POLITECHNIKA WARSZAWSKA

Wydział Elektroniki
i Technik Informacyjnych

ROZPRAWA DOKTORSKA

mgr inż. Łukasz Skonieczny

Odkrywanie częstych grafów z uwzględnianiem niespójności

Promotor
prof. nzw. dr hab. inż.
Marzena Kryszkiewicz

Warszawa, 2010

Streszczenie

Niniejsza rozprawa dotyczy zagadnienia odkrywania grafów częstych. W rozprawie zawarty jest przegląd istniejących metod odkrywania grafów częstych ze szczególnym uwzględnieniem algorytmów pochodzących z platformy $ParMol$. Ograniczenia większości istniejących algorytmów, polegające na uwzględnianiu tylko grafów spójnych w procesie wyszukiwania grafów częstych, skłoniło autora do podjęcia pracy nad opracowaniem nowych metod odkrywania grafów częstych z uwzględnianiem niespójności. W rozprawie zaproponowano dwa nowe algorytmy: algorytm UGM oraz algorytm UFC, które pozwalają na odkrywanie spójnych i niespójnych grafów częstych. Algorytm UGM opiera się na odkrywaniu częstych wielozbiorów krawędzi, z których następnie budowane są częste grafy spójne i niespójne. Algorytm UFC polega na budowaniu częstych grafów niespójnych na podstawie częstych grafów spójnych. Na pierwszym etapie algorytm UFC odkrywa wyłącznie częste grafy spójne, wykorzystując do tego celu dowolny algorytm odkrywania częstych grafów spójnych. Na kolejnych etapach odkrywane są częste grafy niespójne poprzez dołączanie do grafów częstych kolejnych częstych składowych spójnych.

Duża część rozprawy została poświęcona problemowi badania izomorfizmu z podgrafem, które jest jedną z najważniejszych operacji w algorytmach odkrywania grafów częstych. Przedstawiono rozwiązanie problemu izomorfizmu z podgrafem za pomocą metod rozwiązywania problemu spełniania ograniczeń i zaproponowano optymalizację tych metod, z wykorzystaniem symetrii grafów, w szczególności symetrii grafów niespójnych o wielokrotnych składowych spójnych.

Zaproponowane metody zostały zaimplementowane i przetestowane. Uzyskane i umieszczone w pracy wyniki eksperymentów potwierdzają skuteczność i przydatność zaproponowanych metod, a ich analiza pozwala na wskazanie kierunków kontynuacji badań w tej dziedzinie.

Słowa kluczowe: *odkrywanie wiedzy w grafach, odkrywanie grafów częstych, problem izomorfizmu z podgrafem.*

Abstract

This thesis is centered around methods of frequent graphs mining. It presents a survey of existing methods for frequent graphs discovery, especially algorithms included in the $ParMol$ platform. Most existing methods of frequent graphs discovery are limited to connected graphs only, completely ignoring frequent unconnected graphs. In this thesis, we proposed two new algorithms: UGM and UFC, which discover both connected and unconnected frequent graphs. The UGM algorithm is based on the discovery of frequent multisets of edges, which are later used to determine whether given set of edges can be used to construct a frequent graph or not. The UFC algorithm discovers connected frequent graphs by means of any existing $ParMol$ algorithm and then joins these frequent connected graphs with each other creating unconnected frequent graphs with increasing number of connected components.

The significant part of the thesis is dedicated to solving a subgraph isomorphism problem which is one the most important operation in frequent graphs discovery. We described methods of solving the subgraph isomorphism problem by solving the constraints satisfaction problem and proposed optimization of these methods which utilize graph symmetries, especially symmetries coming from multiple connected components.

The proposed methods has been implemented and verified experimentally. The experiments prove their efficiency and usability. Analysis of results shows directions of further research.

Keywords: *knowledge discovery in graphs, frequent graphs mining, subgraph isomorphism problem.*

Spis treści

1. Wstęp

1.1. Wprowadzenie

Od kilkudziesięciu lat obserwuje się nieustanny wzrost rozmiarów danych gromadzonych w postaci elektronicznej dotyczących różnorakich aspektów działalności człowieka (na przykład handlu i marketingu, telekomunikacji, medycyny, bankowości, ...). Zebrane dane mogą posłużyć do poznania charakteru danej dziedziny, co pozwala na przykład na poprawę jakości usług. Gromadzone dane są poddawane analizie i na tej podstawie wyciągane są wnioski praktyczne. Problemem stała się jednak wielkość danych i szybkość ich napływania, uniemożliwiające tradycyjne, analityczne podejście do przetwarzania danych. Pomocne okazują się nowe metody naukowe znane pod wspólną nazwą *odkrywania wiedzy* w dużych zbiorach danych[1]. Jednym ze środków wydobywania tej wiedzy stały się metody *eksploracji danych*[2]. Eksploracja danych jest etapem odkrywania wiedzy, na którym na podstawie odpowiednio przygotowanych danych, powinny zostać pozyskane wyniki nowe, użyteczne, nietrywialnie i łatwe do interpretacji.

Jednym z najważniejszych problemów eksploracji danych jest odkrywanie *wzorców częstych*[3] [1], będące uogólnieniem zadania *analizy koszyka sklepowego*. W przypadku analizy koszyka sklepowego wejściowa baza danych składa się z transakcji, a każda transakcja jest zbiorem zakupionych produktów. Odkrywanie częstych wzorców ma w tym przypadku za zadanie odkrycie grup produktów, które często występują razem w tej samej transakcji. W tym kontekście odkrywanie wzorców częstych nazywane jest też odkrywaniem *zbiorów częstych*, gdyż polega na znalezieniu zbiorów, które są podzbiorami dużej części zbiorów z wejściowej bazy danych. Wzorce częste są też używane w wielu innych metodach eksploracji danych, np. klasyfikacji [31] lub grupowaniu [45, 46]. Pojęcie wzorca częstego zostało szybko uogólnione i obok terminu *zbioru częstego* pojawiły się terminy *wielozbioru częstego*[4] [14] , *sekwencji*

[1] ang. knowledge discovery in large databases
[2] ang. data mining
[3] ang. frequent patterns
[4] ang. frequent multiset

częstej[5] [2], *drzewa częstego*[6] [78] i wreszcie *grafu częstego*[7] [72]. W przypadku odkrywania grafów częstych wejściowa baza danych składa się z grafów, a grafem częstym jest taki graf, który jest izomorficzny z podgrafami dużej części grafów z wejściowej bazy danych.

1.2. Motywacja oraz cel pracy

Odkrywanie wiedzy w grafach jest ważną i rozwojową dziedziną nauki. Grafy są uniwersalnym narzędziem reprezentowania wielu rzeczywistych obiektów i zjawisk, takich jak na przykład związki chemiczne, sieci telefoniczne, sieci drogowe, struktury portali internetowych, interakcje międzyludzkie. Ontologie, czyli również pewnego rodzaje grafy, stały się popularnym narzędziem reprezentacji wiedzy [32]. Coraz więcej gromadzonych danych jest przechowywanych w postaci grafów, co powoduje, że konieczne staje się dostarczanie efektywnych metod ich przetwarzania i analizy.

Istnieje już liczna grupa efektywnych algorytmów odkrywania częstych grafów. Znakomita większość z nich odkrywa jednak jedynie częste grafy spójne. Eksperymenty przeprowadzone w ramach tej pracy pokazują, że spójne grafy częste stanowią jedynie mały ułamek wszystkich grafów częstych. Ponadto w niektórych zastosowaniach grafy niespójne są bardziej informacyjne niż grafy spójne - na przykład w pracy [68] wykazano, że w pewnych okolicznościach wzorce kontrastowe uzyskane z grafów spójnych i niespójnych cechują się większą zwięzłością reprezentacji niż wzorce uzyskane tylko z grafów spójnych. Celem niniejszej rozprawy jest zaproponowanie efektywnej metody odkrywania częstych grafów, która w odróżnieniu od istniejących rozwiązań, nie pomija częstych grafów niespójnych.

Graf częsty jest grafem, który jest izomorficzny z podgrafami dużej części grafów z wejściowej bazy danych. Zadanie zbadania, czy graf jest izomorficzny z co najmniej jednym podgrafem danego grafu, czyli tak zwany problem izomorfizmu z podgrafem, należy do klasy NP-zupełnych i z tego względu większość algorytmów odkrywania grafów częstych nie wykonuje bezpośrednich testów na izomorfizm z podgrafem. W zamian wykorzystywane są tak zwane *zanurzenia* grafów, czyli pełna informacja o izomorfizmie danego grafu z podgrafami grafów z wejściowej bazy grafów, to znaczy informacja, które wierzchołki danego grafu są przyporządkowane którym wierzchołkom grafów ze wejściowej bazy grafów.

[5] ang. frequent sequence
[6] ang. frequent tree
[7] ang. frequent graph

Jeżeli graf posiada co najmniej jedno zanurzenie w pewnym grafie, wtedy jest izomorficzny z podgrafem tego grafu. Zanurzenia mają tę zaletę, że wystarczy znaleźć je raz i w prosty sposób uaktualniać w momencie utworzenia nowych grafów kandydujących na grafy częsty. Zanurzenia sprawdzają się w przypadku odkrywania częstych grafów spójnych, ale tracą część swych zalet w przypadku odkrywania grafów niespójnych, gdyż ich liczba znacząco wzrasta, a operacja uaktualniania zanurzeń staje się skomplikowana. Z tego powodu kolejnym celem pracy jest zaproponowanie efektywnej metody odkrywania częstych grafów, która w odróżnieniu od istniejących rozwiązań, nie wykorzystuje zanurzeń.

1.3. Zakres i teza pracy

Podstawową tezą pracy jest stwierdzenie:
Możliwe jest stworzenie efektywnego algorytmu odkrywania grafów częstych, który uwzględniałby zarówno spójne, jak i niespójne grafy częste, a wyznaczanie wsparcia grafów realizowałby za pomocą testów na izomorfizm z podgrafem.

Podstawą do zweryfikowania tej tezy miało być zaproponowanie nowych algorytmów odkrywania grafów częstych oraz ich zaimplementowanie i zintegrowanie z platformą $ParMol$ [55]. Platforma $ParMol$ zawiera cztery algorytmy odkrywania częstych grafów spójnych ($Gaston$, $MoFa$, $FFSM$, $gSpan$), spośród których trzy pierwsze wykorzystują zanurzenia zamiast bezpośrednich testów na izomorfizm z podgrafem. Cechami charakterystycznymi zaproponowanych w pracy metod odkrywania grafów częstych miały być:

— odkrywanie zarówno spójnych jak i niespójnych grafów częstych,
— wykorzystanie testów na izomorfizm z podgrafem zamiast zajmujących dużo pamięci zanurzeń.

Aby umożliwić szerszy zakres oceny zaproponowanych metod, do platformy $ParMol$ została także włączona opisana w [77] modyfikacja algorytmu $gSpan$ (nazywana w tej pracy $gSpanUnconnected$), która pozwala na odkrywanie częstych grafów niespójnych. Do wykonywania testów na izomorfizm z podgrafem w zaproponowanych metodach zostały wybrane i przetestowane trzy algorytmy: algorytm istniejący w platformie $ParMol$, algorytm $VF2$ oraz zaproponowany przez autora niniejszej rozprawy algorytm bazujący na rozwiązywaniu problemu spełniania ograniczeń. Ponieważ problem izomorfizmu z podgrafem

należy do klasy NP-zupełnych, zatem, aby zachować efektywność proponowanych metod odkrywania grafów częstych, zostały opracowane techniki umożliwiających ograniczenie liczby wykonywanych testów na izomorfizm z podgrafem.

Podstawowym wkładem autora jest propozycja dwóch nowych algorytmów odkrywania częstych grafów z uwzględnianiem niespójności: UGM i UFC. Na potrzeby algorytmów zostały zaproponowane cztery nowe techniki optymalizacyjne, które potencjalnie mogą znaleźć zastosowanie w innych algorytmach odkrywania grafów częstych. Do oryginalnych elementów pracy należy też opracowanie nowej metody wykorzystania symetrii grafów do rozwiązania problemu izomorfizmu z podgrafem zdefiniowanego jako problem spełniania ograniczeń.

1.4. Układ pracy

W rozdziale 2 przedstawiono podstawowe pojęcia teorii grafów używane w pracy. Ponadto zdefiniowano problem odkrywania grafów częstych oraz opisano najważniejsze zagadnienia z nim związane i ogólne metody rozwiązywania.

W rozdziale 3 dokonano aktualnego przeglądu istniejących algorytmów odkrywania grafów częstych ze szczególnym uwzględnieniem algorytmów znajdujących się w platformie $ParMol$, to jest algorytmów $Gaston$, $MoFa$, $gSpan$, $FFSM$.

W rozdziale 4 przedstawiono dwie nowe propozycje algorytmów odkrywania grafów częstych z uwzględnianiem niespójności: algorytmu UGM i UFC. Algorytm UGM odkrywa jednocześnie częste grafy spójne i niespójne, natomiast algorytm UFC odkrywa najpierw częste grafy spójne za pomocą dowolnego algorytmu z platformy $ParMol$, a następnie łączy odkryte częste grafy spójne tworząc niespójne grafy kandydujące. W rozdziale 4 przedstawiono też cztery techniki poprawiające wydajność algorytmu UGM i pokazano, że trzy z nich można wykorzystać także w algorytmie UFC.

Rozdział 5 jest opisem problemów badania izomorfizmu z grafem oraz izomorfizmu z podgrafem. W rozdziale tym zostały krótko omówione istniejące metody rozwiązania problemu, ze szczególnym uwzględnieniem metody wykorzystującej rozwiązanie problemu spełniania ograniczeń. W rozdziale tym zaproponowano i przedyskutowano także ulepszenia tej metody.

Rozdział 6 jest poświęcony badaniom eksperymentalnym zaproponowanych metod. W ramach eksperymentów wykonano zarówno bezpośrednie porównanie zaproponowanych

algorytmów z istniejącymi algorytmami, jak i zbadano wpływ poszczególnych optymalizacji i heurystyk zawartych w proponowanych algorytmach na ich wydajność. W rozdziale 6 zamieszczone są też najważniejsze wnioski płynące z części eksperymentalnej.

Rozdział 7 stanowi podsumowanie pracy, wnioski oraz propozycje i kierunki dalszych badań.

Ostatnią częścią pracy jest bibliografia.

2. Wprowadzenie do zagadnienia odkrywania grafów częstych

2.1. Podstawowe definicje i własności

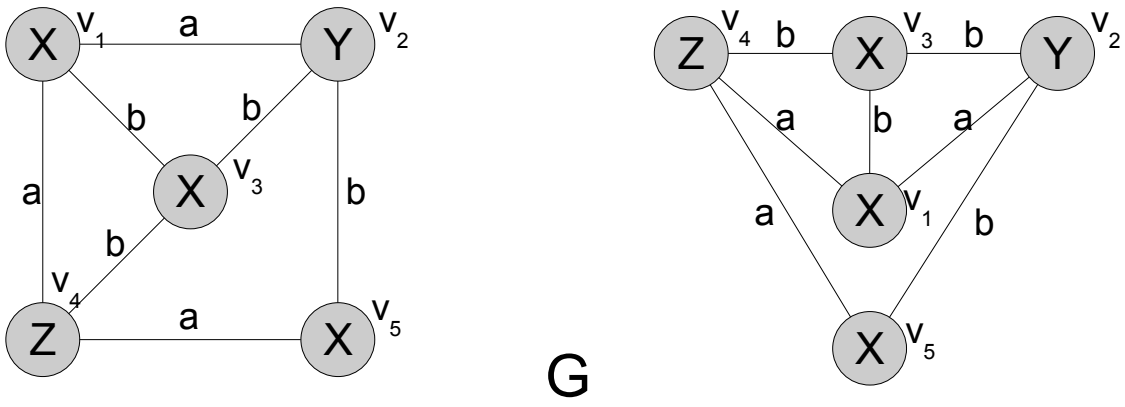
Graf to nieformalnie zbiór wierzchołków, pomiędzy którymi mogą występować krawędzie. W tej pracy pod pojęciem grafu będziemy rozumieli najczęściej *prosty nieskierowany graf etykietowany*, którego definicja przedstawiona jest poniżej.

