UNIWERSYTET JANA DŁUGOSZA w CZĘSTOCHOWIE

Wydział Nauk Ścisłych, Przyrodniczych i Technicznych

Kierunek: Informatyka

Specjalność: Programowanie

Tetiana Mossur

nr albumu: 76716

Efektywność SMT solverów dla klasycznych problemów NP-trudnych

Effectiveness of SMT solvers for classical NP-hard problems

Praca magisterska przygotowana

pod kierunkiem

dr hab. Andrzeja Zbrzeznego

Częstochowa 2024

Podziękowania

Serdecznie dziękuję dr. hab. Andrzejowi Zbrzeznemu za nieocenioną pomoc przy tworzeniu pracy magisterskiej, a także Katedrze Informatyki i Matematyki UJD za wspaniałe lata studiów. Niezmiernie wdzięczna jestem również moim rodzicom i bliskim za wsparcie oraz Siłom Zbrojnym Ukrainy za możliwość napisania tej pracy.

Streszczenie

Celem niniejszej pracy jest analiza i ocena efektywności SMT solverów w rozwiązywaniu klasycznych problemów zaliczanych do klasy NP-trudnych. Przełom w informatyce teoretycznej sprawił, że tego rodzaju zagadnienia stały się przedmiotem intensywnych badań. Niniejsza praca skupia się na zrozumieniu, jak SMT (Satisfiability Modulo Theories) solvery, będące potężnym narzędziem w dziedzinie rozstrzygania logicznego, radzą sobie z tymi wyjątkowo wymagającymi problemami.

Przedmiotem badań w pracy są klasyczne problemy NP-trudne, a celem jest zbadanie i porównanie wydajności trzech SMT solverów - Z3, Yices i cvc5 - w kontekście rozwiązywania konkretnych problemów. Wybór tych narzędzi wynika z ich popularności, wszechstronności i aktywnego udziału w społeczności badawczej.

Słowa kluczowe

Złożoność obliczeniowa, problemy NP-trudne, SMT, SMT-solvery, Z3, Yices, cvc5, Python

Spis treści

Wstęp

W niektórych dziedzinach informatyki, takich jak formalna weryfikacja sprzętu i oprogramowania, wiele istotnych problemów można zredukować do sprawdzenia spełnialności formuły w pewnej logice. Kilka z tych problemów można naturalnie sformułować jako problemy spełnialności w logice propozycjonalnej i rozwiązać bardzo efektywnie przy użyciu nowoczesnych solverów SAT. Inne problemy są bardziej zwięźle sformułowane w logikach klasycznych, takich jak logika pierwszego lub wyższego rzędu, o bardziej złożonym języku, który obejmuje zmienne niebędące wartościami logicznymi, funkcje i symbole predykatów oraz kwantyfikatory. Oczywiście istnieje kompromis między złożonością logiki a zdolnością do automatycznego sprawdzania spełnialności jej formuł.

Praktyczny kompromis można osiągnąć za pomocą fragmentów logiki pierwszego rzędu, które są ograniczone syntaktycznie, na przykład przez zezwolenie tylko na pewne klasy formuł, lub semantycznie, poprzez ograniczenie interpretacji niektórych funkcji i symboli predykatów, lub obie te opcje jednocześnie. Takie ograniczenia mogą uczynić problem spełnialności rozstrzygalnym i, co ważniejsze, umożliwić rozwijanie specjalistycznych procedur spełnialności, które wykorzystują właściwości fragmentu logiki z dużą korzyścią dla wydajności praktycznej, nawet w przypadku wysokiej złożoności obliczeniowej dla najgorszych scenariuszy (worst-case computational complexity). Kiedy są zaangażowane ograniczenia semantyczne, mogą one być pojęte jako ograniczenie interpretacji do modeli z dokładnością do ustalonej teorii (background theory) (np. teorii równości, liczb całkowitych, liczb rzeczywistych, tablic, list itp.). W takich przypadkach mówimy o Spełnialności Moduło Teorii (SMT).

W oparciu o klasyczne rozwiązania dotyczące procedur decyzyjnych dla rozumowania pierwszego rzędu, a także ogromny rozwój technologii rozwiązywania SAT w ciągu minionych dwóch dekad, SMT rozwinęło się przez ostatnie lata w bardzo aktywny obszar badawczy, którego cechą charakterystyczną jest wykorzystywanie metod wnioskowania, właściwych logicznym teoriom o dużym znaczeniu dla problemów stosowanych. Dzięki postępowi w badaniach i technologii SMT, istnieje obecnie kilka potężnych i zaawansowanych solverów SMT (np. Alt-Ergo, Beaver, Boolector, cvc5, MathSAT5, openSMT, SMTInterpol, SONOLAR, STP, veriT, Yices i Z3), które są wykorzystywane w szybko rosnącym szeregu aplikacji. Wśród nich znajdują się obecnie m.in. weryfikacja procesorów, badanie równoważności, ograniczone i nieograniczone sprawdzanie modeli, abstrakcja predykatów, analiza statyczna, automatyczne generowanie przypadków testowych, sprawdzanie typów, harmonogramowanie i optymalizacja. Ostatnie osiągnięcia w dziedzinie SMT były spowodowane kilkoma czynnikami, w tym skupieniem się na podstawowych teoriach i klasach problemów występujących w praktyce, dostosowanie technologii SAT do potrzeb SMT oraz innowacje w zakresie algorytmów i struktur danych [22].

Celem niniejszej pracy jest zbadanie skuteczności SMT solverów w rozwiązywaniu ośmiu

klasycznych problemów NP-trudnych, a mianowicie: Ścieżka Hamiltona w grafie skierowanym, Ścieżka Hamiltona w grafie nieskierowanym, Pokrycie wierzchołkowe, Problem maksymalnej kliki, Problem maksymalnego zbioru niezależnego, Problem Komiwojażera, Kolorowanie grafu, oraz Problem sumy podzbioru.

W pierwszym rozdziale zawarłam teoretyczne fundamenty Satisfiability Modulo Theories (SMT), wprowadzając czytelnika w główne aspekty tej dziedziny. Omówiłam kluczowe pojęcia, biorąc pod uwagę rozwój SMT-solverów i ich praktyczne zastosowania w realnych sytuacjach, takich jak te związane z klasą problemów NP-trudnych.

Drugi rozdział przedstawia podstawowe pojęcia złożoności obliczeniowej oraz koncepcji spełnialności, dwóch kluczowych aspektów niezbędnych do zrozumienia, czym są problemy obliczeniowe w kontekście SMT. Następnie dokonywane jest ogólne wprowadzenie do klasy problemów NP-trudnych, które stanowią centralny temat niniejszej pracy magisterskiej.

W rozdziale trzecim zawarłam szczegółowy przegląd trzech popularnych SMT-solverów: Z3, Yices i cvc5, podkreślając ich cechy, mocne strony i potencjalne ograniczenia. Czytelnik zdobywa wgląd w różnice między tymi narzędziami, co stanowi podstawę dla późniejszych badań.

Rozdział czwarty skupia się na przedstawieniu 8 klasycznych problemów NP-trudnych, a następnie opisuje sposób ich kodowania w języku Python.

Rozdział piąty poświęcony jest praktycznym eksperymentom, wykorzystując wyżej przedstawione solvery do rozwiązania wybranych problemów obliczeniowych. Analiza wyników pozwoli określić efektywność każdego z nich i wyciągnąć wnioski co do ich zastosowania.

ROZDZIAŁ 1

Teoretyczne podstawy Satisfiability Modulo Theories

1.1. Wprowadzenie do SMT

1.1.1. Problem spełnialności

Problem spełnialności formuł boolowskich (SAT) jest ważnym problemem algorytmicznym w teorii złożoności obliczeniowej.

Obiektem problemu SAT jest formuła boolowska składająca się jedynie z nazw zmiennych, nawiasów i operacji $\wedge$ (AND), $\vee$ (OR) oraz $\neg$ (NOT). Problem polega na tym, czy możliwe jest przypisanie wartości false i true do wszystkich zmiennych występujących w formule, tak aby formuła stała się prawdziwa.

Zgodnie z twierdzeniem Cooka, udowodnionym przez Stephena Cooka w 1971 roku, problem SAT dla formuł boolowskich zapisanych w koniunkcyjnej postaci normalnej jest NP-zupełny. Wymóg zapisu w postaci koniunkcyjnej jest ważny, ponieważ, na przykład, dla formuł reprezentowanych w dysjunkcyjnej postaci normalnej, problem SAT jest trywialnie rozwiązywany w czasie liniowym względem rozmiaru formuły.

1.1.2. Satisfiability Modulo Theories

Satisfiability Modulo Theories (SMT) to dziedzina informatyki teoretycznej, która łączy w sobie problem spełnialności logicznej (SAT) z różnymi teoriami matematycznymi [11].

W kontekście SMT, dany jest zestaw ograniczeń logicznych wyrażonych za pomocą formuł logiki pierwszego rzędu oraz dodatkowe ograniczenia wynikające z konkretnych teorii matematycznych, takich jak teoria liczb całkowitych, teoria równań różniczkowych, czy teoria tablic.

Problematyka SMT polega na stwierdzeniu, czy istnieją wartości zmiennych spełniające zarówno ograniczenia logiczne, jak i dodatkowe ograniczenia wynikające z wybranej teorii matematycznej. W przypadku pozytywnej odpowiedzi, rozwiązaniem problemu jest znaczenie konkretnych wartości zmiennych, które spełniają wszystkie warunki.

1.2. Historia i rozwój SMT solverów

Pierwsze próby rozwiązywania problemów SMT miały na celu przekształcenie ich w formuły SAT (na przykład 32-bitowe zmienne zostały zakodowane przez 32 zmienne boolowskie, a odpowiadające im operacje na słowach zostały zakodowane jako niskopoziomowe operacje bitowe) i rozwiązanie formuły za pomocą SAT solvera. Podejście to ma swoje zalety - pozwala na wykorzystanie istniejących solverów SAT bez zmian (As-Is), a także na wykorzystanie wprowadzonych w nich optymalizacji. Z drugiej strony, utrata wysokopoziomowej semantyki leżącej u podstaw teorii oznacza, że solver SAT musi podjąć znaczne działania, aby wywnioskować óczywiste"fakty (takie jak $x + y = y + x$ dla dodawania). Pomysł ten doprowadził do powstania wyspecjalizowanych solverów SMT, które integrują konwencjonalne dowody boolowskie w stylu algorytmu DPLL z solverami specyficznymi dla teorii ("T solvery"), które działają z dysjunkcjami i koniunkcjami predykatów z danej teorii. Satisfiability Modulo Theories(SMT) uogólniają teorię spełnialności boolowskiej (SAT) poprzez dodanie rozumowania równościowego, arytmetyki, bit-wektorów o stałym rozmiarze, tablic, kwantyfikatorów i innych przydatnych teorii pierwszego rzędu. SMT solver jest narzędziem do decydowania o spełnialności formuł w tych teoriach.

1.3. Podstawowe teorie logiczne

Teoria Liniowej Arytmetyki Liczb Całkowitych (Linear Integer Arithmetic) jest fundamentalną gałęzią logiki matematycznej, która zajmuje się manipulacją liczb całkowitych w sposób liniowy, tj. poprzez równania i nierówności liniowe. Problem SMT z teorią LIA polega na określeniu spełnialności kombinacji boolowskiej odpowiednich arytmetycznych formuł atomowych i oznaczane jest symbolem SMT(LIA).

SMT(LIA) jest ważne w weryfikacji oprogramowania i zautomatyzowanym wnioskowaniu, ponieważ większość programów używa zmiennych całkowitoliczbowych i wykonuje na nich operacje arytmetyczne. W szczególności SMT (LIA) ma różne zastosowania w automatycznej analizie rozwiązań, sekwencyjnym sprawdzaniu równoważności i osiągalności stanu w słabych modelach pamięci [3]. Jest to również kluczowa teoria używana przez SMT-solvery do rozwiązywania problemów z zakresu NP-trudnych, takich jak problem plecakowy czy problem pokrycia wierzchołkowego.

Teoria Zbiorów (Sets) pozwala na manipulację zbiorami elementów oraz wykonywanie operacji takich jak dodawanie, usuwanie, sprawdzanie przynależności, czy operacje na zbiorach (np. przekroje, różnice). Jest szczególnie przydatna w problemach, gdzie dane lub zbiory danych mogą być reprezentowane za pomocą zbiorów, takich jak problem sumy podzbioru czy problem planowania zadań [17].

Teoria Zmiennych Boolowskich (Boolean Variables): Teoria Zmiennych Boolowskich umożliwia operacje na zmiennych logicznych, które mogą przyjąć wartość prawda/fałsz. Po-

zwala na konstruowanie formuł logicznych, wyrażeń logicznych oraz wykonywanie operacji takich jak koniunkcja, alternatywa, negacja, implikacja czy równoważność [2].

Teoria Tablic (Arrays): Teoria Tablic umożliwia modelowanie i manipulację strukturami danych, takimi jak tablice. Pozwala na definiowanie tablic, dostęp do ich elementów oraz wykonywanie operacji takich jak przypisanie wartości do elementów tablicy czy odczytywanie wartości z tablicy. Jest przydatna w problemach, gdzie dane są przechowywane w postaci tablic, np. w problemach dotyczących grafów (np. macierzy sąsiedztwa czy list sąsiedztwa), problemach sortowania czy problemach związanych z przetwarzaniem danych [23].

1.4. Zastosowanie SMT w praktyce

SMT solvery są powszechnie uznawane za niezbędne mechanizmy rozumowania dla różnych obszarów zastosowań, w tym weryfikacja mikroprocesorów z potokiem, sprawdzanie równoważności mikrokodu, testowanie biało-skrzynkowe w zastosowaniach związanych z bezpieczeństwem, eksploracja przestrzeni projektowej, a także synteza konfiguracji oraz odkrywanie materiałów kombinatorycznych. [1]. Ich sukces wynika z kilku czynników, w tym zdolności do pracy z bogatszymi reprezentacjami danych oraz możliwością rozszerzenia pojemności poprzez działanie na poziomie wyższym niż tylko logiczne wartości boolowskie.

Rozwiązania oparte na SMT znalazły również zastosowanie w obszarach związanych z problemami NP-trudnymi, które są kluczowe dla wielu dziedzin nauki i technologii. Dzięki swojej zdolności do radzenia sobie z złożonymi problemami decyzyjnymi, SMT solvery są wykorzystywane do modelowania i rozwiązywania problemów trudnych do rozwiązania w sposób klasyczny. Przykładowe zastosowania obejmują optymalizację kombinatoryczną, planowanie zasobów, analizę i weryfikację systemów złożonych oraz projektowanie oprogramowania o wysokiej niezawodności. Ponadto, SMT solvery znajdują zastosowanie w dziedzinach badawczych, takich jak sztuczna inteligencja, gdzie są używane do rozwiązywania problemów logicznych i wnioskowania. Ich rosnąca popularność w obszarze problemów NP-trudnych wynika zarówno z ich wydajności, jak i z możliwości pracy z różnymi teoriami i rodzajami ograniczeń, co czyni je wszechstronnym narzędziem w analizie i rozwiązywaniu złożonych problemów decyzyjnych.

ROZDZIAŁ 2

Problemy NP-trudne

Załóżmy, że jesteśmy menedżerem logistyki, który musi zoptymalizować harmonogram dostaw towarów przez park pojazdów w warunkach ograniczonej sieci dystrybucyjnej obejmującej całe miasto. To pozornie rutynowe zadanie, po bliższym przyjrzeniu się, okazuje się być złożonym problemem optymalizacji kombinatorycznej, charakteryzującym się potrzebą minimalizacji kosztów paliwa, skrócenia czasu podróży i maksymalizacji przepustowości dostaw.

W poszukiwaniu optymalnego rozwiązania mamy do czynienia ze stale rosnącym zestawem zmiennych, w tym nieprzewidywalną dynamiką ruchu, różnymi rozmiarami paczek i dynamicznymi zmianami popytu ze strony klientów. Każda decyzja o ustaleniu konkretnych tras dostawy lub nadaniu priorytetu określonym miejscom docelowym powoduje eksplozję kombinatorycznych możliwości, zmieniając problem optymalizacji logistycznej w przykład problemu NP-trudnego.

Z perspektywy obliczeniowej, problemy NP-trudne są klasą problemów, dla których nie istnieje algorytm wielomianowy, który może zapewnić optymalne rozwiązanie we wszystkich przypadkach. Złożoność ta jest wyraźnie widoczna w problemach takich jak optymalizacja tras, gdzie ogromna przestrzeń rozwiązań nie pozwala na proste rozwiązanie. Złożoność takich problemów logistycznych odzwierciedla szersze trudności występujące w problemach NP-trudnych w różnych obszarach obliczeniowych i optymalizacyjnych.

2.1. Teoria złożoności obliczeniowej

W informatyce teoretycznej i matematyce, teoria złożoności obliczeniowej koncentruje się na klasyfikacji problemów obliczeniowych według ich wykorzystania zasobów i powiązaniu tych klas ze sobą. Problem obliczeniowy to zadanie rozwiązywane przez komputer. Problem obliczeniowy jest rozwiązywalny poprzez mechaniczne zastosowanie kroków matematycznych, takich jak algorytm.

Problem jest uważany za z natury trudny, jeśli jego rozwiązanie wymaga znacznych zasobów, niezależnie od zastosowanego algorytmu. Teoria formalizuje tę intuicję, wprowadzając matematyczne modele obliczeń do badania tych problemów i kwantyfikując ich złożoność obliczeniową, tj. ilość zasobów potrzebnych do ich rozwiązania, takich jak czas i pamięć. Stosowane są również inne miary złożoności, takie jak ilość komunikacji (używana w złożoności komunikacyjnej), liczba bramek w obwodzie (używana w złożoności obwodu) i liczba procesorów (używana w obliczeniach równoległych). Jednym z zadań teorii złożoności oblicze-

niowej jest określenie praktycznych ograniczeń tego, co komputery mogą, a czego nie mogą zrobić. Problem P versus NP, jeden z siedmiu problemów nagrodzonych Millennium Prize, jest poświęcony dziedzinie złożoności obliczeniowej.

2.1.1. Problemy obliczeniowe

Instancje problemu

Problem obliczeniowy może być rozważany jako nieskończony zbiór instancji wraz ze zbiorem (być może pustym) rozwiązań dla każdej instancji. Ciąg wejściowy dla problemu obliczeniowego jest określany jako instancja problemu i nie powinien być mylony z samym problemem. W teorii złożoności obliczeniowej, problem odnosi się do abstrakcyjnego pytania, które należy rozwiązać. W przeciwieństwie do tego, instancja problemu jest raczej konkretną wypowiedzią, która może służyć jako dane wejściowe dla problemu decyzyjnego. Dla przykładu, rozważmy problem testowania pierwotności. Instancją jest liczba (np. 15), a rozwiązaniem jest 'tak', jeśli liczba jest pierwsza i 'nie' w przeciwnym razie (w tym przypadku 15 nie jest pierwsza i odpowiedź brzmi 'nie'). Innymi słowy, instancja jest konkretnym wejściem do problemu, a rozwiązanie jest wyjściem odpowiadającym danemu wejściu.

Aby jeszcze bardziej podkreślić różnicę między problemem a instancją, rozważmy następującą instancję decyzyjnej wersji problemu komiwojażera: Czy istnieje trasa o długości co najwyżej 2000 kilometrów przechodząca przez wszystkie 15 największych miast Niemiec? Ilościowa odpowiedź na ten konkretny przypadek problemu jest mało przydatny do rozwiązywania innych przypadków, takich jak pytanie o podróż w obie strony przez wszystkie dzielnice Mediolanu, których łączna długość wynosi co najwyżej 10 km. Z tego powodu teoria złożoności zajmuje się problemami obliczeniowymi, a nie konkretnymi przypadkami problemów.

Reprezentacja instancji problemu

Rozważając problemy obliczeniowe, instancja problemu jest ciągiem znaków w alfabecie. Zazwyczaj przyjmuje się, że alfabet jest alfabetem binarnym (tj. zbiorem $\{0, 1\}$), a zatem ciągi są ciągami bitów. Podobnie jak w prawdziwym komputerze, obiekty matematyczne inne niż ciągi bitów muszą być odpowiednio zakodowane. Na przykład, liczby całkowite mogą być reprezentowane w notacji binarnej, a grafy mogą być kodowane bezpośrednio poprzez ich macierze sąsiedztwa lub poprzez kodowanie ich list sąsiedztwa w zapisie binarnym.

Problemy decyzyjne jako języki formalne

Problem decyzyjny ma tylko dwa możliwe wyjścia, tak lub nie (lub naprzemiennie 1 lub 0) na dowolnym wejściu. Problemy decyzyjne są jednym z głównych przedmiotów badań w teorii złożoności obliczeniowej. Problem decyzyjny jest specjalnym rodzajem problemu obli-

czeniowego, którego odpowiedzią jest ”takłub ńie”. Problem decyzyjny może być postrzegany jako język formalny, elementami którego są przypadki z wynikiem ”tak”, a poza nim są przypadki z wynikiem ńie”. Celem jest podjęcie decyzji, za pomocą algorytmu, czy dany ciąg wejściowy jest elementem rozważanego języka formalnego. Jeśli algorytm rozwiązujący ten problem zwróci odpowiedź twierdzącą, mówi się, że algorytm akceptuje ciąg wejściowy, w przeciwnym razie mówi się, że odrzuca dane wejściowe.

Przykład problemu decyzyjnego jest następujący. Dane wejściowe to dowolny graf. Problem polega na rozstrzygnięciu, czy dany graf jest spójny, czy nie. Formalnym językiem związanym z tym problemem decyzyjnym jest zbiór wszystkich spójnych grafów - aby uzyskać dokładną definicję tego języka, należy zdecydować, w jaki sposób grafy są kodowane jako ciągi binarne.

Problemy funkcyjne

Problem funkcyjny jest problemem obliczeniowym, w którym dla każdego wejścia oczekuje się pojedynczego wyniku, ale wynik jest bardziej złożony niż w przypadku problemu decyzyjnego - to znaczy, wynik nie jest tylko tak lub nie. Godnymi uwagi przykładami są problem komiwojażera i faktoryzacja liczb całkowitych.

