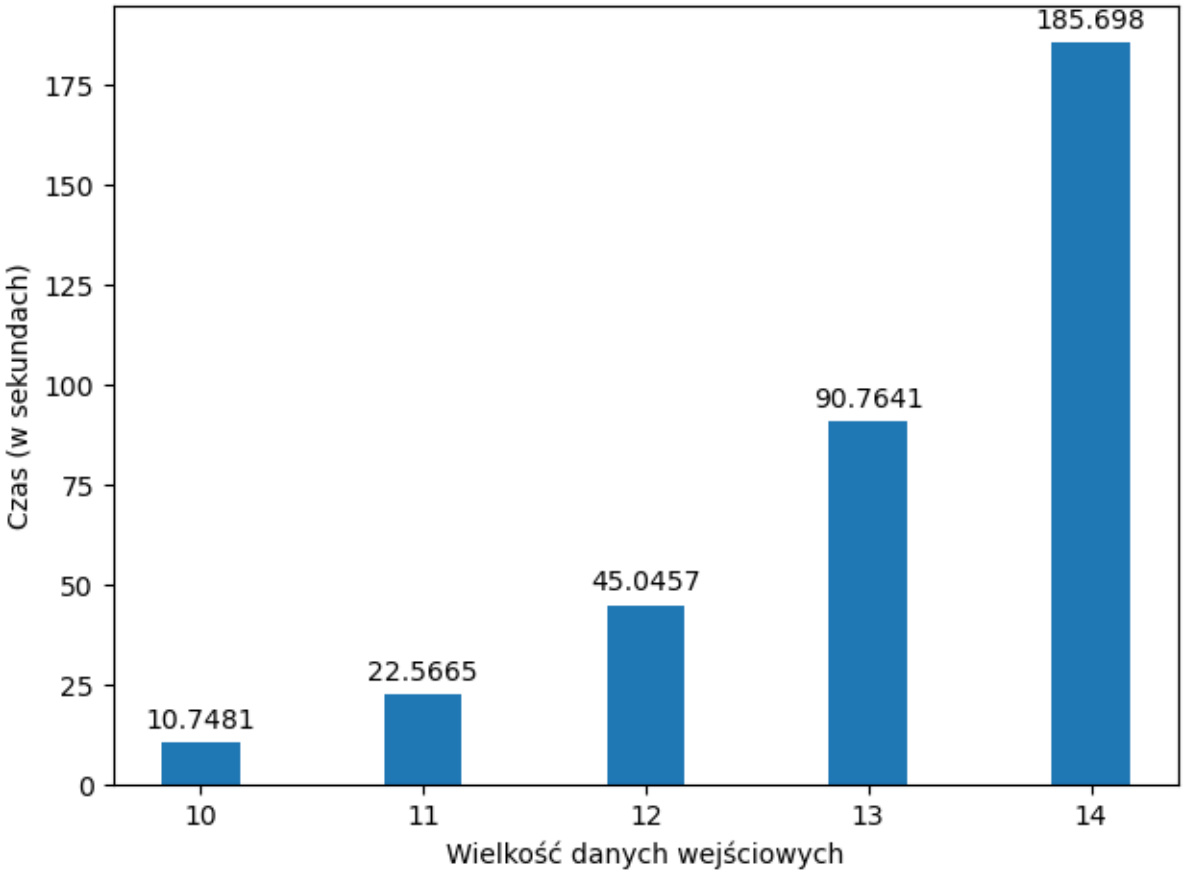

89.6112
80
60
40.1653
40
Czas (w sekundach)
18.3112
20
7.61497
3.03238
0
50
100
200
400
800
Wielkość danych wejściowych