Definicja 2.1. *(prosty nieskierowany graf etykietowany)*
Prostym nieskierowanym grafem etykietowanym G nazywamy czwórkę $G = (V, E, lbl, L)$, gdzie

- V jest zbiorem wierzchołków,
- $E = \{\{v_1, v_2\} | v_1, v_2 \in V, v_1 \neq v_2\}$ jest zbiorem krawędzi,
- $lbl : V \cup E \rightarrow L$ jest funkcją nadającą etykiety wierzchołkom i krawędziom,
- L jest zbiorem etykiet.

Jeżeli w pracy nie zostanie zaznaczone inaczej, graf należy rozumieć jak w definicji 2.1. Jest to zgodne z założeniami przyjętymi w przeważającej części literatury dotyczącej odkrywania wiedzy w grafach, która w znacznej mierze wywodzi się z bio- i chemio-informatyki, gdzie etykietowany graf nieskierowany stosuje się jako model związków chemicznych. Wierzchołki grafu reprezentują wtedy poszczególne atomy, a krawędzie - wiązania między nimi. Wierzchołki są etykietowane nazwami pierwiastków, a krawędzie rodzajem wiązania.

Każdy graf może być jednoznacznie przedstawiony za pomocą *graficznej reprezentacji grafu*. W pracy używana jest następująca graficzna reprezentacja grafu: wierzchołki grafu są reprezentowane w postaci okręgów; symbol umieszczony wewnątrz okręgu oznacza etykietę wierzchołka; krawędzie są reprezentowane w postaci krzywych łączących wierzchołki; symbol umieszczony przy krzywej oznacza etykietę danej krawędzi.



Rysunek 2.1. Dwie różne reprezentacje graficzne tego samego grafu G.

Jeżeli reprezentacja graficzna nie posiada symboli oznaczających etykiety wierzchołków i/lub krawędzi, należy przyjąć, że wszystkie wierzchołki i/lub krawędzie posiadają tę samą etykietę.

Przykład 2.1. Na rysunku 2.1 znajdują się dwie graficzne reprezentacje tego samego grafu $G = (V, E, lbl, L)$, gdzie:

- $V = \{v_1, v_2, v_3, v_4, v_5\}$,
- $E = \{\{v_1, v_2\}, \{v_1, v_3\}, \{v_1, v_4\}, \{v_2, v_3\}, \{v_2, v_5\}, \{v_3, v_4\}, \{v_4, v_5\}\}$,
- $lbl(v_1) = X,\ lbl(v_2) = Y,\ lbl(v_3) = X,\ lbl(v_4) = Z,\ lbl(v_5) = X,$
  $lbl(\{v_1, v_2\}) = a,\ lbl(\{v_1, v_3\}) = b,\ lbl(\{v_1, v_4\}) = a,$
  $lbl(\{v_2, v_3\}) = b,\ lbl(\{v_2, v_5\}) = b,\ lbl(\{v_3, v_4\}) = b,\ lbl(\{v_4, v_5\}) = a,$
- $L = \{X, Y, Z, a, b\}$.

Definicja 2.2. *(stopień wierzchołka)*
W grafie $G = (V, E, lbl, L)$ *stopniem wierzchołka* $v \in V$ (oznaczanym przez $d(v)$) nazywamy liczbę krawędzi $e \in E$, które zawierają wierzchołek v.

Definicja 2.3. *(ścieżka, łańcuch)*
W grafie $G = (V, E, lbl, L)$ *ścieżką łączącą (łańcuchem łączącym)* wierzchołki $v_a, v_b \in V$ nazywamy taki ciąg wierzchołków $(v_1, v_2, v_3, \ldots v_n)$, że
$v_1 = v_a, v_n = v_b, \forall_{i \in \{1, n-1\}}\ \{v_i, v_{i+1}\} \in E$.
Liczbę n nazywamy *długością ścieżki*.

Definicja 2.4. *(graf spójny)*

Graf nazywamy *spójnym*, gdy istnieje ścieżka łącząca każdą parę wierzchołków tego grafu.

Definicja 2.5. *(graf niespójny)*

Graf nazywamy *niespójnym*, gdy nie jest grafem spójnym.

Definicja 2.6. *(graf pusty)*

Graf $G = (V, E, lbl, L)$ nazywamy *grafem pustym*, gdy $|E| = 0$.

Definicja 2.7. *(graf bez wierzchołków)*

Graf $G = (V, E, lbl, L)$ nazywamy *grafem bez wierzchołków*, gdy $|V| = 0$.

Definicja 2.8. *(izomorfizm grafów)*

Grafy $G = (V, E, lbl, L)$ i $G' = (V', E', lbl', L')$ są *izomorficzne* (co oznaczamy przez $G \cong G'$), gdy istnieje bijekcja $\phi : V \to V'$ taka, że:

$\forall_{\{v_1,v_2\}\in E} \ \{\phi(v_1), \phi(v_2)\} \in E' \wedge$

$\forall_{v\in V} \ lbl(v) = lbl'(\phi(v)) \wedge$

$\forall_{e\in E} \ lbl(e) = lbl'(\phi(e))$.

Bijekcja ϕ jest nazywana *izomorfizmem* grafu G w graf G'. Jeżeli bijekcja nie istnieje, wtedy grafy G i G' nie są izomorficzne (co oznaczamy przez $G \ncong G'$).

Definicja 2.9. *(automorfizm grafów)*

Automorfizmem grafu G nazywamy każdy izomorfizm grafu G w graf G (w samego siebie). Innymi słowy jest to bijekcja $\phi : V \to V$ (czyli permutacja) taka, że:

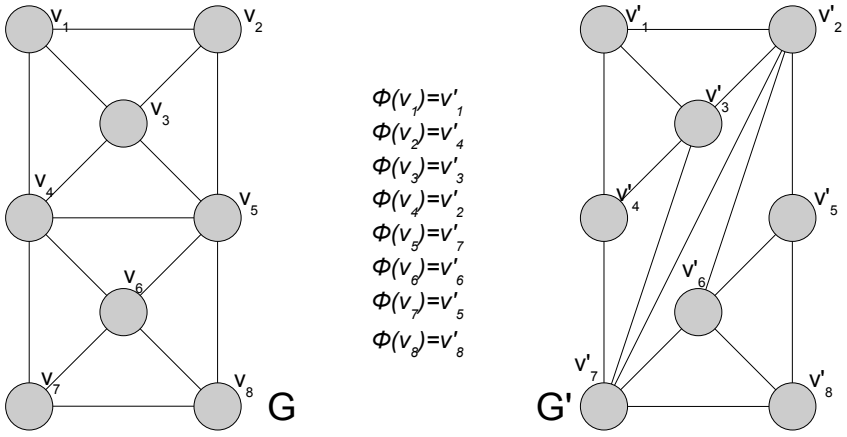
$\forall_{e=\{v_1,v_2\}} \in E \quad \{\phi(v_1), \phi(v_2)\} \in E \wedge$

$\forall_{v\in V} \quad lbl(v) = lbl(\phi(v)) \wedge$

$\forall_{e\in E} \quad lbl(e) = lbl(\phi(e))$.

Każdy graf posiada przynajmniej jeden automorfizm - permutację identycznościową.

Izomorfizm definiuje relację równoważności w zbiorze grafów, a co za tym idzie dzieli zbiór grafów na klasy równoważności. Grafy izomorficzne należą do jednej klasy równoważności. Grafy nieizomorficzne należą do różnych klas równoważności. Wszystkie grafy w jednej klasie równoważności mogą być reprezentowane przez jeden wybrany arbitralnie graf. Pozostałe grafy w tej samej klasie równoważności będziemy nazywali *duplikatami*. Przyjmujemy, że reprezentacja graficzna grafu nie posiadająca symboli przy wierzchołkach będzie reprezentować klasę równoważności. Liczbę klas równoważności w danym zbiorze grafów będziemy nazywać *liczbą grafów nieizomorficznych* w tym zbiorze.



Rysunek 2.2. Przykład izomorfizmu grafów. Grafy G i G' są izomorficzne. ϕ jest jednym z czterech możliwych izomorfizmów G w G'

Przykład 2.2. Widoczne na rysunku 2.2 grafy G i G' są izomorficzne. Istnieją cztery różne izomorfizmy grafu G w graf G':

$\phi_1 : v_1 \to v'_1, v_2 \to v'_4, v_3 \to v'_3, v_4 \to v'_2, v_5 \to v'_7, v_6 \to v'_6, v_7 \to v'_5, v_8 \to v'_8$

$\phi_2 : v_1 \to v'_4, v_2 \to v'_1, v_3 \to v'_3, v_4 \to v'_7, v_5 \to v'_2, v_6 \to v'_6, v_7 \to v'_8, v_8 \to v'_5$

$\phi_3 : v_1 \to v'_5, v_2 \to v'_8, v_3 \to v'_6, v_4 \to v'_2, v_5 \to v'_7, v_6 \to v'_3, v_7 \to v'_1, v_8 \to v'_4$

$\phi_4 : v_1 \to v'_8, v_2 \to v'_5, v_3 \to v'_6, v_4 \to v'_7, v_5 \to v'_2, v_6 \to v'_3, v_7 \to v'_4, v_8 \to v'_1$

Definicja 2.10. *(podgraf)*

Graf $G = (V, E, lbl, L)$ jest podgrafem grafu $G' = (V', E', lbl', L')$, $(G \preceq G')$, gdy

- $V \subseteq V' \wedge$
- $E \subseteq E' \wedge$
- $\forall_{x \in V \cup E} \quad lbl(x) = lbl'(x) \wedge$
- $L \subseteq L'$.

Innymi słowy G' jest grafem powstałym przez usunięcie z grafu G pewnej liczby wierzchołków lub krawędzi. Przykłady podgrafów znajdują się na rysunku 2.3.

Definicja 2.11. *(nadgraf)*

Graf G' jest nadgrafem grafu G, $(G \succeq G')$, gdy graf G jest podgrafem grafu G'.

Definicja 2.12. *(rozszerzenie grafu)*

Graf G' jest *rozszerzeniem* grafu G, gdy graf G' jest nadgrafem grafu G i graf G' powstał z grafu G za pomocą operacji zwanej *rozszerzaniem*, której definicja zależy od kontekstu.

W przeważającej części pracy operacja rozszerzania grafu G oznacza dodanie jednej krawędzi do grafu G.

Definicja 2.13. *(podgraf indukowany wierzchołkowo)*
Graf $G = (V, E, lbl, L)$ jest podgrafem indukowanym wierzchołkowo grafu $G' = (V', E', lbl', L')$, gdy

$V \subseteq V' \wedge$

$E = \{\{v_1, v_2\} \mid v_1, v_2 \in V, \{v_1, v_2\} \in E'\} \wedge$

$\forall_{x \in V \cup E} \quad lbl(x) = lbl'(x) \wedge$

$L \subseteq L'.$

Innymi słowy, G jest grafem powstałym przez usunięcie z grafu G' pewnej liczby wierzchołków oraz wszystkich krawędzi zawierających te wierzchołki. Przykład podgrafu indukowanego wierzchołkowo znajduje się na rysunku 2.3 a).

Definicja 2.14. *(podgraf indukowany krawędziowo)*
Graf $G = (V, E, lbl, L)$ jest podgrafem indukowanym krawędziowo grafu $G' = (V', E', lbl', L')$, gdy

$E \subseteq E' \wedge$

$V = \{v \in V' \mid \exists_{v'} \{v, v'\} \in E\} \wedge$

$\forall_{x \in V \cup E} \quad lbl(x) = lbl'(x) \wedge$

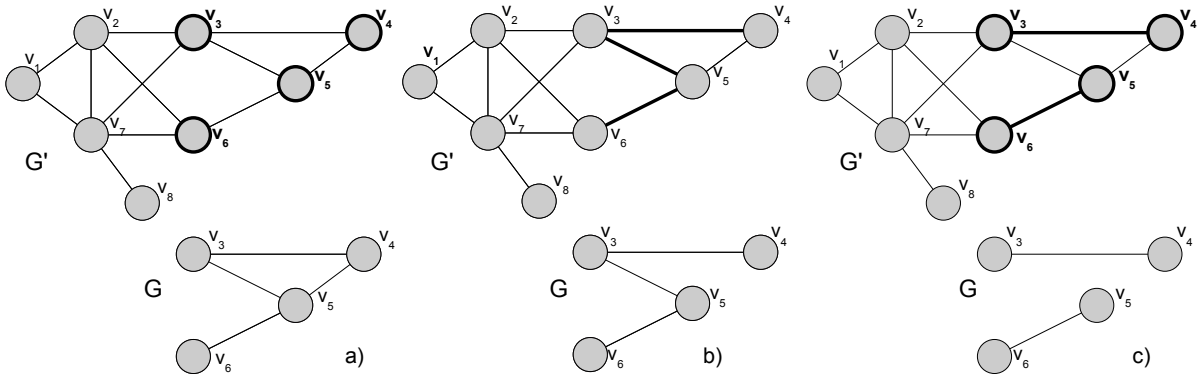
$L \subseteq L'.$

Mówiąc nieformalnie, G jest grafem, którego zbiór krawędzi jest podzbiorem zbioru krawędzi grafu G', natomiast zbiór wierzchołków stanowią wierzchołki należące do tych krawędzi. Przykład podgrafu indukowanego krawędziowo znajduje się na rysunku 2.3 b).

Definicja 2.15. *(składowa spójna)*
Graf CG jest składową spójną grafu G, gdy CG jest grafem spójnym oraz jest podgrafem grafu G oraz nie istnieje graf spójny $G_1 \neq CG$ będący podgrafem grafu G, który jest nadgrafem grafu CG.

Definicja 2.16. *(izomorfizm z podgrafem)*
Graf G jest *izomorficzny z podgrafem* grafu G' ($G \precsim G'$), gdy istnieje taki podgraf grafu G', który jest izomorficzny z grafem G. Innymi słowy, graf $G = (V, E, lbl, L)$ jest izomorficzny z podgrafem grafu $G' = (V', E', lbl', L')$, gdy istnieje injekcja $\phi : V \to V'$ taka, że:

$\forall_{\{v_1, v_2\} \in E} \quad \{\phi(v_1), \phi(v_2)\} \in E' \wedge$



Rysunek 2.3. Typy podgrafów. a) G jest podgrafem grafu G' indukowanym wierzchołkowo, b) G jest podgrafem grafu G' indukowanym krawędziowo, c) G jest podgrafem grafu G'. Wierzchołki i krawędzie, które zostały wybrane do utworzenia podgrafu są pogrubione.

$$\forall_{v \in V} \quad lbl(v) = lbl'(\phi(v)) \wedge$$
$$\forall_{e \in E} \quad lbl(e) = lbl'(\phi(e)).$$

Injekcja ϕ jest nazywana *izomorfizmem* grafu G z podgrafem grafu G'. Jeżeli injekcja nie istnieje, wtedy graf G nie jest izomorficzny z podgrafem grafu G'.

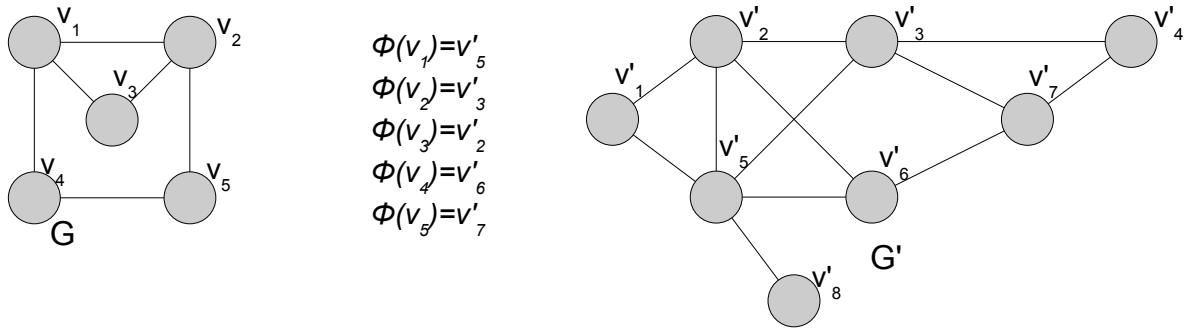
Definicja 2.17. *(zanurzenie)*
Jeżeli ϕ jest izomorfizmem grafu G z podgrafem grafu G', wtedy funkcję $\phi : V \to V'$ nazywamy *zanurzeniem* grafu G w grafie G'.

Izomorfizm z podgrafem definiuje relację częściowego porządku w zbiorze grafów.