Może się wydawać, że pojęcie problemów funkcyjnych jest znacznie bogatsze niż pojęcie problemów decyzyjnych. Nie jest to jednak prawdą, ponieważ problemy funkcyjne można przekształcić w problemy decyzyjne. Na przykład, mnożenie dwóch liczb całkowitych może być wyrażone jako zbiór trójek (a, b, c) takich, że zachodzi relacja $a \cdot b = c$. Podjęcie decyzji, czy dana trójka jest członkiem tego zbioru, odpowiada rozwiązaniu problemu mnożenia dwóch liczb.

Mierzenie rozmiaru instancji

Aby zmierzyć złożoność obliczeniową problemu, konieczne jest zbadanie czasu wykonania najlepszego algorytmu w zależności od rozmiaru instancji. Jest to istotne ze względu na fakt, że czas wykonania zależy od konkretnego zestawu danych wejściowych, a większe instancje problemu zazwyczaj wymagają dłuższego czasu na rozwiązanie. Czas ten (lub pamięć, lub jakakolwiek miara złożoności) jest tradycyjnie wyrażany jako funkcja rozmiaru instancji, zwykle rozumianego jako liczba bitów reprezentujących dane wejściowe. Teoria złożoności skupia się analizie skalowalności algorytmów wraz ze wzrostem rozmiaru danych wejściowych. Przykładowo, jak znacząco zwiększa się czas rozwiązania problemu dla grafu o $2n$ wierzchołkach w porównaniu do grafu o n wierzchołkach.

Przyjmując, że rozmiar danych wejściowych wynosi n, czas potrzebny na rozwiązanie może być wyrażony jako funkcja $f(n)$. Ponieważ czas rozwiązania różnych zestawów danych o tym samym rozmiarze może się różnić, definiuje się najgorszą złożoność czasową $T(n)$ jako maksymalny czas potrzebny do rozwiązania wszystkich danych wejściowych o rozmiarze n.

Jeśli $T(n)$ jest wielomianem w n, to algorytm jest nazywany algorytmem czasu wielomianowego. Praca Cobhama potwierdza, że problem może być rozwiązany przy użyciu realnej ilości zasobów, jeśli dostępny jest algorytm wielomianowy.

2.1.2. Modele i miary złożoności

Maszyna Turinga

Maszyna Turinga to matematyczny model uniwersalnego komputera. Jest to teoretyczne urządzenie, które manipuluje symbolami zawartymi na taśmie. Maszyny Turinga nie są pomyślane jako praktyczna technologia obliczeniowa, ale raczej jako ogólny model maszyny obliczeniowej - od zaawansowanego superkomputera do matematyka z ołówkiem i kartką papieru. Uważa się, że jeśli jakiś problem może zostać rozwiązany za pomocą algorytmu, to istnieje maszyna Turinga, która rozwiązuje ten problem. Faktycznie, jest to stwierdzenie tezy Churcha-Turinga. Co więcej, wiadomo, że wszystko, co można obliczyć na innych znanych nam obecnie modelach obliczeniowych, takich jak maszyna RAM, Conway's Game of Life, automaty komórkowe, rachunek lambda lub dowolny język programowania, można obliczyć na maszynie Turinga. Ponieważ maszyny Turinga są łatwe do analizy matematycznej i uważa się, że są tak potężne, jak każdy inny model obliczeń, maszyna Turinga jest najczęściej używanym modelem w teorii złożoności.

Wiele typów maszyn Turinga jest używanych do definiowania klas złożoności, takich jak deterministyczne maszyny Turinga, probabilistyczne maszyny Turinga, niedeterministyczne maszyny Turinga, kwantowe maszyny Turinga, symetryczne maszyny Turinga i naprzemienne maszyny Turinga. Zasadniczo wszystkie są równie potężne, ale gdy zasoby (takie jak czas lub przestrzeń) są ograniczone, niektóre z nich mogą być potężniejsze od innych.

Deterministyczna maszyna Turinga jest najbardziej podstawową maszyną Turinga, która wykorzystuje ustalony zestaw reguł do określenia swoich przyszłych akcji. Probabilistyczna maszyna Turinga to deterministyczna maszyna Turinga z dodatkowym zapasem losowych bitów. Zdolność do podejmowania probabilistycznych decyzji często pomaga algorytmom rozwiązywać problemy bardziej efektywnie. Algorytmy wykorzystujące losowe bity nazywane są algorytmami losowymi. Niedeterministyczna maszyna Turinga to deterministyczna maszyna Turinga z dodatkową właściwością niedeterminizmu, która pozwala maszynie Turinga na wiele możliwych akcji w przyszłości z danego stanu. Niedeterminizm można rozumieć w ten sposób, że maszyna Turinga na każdym kroku rozgałęzia się na wiele możliwych ścieżek obliczeniowych, a jeśli rozwiąże problem w którejkolwiek z tych gałęzi, mówi się, że rozwiązała problem. Oczywiście model ten nie jest fizycznie możliwym do zrealizowania - to tylko teoretycznie użyteczna abstrakcyjna maszyna, który pozwala na wyróżnienie wielu klas złożoności.

Miary złożoności

W celu dokładnego zdefiniowania, co oznacza rozwiązanie problemu przy użyciu danej ilości czasu i pamięci, stosuje się model obliczeniowy, taki jak deterministyczna maszyna Turinga. Czas wymagany przez deterministyczną maszynę Turinga M na wejściu x jest całkowitą liczbą przejść stanów lub kroków, które maszyna wykonuje zanim się zatrzyma i wyśle odpowiedź (tak lub nie). Mówi się, że maszyna Turinga M działa w czasie $f(n)$, jeśli czas wymagany przez M na każdym wejściu o długości n wynosi co najwyżej $f(n)$. Problem decyzyjny A może być rozwiązany w czasie $f(n)$, jeśli istnieje maszyna Turinga działająca w czasie $f(n)$, która rozwiązuje ten problem. Ponieważ teoria złożoności interesuje się klasyfikacją problemów na podstawie ich trudności, definiuje się zbiory problemów na podstawie pewnych kryteriów. Na przykład, zbiór problemów rozwiązywalnych w czasie $f(n)$ na deterministycznej maszynie Turinga jest oznaczany przez $DTIME(f(n))$.

Analogiczne definicje można stworzyć dla wymagań przestrzennych. Chociaż czas i pamięć są najbardziej znanymi zasobami złożoności, każda miara złożoności może być postrzegana jako zasób obliczeniowy. Miary złożoności są bardzo ogólnie definiowane przez aksjomaty złożoności Bluma. Inne miary złożoności stosowane w teorii złożoności obejmują złożoność komunikacyjną, złożoność obwodu i złożoność drzewa decyzyjnego.

Złożoność algorytmu jest często wyrażana za pomocą notacji duże O.

2.1.3. Historia

Wczesnym przykładem badania złożoności algorytmów jest analiza czasu działania algorytmu Euklidesa przeprowadzona przez Gabriela Lamé w 1844 roku.

Zanim rozpoczęły się rzeczywiste badania wyraźnie poświęcone złożoności problemów algorytmicznych, wielu badaczy opracowało ich podstawy. Najbardziej wpływowym z nich było zdefiniowanie maszyn Turinga przez Alana Turinga w 1936 roku, które okazały się bardzo stabilną i uniwersalną wersją komputera.

Początek systematycznych badań nad złożonością obliczeniową przypisuje się przełomowej pracy z 1965 roku Ón the Computational Complexity of Algorithmsautorstwa Jurisa Hartmanisa i Richarda E. Stearnsa, w której określono definicje złożoności czasowej i pamięciowej oraz udowodniono twierdzenia o hierarchii [5]. Ponadto, w 1965 roku Edmonds zasugerował, by za "dobryalgorytm uznać taki, którego czas działania jest ograniczony wielomianem rozmiaru danych wejściowych [7].

Wcześniejsze prace badające problemy rozwiązywalne przez maszyny Turinga z określonymi ograniczonymi zasobami obejmują [5] definicję liniowych automatów ograniczonych Johna Myhilla (Myhill 1960), badanie podstawowych zbiorów Raymonda Smullyana (1961), a także artykuł Hisao Yamady [20] na temat obliczeń w czasie rzeczywistym (1962). Nieco wcześniej Boris Trakhtenbrot (1956), pionier w tej dziedzinie z ZSRR, badał inną specyficzną miarę złożoności:

”Jednak [moje] początkowe zainteresowanie [teorią automatów] było coraz bardziej odkładane na korzyść złożoności obliczeniowej, ekscytującej fuzji metod kombinatorycznych, odziedziczonych po teorii przełączania, z arsenałem pojęciowym teorii algorytmów. Pomysły te przyszły mi do głowy wcześniej, w 1955 roku, kiedy wymyśliłem termin ”funkcja sygnalizacyjna”, która obecnie jest powszechnie znana jako ”miara złożoności”[19].

W 1967 roku Manuel Blum sformułował zestaw aksjomatów (obecnie znanych jako aksjomaty Bluma) określających pożądane właściwości miar złożoności na zbiorze funkcji obliczalnych i uzyskał ważny rezultat, tak zwane twierdzenie o przyspieszeniu. Dziedzina ta zaczęła się rozwijać w 1971 roku, kiedy Stephen Cook i Leonid Levin udowodnili istnienie praktycznie ważnych problemów, które są NP-zupełne. W 1972 roku Richard Karp posunął tę ideę o krok naprzód w swoim przełomowym artykule Żeducibility Among Combinatorial Problems”, w którym wykazał, że 21 różnych problemów kombinatorycznych i grafowych, z których każdy znany jest z niewykonalności obliczeniowej, są NP-zupełne [7].

2.2. Definicja klasy problemów NP-trudnych

Klasy złożoności stanowią zbiory problemów decyzyjnych, które są uznawane za rozwiązywalne przez algorytmy z określoną ilością zasobów. Wśród kluczowych zasobów wymieniane są przede wszystkim czas i przestrzeń, gdzie czas odnosi się do liczby kroków algorytmu niezbędnych do rozwiązania problemu, a przestrzeń dotyczy ilości pamięci potrzebnej do wykonania algorytmu.

W ramach klasyfikacji klas złożoności, każda z nich jest oparta na zbiorze problemów decyzyjnych, które mogą być rozwiązane w określonym czasie lub zużywają określoną przestrzeń. Na przykład, klasa P obejmuje problemy decyzyjne, które mogą być rozwiązane przez algorytm deterministyczny w czasie wielomianowym, podczas gdy klasa NP zawiera problemy decyzyjne, które mogą być zweryfikowane w czasie wielomianowym przez algorytm niedeterministyczny.

Klasa P charakteryzuje się czasem wielomianowym, co oznacza, że czas działania algorytmu jest ograniczony funkcją wielomianową rozmiaru danych wejściowych. Przykłady problemów z klasy P obejmują sortowanie listy liczb, znajdowanie najkrótszej ścieżki w grafie i określanie, czy liczba jest liczbą pierwszą. Z kolei klasa NP zawiera problemy decyzyjne, które mogą być zweryfikowane w czasie wielomianowym przez algorytm niedeterministyczny.

W ramach klasyfikacji NP, problemy NP-zupełne są uznawane za szczególnie trudne, ponieważ nie tylko znajdują się w klasie NP, ale także każdy inny problem w NP może być w czasie wielomianowym do nich zredukowany. Przykładem problemu NP-zupełnego jest Problem Komiwojażera (TSP), który polega na znalezieniu najkrótszej możliwej trasy, która odwiedza każde miasto z podanej listy. Podczas gdy TSP może być rozwiązany w czasie wykładniczym,

nie jest znany algorytm czasu wielomianowego, który rozwiązuje wszystkie przypadki tego problemu [27].

Z kolei, klasa problemów NP-trudnych zawiera problemy decyzyjne, które są co najmniej tak trudne jak najtrudniejsze problemy, należące do klasy NP. Jest to zbiór problemów, które wymagają znacznie więcej czasu na rozwiązanie niż problemy w klasie NP, nawet przy wykorzystaniu zaawansowanych algorytmów obliczeniowych.

Kluczową cechą problemów NP-trudnych jest możliwość zredukowania każdego problemu w klasie NP do problemu NP-trudnego w czasie wielomianowym. Innymi słowy, problem jest uznawany za NP-trudny, jeśli istnieje możliwość przekształcenia każdego problemu w klasie NP do tego problemu w sposób, który nie wprowadza znaczącego wzrostu złożoności czasowej. Jednakże, rozwiązanie i weryfikacja problemów NP-trudnych zazwyczaj wymagają dużego nakładu obliczeniowego, nawet dla relatywnie niewielkich instancji problemów.

ROZDZIAŁ 3

Przegląd SMT Solverów: Z3, Yices, cvc5

Rozdział ten ma na celu przedstawienie SMT solverów, efektywność których zostanie zbadana w niniejszej pracy. Każdy z nich charakteryzuje się unikatową architekturą oraz zestawem funkcji, co sprawia, że są one wykorzystywane w różnych kontekstach i scenariuszach. Analiza różnic między tymi solverami pozwoli lepiej zrozumieć ich mocne strony, ograniczenia oraz potencjał w kontekście konkretnych zastosowań. W dalszej części rozdziału zostaną omówione kluczowe cechy i mechanizmy każdego z solverów, wraz z ich porównaniem pod kątem wydajności, elastyczności, oraz możliwości generowania i weryfikacji dowodów.

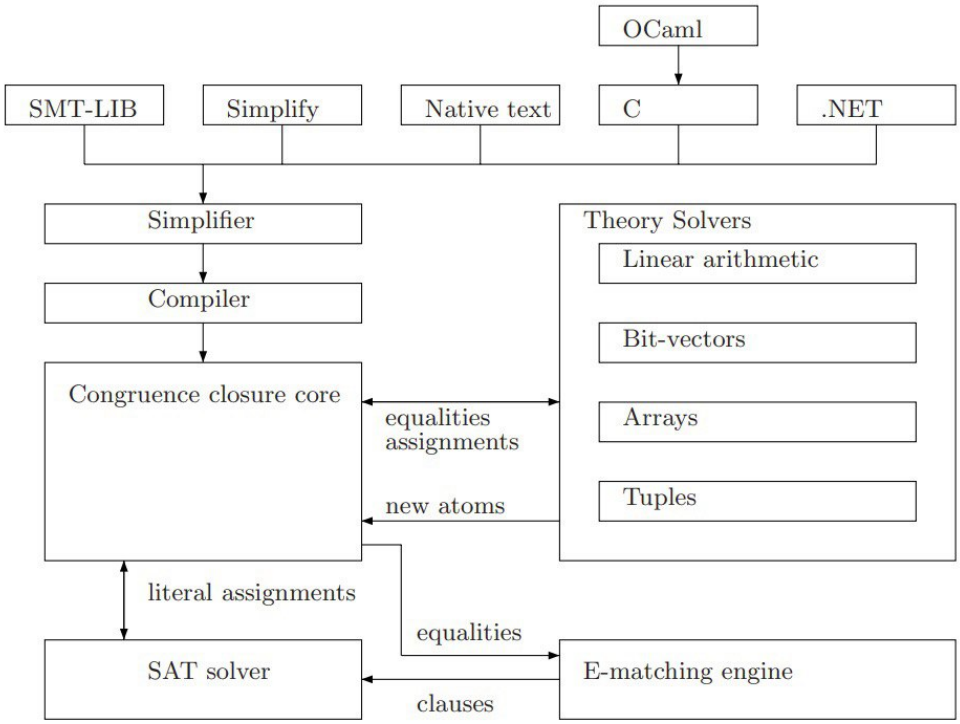
3.1. Z3

Z3 to wydajny SMT solver dostępny bezpłatnie przez Microsoft Research. Z3 jest solverem dla logiki symbolicznej, będącej podstawą wielu narzędzi inżynierii oprogramowania. Solvery SMT polegają na ścisłej integracji wyspecjalizowanych silników walidacyjnych. Każdy silnik jest elementem ogólnej struktury i implementuje wyspecjalizowane algorytmy. Przykładowo, silnik Z3 dla arytmetyki obejmuje Simplex, cięcia i rozumowanie wielomianowe, podczas gdy silnik dla obsługi ciągów znaków i wyrażeń regularnych korzysta z metod symbolicznych pochodnych języków regularnych. Wspólną cechą wielu algorytmów jest sposób, w jaki wykorzystują dwoistość między znajdowaniem rozwiązań spełniających a dowodów odrzucających. Solver ten integruje również silniki do wnioskowań globalnych i lokalnych oraz globalnej propagacji. Z3 jest używany w szerokim zakresie zastosowań inżynierii oprogramowania, obejmując weryfikację programów, walidację kompilatorów, testowanie, fuzzing przy użyciu dynamicznego wykonywania symbolicznego, rozwój oprogramowania oparty na modelach, weryfikację sieci i optymalizację. Z3 jest programem napisanym w C++, a zatem do jego zbudowania potrzebny jest kompilator języka C++. Zapewnia obsługę wielu języków programowania, w tym .NET, C, C++, Java, OCaml, Web Assembly i Python.

3.1.1. Architektura systemu

Z3 integruje nowoczesny solver SAT oparty na DPLL, bazowy solver dla teorii, który obsługuje równości i funkcje nieinterpretowane, specjalistyczne silniki (dla arytmetyki, tablic itp.) oraz maszynę abstrakcyjną E-matching (dla kwantyfikatorów). Schematyczny przegląd Z3 pokazano na rysunku 3.1.

Simplifier. Formuły wejściowe są najpierw przetwarzane przy użyciu niekompletnego,



Rysunek 3.1: Architektura Z3. Źródło: `https://link.springer.com/content/pdf/10.1007/978-3-540-78800-3_24.pdf`

ale wydajnego uproszczenia. Simplifier stosuje standardowe zasady redukcji algebraicznej, takie jak $p \wedge true \rightarrow p$, ale także wykonuje ograniczone uproszczenie kontekstowe, identyfikując definicje równościowe w danym kontekście i redukuje pozostałą formułę przy użyciu definicji, na przykład $x = 4 \wedge q(x) \rightarrow x = 4 \wedge q(4)$. Trywialnie spełnialny spójnik $x = 4$ nie jest kompilowany do jądra, ale zachowany poza nim na wypadek, gdyby klient wymagał modelu do obliczenia x.

Compiler. Uproszczona abstrakcyjna reprezentacja drzewa składniowego formuły jest przekształcana w inną strukturę danych, składającą się ze zbioru klauzul i węzłów domknięcia kongruencji.

Jądro domknięcia kongruencji. Jądro domknięcia kongruencji otrzymuje przypisania prawdy do atomów od solvera SAT. Atomy obejmują równości i formuły atomowe specyficzne dla danej teorii, takie jak nierówności arytmetyczne. Równości stwierdzone przez SAT solver są przekazywane przez jądro domknięcia kongruencji za pomocą struktury danych, którą nazywamy E-grafem. Węzły w E-grafie mogą wskazywać na jeden lub więcej solverów teorii. Gdy dwa węzły są połączone, zbiór odwołań do teorii są łączone, a samo złączenie jest propagowane jako równość do solverów teorii w przecięciu obu zbiorów odwołań. Jądro również propaguje efekty solverów teorii, takie jak wywnioskowane równości oraz atomy przypisane do wartości true lub false.Solvery teorii mogą także generować nowe atomowe wyrażenia w przypadku teorii niekonwekcyjnych. Te atomy są następnie integrowane i zarządzane przez główny solver SAT.

Kombinacja teorii. Tradycyjne metody łączenia solverów teorii opierają się na zdolności tych solverów do generowania wszystkich wynikających równości lub na wprowadzaniu dodatkowych literałów do przestrzeni poszukiwań na etapie wstępnego przetwarzania. Z3 używa nowej metody kombinacji teorii, która przyrostowo dostosowuje modele utrzymywane przez każdą teorię.

SAT Solver. Podziały przypadków logicznych są kontrolowane za pomocą najnowocześniejszego SAT solvera. Solver SAT integruje standardowe metody przycinania wyszukiwania, takie jak dwa obserwowane literały dla wydajnej propagacji ograniczeń boolowskich, lemma learning z wykorzystaniem klauzul konfliktowych, buforowanie faz w celu kierowania podziałami przypadków i wykonuje niechronologiczny backtracking.

Usuwanie klauzul. Instancjonowanie kwantyfikatorów ma skutek uboczny w postaci tworzenia nowych klauzul zawierających nowe atomy w przestrzeni poszukiwań. Z3 usuwa te klauzule wraz z ich atomami i termami, które były bezużyteczne w zamykaniu gałęzi. Klauzule konfliktowe i zawarte w nich literały nie są natomiast usuwane, dlatego instancje kwantyfikatorów, które były przydatne w wywołaniu konfliktów, są zachowywane jako efekt uboczny.

Propagacja relewancji. Solvery oparte na DPLL(T) przypisują wartość boolowską potencjalnie wszystkim atomom pojawiającym się w wyniku. W praktyce niektóre z tych atomów są nieistotne. Z3 ignoruje te atomy dla kosztownych teorii, jak np. wektory bitowe, i reguł wnioskowania, jak instancjonowanie kwantyfikatorów.

Instancjonowanie kwantyfikatorów z użyciem E-matchingu. Z3 wykorzystuje zaawansowaną technikę do rozumowania kwantyfikatorów, która opiera się na E-grafie. Dzięki nowym algorytmom, które identyfikują dopasowania w E-grafach w sposób skuteczny i przyrostowy, Z3 osiąga znaczną przewagę wydajności w porównaniu do innych nowoczesnych SMT solverów.

Theory Solvers. Z3 wykorzystuje liniowy solver arytmetyczny oparty na algorytmie używanym w Yices. Teoria tablic stosuje leniwe instancjonowanie aksjomatów tablicowych. Teoria wektorów bitowych o stałym rozmiarze stosuje bitowanie do wszystkich operacji na wektorach bitowych, z wyjątkiem równości.

Generowanie modeli. Z3 pozwala na generowanie modeli jako części danych wyjściowych. Modele przypisują wartości do stałych na wejściu i generują częściowe grafy dla predykatów oraz symboli funkcji [9].