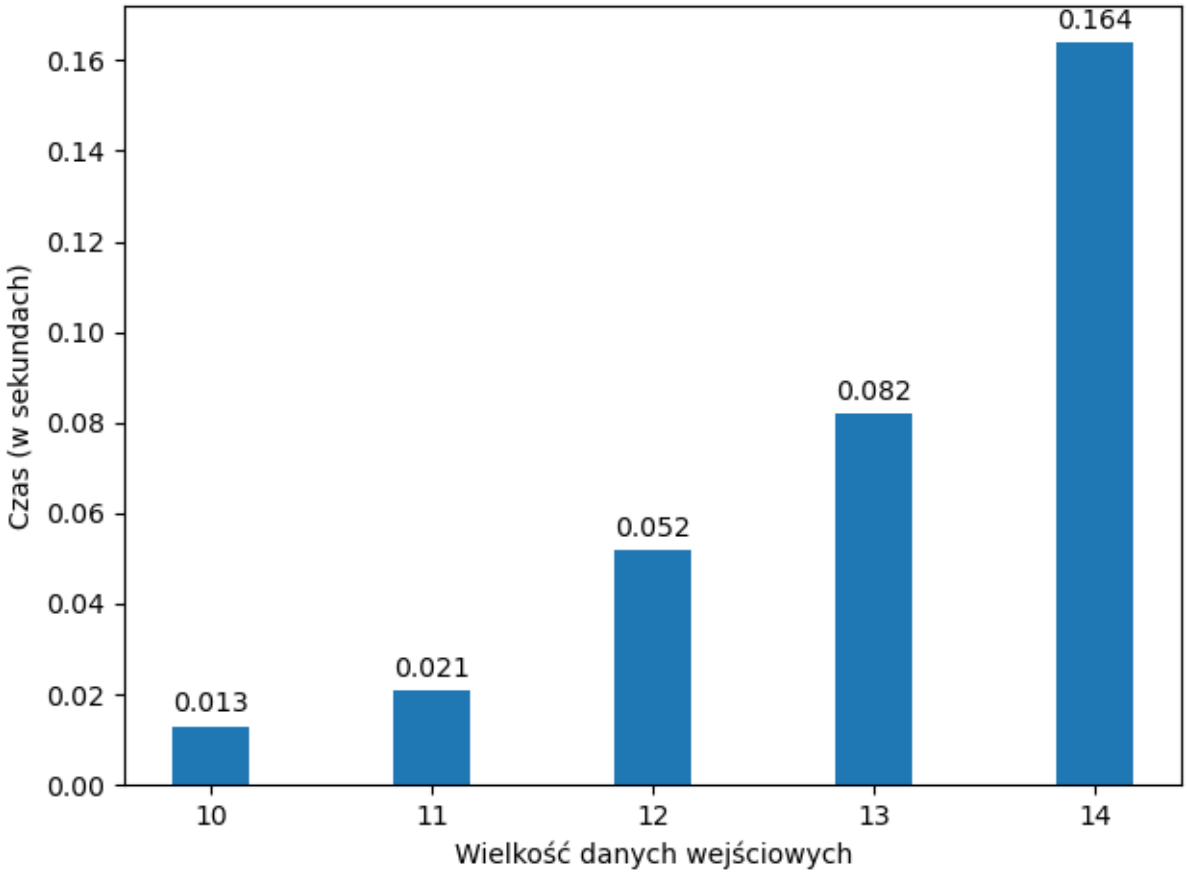


Zależność czasu od wielkości danych jest $O(n \log n)$. SWI-Prolog jest do 7000 razy szybszy.



W tym wypdaku różnica w liczbie inferencji jest bardziej wyraźna i rośnie wykładniczo.


185.698
175
150
125
100
90.7641
75
50
45.0457
25
22.5665
10.7481
0
Czas (w sekundach)
10
11
12
13
14
Wielkość danych wejściowych



Potrzebny czas rośnie wykładniczo, a SWI-Prolog jest około 1000 razy szybszy.

Podsumowanie

Podsumowując implementację udało się zakończyć sukcesem. Zawiera ona pełną funkcjonalność abstrakcyjnej maszyny Warrena, pozwala rozdzielenie kompilacji i wykonywania i pozwala na łatwe testowanie działania i programów. Mimo to pozostaje jeszcze wiele ścieżek rozbudowy.
Zaproponowana implementacja o ile działa, to działa o wiele wolniej od innych łatwo dostępnych implementacji. Dwa możliwe sposoby na przyspieszenie jej działania to: wprowadzenie nowych, bardziej wyspecjalizowanych instrukcji (np. `get_list`) i zmiana sposobu przechowywania kodu z tekstowego na reprezentowanego przez liczby naturalne.
Inną możliwością ulepszenia implementacji jest dodanie nowych funkcjonalności powszechnie dostępnych w innych implementacjach, takich jak: arytmetyka, operacje na ciągach znaków i odcięcia.

Bibliografia

[1] H. Ait-Kaci. Warren's abstract machine. http://wambook.sourceforge.net/wambook.pdf, 1999.

[2] D. H. Warren. An abstract prolog instruction set. http://www.ai.sri.com/pubs/files/641.pdf, 1983.

Zawartość płyty CD

Zawartość płyty CD:
Ten dokument (`praca.pdf`)
Skompilowana implementacja (`wam`)
Pliki z kodem źródłowym (zawartość folderu `src`)
Przykładowe programy (folder `tests`)
Plik Makefile służący do kompilacji (`makefile`)