Przykład 2.3. Widoczny na rysunku 2.4 graf G jest izomorficzny z podgrafem grafu G'. Na rysunku pokazano tylko jeden podgraf grafu G', z którym G jest izomorficzny. Zanurzenie ϕ jest jednym z ośmiu różnych zanurzeń grafu G w grafie G'. Wszystkie zanurzenia grafu G w grafie G' są pokazane na rysunku 2.5.

Definicja 2.18. *(graf wspierający)*
Graf G' jest *grafem wspierającym* graf G, gdy graf G jest izomorficzny z podgrafem grafu G'. Jeśli G' jest grafem wspierającym graf G, wtedy mówimy, że graf G' *wspiera* graf G.

Definicja 2.19. *(zbiór wspierający grafu)*
W danym zbiorze grafów $\mathbb{D}$ *zbiorem wspierającym* grafu G (oznaczanym przez $G.supportingSet$) jest zbiór wszystkich grafów z $\mathbb{D}$, które wspierają graf G.



Rysunek 2.4. Przykład izomorfizmu z podgrafem. Graf G jest izomorficzny z pewnym podgrafem G'. ϕ jest jednym z możliwych zanurzeń grafu G w grafie G'.

Definicja 2.20. *(wsparcie grafu)*
Niech $\mathbb{D}$ będzie zbiorem grafów. Wsparciem grafu G nazywamy liczbę grafów $G' \in \mathbb{D}$, które wspierają graf G.

Definicja 2.21. *(graf częsty)*
Graf G jest częsty w zbiorze $\mathbb{D}$, jeśli jego wsparcie jest większe niż bądź równe progowi wsparcia $minSup$.

Definicja 2.22. *(graf nieczęsty)*
Graf G jest nieczęsty w zbiorze $\mathbb{D}$, jeśli nie jest częsty.

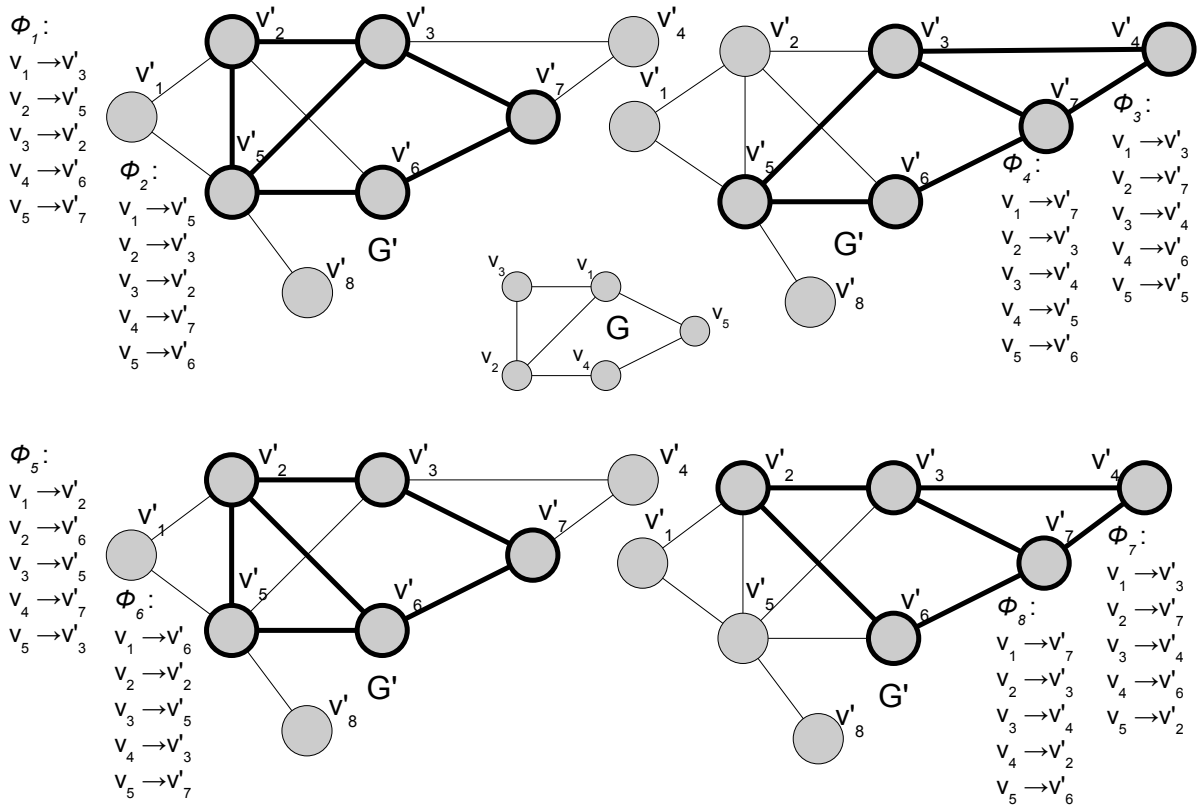
Poniższe pojęcia *deskryptora krawędzi* (definicja 2.23) oraz *wielozbioru deskryptorów krawędzi* (definicja 2.24) zostały wprowadzone przez autora tej pracy na potrzeby algorytmu UGM. Ponieważ są przydatne przy opisie także innych zagadnień zostały umieszczone już w tym rozdziale.

Definicja 2.23. *(deskryptor krawędzi)*
Deskryptorem krawędzi $e = \{v_1, v_2\} \in E$ jest para $(\{lbl(v_1), lbl(v_2)\}, lbl(e))$.

Definicja 2.24. *(wielozbiór deskryptorów krawędzi grafu)*
Wielozbiorem deskryptorów krawędzi danego grafu G (oznaczanym przez $ES(G)$) jest wielozbiór składający się z deskryptorów wszystkich krawędzi grafu G.

Własność 2.1. *Dwa nieizomorficzne grafy mogą mieć ten sam wielozbiór deskryptorów krawędzi.*

Własność 2.2. *Jeśli graf G jest izomorficzny z podgrafem grafu G', wtedy $ES(G) \subseteq ES(G')$.*



Rysunek 2.5. Wszystkie zanurzenia grafu G w grafie G'.

Przykład 2.4. Na rysunku 2.6 są przedstawione cztery nieskierowane grafy z etykietami. Wielozbiory deskryptorów krawędzi tych grafów to odpowiednio:

$ES(G_1) = \{(\{0,0\},4)(\{0,0\},4)(\{0,1\},4)(\{0,1\},4)(\{0,1\},4)(\{1,1\},5)(\{1,1\},5)\}$

$ES(G_2) = \{(\{0,0\},4)(\{0,0\},4)(\{0,0\},4)(\{0,1\},4)(\{1,1\},5)(\{1,1\},5)\}$

$ES(G_3) = \{(\{0,0\},4)(\{0,0\},4)(\{0,0\},4)(\{0,1\},4)(\{1,1\},5)(\{1,1\},5)\}$

$ES(G_4) = \{(\{0,0\},4)(\{0,0\},4)(\{0,1\},4)(\{1,1\},5)\}$

Na przykładzie tych grafów można zaobserwować podane wcześniej własności dotyczące ich wielozbiorów deskryptorów krawędzi.

$ES(G_2) = ES(G_3)$, ale grafy G_2 i G_3 nie są izomorficzne.

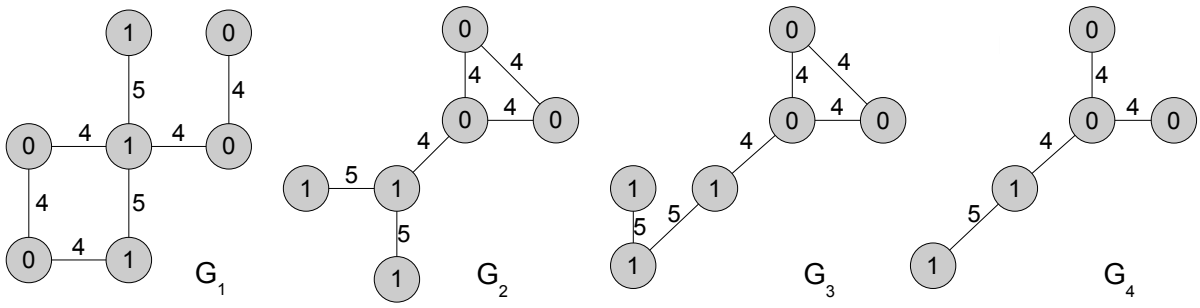
$G_4 \precsim G_2$, zatem $ES(G_4) \subseteq ES(G_2)$.

$G_4 \precsim G_3$, zatem $ES(G_4) \subseteq ES(G_3)$.

$ES(G_4) \subseteq ES(G_1)$, ale G_4 nie jest izomorficzny z podgrafem grafu G_1.

Definicja 2.25. *(wsparcie wierzchołka)*

Niech $\mathbb{D}$ będzie zbiorem grafów. Wsparciem wierzchołka v o etykiecie $lbl(v)$ nazywamy liczbę



Rysunek 2.6. Cztery nieskierowane grafy z etykietami.

grafów ze zbioru $\mathbb{D}$, które posiadają wierzchołek o etykiecie $lbl(v)$. Innymi słowy, wsparcie wierzchołka v jest równe wsparciu grafu o jednym wierzchołku v.

Definicja 2.26. *(wsparcie krawędzi)*
Niech $\mathbb{D}$ będzie zbiorem grafów. Wsparciem krawędzi e o deskryptorze $(\{lbl(v_1), lbl(v_2)\}, lbl(e))$ nazywamy liczbę grafów ze zbioru $\mathbb{D}$, których wielozbiory deskryptorów krawędzi zawierają deskryptor krawędzi e. Innymi słowy, wsparcie krawędzi e jest równe wsparciu grafu o jednej krawędzi e.

2.2. Problem odkrywania grafów częstych

Odkrywanie grafów częstych w danym zbiorze $\mathbb{D}$ polega na znalezieniu wszystkich nieizomorficznych grafów, których wsparcie w zbiorze $\mathbb{D}$ jest równe co najmniej wartości ustalonego progu wsparcia $minSup$. Efektywne odkrywanie grafów częstych uzależnione jest praktycznie od rozwiązania dwóch problemów: generowania potencjalnych kandydatów na grafy częste oraz wyznaczania ich wsparcia. W rozdziale 3 przedstawione zostaną sposoby rozwiązania tych problemów w znanych algorytmach odkrywania grafów częstych. Poniżej jest podany dokładniejszy opis tych problemów i ogólne metody ich rozwiązania.

2.2.1. Generowanie kandydatów

Generowanie kandydatów ma za zadanie utworzyć zbiór grafów, w którym znajdują się wszystkie grafy częste, ale mogą znajdować się też grafy nieczęste. Proces generowania tego zbioru nie musi być jednoetapowy i zwykle przeplata się z procesem wyznaczania wsparcia. Aby odkrywanie grafów częstych było efektywne, zbiór kandydatów powinien mieć dwie

n	t_n	n	t_n
1	1	11	12005168
2	1	12	1018997864
3	2	13	165091172592
4	4	14	50502031367952
5	11	15	29054155657235488
6	34	16	31426485969804308768
7	156	17	6400101570452755789 4928
8	1044	18	245935864153532932683719776
9	12346	19	1787577725145611700547878190848
10	274668	...	...

Tabela 2.1. Liczba klas równoważności t_n ze względu na izomorfizm grafu w zbiorze grafów nieskierowanych o n wierzchołkach.

własności: liczba grafów nieczęstych powinna być jak najmniejsza oraz liczba duplikatów powinna być jak najmniejsza.

Unikanie wytwarzania nieczęstych grafów kandydujących

W przypadku odkrywania grafów liczba potencjalnych kandydatów jest ogromna. Jeśli weźmiemy pod uwagę zbiór wszystkich grafów o n wierzchołkach i tylko jednej etykiecie, wtedy liczność tego zbioru wynosi $2^{\binom{n}{2}}$, a liczba grafów nieizomorficznych w tym zbiorze wynosi t_n, gdzie t_n jest n-tym wyrazem ciągu *Sloane A000088*[1], którego pierwsze wyrazy są podane w tabeli 2.1. Przy większej liczbie etykiet zbiór grafów będzie jeszcze liczniejszy. Generowanie każdego możliwego grafu byłoby nieefektywne. Wszystkie znane metody unikają generowania wszystkich możliwych grafów przez zastosowanie własności, że każdy podgraf grafu częstego też jest częsty, czyli grafy częste mogą powstać poprzez dodawanie wierzchołków i krawędzi wyłącznie do grafów częstych. Sposób tworzenia kandydatów na podstawie odkrytych wcześniej grafów częstych zależy od algorytmu. Wszystkie znane algorytmy zaczynają generowanie kandydatów od grafu bez wierzchołków (to znaczy sprawdzając, czy $|D| \geq minSup$), aby następnie, jeśli jest on częsty, rozszerzyć go w jakiś sposób (najczęściej o jedną krawędź) tworząc zbiór grafów kandydujących [13, 17, 25, 37,

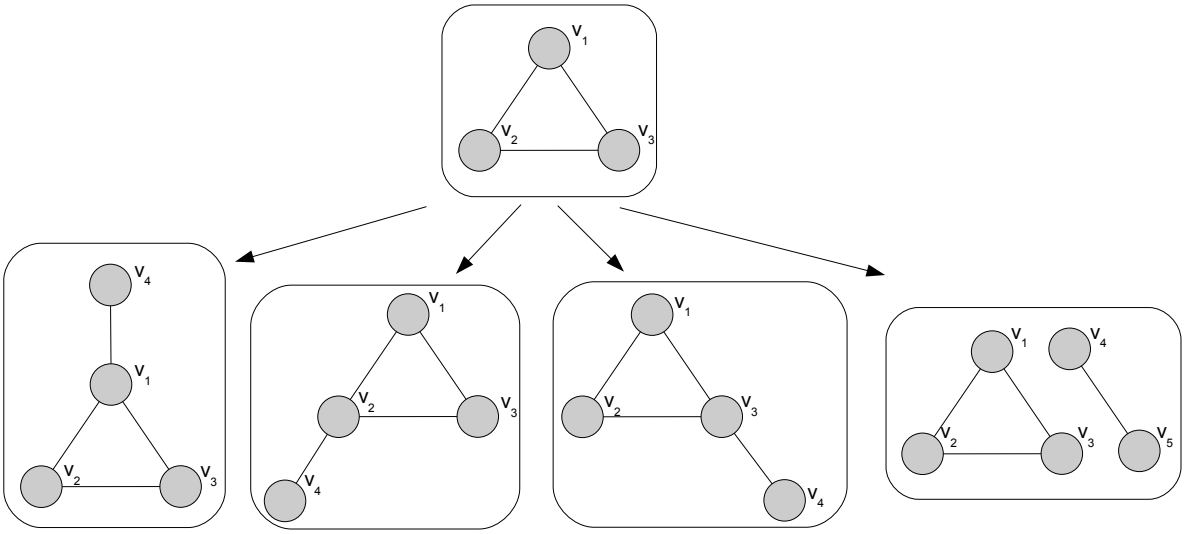
[1] Sloane A000088, http://www.research.att.com/ njas/sequences/A000088

39, 41, 47, 60, 72, 76]. Dla grafów kandydujących wyznaczane jest ich wsparcie i na podstawie tych kandydatów, którzy okazali się grafami częstymi, tworzy się kolejne grafy kandydujące. W ten sposób powstaje drzewo przeszukiwań grafów o korzeniu będącym grafem bez wierzchołków, którego każdy węzeł jest grafem częstym. Unikanie wytwarzania grafów nieczęstych polega przede wszystkim na generowaniu kandydatów na podstawie innych grafów częstych, ale różne algorytmy proponują dodatkowe mechanizmy służące temu celowi. W zaproponowanym w tej pracy algorytmie UGM zastosowane są trzy nowe metody, które zmniejszają liczbę wygenerowanych kandydatów nieczęstych.