3.2. Yices

Yices to SMT solver, który rozwiązuje formuły zawierające niezinterpretowane symbole funkcji z równością, arytmetykę rzeczywistą i całkowitą, wektory bitowe, typy skalarne i krotki. Yices 2 obsługuje zarówno arytmetykę liniową, jak i nieliniową. Został opracowany w Laboratorium Informatyki SRI International przez Bruno Dutertre, Dejana Jovanovica, Stéphane Graham-Lengrand i Iana A. Masona. Yices 2 może przetwarzać dane wejściowe zapisane w notacji SMT-LIB (obsługiwane są zarówno wersje 2.0, jak i 1.2). Alternatywnie można pisać specyfikacje przy użyciu własnego języka specyfikacji Yices 2, który obejmuje krotki i typy skalarne. Yices jest oprogramowaniem open source rozpowszechnianym na licencji GPLv3. Kod źródłowy Yices jest dostępny na GitHub [4].

3.2.1. Architektura systemu

Yices 2 posiada architekturę modularną. Pozwala ona wybrać konkretną kombinację solverów teorii dla swoich potrzeb za pomocą interfejsu API lub pliku wykonywalnego yices. Dzięki API można utrzymywać kilka niezależnych kontekstów równolegle, z których każdy może wykorzystywać różne solvery i ustawienia.

Struktura oprogramowania Yices 2 jest podzielona na trzy główne moduły:

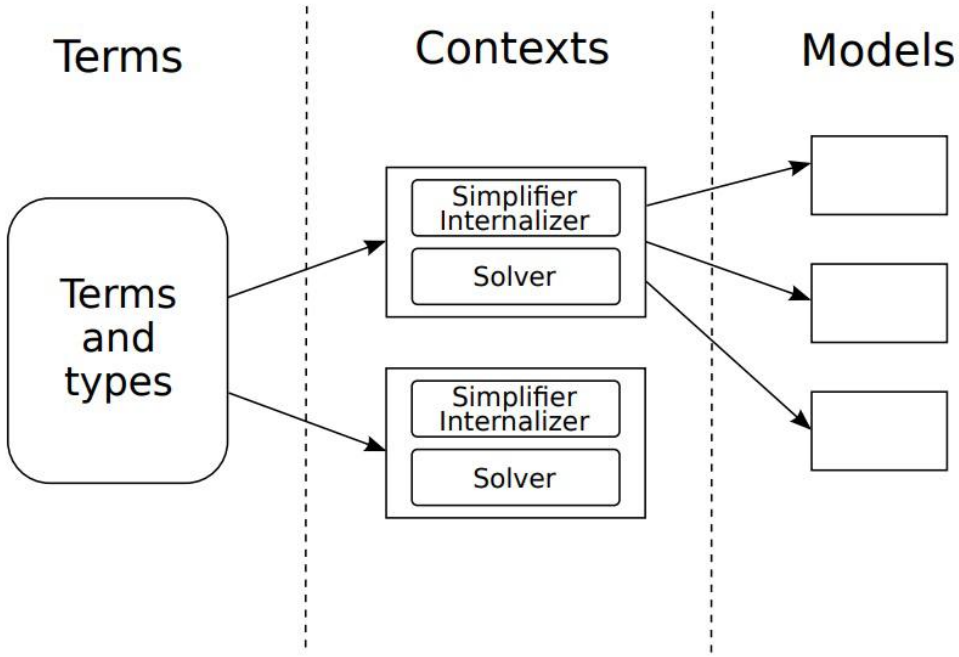
Baza termów Yices 2 utrzymuje globalną bazę danych, w której przechowywane są wszystkie termy i typy. Yices 2 zapewnia interfejs API do konstruowania termów, formuł i typów w tej bazie danych.

Zarządzanie kontekstem Kontekst jest centralną strukturą danych, która przechowuje formuły twierdzeń. Każdy kontekst zawiera zestaw twierdzeń, które mają być sprawdzane pod kątem spełnialności. Interfejs zarządzania kontekstami obsługuje operacje tworzenia i inicjowania kontekstów, przypisywania formuł do kontekstu oraz sprawdzania ich spełnial-

ności. Opcjonalnie kontekst może obsługiwać operacje wycofania twierdzeń przy użyciu mechanizmu push/pop. Kilka kontekstów może być konstruowanych i manipulowanych niezależnie.

Konteksty są wysoce konfigurowalne. Każdy kontekst może być skonfigurowany do obsługi określonej teorii i do korzystania z określonego solvera lub kombinacji solverów.

Zarządzanie modelami Jeśli zbiór formuł zadeklarowanych w kontekście jest spełnialny, to można skonstruować model formuł. Model odwzorowuje symbole formuł na konkretne wartości (np. liczby całkowite, wymierne lub stałe wektorowe). Interfejs API udostępnia funkcje do tworzenia i odpytywania modeli.



Rysunek 3.2: Architektura Yices. Źródło: `https://link.springer.com/chapter/10.1007/978-3-319-08867-9_49`

Rysunek 3.2 przedstawia najwyższy poziom architektury Yices 2, podzielony na trzy główne moduły. Każdy kontekst składa się z dwóch oddzielnych komponentów: Solver wykorzystuje Boolean satisfiability solver i procedury decyzyjne do określania, czy formuły zawarte w kontekście są spełnialne. Upraszczacz/internalizator konwertuje format używany przez bazę danych termów do wewnętrznego formatu używanego przez solver. W szczególności internalizator przepisuje wszystkie formuły do koniunkcyjnej postaci normalnej, z czego korzysta wewnętrzny SAT solver [4].

3.3. cvc5

Narzędzia z kategorii CVC (cooperating validity checker) odgrywają ważną rolę zarówno w badaniach, jak i w praktyce. Najnowsze wcielenie, CVC4, było przepisaną od podstaw wersją CVC3, napisaną w celu stworzenia elastycznej i wydajnej architektury, która mogłaby przetrwać w przyszłości. cvc5, kolejny solver z tej serii, nie jest przepisaniem CVC4 lecz bazuje na jego sprawdzonej architekturze i bazie kodu. W porównaniu do innych solverów SMT, cvc5 zapewnia zróżnicowany zestaw teorii (wszystkie standardowe teorie SMT-LIB i wiele niestandardowych) oraz funkcjonalności poza SMT, takie jak rozumowanie wyższego rzędu i synteza sterowana składnią (SyGuS). Zmiana nazwy raczej odzwierciedla zarówno nowy zespół programistów, jak również znaczącą ewolucję, jaką narzędzie przeszło od czasu opisania CVC4 w 2011 roku. Ponadto, cvc5 zawiera aktualizowaną dokumentację, nowe i ulepszone interfejsy API oraz bardziej przyjazną dla użytkownika instalację. Co najważniejsze, wprowadzono kilka istotnych nowych funkcji. Podobnie jak jego poprzednicy, cvc5 jest dostępny na 3-klauzulowej licencji BSD open source i działa na wszystkich głównych platformach (Linux, macOS, Windows).

cvc5 to wspólny projekt prowadzony przez Uniwersytet Stanforda i Uniwersytet Iowa [1].

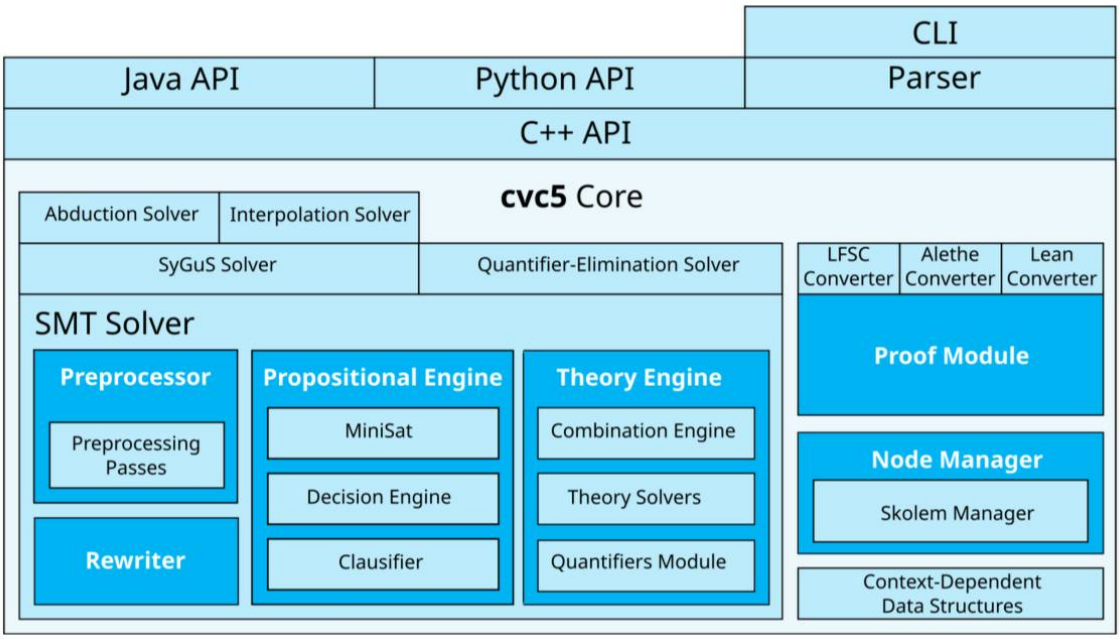
3.3.1. Architektura systemu

cvc5 obsługuje rozumowanie w zakresie formuł bezkwantyfikatorowych i kwantyfikatorowych w szerokim zakresie teorii, w tym wszystkich teorii standaryzowanych w SMT-LIB. Ponadto wspiera kilka niestandardowych teorii i rozszerzeń. Należą do nich, między innymi, logika separacji, teoria sekwencji, teoria zbiorów skończonych i relacji oraz rozszerzenie teorii liczb rzeczywistych o funkcje transcendentalne.

Ogólny przegląd architektury systemu przedstawiono na rysunku 3.3.

Centralnym elementem cvc5 jest moduł SMT Solver, oparty na frameworku CDCL(T) [12] i dostosowanej wersji solvera propozycyjnego MiniSat. Moduł ten składa się z kilku komponentów: modułów Rewriter i Preprocessor, które stosują uproszczenia odpowiednio lokalnie (na poziomie wyrażenia) i globalnie (na całej formule wejściowej); Propositional Engine, który służy jako menedżer dla CDCL(T) SAT solvera; oraz Theory Engine, który zarządza kombinacją teorii i wszystkimi procedurami rozumowania kwantyfikatorowego.

Oprócz standardowego sprawdzania spełnialności, cvc5 zapewnia dodatkową funkcjonalność, taką jak abdukcja, interpolacja, synteza kierowana składnią (SyGuS) oraz eliminacja kwantyfikatorów. Każda z tych funkcji jest zaimplementowana jako dodatkowy solver zbudowany na bazie modułu SMT Solver. SyGuS jest głównym punktem wejścia dla zapytań o syntezę, które kodują problemy SyGuS jako problemy spełnialności (wyższego rzędu) z ograniczeniami semantycznymi i syntaktycznymi. Quantifer Elimination Solver wykonuje eliminację kwantyfikatorów w oparciu o śledzenie instancji kwantyfikatorów przez SMT So-



Rysunek 3.3: Architektura cvc5. Źródło: `https://link.springer.com/chapter/10.1007/978-3-030-99524-9_24`

lver. Abduction Solver i Interpolation Solver są oparte na SyGuS, a zatem są zbudowane jako warstwy nad modułem SyGuS Solver [1].

cvc5 zapewnia API C++ jako główny interfejs, nie tylko dla zewnętrznego oprogramowania klienta, ale także dla własnego parsera i dodatkowych wiązań języka w Javie i Pythonie. cvc5 dostarcza również tekstowy interfejs wiersza poleceń (CLI), zbudowany na parserze, który obsługuje języki wejściowe SMT-LIBv2, SyGuS2 i TPTP. Moduł Proof może wyprowadzać formalne dowody niespełnialności w trzech formatach: Lean 4 [10], Alethe [16] i LFSC [18].

Moduł SMT Solver

Moduł ten odpowiada za obsługę wszystkich zapytań SMT. Jego funkcjonalność obejmuje, oprócz sprawdzania spełnialności, konstruowanie modeli dla formuł spełnialnych oraz wyodrębnianie założeń, rdzeni i obiektów dowodu dla formuł niespełnialnych. Główne składniki modułu są opisane poniżej.

Preprocessor Przed jakąkolwiek weryfikacją spełnialności cvc5 stosuje do każdej formuły z problemu wejściowego sekwencję transformacji zachowujących spełnialność Te transformacje obejmują normalizacje, uproszczenia i przekształcenia formuł między różnymi logikami. Preprocesor wykonuje 34 różne transformacje, które mogą być dostosowywane poprzez opcje konfiguracyjne.

Silnik Propozycyjny Jest to podstawowy silnik CDCL(T) [12], który przekształca abstrakcje logiczne formuł wejściowych w postać CNF i korzysta z MiniSat jako podstawowego solvera

SAT. Silnik ten używa także mechanizmów decyzyjnych do dostosowywania heurystyk wyboru decyzji.

Przepisywacz Ten submoduł jest odpowiedzialny za przekształcanie termów według reguł przepisywania na semantycznie równoważne postacie normalne. W przeciwieństwie do preprocessora działa na termach w trakcie rozwiązywania problemu i jest wykorzystywany przez inne komponenty cvc5 [13].

Silnik Teorii Jest to główny składnik odpowiedzialny za sprawdzanie spójności teorii literałów, zatwierdzonych przez Silnik Propozycyjny. Koordynuje on działanie różnych solverów teorii, które obsługują różnorodne dziedziny matematyczne, takie jak arytmetyka liniowa, nieliniowa, tablice, bit-wektory, typy danych, arytmetyka zmiennoprzecinkowa, zbiory i relacje, logika separacji, stringi i sekwencje oraz funkcje niezinterpretowane.

Solvery Teorii cvc5 obsługuje szeroki zakres teorii, w tym arytmetykę liniową, nieliniową, tablice, bit-wektory, typy danych, arytmetykę zmiennoprzecinkową, zbiory i relacje, logikę separacji, łańcuchy znaków oraz funkcje niezinterpretowane. Ponadto wszystkie teorie przekazują kroki rozumowania do reszty systemu za pośrednictwem Menedżera wnioskowania teorii. Każdy z tych solverów emituje lematy, klauzule konfliktowe i propagowane literały za pośrednictwem tego interfejsu [1].

Proof Module

Moduł Dowodów w cvc5 został zbudowany od podstaw, zastępując system dowodowy CVC4 [6], który był niekompletny i cierpiał z powodu licznych wad architektonicznych. Projektowanie modułu Dowodów cvc5 kierowało się kilkoma zasadami: minimalizacją narzutu czasowego generowania dowodów, zapewnieniem wystarczająco szczegółowych dowodów umożliwiających efektywne sprawdzanie, elastycznością w emitowaniu dowodów w różnych formatach oraz minimalizacją wyłączania komponentów systemu w trybie produkcji. Moduł Dowodów w cvc5 generuje szczegółowe dowody dla prawie wszystkich teorii, reguł przepisywania, przejść wstępnych, wewnętrznych solverów SAT oraz silników kombinacyjnych teorii. Dodatkowo obsługuje on produkowanie dowodów zarówno natychmiastowe, jak i leniwe, z wbudowaną rekonstrukcją dowodów [15]. cvc5 jest w stanie emitować dowody w różnych formatach, w tym tych obsługiwanych przez tester dowodów LFSC [18], oraz asystenty dowodowe Isabelle/HOL [24], Lean 4 [10] i Coq [21].

Node Manager

W cvc5 formuły i termy są reprezentowane jednolicie jako wierzchołki w grafie skierowanym acyklicznym, zarządzane przez Menedżera Węzłów. Menedżer Węzłów zarządza również Menedżerem Skolema, odpowiedzialnym za śledzenie symboli Skolema wprowadzanych podczas rozwiązywania. Węzły są niezmienne i są współdzielone za pomocą techniki hash consing, co zapewnia efektywne zarządzanie pamięcią i sprawdzanie równości skła-

dniowej w czasie stałym. Menedżer Skolema centralnie generuje stałe Skolema, co pozwala na deterministyczne generowanie nowych stałych podczas rozwiązywania i minimalizację ich liczby dla wydajności [14].

Context-Dependent Data Structures

Aby wspierać aplikacje SMT wymagające wielokrotnych sprawdzeń spełnialności podobnych twierdzeń, cvc5 definiuje pojęcie poziomu kontekstu i wdraża kontekstowo zależne struktury danych. Te struktury zachowują się podobnie do odpowiadających im mutowalnych struktur danych w standardowej bibliotece C++, ale są związane z poziomem kontekstu i automatycznie zapisują oraz przywracają swój stan wraz ze zmianą kontekstu [1].

3.4. Analiza porównawcza SMT solverów

Porównując Z3, Yices i cvc5, należy zwrócić uwagę na to, że każdy z tych solverów ma swoje unikalne cechy, które wpływają na sposób ich użytkowania oraz efektywność rozwiązywania problemów.

Z3 wyróżnia się wydajną architekturą opartą na wewnętrznym mechanizmie tzw. tłumaczenia zadań SMT na zadania SAT (ang. Satisfiability Modulo Theories to Boolean Satisfiability). Dzięki wsparciu dla szerokiego zakresu teorii i bogatej dokumentacji, Z3 jest często wybierany w projektach związanych z analizą i weryfikacją oprogramowania, a także w badaniach naukowych z obszaru informatyki teoretycznej i sztucznej inteligencji. Na przykład, firmy z branży technologicznej, takie jak Google i Facebook, korzystają z Z3 do weryfikacji oprogramowania oraz do analizy złożoności algorytmów.

Yices, z kolei, cechuje się prostotą użytkowania oraz wysoką efektywnością w rozwiązywaniu problemów SAT i SMT. Jego architektura koncentruje się na szybkim i niezawodnym rozwiązywaniu problemów, co czyni go popularnym wyborem w projektach związanych z weryfikacją układów cyfrowych oraz w automatycznym generowaniu testów. Przykładowo, firmy z sektora technologii mobilnych, takie jak Samsung i Apple, wykorzystują Yices do weryfikacji poprawności działania swoich urządzeń oraz do optymalizacji procesów projektowania.

Natomiast cvc5, będący inicjatywą naukową, oferuje elastyczną i modułową architekturę, która umożliwia integrację różnorodnych technik, takich jak SyGuS oraz zaawansowane mechanizmy dowodzenia. Jego silnik dedykowany do analizy teorii sprawia, że jest często wybierany w projektach badawczych z zakresu informatyki teoretycznej oraz w badaniach nad metodami rozwiązywania problemów SAT i SMT. Na przykład, instytucje naukowe oraz centra badawcze, takie jak MIT i Stanford University, wykorzystują cvc5 do analizy złożoności obliczeniowej algorytmów oraz do badania właściwości języków programowania.

Te różnice w architekturze i funkcjonalnościach solverów SMT mają istotne implikacje

dla ich wykorzystania w praktyce. Przyjrzenie się konkretnym zastosowaniom tych solverów może dostarczyć cennych wskazówek przy wyborze odpowiedniego narzędzia.

ROZDZIAŁ 4

Kodowanie problemów

Niniejszy rozdział wprowadza czytelnika do ośmiu wybranych problemów w teorii grafów i optymalizacji kombinatorycznej, takich jak:

1. Ścieżka Hamiltona w grafie skierowanym (ang. `Hamiltonian Path`),
2. Ścieżka Hamiltona w grafie nieskierowanym (ang. `Undirected Hamiltonian Path`),
3. Maksymalna klika (ang. `Maximum Clique Size Problem`),
4. Maksymalny zbiór niezależny (ang. `Maximum Independent Set`),
5. Minimalne pokrycie wierzchołkowe (ang. `Minimal Vertex Cover`),
6. Kolorowanie grafu (ang. `Graph Coloring`),
7. Problem Komiwojażera (ang. `Traveling Salesperson Problem`), oraz
8. Problem sumy podzbioru (ang. `Subset Sum`).

Każdy problem, który analizuję w mojej pracy, jest kodowany jako zestaw ograniczeń logicznych, które definiują jego specyfikację i warunki jego rozwiązania. Wykorzystanie tego podejścia pozwala mi na elastyczne definiowanie problemów, wyrażanie ich w sposób zrozumiały dla maszyny, a także szybkie eksperymentowanie z różnymi zestawami ograniczeń oraz łatwe wprowadzanie zmian w modelach problemów.

Każda sekcja zawiera opis algorytmu oraz kod demonstracyjny w języku Python z wykorzystaniem biblioteki Z3 do rozwiązania tych problemów.

W celu optymalizacji procesu programowania oraz podniesienia czytelności kodu, przyjęto praktykę tworzenia katalogu 'utils', który pełni rolę biblioteki narzędziowej. Katalog ten zawiera pliki z zestawami użytecznych funkcji (ang. `utility functions`), które powtarzają się w kodowaniu różnych problemów, umożliwiając ich wielokrotne wykorzystanie i uniknięcie redundancji. Centralizacja tych funkcji w jednym miejscu również ułatwia zrozumienie działania programów dla osób przeglądających kod.

Pierwszy plik z katalogu 'utils' ma nazwę `'read_input.py'`, i zawiera funkcje służące do odczytu danych z plików tekstowych i tworzenia odpowiednich struktur danych na ich podstawie:

Funkcja *read_graph_from_file*(filename) odczytuje graf nieskierowany z pliku o nazwie `filename`. Każda linia pliku reprezentuje krawędź grafu, gdzie liczby oddzielone spacją to numery wierzchołków połączonych krawędzią. Funkcja tworzy i zwraca słownik, gdzie klucze to numery wierzchołków, a wartości to listy sąsiedztwa.

Listing 4.1: Czytanie grafu z pliku

```
def read_graph_from_file(filename):
    with open(filename, 'r') as file:
        lines = file.readlines()

    graph = {}
    for line in lines:
        source, target = [int(x) for x in line.strip().split
           ()]
        if source not in graph:
            graph[source] = []
        if target not in graph:
            graph[target] = []
        graph[source].append(target)
        graph[target].append(source)

    return graph
```

Funkcja *read_digraph_from_file*(filename) odczytuje graf skierowany z pliku o nazwie `filename`. Działa analogicznie do poprzedniej funkcji, ale tworzy graf skierowany.