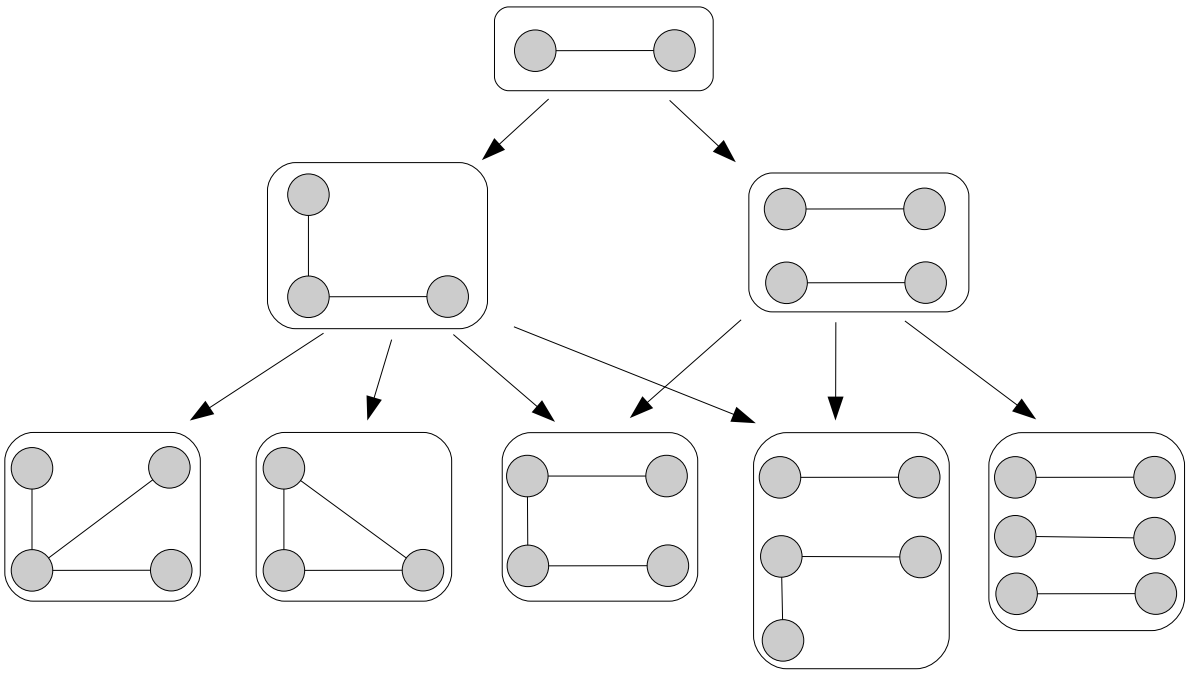
Unikanie wytwarzania duplikatów

Załóżmy, że chcemy wygenerować wszystkie nieizomorficzne grafy z jedną etykietą o n wierzchołkach. Z tabeli 2.1 wynika, że liczba nieizomorficznych grafów o pięciu wierzchołkach wynosi 34. Nie istnieje jednak prosta metoda na bezpośrednie generowanie nieizomorficznych grafów. Pomiędzy każdą parą wierzchołków może istnieć krawędź lub nie. Przy n wierzchołkach istnieje $\binom{n}{2}$ par wierzchołków, zatem liczba grafów uzyskanych w ten sposób wynosi $2^{\binom{n}{2}}$. Zbiór grafów o pięciu wierzchołkach ma zatem liczność 1024, a liczba nieizomorficznych grafów w tym zbiorze to tylko 34. Liczba duplikatów jest więc ogromna, a stosunek wszystkich grafów o n wierzchołkach do liczby nieizomorficznych grafów o n wierzchołkach dąży w granicy do nieskończoności. Problem jest znaczący także w przypadku generowania kandydatów w sposób opisany powyżej, czyli przez rozszerzenie innego grafu częstego. Na rysunku 2.7 pokazane jest rozszerzanie grafu poprzez dodanie jednej krawędzi. Aż trzy z czterech nowo powstałych grafów są izomorficzne. Generowanie kandydatów wymaga więc eliminacji lub unikania tworzenia duplikatów. Jeszcze trudniejszy przykład jest przedstawiony na rysunku 2.8. Ten sam graf może powstać w różnych miejscach drzewa przeszukiwań. Wykrywanie lub unikanie wytwarzania duplikatów nie jest problemem lokalnym, dotyczącym tylko grafów powstałych z danego grafu. Jest ono problemem globalnym - uzyskany graf kandydujący może być izomorficzny z grafem, który powstał na zupełnie innej ścieżce drzewa przeszukiwań.

Istnieją dwa podejścia do wykrywania duplikatów. Pierwsze polega na przechowywaniu wszystkich wygenerowanych grafów, zarówno częstych jak i nieczęstych, oraz wyszukiwaniu wśród nich nowo wygenerowanych grafów za pomocą testów na izomorfizm grafów. Aby uniknąć konieczności wykonywania testów na izomorfizm grafów z wszystkich przechowywanymi grafami, stosuje się struktury indeksujące grafy, na przykład tablice



Rysunek 2.7. Przykład powstawania duplikatów przy tworzeniu kandydatów. Trzy z czterech grafów wygenerowanych z górnego grafu są ze sobą izomorficzne.



Rysunek 2.8. Przykład powstawania duplikatów w różnych gałęziach drzewa przeszukiwań.

mieszające z kluczem będącym zbiorem deskryptorów krawędzi grafu. Drugie podejście jest związane z tak zwaną etykietą kanoniczną grafu. Etykieta kanoniczna grafu G jest obiektem jednoznacznie opisującym klasę równoważności ze względu na izomorfizm, do której należy graf G. Grafy izomorficzne mają tę samą etykietę kanoniczną, a grafy nieizomorficzne mają różne etykiety kanoniczne. Dodatkowo etykiety kanoniczne mają zdefiniowaną relację linowego porządku. Istnieją dwa podejścia do wykrywania duplikatów z zastosowaniem etykiet kanonicznych. Pierwsze polega na przechowywaniu uporządkowanego zbioru etykiet kanonicznych wszystkich wygenerowanych grafów nieizomorficznych i sprawdzaniu czy ten zbiór zawiera etykietę kanoniczną nowo utworzonego grafu. Jeśli zawiera, wtedy utworzony graf jest duplikatem. Jeśli nie zawiera, wtedy utworzony graf nie jest duplikatem, a jego etykieta kanoniczna jest dodawana do zbioru. W drugim podejściu do wykrywania duplikatów nie jest potrzebne przechowywanie zbioru wszystkich etykiet kanonicznych, gdyż kandydaci są generowani w taki sposób, aby unikać tworzenia duplikatów. Etykieta kanoniczna jest zwykle zdefiniowana jako minimalny kod spośród wszystkich kodów opisujących sposób powstawania grafu. W tym podejściu do wykrywania duplikatów, algorytm odkrywania grafów częstych musi gwarantować, że każdy graf częsty o kodzie minimalnym zostanie skonstruowany. Jeśli zatem podczas generowania grafu kandydującego okaże się, że jego kod nie jest minimalny, wtedy graf ten można odrzucić, gdyż albo został on już rozpatrzony albo będzie rozpatrzony później. Etykiety kanoniczne mogą być definiowane na wiele sposobów. Konkretne przykłady definiowania etykiet kanonicznych są podane w rozdziale 3 przy opisie istniejących algorytmów odkrywających grafy częste.

Etykiety kanoniczne są stosowane w większości algorytmów odkrywania grafów częstych. Natomiast wykorzystanie repozytorium grafów zamiast etykiet kanonicznych nie zostało dogłębnie zbadane. W [10] Borgelt zaproponował modyfikację algorytmu $MoFa$ [13], wykorzystującą repozytorium rozpatrzonych grafów kandydujących zamiast etykiet kanonicznych, i wykazał, że takie podejście może być w niektórych przypadkach szybsze.

2.2.2. Wyznaczanie wsparcia

Generalnie istnieją dwie metody wyznaczania wsparcia danego grafu G: wykonywanie bezpośrednich testów na izomorfizm grafu G z podgrafami grafów z wejściowej bazy grafów $\mathbb{D}$ oraz utrzymywanie listy zanurzeń.

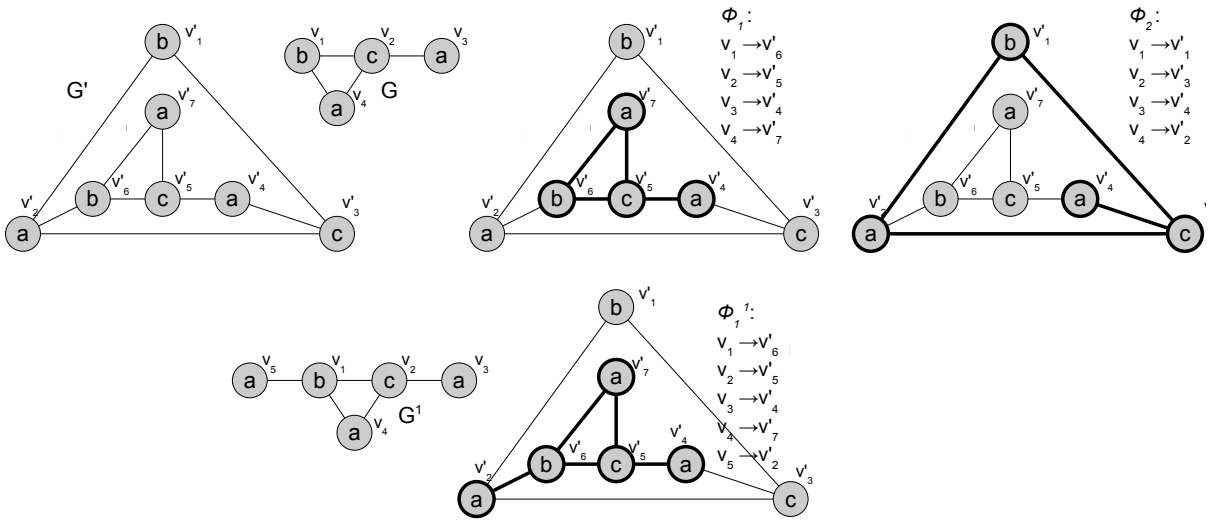
Testy na izomorfizm z podgrafem

W najprostszej postaci wyznaczenie wsparcia grafu G polega na wykonaniu testu na izomorfizm G z podgrafem każdego grafu ze zbioru $\mathbb{D}$. Liczba pozytywnych wyników będzie poszukiwanym wsparciem. Oczywistą optymalizacją jest wykorzystanie własności monotoniczności wsparcia, mówiącą o tym, że graf nie może mieć wsparcia większego niż wsparcie jego podgrafów. Zatem wyznaczenie wsparcia danego grafu G wystarczy ograniczyć do zbioru wspierającego graf, z którego graf G został skonstruowany, lub przecięcia zbiorów wspierających grafów, z których graf G został skonstruwany. W rozdziale 4.1.4 przedstawiam propozycję nowej optymalizacji polegającej na przerywaniu wyznaczania wsparcia dla grafów nieczęstych. Test na izomorfizm z podgrafem jest wykonywany niezależnie dla każdej pary (graf kandydujący, graf potencjalnie wspierający), zatem nie jest wymagający pamięciowo. Może być jednak czasochłonny, gdyż problem izomorfizmu z podgrafem jest NP-zupełny [30].

Utrzymywanie listy zanurzeń

Zgodnie z definicją 2.16 zanurzenie grafu G w grafie G' jest funkcją przyporządkowującą wierzchołki grafu G wierzchołkom pewnego izomorficznego z nim podgrafu grafu G'. Jeśli graf G nie posiada zanurzeń w grafie G', wtedy nie jest przez niego wspierany. Jeśli posiada przynajmniej jedno zanurzenie, wtedy jest wspierany. Szukanie wszystkich zanurzeń danego grafu G w grafie G' jest uogólnieniem testu na izomorfizm grafu G z podgrafem grafu G' i jest też bardziej czasochłonne, gdyż test na izomorfizm z podgrafem może być realizowany jako poszukiwanie tylko jednego, pierwszego zanurzenia. Przewagą zanurzeń jest jednak fakt, że w przypadku grafów tworzonych jako rozszerzenia danego grafu G, proste uaktualnienie zanurzeń grafu G pozwala na uzyskanie wszystkich zanurzeń jego rozszerzeń. Przykład uzyskiwania zanurzeń rozszerzenia jest przedstawiony na rysunku 2.9. Graf G posiada dokładnie dwa zanurzenia w grafie G' - ϕ_1 oraz ϕ_2. Graf G^1 powstaje z grafu G poprzez dodanie nowego wierzchołka a wraz z krawędzią łączącą go z istniejącym wierzchołkiem b. W zanurzeniu ϕ_1 możliwe jest dodanie nowej krawędzi do wierzchołka b, co w efekcie utworzy zanurzenie ϕ_1^1. W zanurzeniu ϕ_2 nie jest możliwe dodanie krawędzi do wierzchołka b, gdyż wszystkie krawędzie zawierające wierzchołek b należą już do zanurzenia. Graf G_1 posiada zatem tylko jedno zanurzenie ϕ_1^1 w grafie G'. Zatem G^1 jest wspierany przez graf G'.

Utrzymywanie listy zanurzeń jest kosztowne pamięciowo. W czasie wyznaczania wsparcia i generowania kandydatów z danego grafu G potrzebna jest lista wszystkich zanurzeń



Rysunek 2.9. Aktualizacja zanurzeń. ϕ_1 i ϕ_2 są zanurzeniami grafu G w grafie G'. Graf G^1 powstał z grafu G przez dodanie krawędzi. Tylko zanurzenie ϕ_1 pozwala na takie rozszerzenie.

grafu G we wszystkich wspierających go grafach. Z tego względu zanurzenia stosuje się praktycznie tylko w przypadku algorytmów przeszukiwania w głąb, dzięki czemu w każdej chwili wystarczy przechowywać zanurzenia tylko jednego grafu. Uaktualnianie zanurzeń jest stosunkowo szybkie i zależy liniowo od liczby zanurzeń. Dodatkową zaletą zanurzeń jest możliwość ich wykorzystania przy tworzeniu grafów kandydujących. Generowanie kandydatów z grafu G nie polega wtedy na wykonywaniu wszystkich możliwych rozszerzeń grafu G, ale wykonaniu wszystkich możliwych rozszerzeń jego zanurzeń, co gwarantuje, że uzyskani kandydaci będą mieli niepuste zbiory wspierające.

Utrzymywanie listy zanurzeń traci część swoich zalet w przypadku odkrywania grafów niespójnych ze względu na znaczny wzrost liczy zanurzeń, a tym samym wzrost zajętości pamięci oraz wzrost złożoności aktualizacji zanurzeń w przypadku rozszerzania grafu o nową składową spójną.

3. Znane algorytmy odkrywania częstych grafów

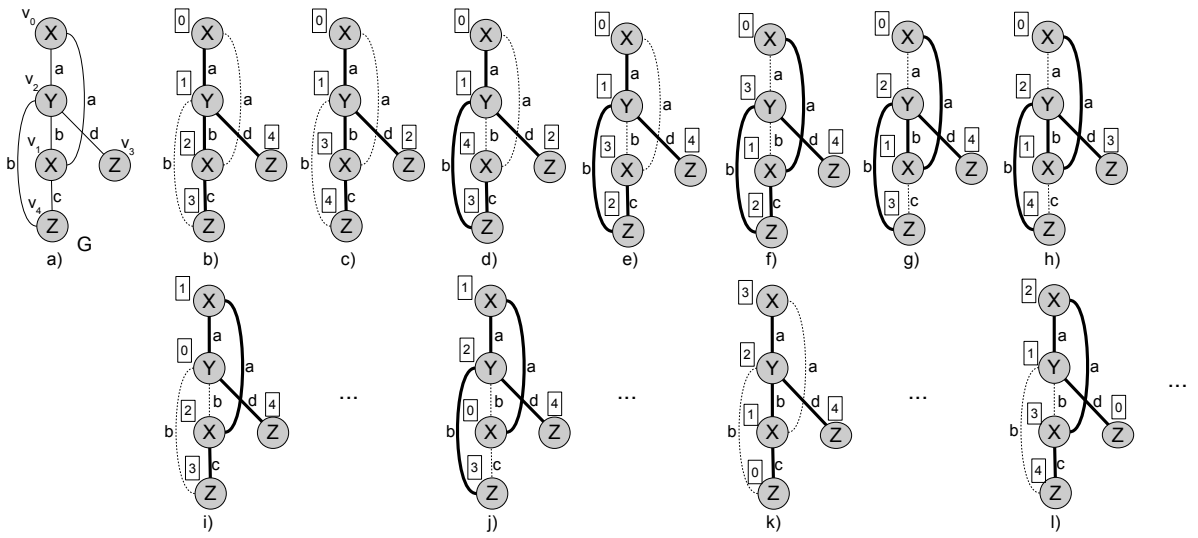
W tym rozdziale omówione są w pierwszej kolejności algorytmy pochodzące z platformy *ParMol* [55], czyli $gSpan$, $FFSM$, $gaston$ i $MoFa$. Na końcu rozdziału zostaną podane i krótko opisane pozostałe ważne algorytmy odkrywania częstych grafów.

3.1. gSpan

Algorytm gSpan [76] (graph-based Substructure pattern mining) odkrywa spójne grafy częste poprzez rekurencyjne rozszerzanie grafów częstych, zaczynając od częstych grafów o jednym wierzchołku. Wykrywanie i unikanie tworzenia duplikatów w drzewie przeszukiwań jest zapewnione przez specjalny rodzaj etykiety kanonicznej grafu - kod DFS.

3.1.1. Kod DFS

Kod DFS grafu G jest ściśle związany z pojęciem *drzewa rozpinającego w głąb* grafu G. Drzewem rozpinającym grafu G nazywamy drzewo, które zawiera wszystkie wierzchołki grafu G, a jego krawędzie należą do podzbioru krawędzi grafu G. Drzewo rozpinające w głąb jest drzewem rozpinającym, które powstaje podczas przeszukiwania w głąb danego grafu. W zależności od kolejności wyboru wierzchołków przy przeszukiwaniu w głąb może istnieć wiele drzew rozpinających w głąb. Drzewo rozpinające w głąb T może posłużyć do numerowania wierzchołków grafu G, w taki sposób, że wierzchołek od którego rozpoczęto przeszukiwanie w głąb ma numer 0, a każdy kolejny nowy wierzchołek ma numer o jeden większy. W analogiczny sposób możliwe jest ponumerowanie krawędzi. Numer wierzchołka v grafu G przy danym drzewie rozpinającym w głąb T będziemy oznaczać przez $t_T(v)$, a numer krawędzi e grafu G przy danym drzewie rozpinającym w głąb T będziemy oznaczać przez $t_T(e)$. Rysunek 3.1 przedstawia różne drzewa rozpinające w głąb grafu G. Rysunki 3.1 od b) do h) przedstawiają wszystkie drzewa rozpinające, które powstały przez przeszukiwanie w głąb rozpoczęte od wierzchołka v_0. Rysunki 3.1 od i) do l) przedstawiają po jednym przykładzie drzew rozpinających, które powstały przez przeszukiwanie w głąb



Rysunek 3.1. Drzewa rozpinające w głąb grafu G.

rozpoczęte od pozostałych wierzchołków. Liczba w ramce umieszczona przy wierzchołku oznacza numer t_T tego wierzchołka.

Definicja 3.1. *(kod DFS)*

Kod DFS grafu G (oznaczany przez $DFSCode_T(G)$) przy danym drzewie rozpinającym w głąb T jest ciągiem uporządkowanych według numeru t_T krawędzi grafu G, przy czym krawędź $e = \{v_i, v_j\}, t_T(v_i) < t_T(v_j)$ grafu G jest reprezentowana w kodzie jako piątka:

$$\begin{cases} (t_T(v_i), t_T(v_j), lbl(v_i), lbl(e), lbl(v_j)), & \text{jeśli krawędź } e \text{ należy do drzewa } T \\ (t_T(v_j), t_T(v_i), lbl(v_j), lbl(e), lbl(v_i)), & \text{w przeciwnym przypadku} \end{cases}$$

Przykład 3.1. Prześledźmy tworzenie przykładowego kodu DFS grafu G z rysunku 3.1 a). Tworzenie kodu DFS wykonuje się równocześnie z poszukiwaniem drzewa rozpinającego w głąb. Prześledźmy przypadek drzewa z rysunku 3.1 b). Algorytm przeszukiwania rozpoczyna się od wierzchołka v_0 i przechodzi krawędzią do wierzchołka v_2. Są to pierwsze odkryte wierzchołki oraz pierwsza odkryta krawędź, zatem $t_T(v_0) = 0, t_T(v_2) = 1, t_T(\{v_0, v_1\}) = 0$. Ta krawędź ma opis $(0, 1, X, a, Y)$. Z wierzchołka v_2 następuje przejście do wierzchołka v_1 o numerze $t_T(v_1) = 2$ krawędzią $(1, 2, Y, b, X)$ o numerze 1. Z wierzchołka v_1 można przejść do v_0, ale ten wierzchołek był już odwiedzony, więc krawędź łącząca v_1 z v_0 nie należy do drzewa rozpinającego, stąd opis tej krawędzi to $(2, 0, X, a, X)$. Kolejna nieodwiedzona krawędź wychodząca z v_1 prowadzi do v_4. Wierzchołek v_4 nie był jeszcze odwiedzany zatem $t_T(v_4) = 3$. Krawędź łącząca v_1 z v_4 ma opis $(2, 3, X, c, Z)$. Z wierzchołka v_4 do wierzchołka v_2 nie przechodzimy, gdyż v_2 był już odwiedzony, zatem

rys. 3.1b	rys. 3.1c	rys. 3.1d	rys. 3.1i	rys. 3.1j	rys. 3.1k	rys. 3.1l
(0,1,X,a,Y)	(0,1,X,a,Y)	(0,1,X,a,Y)	(0,1,Y,a,X)	(0,1,X,a,X)	(0,1,Z,c,X)	(0,1,Z,d,Y)
(1,2,Y,b,X)	(1,2,Y,d,Z)	(1,2,Y,d,Z)	(1,2,X,a,X)	(1,2,X,a,Y)	(1,2,X,b,Y)	(1,2,Y,a,X)
(2,0,X,a,X)	(1,3,Y,b,X)	(1,3,Y,b,Z)	(2,0,X,b,Y)	(2,0,Y,b,X)	(2,0,Y,b,Z)	(2,3,X,a,X)
(2,3,X,c,Z)	(3,0,X,a,X)	(3,4,Z,c,X)	(2,3,X,c,Z)	(2,3,Y,b,Z)	(2,3,Y,a,X)	(3,1,X,b,Y)
(3,1,Z,b,Y)	(3,4,X,c,Z)	(4,0,X,a,X)	(3,0,Z,b,Y)	(3,0,Z,c,X)	(3,1,X,a,X)	(3,4,X,c,Z)
(1,4,Y,d,Z)	(4,1,Z,b,Y)	(4,1,X,b,Y)	(0,4,Y,d,Z)	(2,4,Y,d,Z)	(2,4,Y,d,Z)	(4,1,Z,b,Y)

Tabela 3.1. Kody DFS grafu G z rysunku 3.1 dla drzew rozpinających z rysunków 3.1 b)-3.1 l)

krawędź łącząca te wierzchołki ma opis $(3, 1, Z, b, Y)$. Nie ma innych nieodwiedzonych krawędzi wychodzących z wierzchołka v_4, więc następuje powrót powrót do wierzchołka v_1. Nie ma nieodwiedzonych krawędzi z v_1, więc następuje powrót do v_2. Z wierzchołka v_2 można przejść do nieodwiedzonego jeszcze wierzchołka v_3. Krawędź łącząca te wierzchołki ma opis $(1, 4, Y, d, Z)$. Nie ma innych nieodwiedzonych krawędzi z v_3, więc następuje powrót do wierzchołka v_2, następnie powrót do wierzchołka v_0 i przeszukiwanie w głąb się kończy. Kod DFS dla grafu G jest ciągiem kolejno odkrywanych krawędzi:

$$\begin{aligned} DFSCode_T(G) = \; & ((0,1,X,a,Y),\\ & (1,2,Y,b,X),\\ & (2,0,X,a,X),\\ & (2,3,X,c,Z),\\ & (3,1,Z,b,Y),\\ & (1,4,Y,d,Z)) \end{aligned}$$

3.1.2. Relacja porządkująca w zbiorze kodów DFS i minimalny kod DFS

Zbiór kodów DFS można uporządkować stosując klasyczny porządek leksykograficzny dla ciągów, to znaczy $(a_1, a_2, \ldots, a_n) < (b_1, b_2, \ldots, b_m)$ wtedy i tylko wtedy, gdy istnieje $k \leq min(n, m)$ takie, że $\forall_{i<k}\ a_i = b_i \wedge a_k < b_k$. W przypadku kodów DFS elementy ciągów (a_n) oraz (b_n) są piątkami $(t_T(v_i), t_T(v_j), lbl(v_i), lbl(e), lbl(v_j))$, zatem konieczne jest ustalenie relacji porządkującej piątki, co również można osiągnąć stosując porządek leksykograficzny: $(a_1, a_2, \ldots, a_5) < (b_1, b_2, \ldots, b_5)$ wtedy i tylko wtedy, gdy istnieje $k \leq 5$ takie, że $\forall_{i<k}\ a_i = b_i \wedge a_k < b_k$. Stosując ten porządek dla kodów z tabeli 3.1 uzyskujemy następującą kolejność: najmniejszym kodem DFS jest kod z rysunku 3.1 j), następnym w kolejności jest kod z rysunku 3.1 b), potem kolejno z rysunków 3.1 c), 3.1 d), 3.1 i), 3.1 k) i 3.1 l).

Definicja 3.2. *(minimalny kod DFS)*
Minimalnym kodem DFS grafu G, oznaczanym przez $min(G)$, nazywamy najmniejszy (według podanej wyżej relacji porządkującej) spośród wszystkich kodów DFS grafu G.

Minimalny kod DFS ma własność etykiety kanonicznej grafu. Oznacza to, że dwa izomorficzne grafy mają ten sam minimalny kod DFS, dwa nieizomorficzne grafy mają różne minimalne kody DFS, oraz istnieje relacja porządkująca wśród minimalnych kodów DFS.

3.1.3. Drzewo kodów DFS

Definicja 3.3. *(rozszerzenie kodu DFS)*
Rozszerzeniem danego kodu $DFSCode_T(G) = (a_0, a_1, \ldots a_m)$ nazywamy każdy ciąg w postaci $(a_0, a_1, \ldots a_m, b)$. Rozszerzenie kodu $DFSCode_T(G)$ nazywamy poprawnym, jeżeli jest ono kodem DFS grafu G rozszerzonego o jedną krawędź.

Jeżeli kod DFS α reprezentuje graf G, wtedy poprawne rozszerzenie kodu α reprezentuje rozszerzenie grafu G o jedną krawędź. Autorzy algorytmu $gSpan$ dowodzą, że aby rozszerzenie kodu było poprawne, graf G musi być rozszerzany w określony sposób: krawędź może być dodawana jedynie do wierzchołków z prawej ścieżki drzewa rozpinającego grafu, to znaczy do najkrótszej ścieżki łączącej wierzchołek o numerze t_T równym 0, a wierzchołkiem o największym numerze t_T. Liczba rozszerzeń kodu DFS danego grafu G jest zatem mniejsza niż liczba rozszerzeń grafu G (przy rozszerzaniu o jedną krawędź).

Dany kod DFS może mieć wiele rozszerzeń, zatem wszystkie kody tworzą drzewo. Algorytm $gSpan$ odkrywa częste grafy przez przeszukiwanie w głąb drzewa kodów DFS. Przeszukiwanie drzewa kończy się w węzłach o nieminimalnych kodach oraz w węzłach nieczęstych.

3.1.4. Algorytm gSpan

Wykonanie algorytmu $gSpan$ (algorytm 3.1) rozpoczyna się od wyznaczenia wsparcia pojedynczych wierzchołków oraz pojedynczych krawędzi. Wierzchołki i krawędzie są następnie sortowane według nierosnącego wsparcia. Te, których wsparcie jest poniżej progu $minSup$, są usuwane z grafów zbioru $\mathbb{D}$. Wierzchołki i krawędzie są potem przeetykietowywane w taki sposób, aby porządek wyznaczany z relacji etykiet odpowiadał porządkowi nierosnącego wsparcia. Następnie odkrywane są grafy częste składające się z jednej krawędzi. Grafy te zostają umieszczone w zbiorze F_1 oraz w zbiorze wynikowym

R. Dla każdego takiego grafu wyznaczany jest jego zbiór wspierający $G.supportingSet$ oraz minimalny kod DFS $G.code$, który w tym przypadku będzie jednym z dwóch możliwych, gdyż każdy graf ma tylko dwa wierzchołki. Zbiór F_1 jest sortowany ze względu na pole $G.code$. Następnie ze zbioru F_1 pobierane są kolejne grafy. Dla każdego grafu $G \in F_1$ wywoływana jest procedura $gSpanSubgraphMining$, która odkrywa wszystkie grafy częste, które mogą powstać przez rozszerzanie kodu DFS grafu G. Wynik procedury jest dodawany do R. Następnie z każdego grafu zbioru $\mathbb{D}$ zostają usunięte krawędzie, których deskryptor jest identyczny z deskryptorem jedynej krawędzi grafu G. Usunięcie tych krawędzi nie zmieni wyniku odkrywania grafów częstych, gdyż wszystkie grafy częste zawierające krawędzie o tym deskryptorze już zostały znalezione. Po usunięciu krawędzi, niektóre grafy mogą mieć puste zbiory wierzchołków i należy je usunąć ze zbioru $\mathbb{D}$. Jeśli liczność zbioru $\mathbb{D}$ stanie się mniejsza niż $minSup$ dalsze odkrywanie grafów jest przerywane.

Procedura $gSpanSubgraphMining(G, minSup)$ odkrywa wszystkie częste grafy, które powstają z rozszerzanie kodu DFS grafu G. Na początku sprawdzane jest czy kod DFS $G.code$ jest minimalny. Jeśli kod nie jest minimalny procedura się kończy, gdyż nieminimalny kod oznacza, że graf izomorficzny z G już powstał w innej gałęzi drzewa przeszukiwań przestrzeni grafów. W przypadku kodu minimalnego, graf G jest dodawany do zbioru grafów częstych oraz wywoływana jest procedura $gSpanChildren(G)$, która zwraca grafy o kodach będących rozszerzeniami kodu DFS grafu G. Dla każdego częstego grafu G_c z tak uzyskanego zbioru wywoływana jest rekurencyjnie procedura $gSpanSubgraphMining(G', minSup)$.

Procedura $gSpanChildren(G)$ zwraca wszystkie grafy, które mogą powstać z grafu G poprzez dodanie jednej krawędzi do prawej ścieżki drzewa rozpinającego w głąb, gdyż tylko takie grafy mają kod DFS będący rozszerzeniem kodu DFS grafu G. Dla każdego grafu G', ze zbioru wspierającego graf G, czyli $G.supportingSet$, generowane są rozszerzenia grafu G będące grafami izomorficznymi z podgrafem grafu G'. Autorzy wskazują, że do tego celu może posłużyć dowolny algorytm izomorfizmu z podgrafem, ale sami proponują własny algorytm wyszukujący wszystkie zanurzenia grafu G w grafie G'. Zaproponowany algorytm działa w sposób następujący: na początku znajdowane są wszystkie zanurzenia pierwszej krawędzi kodu $G.code$. Następnie zanurzenia rozszerzane są, jeśli to możliwe, o kolejną krawędź z kodu $G.code$ i tak dalej, aż zostaną dodane wszystkie krawędzie kodu $G.code$. W ten sposób znalezione zostają wszystkie zanurzenia grafu G. Zanurzenia są następnie rozszerzane o jedną krawędź, ale tylko w wierzchołkach znajdujących się na prawej ścieżce drzewa rozpinającego

w głąb, aby uzyskane rozszerzenia były poprawne, to znaczy były kodami DFS. W efekcie uzyskuje się rozszerzenia grafu G wspierane przez graf G'.

Do opisanego algorytmu można wprowadzić kilka dodatkowych optymalizacji, których celem jest przede wszystkim unikanie generowania rozszerzeń o nieminimalnych kodach. Szczegóły znajdują się w [76]. Zaimplementowany w platformie $ParMol$ algorytm posiada wszystkie optymalizacje.