Listing 4.2: Czytanie skierowanego grafu z pliku

```
def read_digraph_from_file(filename):
    with open(filename, 'r') as file:
        lines = file.readlines()

    digraph = {}
    for line in lines:
        source, target = [int(x) for x in line.strip().split
           ()]
        if source not in digraph:
            digraph[source] = []
        if target not in digraph:
            digraph[target] = []
        digraph[source].append(target)

    return digraph
```

Funkcja *read_wgraph_from_file*(filename) odczytuje graf nieskierowany z wagami kra-

wędzi z pliku o nazwie `filename`. Każda linia pliku reprezentuje krawędź grafu, gdzie trzy liczby oddzielone spacją to numery wierzchołków połączonych krawędzią oraz waga tej krawędzi. Funkcja tworzy i zwraca słownik, gdzie klucze to numery wierzchołków, a wartości to listy krotek (target, weight) reprezentujących numery wierzchołków i ich wagi.

Listing 4.3: Czytanie grafu z wagami

```
def read_wgraph_from_file(filename):
    with open(filename, 'r') as file:
    lines = file.readlines()

    wgraph = {}
    for line in lines:
        source, target, weight = [int(x) for x in line.strip
           ().split()]
        if source not in wgraph:
            wgraph[source] = []
        if target not in wgraph:
            wgraph[target] = []
        wgraph[source].append((target, weight))
        wgraph[target].append((source, weight))

    return wgraph
```

Funkcja *read_set_from_file*(filename) odczytuje zbiór liczb całkowitych z pliku o nazwie `filename`. Każda liczba w pliku powinna być oddzielona spacją. Funkcja zwraca listę zawierającą te liczby jako elementy.

Listing 4.4: Czytanie zbioru z pliku

```
def read_set_from_file(filename):
    with open(filename, 'r') as file:
        data = file.read().strip()
        input_set = list(map(int, data.split()))
    return input_set
```

W kolejnym pliku `'constraints.py'` umieszczono funkcje generujące ograniczenia logiczne, które są wielokrotnie stosowane w modelowaniu problemów.

Pierwszą z nich jest funkcja *proper_numbers*(*vertices*), która ma na celu zapewnienie, że wartości przypisane do zmiennych 'vertices' ($v_0, v_1, ..., v_{n-1}$), reprezentujących wierzchołki grafu, mieszczą się w zakresie od 0 do $n-1$, gdzie n oznacza rozmiar zbioru zmiennych.

Listing 4.5: Przypisanie właściwych wartości

```
def proper_numbers(vertices):
    n = len(vertices)
    atoms = []
    for i in range(n):
        atoms.append(z3.And(vertices[i] >= 0, vertices[i] < n
           ))
    bf = z3.And(atoms)
    return bf
```

Kolejną funkcją jest *distinct_vs*$(vertices)$, która zapewnia, że każdy wierzchołek grafu jest odwiedziany dokładnie raz, czyli żadne dwie zmienne v nie mają tej samej wartości:

Listing 4.6: Unikalne wartości

```
def distinct_vs(vertices):
    n = len(vertices)
    atoms = []
    for i in range(n - 1):
        for j in range(i + 1, n):
            atoms.append(vertices[i] != vertices[j])
    bf = z3.And(atoms)
    return bf
```

Funkcja *dir_edge*$(graph, s, t)$ generuje ograniczenia logiczne dla skierowanych krawędzi między wierzchołkami na podstawie podanego grafu. Niech $v_0, v_1, ..., v_{n-1}$ będą zmiennymi reprezentującymi wierzchołki grafu, a E zbiorem wszystkich krawędzi, w przypadku grafu skierowanego zbiór uporządkowanych par (s, t), gdzie s oznacza punkt źródłowy krawędzi, a t jest punktem końcowym:

Listing 4.7: Krawędź skierowana

```
def dir_edge(graph: dict[int, list[int]], s, t):
    atoms = []
    for source in graph:
        for target in graph[source]:
            atoms.append(z3.And([s == source, t == target]))
    bf = z3.Or(atoms)
    return bf
```

Natomiast funkcja *edge*$(graph, s, t)$ generuje ograniczenia dla nieskierowanych krawę-

dzi. Dla zmiennych $v_0, v_1, ..., v_{n-1}$ reprezentujących wierzchołki grafu, i zbioru wszystkich krawędzi E (w przypadku grafu nieskierowanego jest to zbiór nieuporządkowanych par $\{s, t\}$) funkcja ta przedstawia się następująco:

Listing 4.8: Krawędź nieskierowana

```
def edge(graph: dict[int, list[int]], s, t):
    atoms = []
    for source in graph:
        for target in graph[source]:
            atoms.append(z3.And([s == source, t == target]))
            atoms.append(z3.And([s == target, t == source]))
    bf = z3.Or(atoms)
    return bf
```

Funkcja *wedge*$(graph, s, t, w)$ generuje ograniczenie dla etykietowanych nieskierowanych krawędzi pomiędzy dwoma wierzchołkami s i t w grafie, przy użyciu zmiennej w reprezentującej wagę krawędzi:

Listing 4.9: Krawędź etykietowana

```
def wedge(graph: dict[int, list[int]], s, t, w):
    atoms = []
    for source in graph:
        for target, weight in graph[source]:
            atoms.append(z3.And([s == source, t == target, w
               == weight]))
            atoms.append(z3.And([s == target, t == source, w
               == weight]))
    bf = z3.Or(atoms)
    return bf
```

4.1. Problem ścieżki Hamiltona w grafie skierowanym

Problem ścieżki Hamiltona jest zagadnieniem z zakresu teorii złożoności i teorii grafów. Rozstrzyga on, czy graf zawiera ścieżkę Hamiltona, czyli ścieżkę, która odwiedza każdy wierzchołek grafu dokładnie raz.

Nazwa pochodzi od nazwiska W. R. Hamiltona, który opisał grę matematyczną na dwunastościanie, w której jeden z graczy wbija pięć szpilek w dowolnych pięciu kolejnych wierzchołkach, a drugi gracz musi ukończyć ścieżkę, aby utworzyć cykl zawierający wszystkie wierzchołki.

Problem ścieżki Hamiltona ma zastosowanie w wielu dziedzinach, takich jak informatyka, telekomunikacja czy bioinformatyka. Rozwiązanie tego problemu jest często kluczowe w analizie sieci i trasowania w systemach komunikacyjnych [29].

Dla grafu skierowanego $G = (V, E)$, gdzie:

- V jest zbiorem wierzchołków.
- E jest zbiorem skierowanych krawędzi.

Ścieżka Hamiltona jest sekwencją wierzchołków $s_0, s_1, ..., s_{n-1}$ taką, że:

1. Każdy wierzchołek z V występuje dokładnie raz.
2. Dla każdej pary kolejnych wierzchołków s_i i s_{i+1} w sekwencji istnieje skierowana krawędź od s_i do s_{i+1} w E.

Dodatkowo, ścieżka Hamiltona może zaczynać się z dowolnego wierzchołka i kończyć w dowolnym innym wierzchołku.

Aby zakodować problem decyzyjny istnienia ścieżki Hamiltona w grafie skierowanym użyjemy n zmiennych $v_0, \ldots, v_{n-1}$, gdzie n to liczba wierzchołków w grafie. Poniższe podformuły służą do zapewnienia odpowiednich warunków dla zmiennych używanych w rozwiązaniu problemu.

Przedstawione oznaczenia używane w formułach oznaczają:

- n - rozmiar zbioru zmiennych,
- v_j - zmienna oznaczająca wierzchołek grafu,
- E - zbiór krawędzi w grafie,
- s i t - końce krawędzi.

proper_numbers gwarantuje, że każda zmienna v_j znajduje się w zakresie od 0 do $n-1$, gdzie n jest rozmiarem zbioru zmiennych.

$$\textit{proper\_numbers}(n) = \left(\bigwedge_{j=0}^{n-1} (v_j \geq 0 \wedge v_j < n) \right)$$

distinct_vs zapewnia, że każda para zmiennych v_i i v_j jest różna od siebie, eliminując powtarzające się wartości między zmiennymi.

$$\textit{distinct\_vs}(n) = \left(\bigwedge_{i=0}^{n-1} \bigwedge_{j=i+1}^{n} (v_i \neq v_j) \right)$$

dir_edges sprawdza, czy istnieją krawędzie w grafie o zadanych wierzchołkach s i t, zapewniając, że dla każdej zmiennej v_i istnieje skierowana krawędź od niej do kolejnej zmiennej v_{i+1}.

$$\textit{dir\_edges}(n, E) = \left(\bigwedge_{i=0}^{n-1} \bigvee_{(s,t)\in E} (v_i = s \wedge v_{i+1} = t) \right)$$

Zatem cała formuła kodująca ścieżkę Hamiltona dla grafu skierowanego zawierającego n wierzchołków i zbiór krawędzi E jest następująca:

$$\textit{HamPath}(n, E) = \textit{proper\_numbers}(n) \wedge \textit{distinct\_vs}(n) \wedge \textit{dir\_edges}(n, E)$$

Kodowanie problemu ścieżki Hamiltona opiera się na generowaniu powyższych ograniczeń logicznych, które muszą zostać spełnione, aby znaleźć poszukiwaną ścieżkę Hamiltona. W tym celu w programie *hampath.py* stosują się trzy funkcje z katalogu 'utils':

proper_numbers(*vertices*), *distinct_vs*(*vertices*) oraz *dir_edge*(*graph*, *s*, *t*).

Funkcja *check_hampath*(*graph*) odpowiada za główny algorytm rozwiązujący problem.

W implementacji funkcji *check_hampath*(*graph*) wykorzystuje się strukturę danych reprezentującą graf za pomocą słownika, gdzie klucze odpowiadają wierzchołkom, a wartości są listami sąsiedztwa.

Inicjalizacja zmiennych: na początku funkcja wyznacza liczbę wierzchołków grafu n na podstawie jego rozmiaru; tworzy wektor zmiennych całkowitoliczbowych $vertices$ za pomocą metody $z3.IntVector$, gdzie każda zmienna odpowiada jednemu wierzchołkowi grafu.

Dodawanie ograniczeń: dodaje wyżej opisane ograniczenia do solvera Z3.

Rozwiązywanie problemu: solver Z3 sprawdza spełnialność wszystkich dodanych ograniczeń. Jeśli znajdzie rozwiązanie ($'z3.sat'$), oznacza to istnienie ścieżki Hamiltona w grafie. W przypadku braku rozwiązania, informuje o tym użytkownika.

Prezentacja wyników: jeśli solver znalazł rozwiązanie, funkcja wyświetla znalezioną ścieżkę Hamiltona, prezentując kolejność odwiedzania wierzchołków. W przeciwnym przypadku wyświetla wynik $'unsat'$.

Listing 4.10: Funkcja *check_hampath*(*graph*)

```
1 def check_hampath(graph: dict[int, list[int]]):
2
3     n = len(graph)
4     vertices = z3.IntVector("v", n)
5
```

```
    solver = z3.Solver()

    solver.add(proper_numbers(vertices))

    solver.add(distinct_vs(vertices))

    edges = []
    for i in range(n - 1):
        edges.append(dir_edge(graph, vertices[i], vertices[i
           + 1]))
    solver.add(z3.And(edges))

    smt2_representation = solver.to_smt2()
    file_name = f'hampath_state.smt2'
    with open(file_name, 'w') as file:
        file.write("(set-logic ALL)\n")
        file.write(smt2_representation)
    file.close()

    result = solver.check()
    if result == z3.sat:
        model = solver.model()
        vertex_values = [(idx, model[v].as_long()) for idx, v
            in enumerate(vertices)]
        sorted_verticess = sorted(vertex_values)
        print("Hamiltonian Path:")
        for idx, value in sorted_verticess[:-1]:
            print(f"v__{value}", end=' -> ')
        print(f"v__{sorted_verticess[-1][1]}")
    else:
        print(result)
```

4.2. Problem ścieżki Hamiltona w grafie nieskierowanym

W przypadku grafów nieskierowanych krawędzie nie mają określonego kierunku, dlatego ścieżka Hamiltona może przechodzić przez krawędź w obie strony.

Dla grafu nieskierowanego $G = (V, E)$, gdzie:

- V jest zbiorem wierzchołków.

- E jest zbiorem krawędzi.

Ścieżka Hamiltona jest sekwencją wierzchołków $s_1, s_2, ..., s_n$ taką, że:

1. Każdy wierzchołek z V występuje dokładnie raz.
2. Dla każdej pary kolejnych wierzchołków s_i i s_{i+1} w sekwencji istnieje krawędź między s_i a s_{i+1} w E.

Dodatkowo, ścieżka Hamiltona może zaczynać się z dowolnego wierzchołka i kończyć w dowolnym innym wierzchołku.

W kontekście modelowania problemu ścieżki Hamiltona w grafie nieskierowanym należy zapewnić możliwość przechodzenia przez krawędź w obie strony, co można osiągnąć poprzez stworzenie formuły logicznej, która będzie uwzględniała możliwość istnienia krawędzi w E pomiędzy wierzchołkami v_i a v_{i+1} w obu kierunkach:

$$\mathit{edges}(n, E) = \left(\bigwedge_{i=0}^{n-1} \bigvee_{\{s,t\} \in E} (v_i = s \wedge v_{i+1} = t) \vee (v_i = t \wedge v_{i+1} = s) \right)$$

Zatem cała formuła kodująca ścieżkę Hamiltona dla grafu nieskierowanego zawierającego n wierzchołków i zbiór krawędzi E jest następująca:

$$\mathit{UHamPath}(n, E) = \mathit{proper\_numbers}(n) \wedge \mathit{distinct\_vs}(n) \wedge \mathit{edges}(n, E)$$

4.3. Problem maksymalnej kliki w grafie nieskierowanym

Klika w grafie nieskierowanym G to pełny podgraf, czyli taki, w którym każde dwa wierzchołki są połączone krawędzią. Problem kliki polega na sprawdzeniu, czy graf zwiera klikę o określonym rozmiarze. Maksymalna klika to taka, która zawiera największą możliwą liczbę wierzchołków [25].

Dla grafu nieskierowanego $G = (V, E)$, gdzie:

- V jest zbiorem wierzchołków.
- E jest zbiorem krawędzi.

Maksymalną kliką w grafie G jest podzbiór $C \subseteq V$ taki, że:

1. Dla każdej pary wierzchołków s i t w C istnieje krawędź między s a t w E, tj. $\{s, t\} \in E$.
2. C nie można rozszerzyć przez dodanie dowolnego wierzchołka $w \in V$ takiego, że $C \cup \{w\}$ tworzą klikę, tj. nie istnieje żaden wierzchołek w sąsiadujący z jakimkolwiek wierzchołkiem z C, który nie występuje już w C.

Aby zakodować problem decyzyjny istnienia kliki o rozmiarze k użyjemy k zmiennych $v_0, \ldots, v_{k-1}$, gdzie $v_0, \ldots, v_{k-1}$, gdzie $0 < k \leq n$.

Formuła logiczna kodująca problem poszukiwania maksymalnej kliki zapewnia, że każda zmienna v_j ma właściwą wartość i wszystkie zmienne v_j są różne od siebie (formuły *proper_numbers* oraz *distinct_vs* zostały już wcześniej omówione w opisie problemu ścieżki Hamiltona (Hamiltonian Path), dlatego też nie będę ich ponownie rozważać, zgodnie z zasadą unikania nadmiernego powtórzenia.) Druga część formuły zapewnia, że dla każdej pary wierzchołków v_i i v_j w zbiorze kandydatów na klikę o rozmiarze k istnieje krawędź między nimi. Ten warunek jest sprawdzany poprzez iterację przez wszystkie pary $\{s, t\}$ w zbiorze krawędzi E i sprawdzenie, czy v_i i v_j są końcami tej krawędzi.

$$\textit{MaxClique}(n, E, k) = \textit{proper\_numbers}(k) \land \textit{distinct\_vs}(k) \land \left(\bigwedge_{i=0}^{k-1} \bigwedge_{j=i+1}^{k} \bigvee_{\{s,t\} \in E} ((v_i = s \land v_j = t) \lor (v_j = s \land v_i = t)) \right)$$

Funkcja *check_maxclique*$(graph, k)$ w programie *maxclique.py* przyjmuje graf reprezentowany jako słownik oraz liczbę całkowitą k reprezentującą rozmiar kliki do sprawdzenia. Inicjalizuje solver Z3 i definiuje zmienne całkowite $vertices$ do reprezentowania wierzchołków grafu. Następnie dodaje ograniczenia, importowane z pliku `constraints.py`, aby każda zmienna reprezentowała inny wierzchołek grafu: *proper_numbers* i *distinct_vs*. Następnie generuje ograniczenia, aby zapewnić, że wybrane wierzchołki tworzą klikę o rozmiarze k, dodając krawędzie między wszystkimi parami wierzchołków w klice. Sprawdza spełnialność ograniczeń za pomocą solvera Z3. Jeśli znaleziono przypisanie spełniające, co oznacza istnienie kliki o rozmiarze k, wyświetla szczegóły kliki wraz z modelem uzyskanym od solvera. W przeciwnym razie informuje, że w grafie nie istnieje klika o danym rozmiarze. Na koniec zapisuje stan solvera do pliku w formacie SMT-LIB dla dalszych eksperymentów.

Listing 4.11: Funkcja *check_maxclique*$(graph, k)$

```
1  def check_clique(graph: dict[int, list[int]], k):
2      n = len(graph)
3
4      vertices = z3.IntVector("v", k)
5      solver = z3.Solver()
6
7      solver.add(proper_numbers(vertices))
8      solver.add(distinct_vs(vertices))
9
10     edges = []
11     for i in range(k):
```

```
12          for j in range(i + 1, k):
13              edges.append(edge(graph, vertices[i], vertices[j
                    ]))
14      solver.add(z3.And(edges))
15
16      result = solver.check()
17      if result == z3.sat:
18          print('Znaleziono klike o rozmiarze', k)
19          model = solver.model()
20          clique = [model[vertices[i]].as_long() for i in range
                (k)]
21          print(clique)
22      else:
23          print('Nie znaleziono kliki o rozmiarze', k)
24          model = None
25
26      smt2_representation = solver.to_smt2()
27      file_name = f'maxclique-{n}-{k}.smt2'
28      with open(file_name, 'w') as file:
29          file.write("(set-logic ALL)\n")
30          file.write(smt2_representation)
31      file.close()
32
33      return (result, model)
```

Funkcja *main*() koordynuje proces znajdowania maksymalnej kliki w grafie odczytanym z pliku. Upewnia się, że podane są poprawne argumenty wiersza poleceń, a następnie przystępuje do odczytu grafu z określonego pliku. Iteruje po możliwych rozmiarach kliki od 2 do liczby wierzchołków w grafie. Dla każdego rozmiaru kliki k wywołuje funkcję *check_maxclique*(*graph*, *k*), aby sprawdzić, czy istnieje klika o rozmiarze k w grafie. Jeśli taka klika zostanie znaleziona, funkcja wyświetla jej szczegóły wraz z modelem uzyskanym od solvera.

Ważnym aspektem funkcji *main*() jest cykl for, który iteruje po możliwych rozmiarach kliki. Wewnątrz tego cyklu, po sprawdzeniu, czy istnieje klika o aktualnym rozmiarze k, następuje sprawdzenie wyniku $result$. Jeśli wynik jest inny niż 'sat' (co oznacza, że nie znaleziono spełnienia ograniczeń), funkcja przerywa dalsze iteracje, ponieważ nie ma potrzeby kontynuowania poszukiwań. W momencie, gdy nie można znaleźć kliki o określonym rozmiarze, dalsze poszukiwania dla większych rozmiarów kliki są bezcelowe, ponieważ każda klika musiałaby zawierać wierzchołki kliki mniejszych. Dlatego przerwanie cyklu w momencie, gdy

wynik jest inny niż 'sat', optymalizuje działanie funkcji *main*() i przyspiesza proces znajdowania maksymalnej kliki w grafie.

Listing 4.12: Funkcja *main*() w *maxclique.py*

```
def main():
    if len(sys.argv) != 2:
        print("Usage: python3 maxclique.py <filename>")
        return

    filename = sys.argv[1]
    graph = read_graph_from_file(filename)
    n = len(graph)

    for k in range(2, n + 1):
        result, model = check_clique(graph, k)
        if result != z3.sat:
            break
```

4.4. Problem maksymalnego zbioru niezależnego w grafie nieskierowanym

Niezależny zbiór w grafie nieskierowanym jest podzbiorem wierzchołków grafu G, z których żadne dwa nie sąsiadują ze sobą. Problem maksymalnego zbioru niezależnego ma na celu obliczenie rozmiaru największego niezależnego zbioru w danym grafie. Rozmiar niezależnego zbioru to liczba zawartych w nim wierzchołków [8].

Problem maksymalnego niezależnego zbioru i jego dopełnienie, problem minimalnego pokrycia wierzchołkowego, są zaangażowane w dowodzenie złożoności obliczeniowej wielu problemów teoretycznych [26].

Dla grafu $G = (V, E)$, niezależny zbiór S jest maksymalnym jeżeli dla $s \in V$ jedno z poniższych twierdzeń jest prawdziwe [28]:

- $s \in S$;
- $N(s) \cap S \neq \emptyset$, gdzie $N(s)$ oznacza sąsiadów s.

Aby zakodować problem decyzyjny maksymalnego niezależnego zbioru o rozmiarze k użyjemy n zmiennych $v_0, \ldots, v_{n-1}$, gdzie n to liczba wierzchołków w grafie.

Formuła logiczna kodująca problem maksymalnego zbioru niezależnego składa się z trzech podformuł. Pierwsza i druga zapewniają, że każda zmienna ma właściwą wartość, a wszystkie

zmienne są różne od siebie odpowiednio. Trzecia podformuła zapewnia, że dla każdej pary zmiennych v_i i v_j w zbiorze kandydatów na maksymalny zbiór niezależny o rozmiarze k, nie istnieje krawędź między nimi w E.

$$\textit{MaxIndSet}(n, E, k) = \textit{proper\_numbers}(n) \land \textit{distinct\_vs}(n) \land \left(\bigwedge_{i=0}^{k-1} \bigwedge_{j=i+1}^{k} \bigvee_{\{s,t\} \in E} \neg((v_i = s \land v_j = t) \lor (v_j = s \land v_i = t)) \right)$$

Funkcja *check_maxindset*$(graph, k)$ w programie *maxindset.py* przyjmuje graf oraz rozmiar potencjalnego maksymalnego zbioru niezależnego. Tworzy wektor zmiennych całkowitoliczbowych $vertices$, dodaje odpowiednie ograniczenia (wszystkie wierzchołki muszą być w zakresie od 0 do n i nie mogą się powtarzać), a następnie definiuje warunki, które wykluczają istnienie krawędzi między wierzchołkami w potencjalnym zbiorze niezależnym. Po dodaniu ograniczeń solver próbuje znaleźć rozwiązanie. Po sprawdzeniu wyniku działania solvera, jeśli udało się znaleźć maksymalny zbiór niezależny, program wypisuje ten zbiór na ekranie. W przeciwnym przypadku kończy działanie, nie wypisując żadnego wyniku. Program zapisuje stan rozwiązania w pliku o formacie .smt2.

Listing 4.13: Funkcja *check_maxindset*$(graph, k)$

```
1  def check_maxindset(graph: dict[int, list[int]], k):
2      n = len(graph)
3
4      vertices = z3.IntVector("v", n)
5      solver = z3.Solver()
6
7      solver.add(proper_numbers(vertices))
8      solver.add(distinct_vs(vertices))
9
10     no_edges = []
11     for i in range(k):
12         for j in range(i + 1, k):
13                 no_edges.append(z3.Not(edge(graph, vertices[i
                       ], vertices[j])))
14     solver.add(z3.And(no_edges))
15
16     result = solver.check()
17     if result == z3.sat:
18         model = solver.model()
```

```
19          maxindset = [model[vertices[i]].as_long() for i in
                range(k)]
20          print(maxindset)
21      else:
22          print(result)
23          model = None
24
25      smt2_representation = solver.to_smt2()
26      file_name = f'maxindset_{n}_{k}.smt2'
27      with open(file_name, 'w') as file:
28          file.write("(set-logic ALL)\n")
29          file.write(smt2_representation)
30      file.close()
31
32      return result, model
```

W programie *maxindset.py* funkcja *main*() ma analogiczną rolę do funkcji *main*() z programu *maxclique.py*. Obydwie funkcje mają taką samą strukturę kontrolną i wykonują te same czynności, ale w odniesieniu do różnych problemów.

Listing 4.14: Funkcja *main*() w *maxindset.py*

```
1  def main():
2      if len(sys.argv) != 2:
3          print("Usage: python3 maxindset.py <filename>")
4          return
5
6      filename = sys.argv[1]
7      graph = read_graph_from_file(filename)
8      n = len(graph)
9
10     for k in range(2, n + 1):
11         result, model = check_maxindset(graph, k)
12         if result != z3.sat:
13             break
```

4.5. Problem pokrycia wierzchołkowego

Pokrycie wierzchołkowe grafu nieskierowanego $G = (V, E)$ to podzbiór $V' \subseteq V$ taki, że jeżeli $\{s, t\} \in E$, to $s \in V'$ lub $t \in V'$ (lub oba). To znaczy, że każdy wierzchołek "po-

krywa”przylegające krawędzie, a pokrycie wierzchołkowe dla G jest zbiorem wierzchołków, który pokrywa wszystkie krawędzie z E. Rozmiar pokrycia to liczba wierzchołków w nim zawartych.

Problem pokrycia wierzchołkowego polega na znalezieniu minimalnego pokrycia w danym grafie. Dla tego problemu optymalizacyjnego, odpowiadający mu problem decyzyjny zadaje pytanie czy graf ma pokrycie o danym rozmiarze k [23].

Aby zakodować problem decyzyjny czy graf ma pokrycie o rozmiarze k użyjemy k zmiennych $v_0, \ldots, v_{k-1}$, gdzie $0 < k \leq n$.

Formuła logiczna kodująca problem pokrycia wierzchołkowego zapewnia, że każda zmienna przyjmuje właściwą wartość oraz wszystkie zmienne są różne od siebie (*proper_numbers* oraz *distinct_vs*).

Druga część formuły gwarantuje, że dla każdej krawędzi w E co najmniej jeden z jej końców należy do V':

$$\textit{VertexCover}(n, E, k) = \textit{proper\_numbers}(k) \wedge \textit{distinct\_vs}(k) \wedge \left(\bigwedge_{\{s,t\} \in E} \left(\bigvee_{j=0}^{k-1} (v_j = s \vee v_j = t) \right) \right)$$

Funkcja *check_vertexcover*$(graph, k)$ w programie *vertexcover.py* tworzy solver Z3 i dodaje ograniczenia dotyczące właściwych numerów i różności zmiennych. Następnie dla każdej krawędzi grafu, tworzy formułę logiczną, która wymusza, że co najmniej jeden wierzchołek z pokrycia musi pokrywać daną krawędź.

Po znalezieniu pokrycia, program tworzy plik .smt2 z reprezentacją formuły i wynikami, a następnie wypisuje pokrycie wierzchołkowe, jeśli zostało znalezione. W przeciwnym razie wyświetla odpowiedni komunikat.

Listing 4.15: Funkcja *check_vertexcover*$(graph, k)$

```
1  def check_vertexcover(graph: dict[int, list[int]], k):
2      n = len(graph)
3      vertices = z3.IntVector('v', k)
4      solver = z3.Solver()
5
6      solver.add(proper_numbers(vertices))
7      solver.add(distinct_vs(vertices))
8
9      for source in graph:
10         for target in graph[source]:
11             vertex_in_cover = False
12             for j in range(k):
```

```
13                vertex_in_cover = z3.Or(vertex_in_cover, z3.
                      Or(vertices[j] == source, vertices[j] ==
                      target))
14            solver.add(vertex_in_cover)
15
16
17    smt2_representation = solver.to_smt2()
18    file_name = f'vertexcover_{n}_{k}.smt2'
19    with open(file_name, 'w') as file:
20        file.write("(set-logic ALL)\n")
21        file.write(smt2_representation)
22    file.close()
23
24    result = solver.check()
25    if result == z3.sat:
26        print('Znaleziono pokrycie o rozmiarze', k)
27        model = solver.model()
28        vertex_cover = [model[v].as_long() for v in vertices]
29        print(vertex_cover)
30    else:
31        print('Nie znaleziono pokrycia o rozmiarze', k)
32        model = None
```

Funkcja *main*() programu *vertexcover.py* jest odpowiedzialna za przetwarzanie danych wejściowych oraz wywołanie funkcji, która sprawdza pokrycie wierzchołkowe dla różnych rozmiarów k. Po ustaleniu liczby wierzchołków n w grafie, następuje pętla iterująca od największego możliwego rozmiaru pokrycia k do 1. W każdej iteracji pętli sprawdzane jest pokrycie wierzchołkowe dla danego rozmiaru k, wywołując funkcję *check_vertexcover*(*graph*, *k*). Jeśli wynik sprawdzenia nie jest sat (czyli nie znaleziono pokrycia), pętla zostaje przerwana, ponieważ dalsze sprawdzanie dla mniejszych rozmiarów pokrycia nie ma sensu.

Listing 4.16: Funkcja *main*() programu *vertexcover.py*

```
1 def main():
2     if len(sys.argv) != 2:
3         print("Usage: python3 vertexcover.py <filename>")
4         return
5
6     filename = sys.argv[1]
7     graph = read_graph_from_file(filename)
8     n = len(graph)
```

```
9
10      for k in range(n, 0, -1):
11          result, model = check_vertexcover(graph, k)
12          if result != z3.sat:
13              break
```

4.6. Problem kolorowania grafu

Projektanci map dążą do używania jak najmniejszej liczby kolorów podczas kolorowania krajów na mapie, z zastrzeżeniem, że jeśli dwa kraje mają wspólną granicę, muszą mieć różne kolory. Problem ten można zamodelować za pomocą grafu nieskierowanego $G = (V, E)$, w którym każdy wierzchołek reprezentuje kraj, i wierzchołki, których kraje mają wspólną granicę, ze sobą sąsiadują.

Wtedy k-kolorowanie jest funkcją $c\colon V \to \{1, 2, ..., k\}$ taką, że $c(u) \neq c(v)$ dla każdej krawędzi $\{u, v\} \in E$. Innymi słowy, liczby $1, 2, ..., k$ oznaczają k kolorów, zaś sąsiednie wierzchołki muszą mieć różne przypisane numery. Problem kolorowania grafu polega na określeniu minimalnej liczby kolorów potrzebnej do pokolorowania danego grafu.

Aby zakodować problem decyzyjny kolorowania grafu używając liczby kolorów k użyjemy n zmiennych $c_0, \ldots, c_{n-1}$, gdzie n to liczba wierzchołków w grafie.

Formuła logiczna kodująca problem kolorowania grafu zapewnia, że każda zmienna c_j reprezentująca kolor wierzchołka j przyjmuje wartość z przedziału od 1 do k, gdzie k to liczba kolorów. Ponadto, dla każdej krawędzi s, t w E, kolor wierzchołka s jest różny od koloru wierzchołka t:

$$\textit{GraphColoring}(n, E) = \left(\bigwedge_{j=1}^{n} (c_j \geq 1 \wedge c_j \leq k) \right) \wedge \left(\bigwedge_{\{s,t\} \in E} (c_s \neq c_t) \right)$$

Program *graph_coloring.py* służy do kolorowania grafu minimalną możliwą liczbą kolorów. Oto opis tego programu:

Importuje niezbędne moduły i funkcje, w tym bibliotekę `networkx` do manipulacji grafami oraz `matplotlib.pyplot` do rysowania grafów.

Definiuje funkcję *main*(), która jest punktem wejścia do programu. Sprawdza ona, czy została podana prawidłowa liczba argumentów wiersza poleceń i wczytuje nazwę pliku zawierającego graf. Następnie iteruje po różnych możliwych liczbach kolorów (od 1 do liczby wierzchołków w grafie), aby znaleźć poprawne pokolorowanie grafu. Jeśli tak, program kończy działanie i wyświetla znalezione pokolorowanie. Jeśli nie, iteruje dalej.

Ta strategia wyszukiwania do pierwszego wyniku sat jest zastosowana ze względu na fakt, że poszukujemy minimalnej liczby kolorów potrzebnych do pokolorowania grafu. Je-

śli znajdziemy rozwiązanie sat, oznacza to, że znaleźliśmy minimalną liczbę kolorów. Wtedy możemy przerwać dalsze iteracje i zakończyć program, ponieważ nie ma potrzeby kontynuowania poszukiwań.

Listing 4.17: Funkcja *main*() w *graph_coloring.py*

```
def main():
    if len(sys.argv) != 2:
        print("Usage: python3 graph_coloring.py <filename>")
        return

    filename = sys.argv[1]
    graph = read_graph_from_file(filename)
    n = len(graph)

    for k in range(1, n + 1):
        result, model = check_coloring(graph, k)
        if result != z3.unsat:
            break
```

Definiuje funkcję *check_coloring*(*graph*, *k*), która sprawdza, czy można pokolorować dany graf graph za pomocą maksymalnie k kolorów. W tym celu tworzy wektor zmiennych całkowitoliczbowych $vertex\_color$, reprezentujących kolor każdego wierzchołka grafu. Następnie tworzy instancję obiektu $Solver()$ z biblioteki Z3 i dodaje ograniczenia zapewniające, że każdy kolor ma wartość od 1 do k, oraz że żadne dwa sąsiednie wierzchołki nie mają tego samego koloru.

Tworzy również plik .smt2 zawierający reprezentację problemu w formacie SMT-LIB 2.0, który może być używany przez inne narzędzia do rozwiązywania problemów ograniczeń logicznych.

Następnie sprawdza, czy istnieje rozwiązanie spełniające wszystkie ograniczenia. Jeśli tak, otrzymuje model kolorowania grafu i rysuje go przy użyciu biblioteki $matplotlib$, pokazując wierzchołki w różnych kolorach zgodnie z modelem. Jeśli nie, wyświetla komunikat informujący o tym, że nie znaleziono poprawnego pokolorowania grafu.

Listing 4.18: Funkcja *check_coloring*(*graph*, *k*)

```
def check_coloring(graph, k):
    vertices = list(graph.keys())
    n = len(vertices)

    vertex_color = z3.IntVector('c', n)

    solver = z3.Solver()
```

```
8
9       for i in range(n):
10          solver.add(vertex_color[i] >= 1)
11          solver.add(vertex_color[i] <= k)
12
13      for i in range(n):
14          for neighbor in graph[vertices[i]]:
15              solver.add(vertex_color[i] != vertex_color[
                    vertices[neighbor]])
16
17      smt2_representation = solver.to_smt2()
18      file_name = f'graphcoloring_{n}_{k}.smt2'
19      with open(file_name, 'w') as file:
20          file.write("(set-logic ALL)\n")
21          file.write(smt2_representation)
22      file.close()
23
24      result = solver.check()
25
26      if result == z3.sat:
27          model = solver.model()
28          print(model)
29          draw_graph(graph, vertex_color, model)
30      else:
31          print(result)
32          model = None
33
34      return result, model
```

Funkcja `draw_graph` rysuje graf z pokolorowanymi wierzchołkami na podstawie modelu uzyskanego z rozwiązania problemu pokolorowania grafu. Najpierw tworzy obiekt grafu, dodając wierzchołki i krawędzie. Następnie przypisuje kolory wierzchołkom na podstawie modelu i rysuje graf z odpowiednimi kolorami. Ta funkcja pomaga w wizualizacji wyników pokolorowania grafu, ułatwiając ich zrozumienie.

Listing 4.19: Funkcja *draw_graph*

```
1 def draw_graph(graph, vertex_color, model):
2     G = nx.Graph()
3     for vertex, neighbors in graph.items():
4         G.add_node(vertex)
```

```
5           for neighbor in neighbors:
6               G.add_edge(vertex, neighbor)
7
8       colors = [model.evaluate(vertex_color[v]).as_long() for v
            in graph]
9
10      nx.draw(G, with_labels=True, node_color=colors, cmap = '
            tab10')
11      plt.show()
```

4.7. Problem Komiwojażera

Rozwiązanie Problemu Komiwojażera jest kluczowym zagadnieniem optymalizacyjnym, które ma szerokie zastosowanie w różnych dziedzinach, od logistyki po planowanie tras w sieciach komunikacyjnych. Jest to szczególny przypadek problemu cyklu Hamiltona, gdzie cykl ten musi minimalizować sumaryczny koszt podróży.

Formalnie, cykl hamiltonowski grafu nieskierowanego $G = (V, E)$ jest prostym cyklem, który zawiera każdy wierzchołek w V. Mówi się, że graf, zawierający cykl Hamiltona, jest hamiltonowskim, a w przeciwnym razie jest niehamiltonowskim.

W problemie komiwojażera, komiwojażer musi odwiedzić n miast. Zamodelujmy ten problem jako nieskierowany etykietowany graf z n wierzchołkami, tak aby komiwojażer wykonał trasę, czyli cykl hamiltonowski, odwiedzając każde miasto dokładnie raz i kończąc w mieście początkowym. Koszt podróży z miasta i do miasta j jest nieujemną liczbą całkowitą $c(i, j)$. W wersji optymalizacyjnej problemu, zadaniem jest wybrać trasę, której sumaryczny koszt, tzn. suma wag poszczególnych krawędzi, jest minimalny.

Dla pełnego grafu nieskierowanego $G = (V, E)$, gdzie:

- V jest zbiorem etykietowanych wierzchołków.
- E jest zbiorem krawędzi.

trasa o minimalnym koszcie jest sekwencją wierzchołków $s_1, s_2, ..., s_n$ taką, że:

1. Każdy wierzchołek z V występuje dokładnie raz.
2. Dla każdej pary kolejnych wierzchołków s_i i s_{i+1} w sekwencji istnieje krawędź między s_i a s_{i+1} w E.
3. Istnieje krawędź między ostatnim s_n a pierwszym s_1 wierzchołkiem w E.
4. Suma wag wszystkich krawędzi jest co najwyżej k.

Aby zakodować problem decyzyjny istnienia trasy komiwojażera o koszcie co najwyżej k użyjemy n zmiennych $v_0, \ldots, v_{n-1}$, gdzie n to liczba wierzchołków w grafie.

Formuła logiczna kodująca problem komiwojażera zapewnia, że każda zmienna v_j reprezentująca wierzchołek w ścieżce ma właściwą wartość i wszystkie zmienne są różne od siebie (podformuły *proper_numbers* oraz *distinct_vs* zostały już wcześniej omówione). Podformuła *edges* gwarantuje, że istnieje krawędź między kolejnymi wierzchołkami w ścieżce. Dodatkowo, formuła uwzględnia warunek zamknięcia ścieżki, czyli istnienie krawędzi między ostatnim a pierwszym wierzchołkiem. Ostatnia część formuły ogranicza sumę wag krawędzi $c(s,t)$ na ścieżce do k.

$$\begin{aligned}
&\textit{TSP}(n, c, E, k) = \textit{proper_numbers}(n) \land \textit{distinct_vs}(n) \land \textit{edges}(n, E) \land \\
&\left(\bigvee_{\{s,t\} \in E} ((v_{n-1} = s \land v_0 = t) \lor (v_0 = s \land v_{n-1} = t)) \right) \land \\
&\left(\bigwedge_{\{s,t\} \in E} \sum c(s,t) <= k \right)
\end{aligned}$$

Funkcja *check_tsp*$(\textit{graph}, k)$ w programie *tsp.py* sprawdza, czy istnieje trasa komiwojażera dla danego grafu i maksymalnego kosztu k. Tworzy ona zmienne reprezentujące wierzchołki i dodaje do solvera odpowiednie ograniczenia, takie jak poprawność numerów wierzchołków i ich różnorodność.

Następnie funkcja tworzy listę krawędzi (odpowiednio zakodowanych z pomocą funkcji *wedge*$(\textit{graph}, s, t, w)$ z pliku `constraints.py`) reprezentujących poszczególne odcinki trasy komiwojażera w grafie. Dodaje także ograniczenie, aby suma kosztów wszystkich krawędzi była mniejsza lub równa k. Po znalezieniu rozwiązania, funkcja wyświetla trasę komiwojażera oraz koszt całej trasy.

Listing 4.20: Funkcja *check_tsp*$(\textit{graph}, k)$

```
1  def check_tsp(graph: dict[int, list[int]], k):
2      n = len(graph)
3      vertices = z3.IntVector('v', n)
4
5      solver = z3.Solver()
6
7      solver.add(proper_numbers(vertices))
8      solver.add(distinct_vs(vertices))
9
10     edges = []
11     total_cost = z3.IntVal(0)
```

```
12
13      # Ścieżka Hamiltona z dodawaniem kosztów tras (wag
            ękrawdzi)
14      for i in range(n - 1):
15          c = z3.Int(f"w_{vertices[i]}_{vertices[i + 1]}")
16          tour_i = wedge(graph, vertices[i], vertices[i + 1], c
                )
17          edges.append(tour_i)
18          total_cost += c
19
20      # Dodawanie trasy do pierwszego miasta z powrotem oraz
            jej kosztu
21      c_back = z3.Int(f"w_{vertices[n - 1]}_{vertices[0]}")
22      tour_back = wedge(graph, vertices[n - 1], vertices[0],
            c_back)
23      edges.append(tour_back)
24      total_cost += c_back
25
26      solver.add(z3.And(edges))
27
28      solver.add(total_cost <= k)
29
30      smt2_representation = solver.to_smt2()
31      file_name = f'tsp_{n}_{k}.smt2'
32      with open(file_name, 'w') as file:
33          file.write("(set-logic ALL)\n")
34          file.write(smt2_representation)
35      file.close()
36
37      result = solver.check()
38      if result == z3.sat:
39          model = solver.model()
40          vertex_values = [(idx, model[v].as_long()) for idx, v
                in enumerate(vertices)]
41          sorted_vertices = sorted(vertex_values)
42          print("Tour:")
43          for idx, value in sorted_vertices:
44              print(f"v__{value}", end=' -> ')
45          print(f"v__{sorted_vertices[0][1]}")
```

```
46
47          # Koszt sumaryczny z modelu
48          cost = 0
49          for decl in model.decls():
50              if decl.name().startswith("w_"):
51                  value = model[decl].as_long()
52                  cost += value
53          print("Cost:", cost)
54      else:
55          print(result)
56          model = None
57
58      return result, model
```

4.8. Problem sumy podzbioru

W ogólnym sformułowaniu, problem sumy podzbiorów przyjmuje jako dane wejściowe skończony multizbiór liczb całkowitych S i całkowitą wartość docelową t, a pytanie polega na rozstrzygnięciu, czy istnieje podzbiór $S' \subseteq S$, którego elementy sumują się dokładnie do t.

Dany $S = s_0, s_1, \ldots, s_{n-1}$ to zbiór liczb całkowitych. Zmienna x_j oznacza, czy s_j zostanie wybrane do podzbioru (jeśli $x_j = 1$) lub nie (jeśli $x_j = 0$).

Aby zakodować problem decyzyjny istnienia podzbioru o sumie t użyjemy n zmiennych $x_0, \ldots, x_{n-1}$, gdzie n to liczba elementów w danym zbiorze.

Formuła logiczna problemu sumy podzbioru zapewnia, że każda zmienna x_j przyjmuje wartość logiczną 0 lub 1, co reprezentuje decyzję dotyczącą wyboru liczby s_j do podzbioru. Następnie formuła gwarantuje, że suma iloczynów x_i i s_i dla wszystkich k elementów ze zbioru S jest równa wartości t, co odpowiada szukanemu podzbiorowi.

$$\textit{SubsetSum}(n,t) = \left(\bigwedge_{j=0}^{n-1}(x_j = 0 \vee x_j = 1)\right) \wedge \left(\sum_{i=0}^{n}(x_i * s_i) = t\right)$$

Funkcja *check_subsetsum*$(\textit{input\_set}, t, k)$ w programie *subsetsum.py* jest przeznaczona do sprawdzenia, czy istnieje podzbiór zbioru wejściowego, którego suma równa się określonej wartości t. Tworzy formułę logiczną, która zakłada, że każdy element podzbioru może być 0 lub 1, a następnie dodaje ograniczenie, że suma iloczynów elementów zbioru i ich odpowiadających zmiennych musi być równa t. Po rozwiązaniu problemu generuje plik .smt2 z reprezentacją problemu oraz zwraca wynik i ewentualny model, jeśli problem jest rozwią-

zalny. Na koniec drukuje wynik w postaci sumy oraz podzbioru, który spełnia warunki, lub informację o nierozwiązywalności problemu.