Algorytm 3.1 gSpan($\mathbb{D}, minSup$)

posortuj i przeetykietuj wierzchołki i krawędzie grafów z $\mathbb{D}$ wg nierosnącego wsparcia;
usuń nieczęste wierzchołki i nieczęste krawędzie ze wszystkich grafów zbioru $\mathbb{D}$;
$F_1 \leftarrow$ częste grafy o jednej krawędzi;
posortuj F_1 wg kodów DFS;
$R \leftarrow F_1$;
for all $G \in F_1$ **do**
  $R \leftarrow R \cup gSpanSubgraphMining(G, minSup)$
  usuń z grafów $\mathbb{D}$ krawędzie o deskryptorach takich samych jak deskryptor jedynej krawędzi grafu G;
  usuń ze zbioru $\mathbb{D}$ grafy o pustych zbiorach wierzchołków;
  if $|\mathbb{D}| < minSup$ **then**
    break;
  end if
end for
return R;

Algorytm 3.2 gSpanSubgraphMining(G,minSup)

if $G.code$ nie jest minimalny **then**
  return $\emptyset$;
end if
$R \leftarrow \emptyset$;
$R \leftarrow R \cup \{G\}$;
for all $G_c \in gSpanChildren(G)$ **do**
  if $|G_c.supportingSet| \geq minSup$ **then**
    $R \leftarrow R \cup gSpanSubgraphMining(G_c)$;
  end if
end for
return R;

3.1.5. Odkrywanie niespójnych grafów częstych

W pracy [77] została zasygnalizowana możliwość modyfikacji algorytmu $gSpan$, tak aby odkrywane były również częste grafy niespójne. W tej pracy ten algorytm będziemy nazywać *gSpanUnconnected*. Algorytm *gSpanUnconnected* wymaga przedefiniowania pojęcia

Algorytm 3.3 gSpanChildren(G)

$R \leftarrow \emptyset$;
for all $G' \in G.supportingSet$ **do**
  for all graf $G_c \precsim G'$ taki, że kod DFS $G_c.code$ jest rozszerzeniem kodu $G.code$ **do**
    $G_c.supportingSet \leftarrow G_c.supportingSet \cup \{G'\}$;
    $R \leftarrow R \cup \{G_c\}$;
  end for
end for
return R;

kodu DFS grafu. Każdy graf G jest reprezentowany jako zbiór składowych spójnych $G = \{CG_1, CG_2, \ldots, CG_n\}$. Kod DFS grafu G jest reprezentowany jako uporządkowana lista kodów DFS jego spójnych składowych. Kod DFS grafu G jest minimalny, gdy kody DFS wszystkich jego składowych spójnych są minimalne. Kod DFS $s = (s_0, s_1, \ldots s_n), s_0 \leq s_1 \leq \ldots \leq s_n$ może być rozszerzony na dwa sposoby:

— dodanie do listy s kodu $s_{n+1} \geq s_n$, który reprezentuje graf o jednej krawędzi;

— rozszerzanie kodów $s_0, \ldots s_n$ w taki sam sposób jak w algorytmie $gSpan$

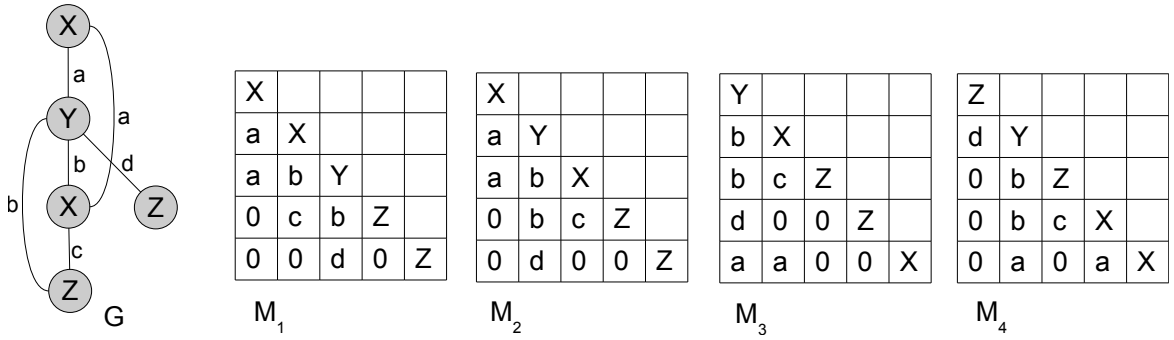
3.2. FFSM

Algorytm $FFSM$ [37] (Fast Frequent Subgraph Mining) odkrywa grafy częste poprzez zarówno rozszerzanie jak i łączenie wcześniej odkrytych grafów częstych. Grafy są reprezentowane w postaci pewnego rodzaju macierzy sąsiedztwa (ang. adjacency matrix), a unikanie tworzenia duplikatów jest zapewnione przez odrzucanie grafów, których reprezentacja nie jest w postaci postaci kanonicznej macierzy sąsiedztwa (ang. canonical adjacency matrix). Do wyznaczania wsparć grafów kandydujących wykorzystywana jest lista ich zanurzeń w grafach wejściowej bazy grafów.

3.2.1. Kanoniczna macierz sąsiedztwa

Definicja 3.4. *(macierz sąsiedztwa)*
Macierz sąsiedztwa grafu G o n wierzchołkach, oznaczana jako $M(G)$, jest macierzą $n \times n$, w której:

— przekątna, czyli elementy $m_{ii}, 0 < i < n$, zawiera etykiety wierzchołków grafów; $m_{ii} = lbl(v_i)$;



Rysunek 3.2. Cztery wybrane macierze sąsiedztwa grafu G.

— część pod przekątną, czyli elementy $m_{ij}, 0 < j < i < n$, zawierają etykiety krawędzi pomiędzy wierzchołkami i oraz j, lub 0, jeśli krawędź nie istnieje;

— część nad przekątną, czyli elementy $m_{ij}, 0 < i < j < n$, jest pusta;

Element m_{ij} jest elementem macierzy $M(G)$ znajdującym się w $i-tym$ wierszu i $j-tej$ kolumnie. Ponieważ elementy ponad przekątną są zawsze równe 0 macierz $M(G)$ jest macierzą trójkątną.

Każdy graf o n wierzchołkach posiada co najwyżej $n!$ różnych macierzy sąsiedztwa, co wynika z liczby permutacji wierzchołków znajdujących się na przekątnej macierzy. Na rysunku 3.2 przedstawione są cztery przykładowe macierze sąsiedztwa grafu G.

Definicja 3.5. *(kod macierzy sąsiedztwa)*
Kodem macierzy sąsiedztwa M, oznaczanym jako $code(M)$, jest ciąg elementów macierzy M w następującej kolejności: $m_{00}, m_{10}, m_{11}, m_{20}, m_{21}, m_{22}, \ldots, m_{n0}, \ldots m_{nn}$.

Definicja 3.6. *(CAM, kanoniczna macierz sąsiedztwa)*
Kanoniczną macierzą sąsiedztwa (ang. canonical adjacency matrix) grafu G, oznaczaną jako $CAM(G)$, jest ta spośród wszystkich macierzy sąsiedztwa grafu G, która ma największy kod ze względu na porządek leksykograficzny.

Kod kanonicznej macierzy sąsiedztwa spełnia warunki etykiety kanonicznej.

Przykład 3.2. Na rysunku 3.2 przedstawione są cztery wybrane macierze sąsiedztwa grafu G. Wszystkich macierzy sąsiedztwa jest $5! = 120$. Kody przedstawionych macierzy to kolejno:
$code(M_1) = (X, a, X, a, b, Y, 0, c, b, Z, 0, 0, d, 0, Z)$
$code(M_2) = (X, a, Y, a, b, X, 0, b, c, Z, 0, d, 0, 0, Z)$
$code(M_3) = (Y, b, X, b, c, Z, d, 0, 0, Z, a, a, 0, 0, X)$

$code(M_4) = (Z, d, Y, 0, b, Z, 0, b, c, X, 0, a, 0, a, X)$

$code(M_1) < code(M_2) < code(M_3) < code(M_4)$.

Kod macierzy M_4 jest największy z kodów wszystkich możliwych macierzy sąsiedztwa. M_4 jest kanoniczną macierzą sąsiedztwa grafu G. $CAM(G) = M_4$.

Definicja 3.7. *(maksymalna podmacierz właściwa)*
Niech M będzie macierzą sąsiedztwa grafu G o n wierzchołkach, a K będzie macierzą, która powstaje z M przez wyzerowanie ostatniego niezerowego elementu z ostatniego wiersza macierzy M, który nie leży na przekątnej. Maksymalną podmacierzą właściwą macierzy M nazywamy macierz K, jeśli co najmniej jeden element z ostatniego wierzsza macierzy K, który nie należy do przekątnej macierzy K, jest niezerowy. W przeciwnym przypadku maskymalną podmacierzą właściwą macierzy M jest macierz powstała z K poprzez usunięcie ostatniego wiersza i ostatniej kolumny macierzy K.

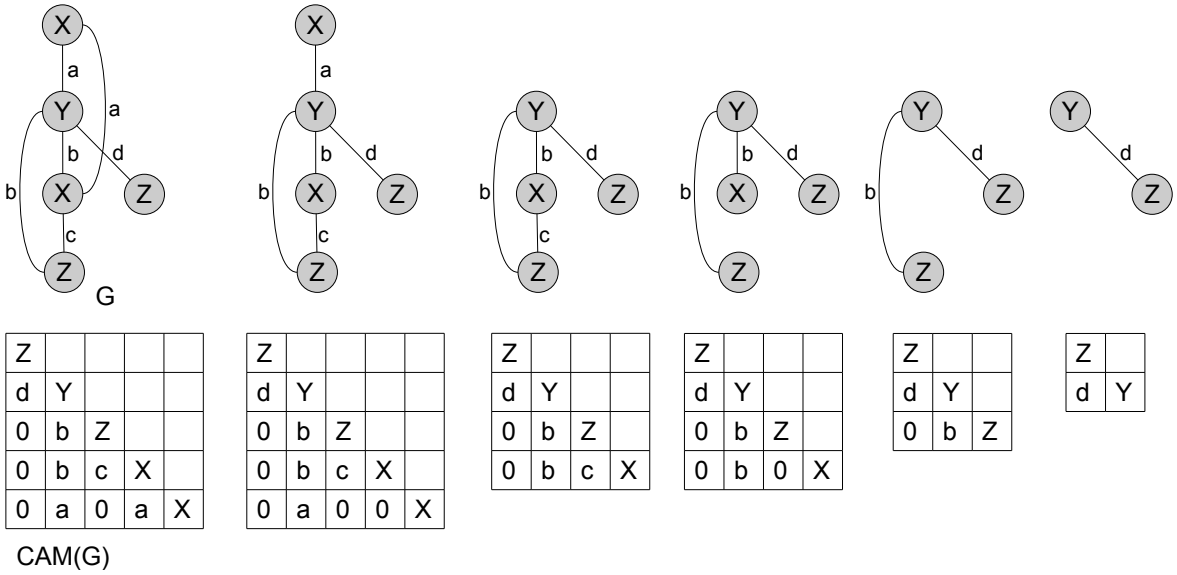
Jeśli macierz M jest macierzą sąsiedztwa grafu G, wtedy maksymalna podmacierz właściwa K macierzy M reprezentuje graf G^-, który powstał przez usunięcie jednej krawędzi z grafu G. Jeśli $M = CAM(G)$ wtedy $K = CAM(G^-)$.

Na rysunku 3.3 przedstawiony jest graf G wraz z jego macierzą $CAM(G)$ oraz łańcuch kolejnych maksymalnych podmacierzy właściwych. Każdy kolejny graf ma o jedną krawędź mniej. Podczas odkrywania grafów częstych będziemy budować tę relację w odwrotnej kolejności, przy czym efektem nie będzie łańcuch macierzy CAM lecz drzewo macierzy CAM, gdyż dwie różne macierze CAM mogą mieć tę samą maksymalną podmacierz właściwą. Przykładowe drzewo złożone z macierzy CAM znajduje się na rysunku 3.4.

Definicja 3.8. *(suboptymalna kanoniczna macierz sąsiedztwa, suboptymalna CAM)*
Macierz sąsiedztwa M jest suboptymalną kanoniczną macierzą sąsiedztwa (suboptymalną CAM), gdy nie jest kanoniczną macierzą sąsiędztwa, ale jej maksymalna podmacierz właściwa jest kanoniczną macierzą sąsiedztwa.

3.2.2. Generowanie kandydatów

Autorzy $FFSM$ zdefiniowali dwie operacje: rozszerzania grafu $FFSM\_Extension$ oraz łączenia grafów $FFSM\_Join$, które łącznie pozwalają na zbudowanie drzewa zawierającego macierze CAM oraz suboptymalne macierze CAM, rozpoczynając od grafu z pustym zbiorem wierzchołków.

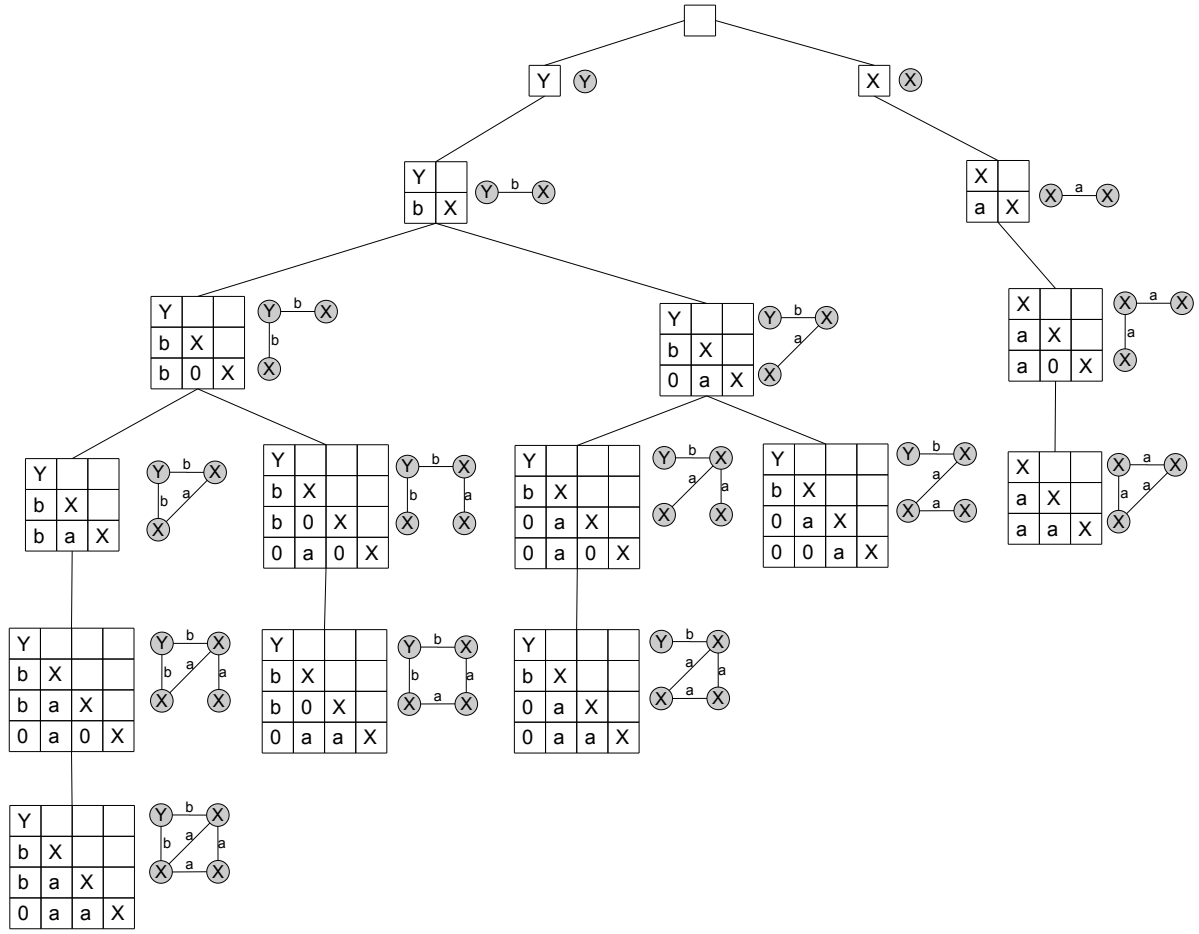


Rysunek 3.3. Maksymalne podmacierze właściwe oraz grafy przez nie reprezentowane. Idąc od lewej do prawej każda macierz jest maksymalną podmacierzą właściwą macierzy poprzedniej.

Operacje rozszerzania i e łączenia do powstawania zarówno CAM jak i suboptymalnych CAM, ale tylko CAM są kandydatami na grafy częste. Jeśli kandydat okaże się nieczęsty nie jest już wykorzystywany do generowania kolejnych kandydatów. Autorzy dowodzą, że takie generowanie kandydatów nie pomija żadnych grafów częstych.