Listing 4.21: Funkcja *check_subsetsum*$(input\_set, t, k)$

```
def check_subsetsum(input_set, t):
    n = len(input_set)
    vars = z3.IntVector('x', n)

    solver = z3.Solver()

    for var in vars:
        solver.add(z3.Or(var == 0, var == 1))

    subset_sum = z3.Sum([vars[i] * input_set[i] for i in
       range(n)])
    solver.add(subset_sum == t)

    folder_name = f'subsetsum_{n}'
    os.makedirs(folder_name, exist_ok=True)

    smt2_representation = solver.to_smt2()
    file_name = f'{folder_name}/subsetsum_{n}_{t}.smt2'
    with open(file_name, 'w') as file:
        file.write("(set-logic ALL)\n")
        file.write(smt2_representation)
    file.close()

    result = solver.check()
    if result == z3.sat:
        model = solver.model()
        subset = [input_set[i] for i in range(n) if model[
           vars[i]] == 1]
        print(t, subset)
    else:
        print(t, result)
        model = None

    return result, model
```

ROZDZIAŁ 5

Eksperymenty i analiza wyników

5.1. Przeprowadzenie eksperymentów i zbieranie danych

Przeprowadzenie eksperymentów zostało wykonane na laptopie Dell z procesorem Intel Core i5-1135G7 z częstotliwością 2.40GHz i 16 GB pamięci RAM. Wybór tego sprzętu został podyktowany jego dostępnością oraz wystarczającymi parametrami technicznymi do wykonania zaplanowanych testów.

W procesie generowania danych wejściowych, takich jak grafy, przyjęto zastosowanie biblioteki igraph, która stanowi potężne narzędzie w dziedzinie analizy grafów. Biblioteka ta oferuje wszechstronne możliwości tworzenia, manipulowania oraz analizy grafów, co czyni ją popularnym wyborem wśród badaczy i praktyków zajmujących się analizą sieci oraz grafów. Zaawansowane funkcje igraph umożliwiają generowanie różnorodnych typów grafów, w tym grafów skierowanych i nieskierowanych, o różnych rozmiarach i strukturach. Dodatkowo, igraph dostarcza mechanizmy do manipulacji wierzchołkami oraz krawędziami grafu, co pozwala na elastyczne dostosowanie danych wejściowych do potrzeb konkretnego badania czy eksperymentu.

Zdecydowano się generować dwa rodzaje grafów - Barabasi-Alberta i Erdos-Rényi'ego - w celu przeprowadzenia eksperymentów nad zróżnicowanymi właściwościami strukturalnymi. Ta strategia pozwala na badanie efektywności SMT-solverów w różnych warunkach oraz umożliwia pełniejszą analizę.

Grafy Barabasi-Alberta są generowane na podstawie modelu preferencyjnego przyrostowego, zaproponowanego przez Alberta-László Barabási i Réka Albert w 1999 roku.

W tym modelu nowe wierzchołki dołączają się do grafu, preferując dołączanie do wierzchołków, które już mają wiele krawędzi, co prowadzi do powstania "hubówlub wierzchołków o wysokim stopniu. Parametr m określa liczbę nowych krawędzi do dodania dla każdego nowego wierzchołka. W funkcji *generate_graph*$()$ używamy $i \div 2$ jako wartość m.

Grafy Barabasi-Alberta są często stosowane do modelowania sieci skomplikowanych, takich jak sieci społecznościowe, internetowe, czy biologiczne.

Grafy Erdos-Rényi'ego są generowane na podstawie modelu losowych grafów, zaproponowanego przez Paula Erdősa i Alfréda Rényiego w 1959 roku. W tym modelu każda możliwa krawędź między wierzchołkami grafu ma jednakowe prawdopodobieństwo istnienia. Parametry n i m określają odpowiednio liczbę wierzchołków i krawędzi grafu. W funkcji *generate_graph*$()$ ustawiamy p na wartość równą 0.98, co oznacza, że prawdopodobieństwo istnienia krawędzi między dowolnymi dwoma wierzchołkami jest bardzo wysokie, co skut-

kuje gęstym grafem. Grafy Erdos-Rényi'ego są często używane w badaniach nad teorią grafów, a także w symulacjach losowych procesów, takich jak transmisja informacji w sieciach komputerowych czy analiza przypadkowych struktur.

Funkcja *generate_graph*() służy do generowania dwóch typów nieskierowanych grafów, omówionych wyżej, o różnych rozmiarach z zakresu od 4 do 24 z krokiem co 2, oraz zapisywania ich w formacie listy krawędzi (edgelist) do odpowiednich plików tekstowych.

```
def generate_graph():
    for i in range(4, 25, 2):
        g1 = ig.Graph.Barabasi(i, i // 2)
        g1.write_edgelist(f'barabasi/barabasi_{i}.txt')

        g2 = ig.Graph.Erdos_Renyi(n = i, p = 0.98)
        g2.write_edgelist(f'erdos-renyi/erdos-renyi_{i}.txt')
```

Listing 5.1: Funkcja *generate_graph*()

Funkcja *generate_digraph*() służy do generowania skierowanych grafów. Jest podobna do funkcji *generate_graph*(), z wyjątkiem tego, że generuje ona skierowane krawędzie.

```
def generate_digraph():
    for i in range(4, 25, 2):
            g1 = ig.Graph.Barabasi(i, i // 2, directed=True)
            g1.write_edgelist(f'barabasi/barabasi_{i}.txt')

            g2 = ig.Graph.Erdos_Renyi(n=i, p=0.98, directed=
               True)
            g2.write_edgelist(f'erdos_renyi/erdos_renyi_{i}.
               txt')
```

Listing 5.2: Funkcja *generate_digraph*()

Funkcja *append_weights*(*folder_path*) została stworzona w celu przygotowania danych testowych do analizy efektywności solverów w rozwiązywaniu problemu Komiwojażera. Odczytuje ona zawartość każdego pliku linia po linii, przypisując wierzchołki źródłowe i docelowe każdej krawędzi do zmiennych s i t. Następnie generuje losową wagę dla każdej krawędzi w zakresie od 10 do 20. Ostatecznie funkcja zapisuje każdą krawędź wraz z jej wagą do pliku, nadpisując pierwotną zawartość pliku.

```
def append_weights(folder_path):
    for filename in os.listdir(folder_path):
        if filename.endswith('.txt'):
            filepath = os.path.join(folder_path, filename)
            with open(filepath, 'r') as f:
```

```
6                 lines = f.readlines()
7             with open(filepath, 'w') as f:
8                 for line in lines:
9                     s, t = map(int, line.strip().split())
10                    w = random.randint(10, 20)
11                    f.write(f'{s} {t} {w}\n')
```

Listing 5.3: Funkcja *append_weights*(*folder_path*)

Do generowania zestawów liczb całkowitych o różnych rozmiarach, przeznaczonych do eksperymentów nad problemem *SubsetSum* służy funkcja *generate_sets*(). Zestawy te są tworzone w sposób losowy w zakresie od 5 do 55 liczb całkowitych, z krokiem co 5, korzystając z funkcji random.sample(range(1, n * 10), n). Funkcja random.sample() zwraca losowy podzbiór n elementów z zakresu od 1 do n * 10. Każdy zestaw liczb jest reprezentowany jako zbiór. Następnie funkcja zapisuje każdy wygenerowany zestaw liczb do pliku tekstowego o nazwie odpowiadającej rozmiarowi zestawu, gdzie każda liczba jest oddzielana spacją.

```
1 def generate_sets():
2     for n in range(5, 56, 5):
3         int_set = set(random.sample(range(1, n * 10), n))
4 
5         filename = f'{n}.txt'
6         with open(filename, 'w') as file:
7             file.write(' '.join(map(str, int_set)))
```

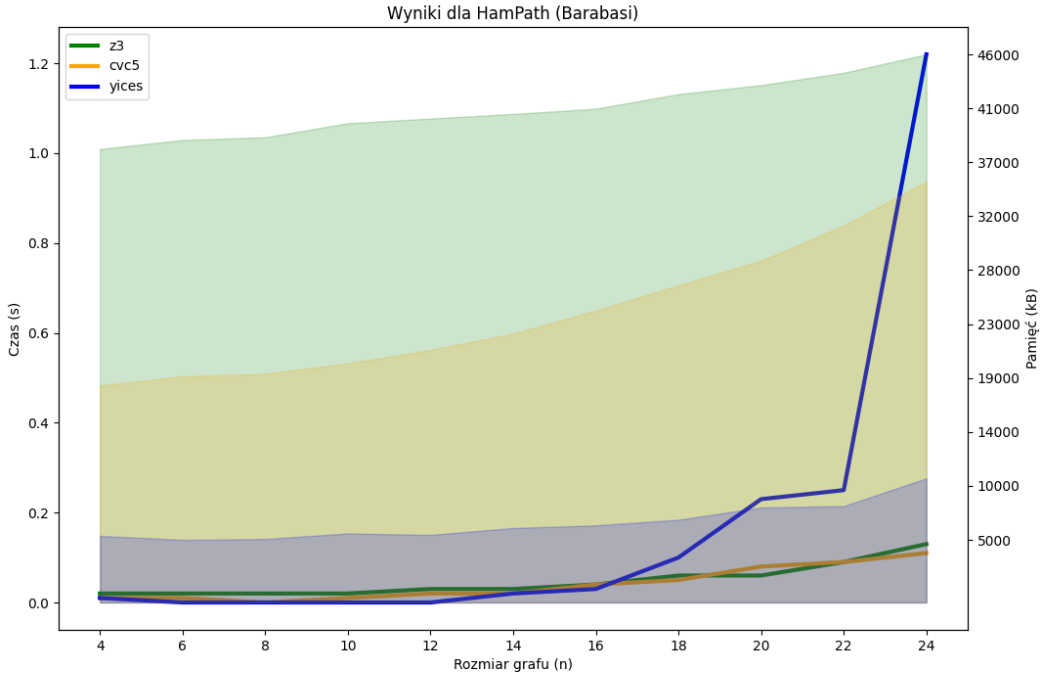
Listing 5.4: Funkcja *generate_sets*()

5.2. Analiza zebranych danych i porównanie efektywności solverów

5.2.1. Wyniki dla Problemu ścieżki Hamiltona w grafie skierowanym

Eksperymenty nad problemem ścieżki Hamiltona w grafach skierowanych, generowanych metodą Barabási-Alberta, wykazały interesujące zależności. W przypadku grafów tego typu, solver Yices zaczął wykazywać wydłużony czas działania od 20 wierzchołków. To sugeruje, że rozwiązanie problemu ścieżki Hamiltona w grafach skierowanych, zwłaszcza o większym rozmiarze, może być wymagające dla tego solvera, prowadząc do wydłużonego czasu działania.

Solvery Z3 i cvc5 wykazywały podobne czasy działania, osiągając odpowiedzi niemal równie szybko, typowo w granicach 0.1 sekundy (rysunek 5.1).



Rysunek 5.1: Wyniki eksperymentów dla grafów typu Barabási-Alberta. Opracowanie własne

Niemniej jednak, warto zauważyć, że Yices zużywał najmniej pamięci w porównaniu do innych solverów, co może stanowić jego zaletę w przypadku problemów z ograniczeniami zasobowymi.

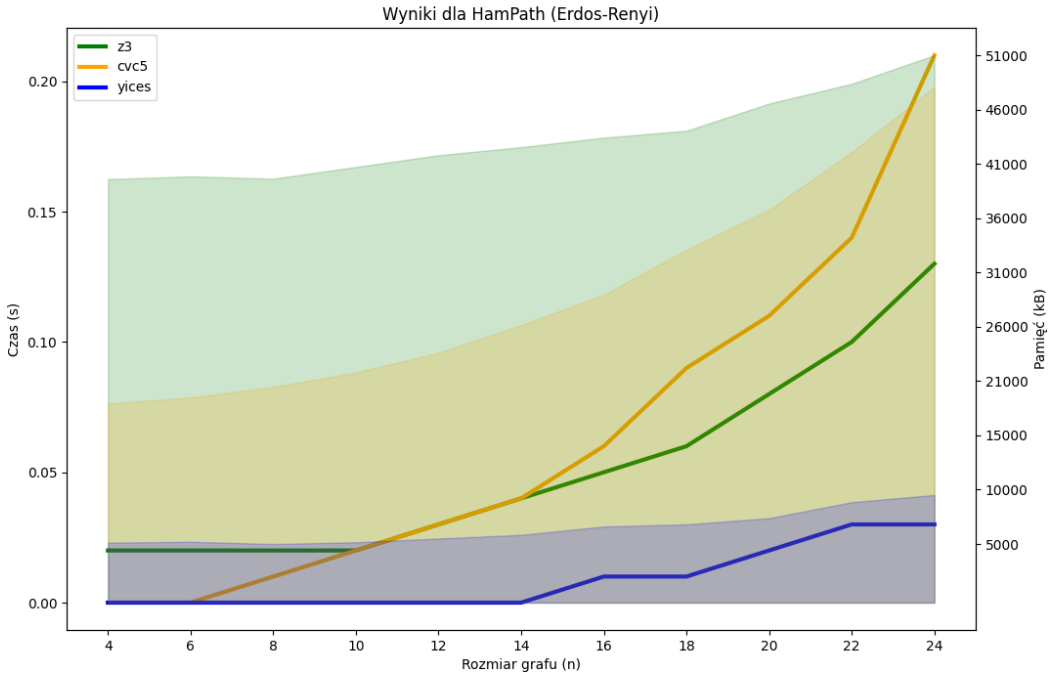
W przypadku grafów Erdős-Rényi, wyniki eksperymentów demonstrują, że Yices wykazywał się najwyższą wydajnością w rozwiązywaniu problemów ścieżki Hamiltona dla grafów tego typu, podczas gdy Z3 i cvc5 były od niego nieco wolniejsze. Ponadto, cvc5 zużywał więcej pamięci niż Yices, a Z3 jeszcze więcej, co można zaobserwować na rysunku 5.2. Można więc założyć, że pod względem zużycia zasobów pamięciowych Yices jest lepszy niż pozostałe solvery.

5.2.2. Wyniki dla Problemu ścieżki Hamiltona w grafie nieskierowanym

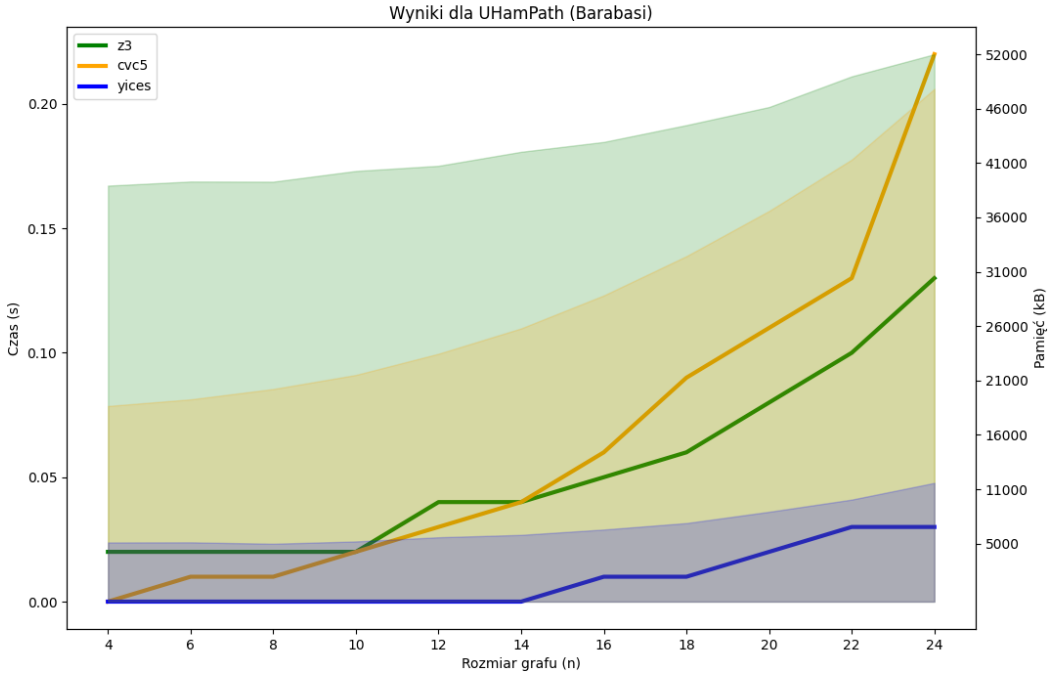
Wyniki eksperymentów nad problemem ścieżki Hamiltona w grafach nieskierowanych, uzyskane dla obu typów grafów, są bardzo podobne. W obu przypadkach solvery wykazały się zbliżoną szybkością działania, co sugeruje, że skomplikowanie grafu nie miało znaczącego wpływu na czas rozwiązania problemu.

Mimo że wszystkie solvery udzieliły odpowiedzi w czasie rzędu 0.3 sekundy, na wykresie 5.3 można zauważyć różnice w ich efektywności. Szczególnie wyraźna jest szybkość działania solvera Yices, który uzyskał najlepsze rezultaty w porównaniu z pozostałymi.

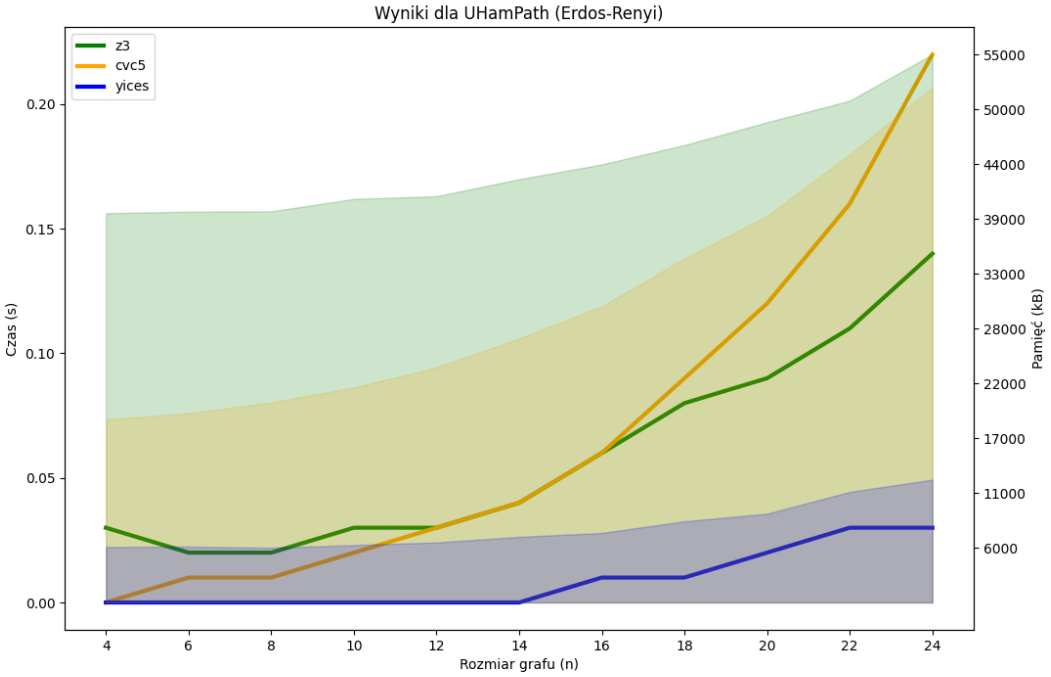
Najbardziej znaczącą różnicę zaobserwowano w zużyciu pamięci przez poszczególne solvery, co jest widoczne na rysunku 5.4. Yices po raz kolejny okazał się być najwydajniejszy, zu-



Rysunek 5.2: Wyniki eksperymentów dla grafów typu Erdős-Rényi'ego. Opracowanie własne



Rysunek 5.3: Wyniki eksperymentów dla grafów typu Barabási-Alberta. Opracowanie własne



Rysunek 5.4: Wyniki eksperymentów dla grafów typu Erdős-Rényi'ego. Opracowanie własne

żywając najmniej zasobów pamięciowych, zwłaszcza w przypadku mniejszych danych. Z tego możemy wywnioskować, że Yices wykazuje się wysoką wydajnością przy rozwiązywaniu problemów o niewielkim rozmiarze. Natomiast Z3 wykazał się największym zużyciem pamięci.

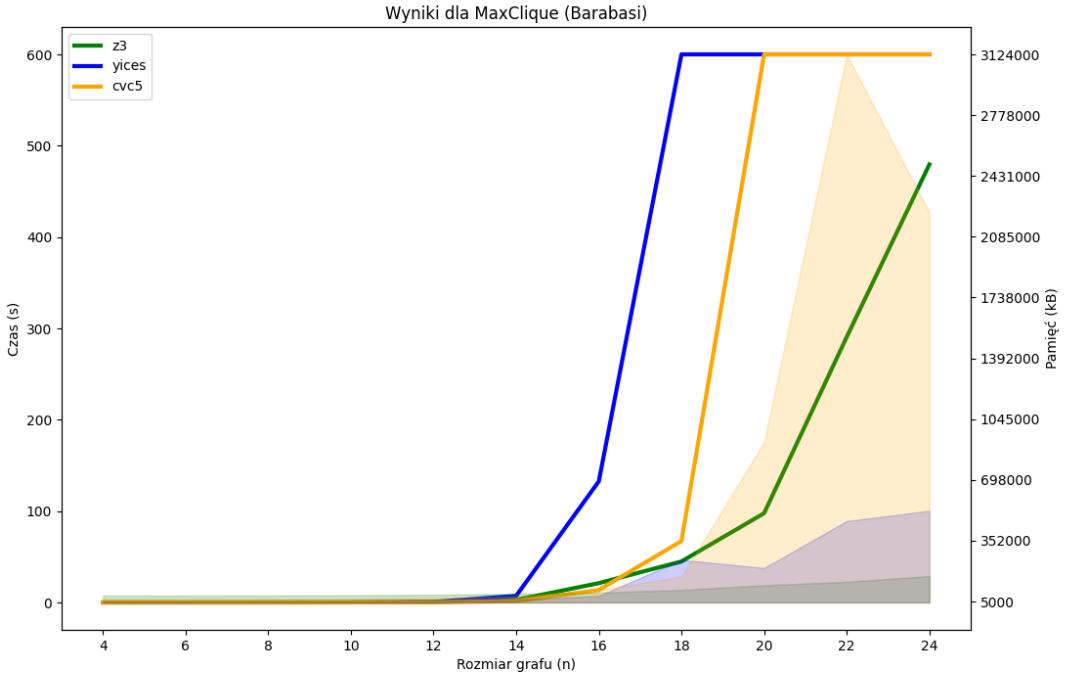
5.2.3. Wyniki dla Problemu maksymalnej kliki

W badaniach nad problemem maksymalnej kliki dla grafów typu Barabási-Alberta, eksperymenty wykazały, że solver Z3 okazał się najszybszym spośród analizowanych. Obserwowano wyraźną tendencję wzrostową czasu wykonania wraz z rosnącą liczbą zmiennych, co potwierdzało złożoność obliczeniową problemu.