Operacja FFSM_Join

Operacja $FFSM\_Join$ generuje zbiór nowych grafów poprzez połączenie ze sobą dwóch grafów G_1 i G_2. Wynikowy zbiór zawiera maksymalnie dwa grafy. Według algorytmu grafy G_1 i G_2 mogą być złączone tylko wtedy, gdy ich macierze sąsiedztwa mają dokładnie tę samą maksymalną podmacierz sąsiedztwa. Autorzy algorytmu $FFSM$ definiują trzy przypadki złączeń, które schematycznie są przedstawione na rysunku 3.5. Rodzaj złączenia jest wybierany w zależności od rodzaju łączonych macierzy sąsiedztwa. Autorzy definiują dwa rodzaje macierzy - macierz wewnętrzną oraz macierz zewnętrzną. Macierz wewnętrzna to macierz, która w ostatnim wierszu posiada co najmniej dwie krawędzie. Macierz zewnętrzna to macierz, która w ostatnim wierszu posiada tylko jedną krawędź. Pierwszy przypadek operacji $FFSM\_Join$, oznaczony na rysunku 3.5 jako $j.1$, dotyczy łączenia macierzy wewnętrznych. W tym przypadku macierzą wynikową jest macierz pierwsza, w której zamiast elementów zerowych wystepują elementy z tej samej pozycji z macierzy drugiej. Drugi przypadek - $j.2$ - dotyczy łączenia macierzy wewnętrznej z macierzą zewnętrzną. W tym przypadku macierzą



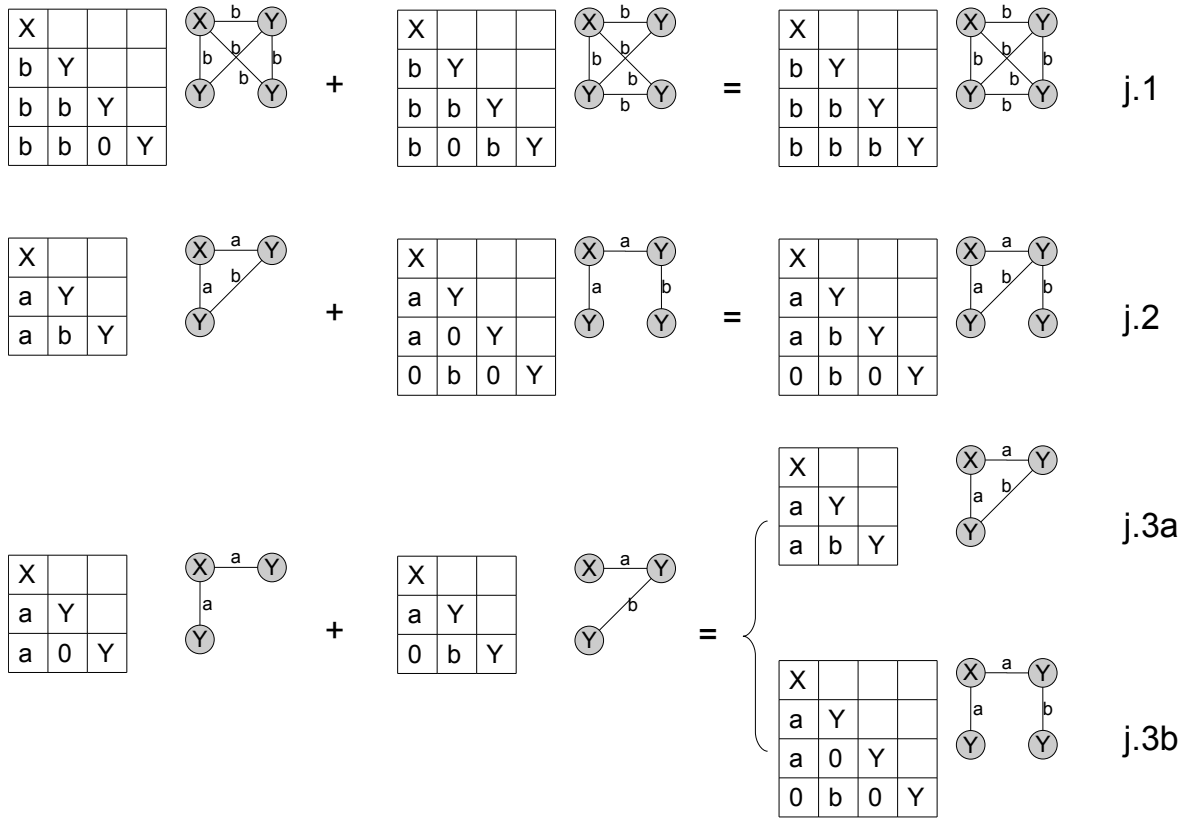
Rysunek 3.4. Drzewo złożone z CAM utworzone na podstawie przykładowego zbioru grafów. Krawędzie drzewa oznaczają relację typu maksymalna podmacierz właściwa.

wynikową jest macierz pierwsza, w której dodano ostatni wiersz i kolumnę z wartościami pobranymi z macierzy drugiej. Trzeci przypadek składa się z dwóch wariantów - $j.3a$ i $j.3b$ - dotyczy łączenia macierzy zewnętrznych. W przypadku $j.3a$ macierze łączy się tak samo jak w złączeniu $j.1$, a w przypadku $j.3b$ - tak jak w przypadku $j.2$ z tą różnicą, że w dodawanym do macierzy wierszu i kolumnie należy dostawić 0 przed ostatnim elementem.

Operacja FFSM_Extension

Operacja $FFSM\_Extension$ generuje zbiór nowych grafów przez dodanie do grafu G jednej krawędzi. Aby rozszerzenie macierzy było możliwe musi ona być macierzą zewnętrzną, czyli posiadającą tylko jedną krawędź w ostatnim wierszu. Rozszerzenie macierzy polega na dodaniu jednego nowego wierzchołka i połączeniu go krawędzią z ostatnim wierzchołkiem macierzy. Operacja rozszerzenia generuje tyle nowych macierzy, ile jest różnych par (etykieta wierzchołka, etykieta krawędzi). Rozszerzenie macierzy o wielkości $n \times n$ o wierzchołek

o etykiecie $lbl(v)$ i krawędź o etykiecie $lbl(e)$ powoduje powstanie nowej macierzy $(n+1) \times (n+1)$ poprzez dodanie jednego wiersza i jednej krawędzi. Dodany wiersz składa się z samych zer z wyjątkiem dwóch ostatnich elementów, którymi są $lbl(e)$ i $lbl(v)$.



Rysunek 3.5. Trzy przypadki łączenia grafów w operacji FFSM_Join.

3.2.3. Wyznaczanie wsparcia

Algorytm $FFSM$ przechowuje listę zanurzeń, która jest aktualizowana w czasie wykonywania operacji $FFSM\_Join$ i $FFSM\_Extensions$. Wsparciem grafu kandydującego G_c jest liczba takich grafów G' z wejściowej bazy grafów, że G_c posiada zanurzenie w G'. Zanurzenia służą dodatkowo jako wskazówka do generowania rozszerzeń w operacji $FFSM\_Extensions$. Dzięki temu operacja rozszerzania nie musi generować wszystkich kandydatów, które wynikają z liczby par (etykieta wierzchołka, etykieta krawędzi), a tylko te, które występują w zanurzeniach.

3.2.4. Algortym FFSM

Wykonanie algorytmu $FFSM$ (algorytm 3.4) rozpoczyna się od znalezienia częstych grafów o jednej krawędzi oraz częstych grafów o jednym wierzchołku. Grafy reprezentowane są w postaci kanonicznych macierzy sąsiedztwa CAM. Zbiór częstych grafów o jednym wierzchołku zostaje dodany do zbioru wynikowego R. Grafy o jednej krawędzi są przekazywane do procedury $FFSM\_Explore$, której wynik jest dodawany do zbioru wynikowego R.

Procedura $FFSM\_Explore(P)$ zwraca wszystkie grafy częste, które powstają przez rozszerzanie i łączenie grafów ze zbioru P. Grafy ze zbioru P, których macierze sąsiedztwa nie są kanoniczne, są pomijane. Wszystkie pozostałe zostają umieszczone w zbiorze wynikowym. Określenie, czy macierz jest kanoniczna, wymaga utworzenia wszystkich macierzy sąsiedztwa danego grafu i znalezienia macierzy o największym kodzie. Tworzony jest także zbiór nowych kandydatów C. Do zbioru C zostają dodane grafy powstałe w wyniku operacji $FFSM\_Join$ na każdej parze macierzy z zbioru P, w której co najmniej jedna macierz jest CAM, oraz grafy powstałe w wyniku operacji $FFSM\_Extensions$ na każdej CAM ze zbioru P. Operacje te wyznaczają jednocześnie wsparcie powstałych kandydatów. Kandydaci nieczęści oraz kandydaci, których macierze sąsiedztwa nie są ani kanoniczne ani suboptymalne są usuwanie ze zbioru C. Następnie rekurencyjnie jest wywoływana procedura $FFSM\_Explore(C)$, a jej wynik jest dodawany do zbioru wynikowego.

Algorytm 3.4 FFSM($\mathbb{D}, minSup$)

$R \leftarrow CAM$ częstych grafów o jednym wierzchołku;
$P \leftarrow CAM$ częstych grafów o jednej krawędzi;
$R \leftarrow R \cup FFSM\_Explore(P)$;
return R;

3.3. MoFa

Algorytm $MoFa$ [13] (Molecule Fragment miner), znany też pod nazwą $MoSS$ [12] (Molecular Substructure Mining), odkrywa spójne grafy częste poprzez rekurencyjne rozszerzanie grafów częstych o jedną krawędź. Algorytm wykorzystuje zanurzenia zarówno do wyznaczania wsparcia, jak i generowania kandydatów, co gwarantuje, że mają one niezerowe wsparcia. Do ograniczenia powstawania duplikatów stosowana jest prosta zasada mówiąca,

Algorytm 3.5 FFSM_Explore(zbiór macierzy sąsiedztwa P)

```
R ← ∅;
for all X ∈ P do
  if X.isCAM() then
    R ← R ∪ {X};
    C ← ∅;
    for all Y ∈ P do
      C ← C ∪ FFSM_Join(X, Y);
    end for
    C ← C ∪ FFSM_Extension(X);
    w zbiorze C pozostaw tylko częste CAM i częste suboptymalne CAM;
    R ← R ∪ FFSM_Explore(C);
  end if
end for
return R;
```

że nie trzeba rozszerzać grafu przez dodanie krawędzi do wierzchołka o numerze k, jeśli wcześniej graf był rozszerzany przez dodanie krawędzi do wierzchołka o numerze wyższym niż k. Ten mechanizm eliminuje powstawianie duplikatów tylko w pewnym stopniu. W zależności od wersji, algorytm albo zupełnie ignoruje problem powstawania duplikatów i dopuszcza występowanie grafów izomorficznych w wynikowym zbiorze grafów częstych, albo wykrywa je za pomocą testów na izomorfizm grafu lub etykiet kanonicznych.

3.3.1. Generowanie kandydatów

Generowanie kandydatów polega na rozszerzaniu zanurzeń danego grafu częstego o jedną krawędź. Każdy graf częsty ma ponumerowane wierzchołki w takiej kolejności, w jakiej były do niego dodawane. Z każdym grafem częstym G związany jest też numer wierzchołka, nazwijmy go $lastV$, którego wartość jest wyznaczona w następujący sposób:

— jeśli graf G powstał przez rozszerzenie grafu G^- polegające na dodaniu nowej krawędzi pomiędzy dwa istniejące wierzchołki grafu G^-, wtedy $lastV$ przyjmuje większy spośród numerów tych wierzchołków;

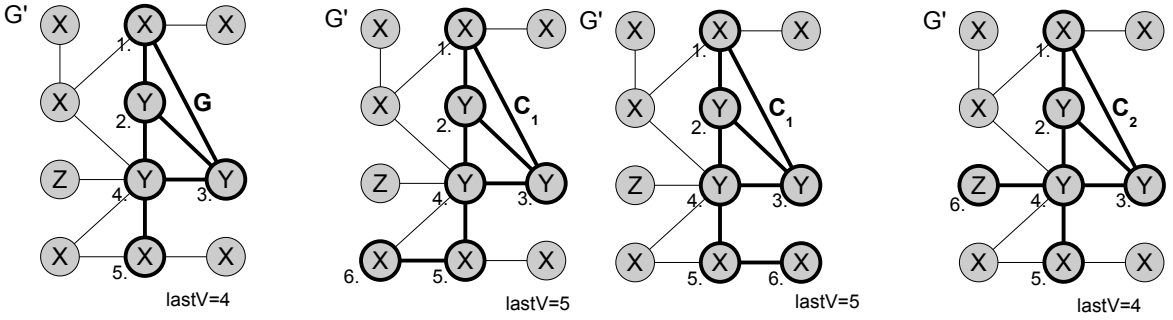
— jeśli graf G powstał przez rozszerzenie grafu G^- polegające na dodaniu nowej krawędzi pomiędzy istniejący wierzchołek v grafu G^- i nowo dodany wierzchołek, wtedy $lastV$ przyjmuje numer wierzchołka v.

Według autorów algorytmu $MoFa$ przy podanym numerowaniu wierzchołków i sposobie wyznaczania wartości $lastV$ dozwolone i wystarczające do uzyskania wszystkich grafów częstych są tylko rozszerzenia dodające jedną krawędź spełniające dwa warunki:

1. W rozszerzaniu zanurzeń biorą udział tylko te wierzchołki, których numery są nie mniejsze niż $lastV$.
2. Do wierzchołka o numerze $lastV$ mogą zostać dodane tylko te krawędzie, które są nie mniejsze niż największa z krawędzi wychodzących z tego wierzchołka, przy czym relację większości wśród krawędzi określa się na podstawie etykiety tej krawędzi oraz etykiety wierzchołka docelowego.

Wygenerowane rozszerzenia są grupowane w klasy równoważności według następującego schamatu. Do jednej klasy równoważności należą wszystkie rozszerzenia, które polegały na dodaniu nowego wierzchołka o danej etykiecie lv i połączeniu go krawędzią o danej etykiecie el do wierzchołka o danym numerze k. Do jednej klasy równoważności należą także wszystkie rozszerzenia, które polegały na dodaniu krawędzi o danej etykiecie el pomiędzy wierzchołki o danych numerach k i l. Tak skonstruowane klasy równoważności nie są równoważne z klasami równoważności generowanymi przez izomorfizm grafów. Wszystkie grafy należące do jednej klasy równoważności są izomorficzne, ale dwa izomorficzne grafy mogą należeć do różnych klas równoważności. Klasa równoważności reprezentuje graf kandydujący, a elementy tej klasy są zanurzeniami tego grafu w grafach wejściowej bazy danych.

Przykład 3.3. Szukamy rozszerzeń zanurzenia grafu G w grafie G' z rysunku 3.6. Kolejność, w jakiej wierzchołki były dodawane do grafu G jest zaznaczona numerami przy wierzchołkach. Jako ostatni został dodany wierzchołek nr 5 poprzez dodanie krawędzi do wierzchołka nr 4, zatem $lastV = 4$. Nowe krawędzie mogą być dodane tylko do wierzchołków o numerze nie mniejszym niż 4, czyli w tym przypadku tylko do wierzchołków nr 4 lub nr 5. Od wierzchołka nr 5 prowadzą dwie nowe krawędzie - obie do wierzchołków o etykiecie X. Powstaną zatem dwa rozszerzenia należące do jednej klasy równoważności. W efekcie powstanie nowy kandydat C_1, który ma dwa zanurzenia w grafie G'. Natomiast od wierzchołka nr 4 prowadzą trzy nowe krawędzie: $Y - X$, $Y - Z$, $Y - X$, ale tylko krawędź prowadząca do wierzchołka o etykiecie Z spełnia warunek rozszerzenia, gdyż największą krawędzią wychodzącą z wierzchołka nr 4 jest $Y - Y$. Rozszerzenie o krawędź $Y - Z$ prowadzi do powstania kandydata C_2 o jednym zanurzeniu w G'.



Rysunek 3.6. Graf częsty G wraz z jego zanurzeniem w G'. C_1 i C_2 są jedynymi kandydatami, które mogą zostać wygenerowane z G przy podanym numerowaniu wierzchołków.