Obserwowano, że rozmiar maksymalnej kliki zwiększał się w sposób regularny, zazwyczaj o jeden wierzchołek, w miarę rosnącej liczby wierzchołków w grafie. Ta regularność wzrostu rozmiaru kliki jest charakterystyczna dla tego typu sieci, gdzie nowe wierzchołki preferują łączenie się z istniejącymi wierzchołkami o większym stopniu.

Na rysunku 5.5 przedstawiono wyniki eksperymentów dla danych grafów.

W przypadku ustawienia limitu czasu na 600 sekund, solver Yices nie był w stanie wykonać obliczeń dla grafów zawierających 16 wierzchołków, podczas gdy solver cvc5 osiągnął swoje ograniczenie na 18 wierzchołkach. Oznacza to, że Yices nie był w stanie znaleźć rozwiązania w przewidzianym czasie nawet dla grafów o stosunkowo niewielkim rozmiarze, podczas gdy cvc5 wykazał lepszą skalowalność, ale również osiągnął swoje ograniczenie dla większych grafów.

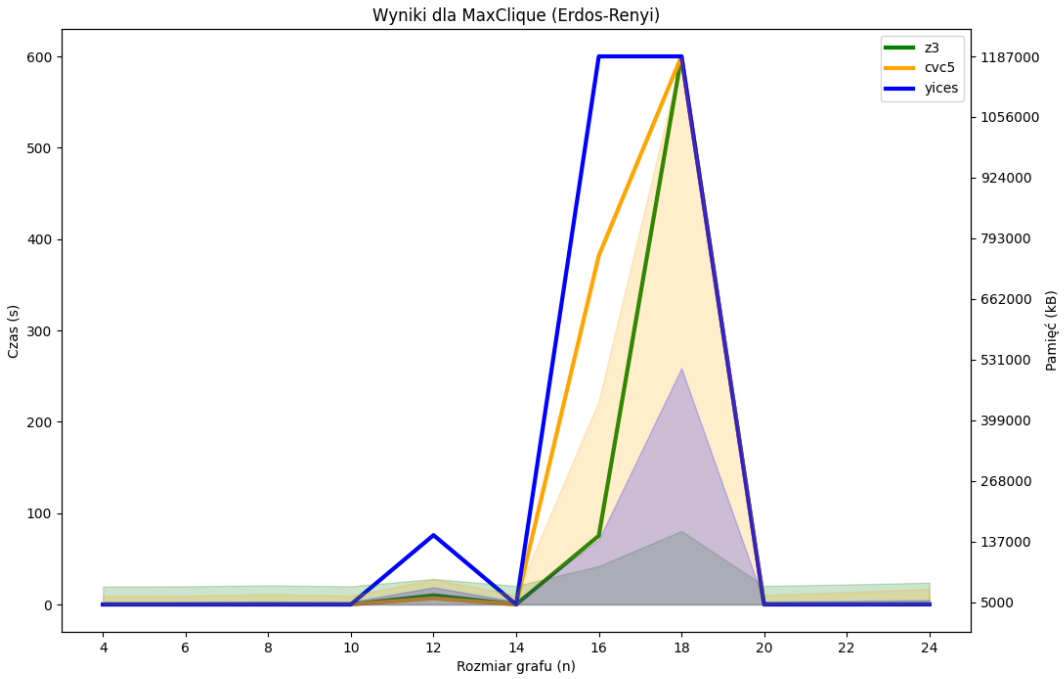


Rysunek 5.5: Wyniki eksperymentów dla grafów typu Barabási-Alberta. Opracowanie własne

Eksperymenty dla grafów typu Erdős-Rényi'ego wykazały interesującą zależność czasu działania solverów od rozmiaru poszukiwanej kliki, szczególnie w przypadku braku istnienia kliki o zadanej wielkości. Im większy był rozmiar poszukiwanej kliki, tym dłużej zajmowało czasu na stwierdzenie, że problem jest niespełnialny. W takich sytuacjach, gdzie solver musiał sprawdzić większą liczbę możliwych kombinacji w poszukiwaniu niespełnialnego warunku, czas wykonania wzrastał wykładniczo wraz ze wzrostem rozmiaru poszukiwanej kliki.

Widoczne na rysunku 5.6, że rozmiar maksymalnej kliki nie podlegał takiej regularności, jak w grafach Barabási-Alberta, ponieważ struktura tych grafów opiera się na losowych połączeniach między wierzchołkami. W związku z tym, rozmiar maksymalnej kliki mógł się zmieniać losowo dla każdego kolejnego grafu. Dla pewnych konfiguracji, gdzie istniały większe kliki, czas potrzebny na znalezienie maksymalnej kliki był dłuższy, co można zaobserwować na wykresie, ponieważ w procesie rozwiązywania problemu, solver musiał uwzględniać większą liczbę wierzchołków i krawędzi, co prowadziło do wydłużenia czasu wykonania. Z3 nadal był najszybszym spośród badanych.

Wnioskiem z tych eksperymentów jest, że czas działania solverów Z3, Yices i cvc5 zależy nie tyle od liczby wierzchołków, co od rozmiaru poszukiwanej kliki, zwłaszcza w przypadku braku istnienia kliki o zadanej wielkości. Dla przypadków, gdzie poszukiwana klika istniała, czasy działania były bardziej stabilne i niezależne od rozmiaru poszukiwanej kliki.



Rysunek 5.6: Wyniki eksperymentów dla grafów typu Erdős-Rényi'ego. Opracowanie własne

5.2.4. Wyniki dla Problemu maksymalnego niezależnego zbioru

W wynikach eksperymentów dla problemu maksymalnego niezależnego dla grafów Barabási-Alberta, obserwowano interesujące różnice w ich wydajności.

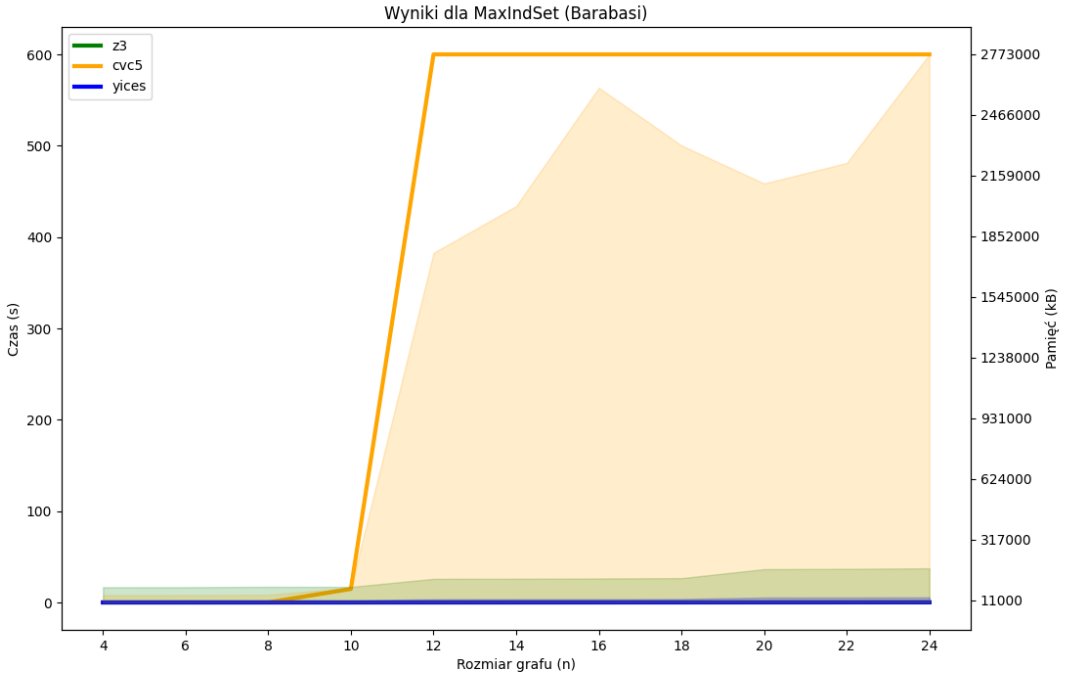
Na podstawie danych z wykresu 5.7 można stwierdzić, że zarówno Z3, jak i Yices wykazały podobnie szybkie reakcje, udzielając odpowiedzi w ciągu zaledwie 0.3 sekundy. Jednak warto zwrócić uwagę na ten fakt, że analizowane grafy miały małe rozmiary niezależnych zbiorów, z reguły zawierające 2-4 wierzchołki.

Natomiast cvc5, mimo że jest potężnym solverem SMT, napotkał znaczne trudności w rozwiązywaniu problemu dla większych instancji grafów. Już na 12 wierzchołkach cvc5 nie był w stanie udzielić odpowiedzi w określonym limicie czasu 600 sekund, szczególnie dla mniejszych wartości parametru k (rozmiar zbioru). To znaczące opóźnienie w stosunku do Z3 i Yices jest interesującym spostrzeżeniem, co może sugerować, że cvc5 ma trudności w radzeniu sobie z tym konkretnym problemem w przypadku bardziej złożonych instancji.

W przypadku grafów Erdős-Rényi, obserwowana sytuacja jest podobna do wyników uzyskanych dla grafów Barabási-Alberta. Niezależne zbiory w tych grafach były jeszcze mniejsze, co miało wpływ na szybkość działania solverów.

Z wykresu 5.8 wynika, że zarówno Z3, jak i Yices, wykazywały błyskawiczną reakcję, udzielając odpowiedzi praktycznie natychmiastowo.

Natomiast cvc5 wykazał zmienność w swojej wydajności. Mimo że niektóre odpowiedzi



Rysunek 5.7: Wyniki eksperymentów dla grafów typu Barabási-Alberta. Opracowanie własne

udzielał szybko, od grafu z 10 wierzchołkami nagle zaczęły pojawiać się dłuższe czasy oczekiwania, a w przypadku niektórych instancji problemu czas działania przekroczył 10 minut.

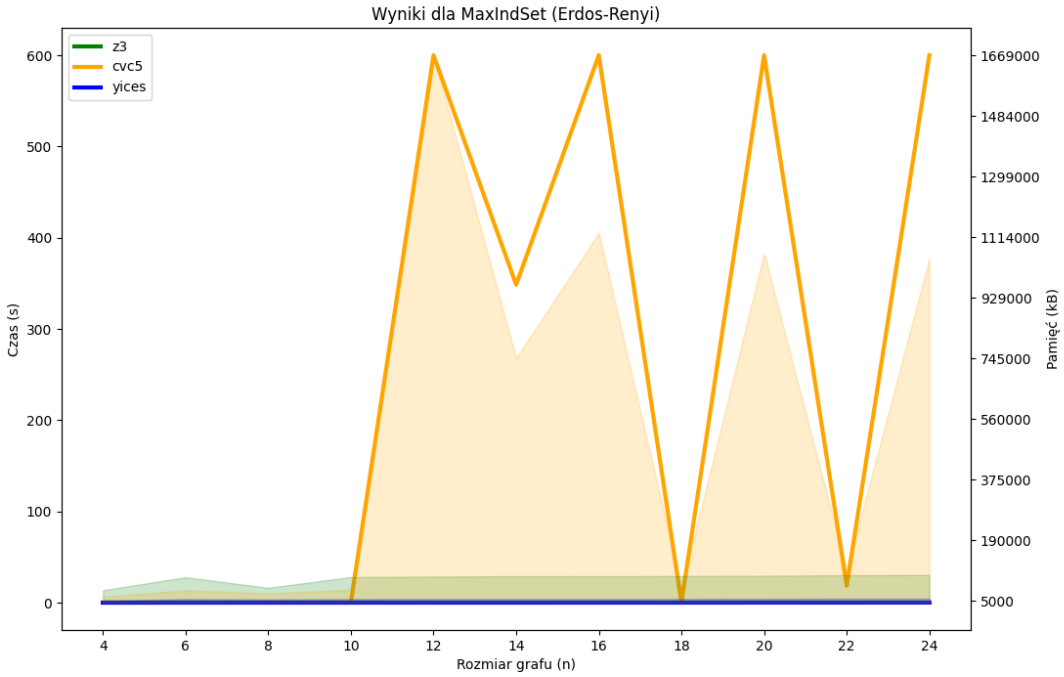
Z tej analizy można wnioskować, że szybkość działania solverów Z3 i Yices jest prawdopodobnie związana z małymi rozmiarami niezależnych zbiorów zawartych w danych grafach. Na podstawie tych obserwacji można przypuszczać, że taka cecha znacząco ułatwia rozwiązanie problemu maksymalnego niezależnego zbioru. W kontekście naukowym taka hipoteza wskazuje na istotność rozmiaru zbioru w kwestii efektywnego przetwarzania problemów optymalizacyjnych.

5.2.5. Wyniki dla Problemu pokrycia wierzchołkowego

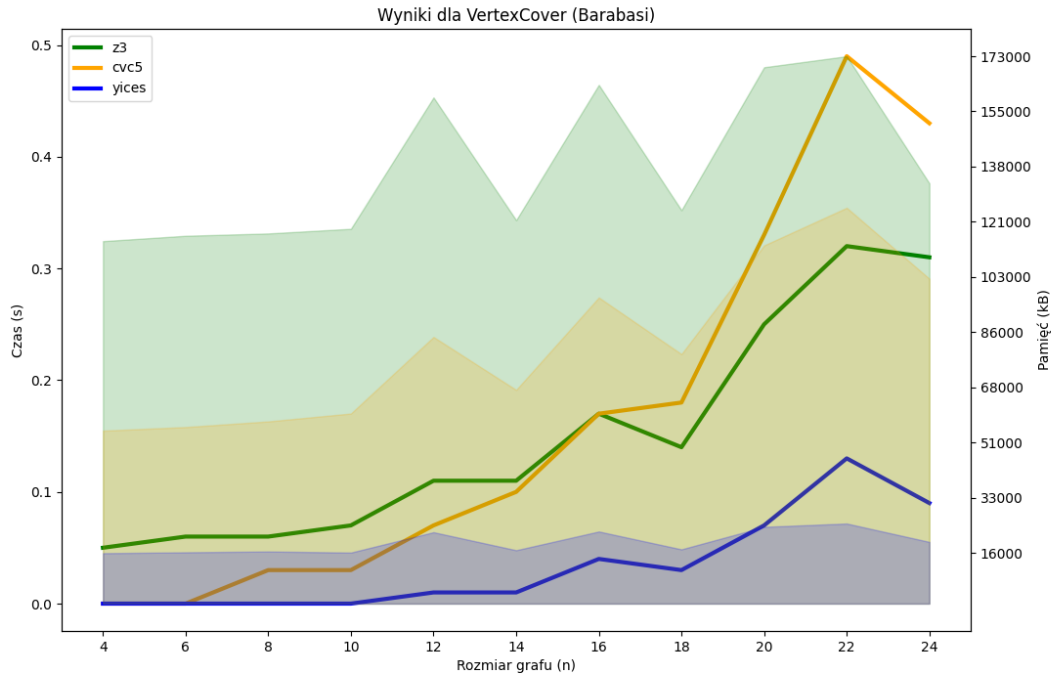
W wynikach eksperymentów dotyczących problemu pokrycia wierzchołkowego, wszystkie testowane solvery wykazywały zdolność do udzielania natychmiastowych odpowiedzi. Jednakże, interesujące różnice w wydajności można było zaobserwować w kontekście wykorzystania zasobów pamięci.

Analiza szczegółowych danych wykazała istotne różnice w szybkości działania oraz wykorzystaniu zasobów pamięci między solverami. Jak przedstawiono na wykresach 5.9 oraz 5.10, Yices wyróżniał się jako najszybszy spośród testowanych narzędzi dla obu typów grafów, udzielając odpowiedzi w bardzo krótkim czasie i wykorzystując dziesięciokrotnie mniej pamięci w porównaniu do pozostałych solverów.

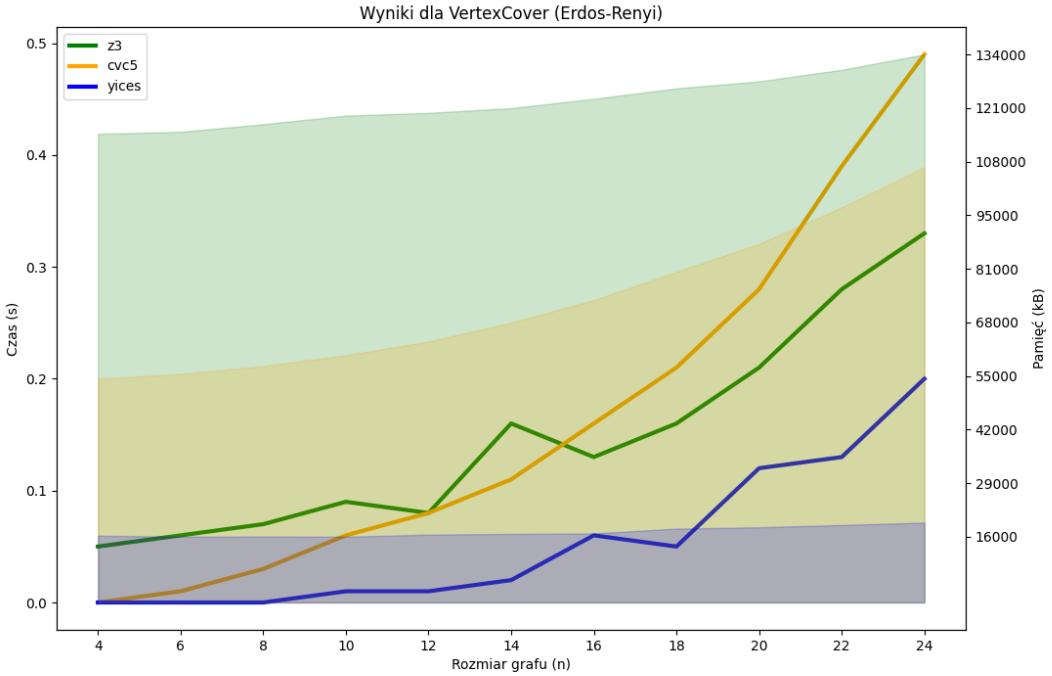
Takie wnioski sugerują, że Yices może być szczególnie przydatny w przypadku proble-



Rysunek 5.8: Wyniki eksperymentów dla grafów typu Erdős-Rényi'ego. Opracowanie własne



Rysunek 5.9: Wyniki eksperymentów dla grafów typu Barabási-Alberta. Opracowanie własne



Rysunek 5.10: Wyniki eksperymentów dla grafów typu Erdős-Rényi'ego. Opracowanie własne

mów związanych z pokryciem wierzchołkowym, ze względu na swoją wydajność i oszczędne zarządzanie zasobami.

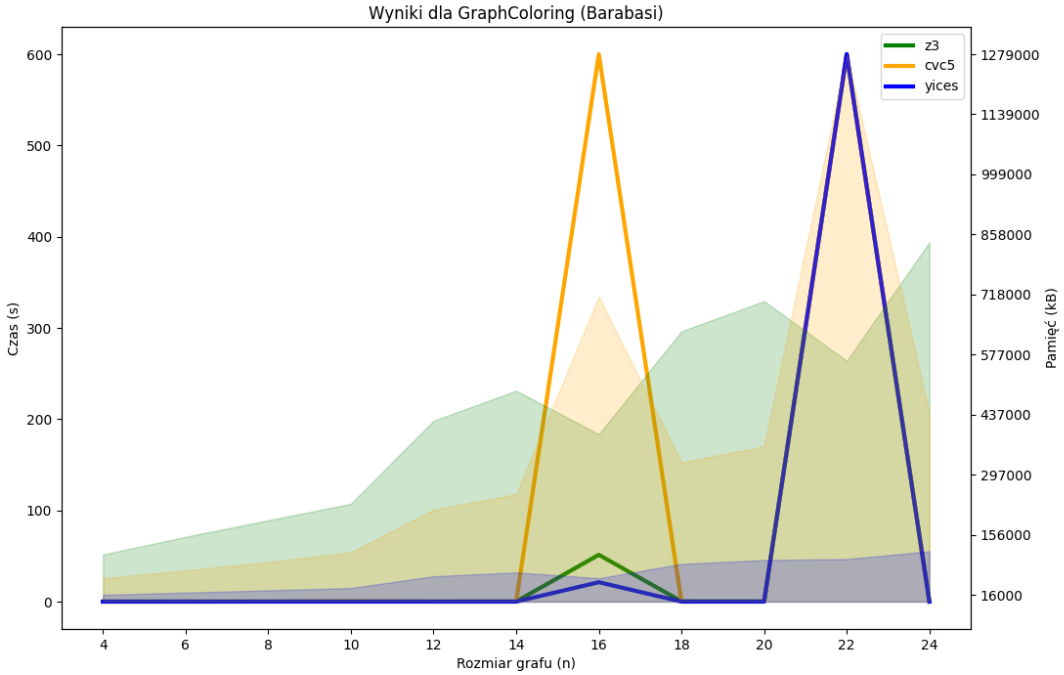
5.2.6. Wyniki dla Problemu kolorowania grafu

Wyniki eksperymentów nad problemem kolorowania grafów wykazały znaczące różnice w wykorzystaniu zasobów pamięci przez różne solvery. Natomiast, zgodnie z wykresami 5.11 i 5.12, czas potrzebny na uzyskanie odpowiedzi był prawie taki sam dla wszystkich solverów.

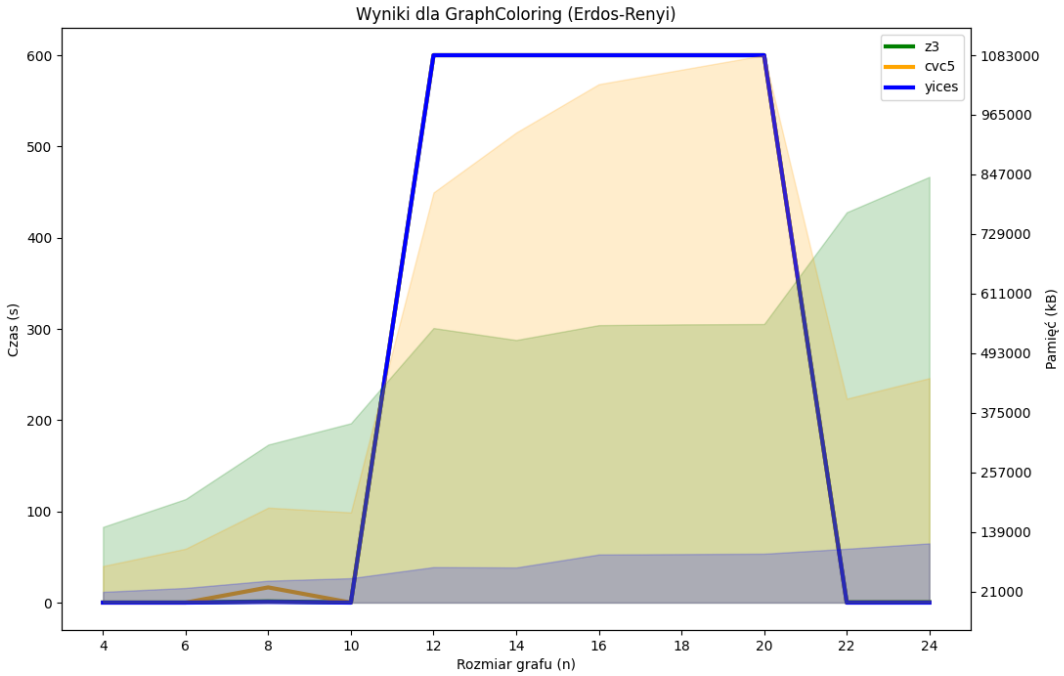
W przypadku, gdy nie było możliwości pokolorowania wszystkich wierzchołków (unsat), solvery szybko udzielały odpowiedzi, co sugeruje ich zdolność do efektywnego wykrywania niespełnionych ograniczeń.

Jednakże, w sytuacjach, gdzie istniała możliwość pokolorowania grafu (sat), zaobserwowano wydłużenie czasu potrzebnego na uzyskanie odpowiedzi od solverów, zwłaszcza w przypadku, gdy liczba dostępnych kolorów była ograniczona. Im mniejsza liczba kolorów była potrzebna do pokolorowania grafu, tym dłużej solvery szukały odpowiedzi.

Na podstawie danych z wykresu 5.12 można stwierdzić, że cvc5 wyróżniał się potrzebą znacznie większej ilości pamięci w porównaniu do innych solverów, co wskazuje na jego ograniczenia w efektywnym zarządzaniu zasobami w przypadku tego konkretnego problemu kolorowania grafów.



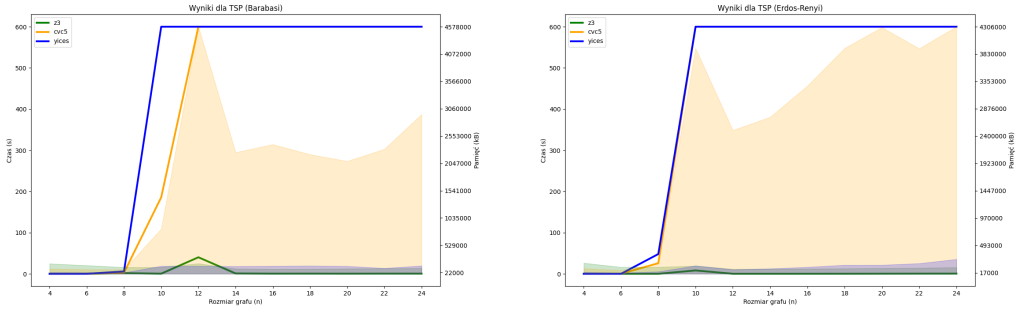
Rysunek 5.11: Wyniki eksperymentów dla grafów typu Barabási-Alberta. Opracowanie własne



Rysunek 5.12: Wyniki eksperymentów dla grafów typu Erdős-Rényi'ego. Opracowanie własne

5.2.7. Wyniki dla Problemu Komiwojażera

Wyniki eksperymentów dotyczących problemu komiwojażera (TSP) wykazały znaczną różnicę w wydajności między solverem Z3 a pozostałymi. Z3 był w stanie błyskawicznie udzielać odpowiedzi nawet dla problemów o znacznych rozmiarach, co sugeruje jego skuteczność w rozwiązywaniu TSP. Natomiast Yices i cvc5 zaczęły wykazywać długie czasy rozważania już od 8 wierzchołków i nawet na niewielkich wartościach k, szczególnie w przypadku niespełnialności. To zjawisko może wskazywać na pewne ograniczenia wydajnościowe Yices i cvc5 w przypadku problemów, w których występują niespełnialne instancje.



(a) Barabási-Alberta (b) Erdős-Rényi'ego

Rysunek 5.13: Wyniki dla TSP. Opracowanie własne

5.2.8. Wyniki dla Problemu sumy podzbioru

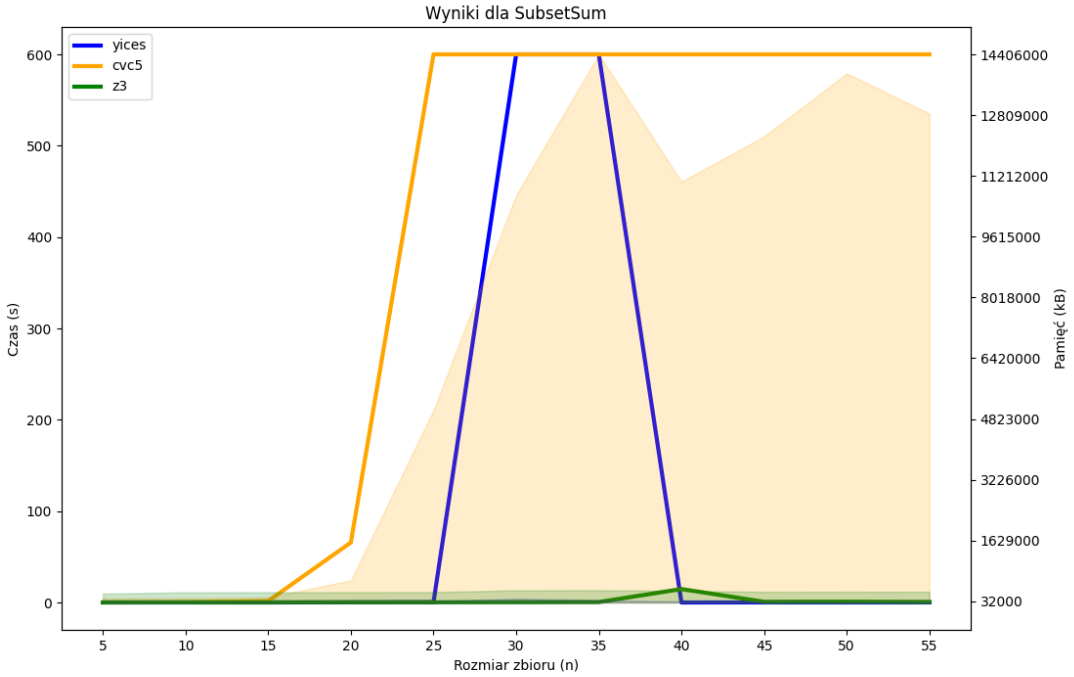
W badaniach dotyczących problemu *SubsetSum* zaobserwowano, że solver Z3 wykazywał się najwyższą wydajnością zarówno pod względem czasu, jak i zużycia pamięci. Yices również osiągnął zadowalające wyniki, jednakże na zbiorach danych o rozmiarach 30 i 35 nie był w stanie wykonać zadania. Natomiast solver cvc5 nie radził sobie już przy zbiorze o rozmiarze 20, zużywając przy tym znacznie większą ilość pamięci w porównaniu do innych solverów.

Powyższe wyniki są dobrze zilustrowane na wykresie 5.14. Te różnice w wydajności mogą być istotne w kontekście praktycznego zastosowania solverów w rozwiązywaniu problemów związanych z arytmetyką.

5.3. Identyfikacja czynników wpływających na efektywność

Z przeprowadzonych eksperymentów można wywnioskować kilka istotnych kwestii wpływających na efektywność solverów w rozwiązywaniu problemów NP-trudnych:

1. **Rozmiar instancji problemu.** Analiza eksperymentów wskazuje, że wraz z rosnącym rozmiarem problemu, tj. liczbą wierzchołków w grafie lub liczbą elementów w zbio-



Rysunek 5.14: Wyniki eksperymentów dla zbiorów o rozmiarach 5, 10, ..., 55. Opracowanie własne

rze, czas rozwiązania przez solvery znacząco się wydłuża. Dla bardziej złożonych problemów czas działania solverów może być znacznie dłuższy.

2. **Wpływ struktury grafu.** Struktura grafu, na przykład sposób jego generowania lub złożoność, może mieć znaczący wpływ na wydajność obliczeniową. Pewne solvery osiągają lepsze wyniki podczas pracy z określonymi typami grafów, co sugeruje, że istnieje zależność między strukturą grafu a wydajnością solvera.

3. **Rodzaj problemu.** Różnice w wydajności solverów w zależności od rodzaju problemu mogą być zauważalne w kontekście różnorodnych klas problemów NP-trudnych. Przykładowo, eksperymenty wykazały, że solvery znacznie szybciej radziły sobie z rozwiązaniem problemu *SubsetSum*, niż z problemami grafowymi, nawet przy dużych zestawach danych wejściowych. Te różnice mogą być istotne podczas analizy wydajności solverów w zróżnicowanych scenariuszach problemowych.

4. **Zużycie zasobów.** Oprócz czasu działania, istotnym czynnikiem jest również zużycie pamięci przez solvery. Chociaż solver Z3 zużywał więcej pamięci, to utrzymywał stabilnie szybkie czasy odpowiedzi. Z kolei Yices zużywał mniej pamięci, jednak dla większych instancji problemów nie radził sobie ze skutecznością. Natomiast solver cvc5 charakteryzował się zarówno dużym zużyciem pamięci, jak i dłuższymi czasami odpowiedzi,

co sugeruje, że może mieć ograniczoną wydajność w rozwiązywaniu problemów NP-trudnych.

5. **Różnice w trudności sprawdzania spełnialności i niespełnialności.** Warto zauważyć, że w większości przypadków szukanie niespełnialności może być trudniejsze niż spełnialności. To oznacza, że nawet w przypadku krótkiego czasu działania solvera, rozwiązanie problemu 'unsat' może być bardziej wymagające i czasochłonne niż 'sat'. Jednak nie zawsze sytuacja wygląda identycznie dla wszystkich problemów. Na przykład, w przypadku problemu sumy podzbioru, szukanie spełnialności zajmowało solverom więcej czasu niż niespełnialności.

Na podstawie wyników eksperymentów, Z3 w większości przypadków okazał się być najbardziej uniwersalnym solverem dla różnorodnych problemów NP-trudnych. Zarówno dla grafów typu Barabási-Alberta, jak i Erdős-Rényi'ego, Z3 osiągał dobre rezultaty. Jego wszechstronność pozwala na skuteczne rozwiązanie problemów, co czyni go atrakcyjnym wyborem dla wielu zastosowań. Ponadto, Z3 pokazywał tendencję do efektywnego wykorzystania zasobów pamięci, co jest istotne w kontekście problemów o dużej skali. Z uzyskanych danych wynika, że Z3 może być skutecznym narzędziem do rozwiązywania złożonych problemów NP-trudnych w praktyce.

Wszystkie pliki, kod źródłowy, eksperymenty i inne zasoby związane z niniejszą pracą znajdują się na platformie GitHub pod następującym linkiem:

`https://github.com/mossurtt/smt_solvers_for_np-hard/tree/main/experiments`.

Zakończenie

Celem mojej pracy było zbadanie efektywności SMT solverów w rozwiązywaniu klasycznych problemów NP-trudnych. W tym celu zakodowałam osiem wybranych zagadnień z tej kategorii, w tym siedem z teorii grafów i jeden z optymalizacji kombinatorycznej, i przeprowadziłam serię eksperymentów mających na celu zrozumienie, jak solvery Z3, Yices i cvc5 radzą sobie z tymi problemami obliczeniowymi.

Uważam, że udało mi się zrealizować ten cel, gdyż analiza wyników eksperymentów pozwoliła mi na dokładne zrozumienie wydajności poszczególnych solverów w różnych scenariuszach. Dzięki temu mam lepsze rozeznanie w tym, jakie rodzaje problemów mogą być rozwiązywane efektywnie przez poszczególne narzędzia, co może być przydatne w praktycznych zastosowaniach oraz w dalszych badaniach naukowych.

Wyzwaniem naukowym jest rozwój kodowania większej liczby problemów NP-trudnych z różnych dziedzin oraz przeprowadzenie eksperymentów na rozleglejszych zestawach danych wejściowych. Rozszerzenie zbioru problemów pozwoliłoby na bardziej wszechstronne zbadanie możliwości i ograniczeń poszczególnych solverów SMT w kontekście problemów NP-trudnych. Dodatkowo, większa różnorodność problemów pozwoliłaby na lepsze zrozumienie zachowania solverów w różnych scenariuszach, co może prowadzić do lepszych praktycznych zastosowań tych narzędzi w rozwiązywaniu rzeczywistych problemów.

Bibliografia - artykuły i referaty konferencyjne

[1] Haniel Barbosa i in. „cvc5: A Versatile and Industrial-Strength SMT Solver”. W: *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*. Red. Dana Fisman i Grigore Rosu. T. 13243. Lecture Notes in Computer Science. Springer, 2022, s. 415–442. DOI: 10.1007/978-3-030-99524-9_24. URL: https://doi.org/10.1007/978-3-030-99524-9%5C_24.

[2] Randal E. Bryant. „Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams”. W: *ACM Comput. Surv.* 24.3 (1992), s. 293–318. DOI: 10.1145/136035.136043. URL: https://doi.org/10.1145/136035.136043.

[3] Shaowei Cai, Bohan Li i Xindi Zhang. „Local Search for SMT on Linear Integer Arithmetic”. W: *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part II*. Red. Sharon Shoham i Yakir Vizel. T. 13372. Lecture Notes in Computer Science. Springer, 2022, s. 227–248. DOI: 10.1007/978-3-031-13188-2_12. URL: https://doi.org/10.1007/978-3-031-13188-2%5C_12.

[4] Bruno Dutertre. „Yices 2.2”. W: *Computer Aided Verification*. Red. Armin Biere i Roderick Bloem. Cham: Springer International Publishing, 2014, s. 737–744. ISBN: 978-3-319-08867-9.

[5] Lance Fortnow i Steven Homer. „A Short History of Computational Complexity”. W: *Bull. EATCS* 80 (2003), s. 95–133.

[6] Liana Hadarean i in. „Fine Grained SMT Proofs for the Theory of Fixed-Width Bit-Vectors”. W: *Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings*. Red. Martin Davis i in. T. 9450. Lecture Notes in Computer Science. Springer, 2015, s. 340–355. DOI: 10.1007/978-3-662-48899-7_24. URL: https://doi.org/10.1007/978-3-662-48899-7%5C_24.

[7] Richard M. Karp. „Combinatorics, Complexity, and Randomness”. W: *Commun. ACM* 29.2 (1986), s. 97–109. DOI: 10.1145/5657.5658. URL: https://doi.org/10.1145/5657.5658.

[8] A. D. Korshunov. „Coefficient of internal stability of graphs”. W: *Cybernetics* 10.1 (1974), s. 19–33. ISSN: 1573-8337. DOI: 10.1007/BF01069014. URL: https://doi.org/10.1007/BF01069014.

[9] Leonardo de Moura i Nikolaj Bjørner. „Z3: an efficient SMT solver". W: *2008 Tools and Algorithms for Construction and Analysis of Systems*. Springer, Berlin, Heidelberg, mar. 2008, s. 337–340. URL: https://www.microsoft.com/en-us/research/publication/z3-an-efficient-smt-solver/.

[10] Leonardo de Moura i Sebastian Ullrich. „The Lean 4 Theorem Prover and Programming Language". W: *Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings*. Red. André Platzer i Geoff Sutcliffe. T. 12699. Lecture Notes in Computer Science. Springer, 2021, s. 625–635. DOI: 10.1007/978-3-030-79876-5_37. URL: https://doi.org/10.1007/978-3-030-79876-5%5C_37.

[11] Leonardo Mendonça de Moura i Nikolaj S. Bjørner. „Satisfiability modulo theories: introduction and applications". W: *Commun. ACM* 54.9 (2011), s. 69–77. DOI: 10.1145/1995376.1995394. URL: https://doi.org/10.1145/1995376.1995394.

[12] Robert Nieuwenhuis, Albert Oliveras i Cesare Tinelli. „Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(*T*)". W: *J. ACM* 53.6 (2006), s. 937–977. DOI: 10.1145/1217856.1217859. URL: https://doi.org/10.1145/1217856.1217859.

[13] Andres Nötzli i in. „Syntax-Guided Rewrite Rule Enumeration for SMT Solvers". W: *Theory and Applications of Satisfiability Testing - SAT 2019 - 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings*. Red. Mikolás Janota i Inês Lynce. T. 11628. Lecture Notes in Computer Science. Springer, 2019, s. 279–297. DOI: 10.1007/978-3-030-24258-9_20. URL: https://doi.org/10.1007/978-3-030-24258-9%5C_20.

[14] Andrew Reynolds i in. „Reductions for Strings and Regular Expressions Revisited". W: *2020 Formal Methods in Computer Aided Design, FMCAD 2020, Haifa, Israel, September 21-24, 2020*. IEEE, 2020, s. 225–235. DOI: 10.34727/2020/ISBN.978-3-85448-042-6_30. URL: https://doi.org/10.34727/2020/isbn.978-3-85448-042-6%5C_30.

[15] Andrew Reynolds i in. „Scaling Up DPLL(T) String Solvers Using Context-Dependent Simplification". W: *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*. Red. Rupak Majumdar i Viktor Kuncak. T. 10427. Lecture Notes in Computer Science. Springer, 2017, s. 453–474. DOI: 10.1007/978-3-319-63390-9_24. URL: https://doi.org/10.1007/978-3-319-63390-9%5C_24.

[16] Hans-Jörg Schurr i in. „Alethe: Towards a Generic SMT Proof Format (extended abstract)". W: *Proceedings Seventh Workshop on Proof eXchange for Theorem Proving, PxTP 2021, Pittsburg, PA, USA, July 11, 2021*. Red. Chantal Keller i Mathias Fleury.

T. 336. EPTCS. 2021, s. 49–54. DOI: 10.4204/EPTCS.336.6. URL: https://doi.org/10.4204/EPTCS.336.6.

[17] Brent Smith. „Review of how to prove it: a structured approach by Daniel J. Velleman (Cambridge University Press, 2006)". W: *SIGACT News* 40.1 (2009), s. 18–20. DOI: 10.1145/1515698.1515703. URL: https://doi.org/10.1145/1515698.1515703.

[18] Aaron Stump i in. „SMT proof checking using a logical framework". W: *Formal Methods Syst. Des.* 42.1 (2013), s. 91–118. DOI: 10.1007/S10703-012-0163-3. URL: https://doi.org/10.1007/s10703-012-0163-3.

[19] Boris A. Trakhtenbrot. „From Logic to Theoretical Computer Science - An Update". W: *Pillars of Computer Science, Essays Dedicated to Boris (Boaz) Trakhtenbrot on the Occasion of His 85th Birthday*. Red. Arnon Avron, Nachum Dershowitz i Alexander Rabinovich. T. 4800. Lecture Notes in Computer Science. Springer, 2008, s. 1–38. DOI: 10.1007/978-3-540-78127-1_1. URL: https://doi.org/10.1007/978-3-540-78127-1%5C_1.

[20] Hisao Yamada. „Real-Time Computation and Recursive Functions Not Real-Time Computable". W: *IRE Trans. Electron. Comput.* 11.6 (1962), s. 753–760. DOI: 10.1109/TEC.1962.5219459. URL: https://doi.org/10.1109/TEC.1962.5219459.

Bibliografia - książki

[21] Yves Bertot i Pierre Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004. ISBN: 978-3-642-05880-6. DOI: 10.1007/978-3-662-07964-5. URL: https://doi.org/10.1007/978-3-662-07964-5.

[22] Edmund M. Clarke i in. *Handbook of Model Checking*. Springer, 2018. DOI: 10.1007/978-3-319-10575-8. URL: https://doi.org/10.1007/978-3-319-10575-8.

[23] Thomas H. Cormen i in. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009. ISBN: 978-0-262-03384-8. URL: http://mitpress.mit.edu/books/introduction-algorithms.

[24] Tobias Nipkow, Lawrence C. Paulson i Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. T. 2283. Lecture Notes in Computer Science. Springer, 2002. ISBN: 3-540-43376-7. DOI: 10.1007/3-540-45949-9. URL: https://doi.org/10.1007/3-540-45949-9.

[25] Michael Sipser. *Introduction to the theory of computation*. PWS Publishing Company, 1997. ISBN: 978-0-534-94728-6.

[26] Steven Skiena. *The Algorithm Design Manual, Third Edition*. Texts in Computer Science. Springer, 2020. ISBN: 978-3-030-54255-9. DOI: 10.1007/978-3-030-54256-6. URL: https://doi.org/10.1007/978-3-030-54256-6.

Bibliografia - strony internetowe

[27] Ajin Sunny. *Complexity Classes — Not just your regular Big-O*. https://medium.com/@ajin.sunny/complexity-classes-not-just-your-regular-big-o-9cb217097ed9. dostęp: 14 marzec 2024. 2023.

[28] Eric Vigoda. *Luby's Alg. for Maximal Independent Sets using Pairwise Independence*. https://faculty.cc.gatech.edu/~vigoda/RandAlgs/MIS.pdf. dostęp: 27 luty 2024. 2006.

[29] Wikipedia contributors. *Hamiltonian path problem — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Hamiltonian_path_problem&oldid=1190136070. dostęp: 16 luty 2024. 2023.

Spis rysunków

Spis listingów