3.3.2. Algorytm MoFa

Wykonanie algorytmu $MoFa$ (algorytm 3.6) rozpoczyna się od znalezienia wszystkich częstych grafów o jednym wierzchołku wraz z ich zanurzeniami. Grafy te zostają posortowane niemalejąco według wsparcia i dodane do zbioru wynikowego R. Następnie dla każdego z tych grafów wykonywana jest procedura $MoFaExtension$, której wynik jest dodawany do zbioru wynikowego R. Ponieważ do zbioru wynikowego R mogą wielokrotnie być dodawane grafy identyczne ze względu na izomorfizm, zostają one odfiltrowane ze zbioru wynikowego przed zakończeniem algorytmu.

Algorytm 3.6 MoFa($\mathbb{D}, minSup$)

$F_1 \leftarrow$ częste grafy o jednym wierzchołku;
posortuj F_1 wg niemalejącego wsparcia;
$R \leftarrow F_1$;
for all $G \in F_1$ **do**
  $R \leftarrow R \cup MoFaExtension(G)$;
end for
usuń duplikaty ze zbioru R;
return R;

Procedura $MoFaExtension(G)$ ma za zadanie wygenerować wszystkie częste grafy, które mogą powstać przez rozszerzanie grafu G. Procedura $MoFaExtension$ jest wykonywana rekurencyjnie. Każde wywołanie generuje rozszerzenia grafu G o jedną krawędź. Na początku sprawdzane jest, czy zbiór wspierający G jest wystarczająco liczny. Jeśli nie jest, procedura się kończy. W przeciwnym przypadku G dodawany jest do zbioru wynikowego i rozpoczyna się generowanie rozszerzeń. Rozszerzenia są tworzone na podstawie zanurzeń grafu G. Z każdym zanurzeniem związana jest zmienna $lastV$ przechowująca numer wierzchołka,

który jako ostatni został wybrany do dodania nowej krawędzi. Z danego zanurzenia generowane są wszystkie dozwolone rozszerzenia o jedną krawędź, to jest takie, które spełniają warunki podane w rozdziale 3.3.1. Wszystkie rozszerzenia są następnie grupowane w klasy równoważności w sposób opisany w poprzedniej sekcji. Każda klasa reprezentuje nowego kandydata na graf częsty. Dla każdego kandydata wywoływana jest ponownie procedura $MoFaExtension$.

Algorytm 3.7 MoFaExtension(G)

if $|G.supportingSet| < minSup$ **then**
  return $\emptyset$;
end if
$R \leftarrow G$;
$E \leftarrow \emptyset$;
for all $G' \in G.supportingSet$ **do**
  for all zanurzenie $e \in G.embeddings[G']$ **do**
    $E \leftarrow E \cup$ dozwolone rozszerzenia zanurzenia e o jedną krawędź;
  end for
end for
pogrupuj E na klasy równoważności EquivalenceClasses;
for all $class \in EquivalenceClasses$ **do**
  $(G_r, G', \phi) \leftarrow class.first$; $/*$ gdzie $class$ jest zbiorem zanurzeń, ϕ jest zanurzeniem grafu G_r (rozszerzenia grafu G) w grafie G' $*/$
  $R \leftarrow R \cup MoFaExtension(G_r)$;
end for
return R;

3.3.3. Modyfikacje algorytmu MoFa

Podstawową wadą algorytmu $MoFa$ jest generowanie kandydatów, które często prowadzi do wielokrotnego rozpatrywania izomorficznych kandydatów. Nawet jeśli zostanie wykryte, że rozpatrywany graf częsty jest izomorficzny z wykrytym wcześniej grafem częstym, nie można w tym miejscu przerwać przeszukiwania, gdyż ten graf ma prawdopodobnie inne numerowanie wierzchołków niż poprzednio wykryty graf, zatem będzie prowadził do powstania innych kandydatów. Prosty mechanizm ograniczania redundancji przeszukiwania został zaproponowany w [12]. Nosi on nazwę *przycinania równoważnych węzłów o wpólnym rodzicu* (ang. equivalent siblings pruning) i polega na usuwaniu izomorficznych kandydatów wygenerowanych z danego grafu częstego. Wśród grafów izomorficznych pozostawiany jest tylko ten najmniej ograniczający rozszerzanie, a więc z najmniejszą wartością $lastV$. Im mniejsza wartość $lastV$, tym więcej potencjalnych rozszerzeń. W [9] zauważono, że

numerowanie wierzchołków zaproponowane w $MoFa$ można wykorzystać do stworzenia kanonicznej etykiety grafu. Zaproponowano ogólną rodzinę etykiet kanonicznych, do której należy też etykieta kanoniczna stosowana w algorytmie $gSpan$. Dla takiej etykiety istnieje pojęcie minimalności oraz procedura sprawdzania minimalności. Wykrycie kandydata o nieminimalnej etykiecie przerywa przeszukiwanie danej gałęzi drzewa przeszukiwań. Przy zastosowaniu etykiety kanonicznej autorzy raportują wzrost wydajności od kilkudziesięciu do kilkuset procent w stosunku do algorytmu $MoSS$. W [12] zaproponowano modyfikację prowadzącą do odkrywania zamkniętych grafów częstych. Metoda ta została rozwinięta w [11].

3.4. Gaston

Algorytm $Gaston$ [60] (Graph/Sequence/Tree extraction) odkrywa częste grafy spójne w sposób wielofazowy. Najpierw odkrywane są częste łańcuchy proste, następnie częste drzewa i wreszcie częste grafy z cyklami. Podstawową zaletą tego podejścia jest fakt, że zarówno dla łańcuchów, jak i dla drzew, istnieją szybkie metody generowania kandydatów częstych bez powtórzeń. Problem wykrywania i unikania tworzenia duplikatów pojawia się dopiero podczas generowania grafów. Dodatkowo podane struktury posiadają naturalną relację zawierania: łańcuchy są częścią drzew, drzewa są częścią grafów. Zatem częste drzewa mogą powstać tylko z częstych łańcuchów, częste grafy - z częstych łańcuchów lub z częstych drzew. Do wyznaczania wsparcia grafów kandydujących wykorzystywane są zanurzenia, które na potrzebę tego algorytmu przechowywane są strukturze drzewiastej.

3.4.1. Generowanie kandydatów

Algorytm $Gaston$ stosuje oddzielne metody generowania kandydatów dla każdego rodzaju struktur: łańcuchów prostych, drzew i grafów.

W przypadku łańcuchów prostych stosowana jest metoda zaczerpnięta z algorytmu odkrywania częstych łańcuchów prostych $MolFea$ [44]. Metoda ta wykorzystuje pojęcie tzw. unikalnego poprzednika łańcucha prostego. *Unikalnym poprzednikiem łańcucha prostego* $v_1, e_1, v_2, \ldots, e_{n-1}, v_n$ jest:

$$\begin{cases} v_1, e_1, v_2, \ldots, e_{n-2}, v_{n-1}, & \text{jeśli } (lbl(v_1), lbl(e_1), lbl(v_2), \ldots, lbl(e_{n-2}), lbl(v_{n-1})) \\ & \quad < (lbl(v_n), lbl(e_{n-1}), \ldots, lbl(e_2), lbl(v_2)) \\ v_2, e_2, v_3, \ldots, e_{n-1}, v_n, & \text{w przeciwnym przypadku.} \end{cases}$$

Kandydujące łańcuchy proste powstają przez rozszerzanie częstych łańcuchów prostych o jedną krawędź. Załóżmy, że rozszerzany jest dany łańcuch prosty p. Jeśli p jest symetryczny, wtedy rozszerzany jest tylko jeden koniec łańcucha, dając w wyniku jeden nowy łańcuch kandujący. Jeśli p jest niesymetryczny, rozszerzane są oba końce łańcucha, dając w efekcie dwa nowe łańcuchy kandydujące. Jeśli okaże się, że p nie jest unikalnym poprzednikiem powstałego łańcucha, taki łańcuch jest odrzucany, gdyż został on już lub dopiero zostanie wygenerowany z innego łańcucha.

Generowanie drzew kandydujących opiera się na rozwinięciu metod znanych z algorytmów odkrywania częstych drzew, w szczególności algorytmów $Unot$ [3] i $uFreqt$ [59]. Drzewa kandydujące powstają bądź z częstych łańcuchów prostych, bądź z częstych drzew. Szczegóły są opisane w [60].

Generowanie grafów kandydujących polega na rozszerzaniu częstych łańcuchów prostych, częstych drzew oraz częstych grafów oraz sprawdzaniu, czy wygenerowany graf nie był już wcześniej rozpatrywany. Nie stosuje się rozszerzeń wprowadzających nowy wierzchołek. Do wykrywania duplikatów używana jest struktura mieszająca zawierająca etykiety kanoniczne grafów uzyskiwane za pomocą programu NAUTY[1].

3.4.2. Przechowywanie zanurzeń

Struktura przechowująca zanurzenia danego grafu G jest wielopoziomowa i rozrasta się wraz z rozszerzaniem grafu. Jeśli graf G posiada tylko jeden wierzchołek (powiedzmy v), wtedy ta struktura składa się z jednego poziomu, który zawiera listę zanurzeń tego wierzchoła we wszystkich grafach zbioru $\mathbb{D}$. Pojedyncze zanurzenie na tej liście reprezentowane jest jako trójka $(-, G', v')$, gdzie G' jest grafem, w którym jest zanurzony graf G, a v' jest wierzchołkiem grafu G' o etykiecie identycznej z etykietą wierzchołka v.

Po każdym rozszerzeniu grafu G o nową krawędź do struktury dodawany jest jeden poziom. Jeśli rozszerzenie grafu G polegało na dodaniu jednego nowego wierzchołka v i jednej nowej krawędzi, tworząc nowy graf G_c, wtedy ten nowy poziom zawiera listę trójek $t = (parent, G', v')$, gdzie $parent$ jest indeksem trójki z poziomu wyżej, G' jest grafem, w którym jest zanurzony graf G_c, a v' wierzchołkiem, któremu przyporządkowany jest wierzchołek v w zanurzeniu grafu G_c w grafie G'. Wartości przyporządkowań pozostałych wierzchołków tego zanurzenia odczytuje się z poziomu wyżej, z trójki o indeksie równym $parent$.

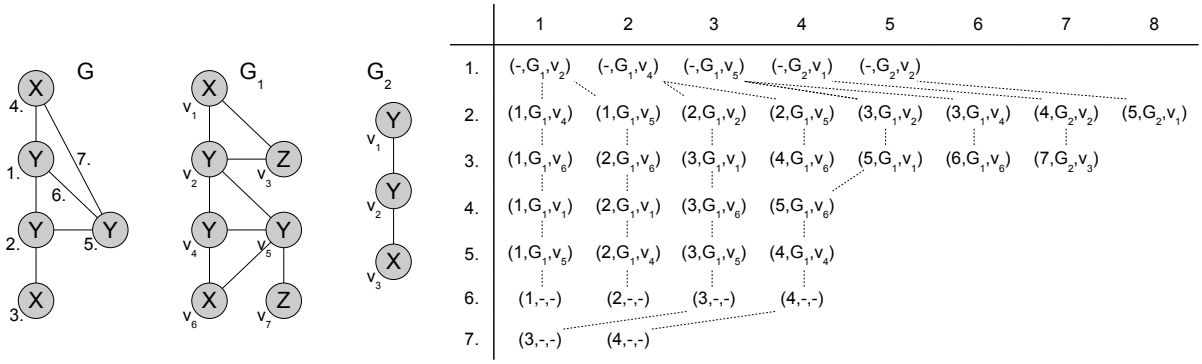
[1] NAUTY, http://www.cs.sunysb.edu/~algorith/implement/nauty/implement.shtml

Jeśli rozszerzenie grafu G polegało na dodaniu jednej nowej krawędzi pomiędzy wierzchołki istniejące w grafie G, tworząc nowy graf G_c, wtedy nowy poziom zawiera listę trójek $t = (parent, -, -)$, gdzie $parent$ jest indeksem trójki z poziomu wyżej. Zanurzenia grafu G_c w grafy ze zbioru $\mathbb{D}$ odczytuje się wtedy z poziomu wyżej, z trójki o indeksie równym $parent$.

Na rysunku 3.7 przedstawiona jest struktura przechowująca zanurzenia grafu G w grafach G_1 i G_2 z wejściowej bazy grafów. Liczby przy wierzchołkach i krawędziach pokazują w jakiej kolejności był budowany graf G i odpowiadają wierszom w strukturze przechowującej zanurzenia. Na początku G był grafem o jednym wierzchołku i posiadał pięć zanurzeń w grafach z wejściowej bazy grafów. Kolejne rozszerzenia tego grafu o jednym wierzchołku były rozszerzeniami wprowadzającymi nową krawędź wraz z nowym wierzchołkiem, co przedstawiają poziomy od 2 do 5 struktury zanurzeń. Ostatnie dwa rozszerzenia - poziom 6 i 7 - są rozszerzeniami o jedną krawędź bez wprowadzania nowego wierzchołka. Zanurzenia grafu G znajdują się na poziomie 7. Graf G posiada dwa zanurzenia w grafie G_1 i żadnego w grafie G_2. Stąd wsparcie tego grafu wynosi 1. Aby zrekonstruować zanurzenia grafu G, należy prześledzić wszystkie poziomy ze struktury zanurzeń wybierając na każdym poziomie zanurzenie wskazywane przez identyfikator $parent$ z poziomu wyżej. Na poziomie 7 są dwa zanurzenia, zrekonstruujemy tylko jedno z nich: $(3, -, -)$. Pole $parent$ ma wartość 3, należy więc wziąć trzecią trójkę z poziomu 6, czyli $(3, -, -)$. Z poziomu 5 należy więc wziąć trzecią trójkę, czyli $(3, G_1, v_5)$, z poziomu 4 należy wziąć trzecią trójkę, czyli $(3, G_1, v_6)$, z poziomu 3 należy wziąć trzecią trójkę, czyli $(3, G_1, v_1)$, z poziomu 2 należy wziąć trzecią trójką, czyli $(2, G_1, v_2)$, a z poziomu 1 należy wziąć drugą trójkę, czyli $(-, G_1, v_4)$. Zatem w rozpatrywanym zanurzeniu grafu G w grafie G_1 pierwszy wierzchołek grafu G odpowiada wierzchołkowi $v_4 \in G_1$, drugi wierzchołek grafu G odpowiada wierzchołkowi $v_2 \in G_1$, trzeci wierzchołek odpowiada wierzchołkowi $v_1 \in G_1$, czwarty wierzchołek odpowiada wierzchołkowi $v_6 \in G_1$, a piąty wierzchołek odpowiada wierzchołkowi $v_5 \in G_1$.

3.4.3. Algorytm Gaston

W algorytmie $Gaston$ operacja rozszerzania grafu (łańcucha, drzewa, grafu z cyklami) oznacza dodanie jednej krawędzi do struktury. Sposób rozszerzenia nazywamy $przyrostem$ i definiujemy w następujący sposób:



Rysunek 3.7. Struktura zanurzeń grafu G w grafach G_1 i G_2.

Definicja 3.9. *(przyrost)*

Jeżeli graf $G = (V, E, lbl, L)$ ma być rozszerzony przez dodanie jednej nowej krawędzi o etykiecie el pomiędzy wierzchołki $v_i \in V$ i $v_j \in V$, wtedy *przyrost* jest czwórką $l^G = (v_i, v_j, el, -)$.

Jeżeli graf G ma być rozszerzony przez dodanie jednego nowego wierzchołka $v \notin V$ o etykiecie vl i jednej krawędzi o etykiecie el pomiędzy wierzchołki $v_i \in V$ i v, wtedy przyrost jest czwórką $l^G = (v_i, -, el, vl)$.

Graf, który powstał przez roszszerzenie grafu G o przyrost l^G będziemy oznaczali jako $l^G(G)$. Jeśli graf $l^G(G)$ jest częsty, wtedy przyrost l^G nazywamy *przyrostem częstym*. Przyrost l^G jest *dozwolony*, gdy rozszerzenie $l^G(G)$ spełnia warunki rozszerzania z rozdziału 3.4.1.

Dla danego grafu G, zbiór L jest zbiorem wszystkich przyrostów, pozwalających na wygenerowanie częstych rozszerzeń tej struktury. Z każdym przyrostem $l^G \in L$ grafu G jest związana lista zanurzeń $embeddingList[l^G]$, która jest ostatnim poziomem ze struktury zanurzeń grafu G rozszerzonego o l_G. Lista $embeddingList[l^G]$ jest zatem listą trójek $t = (parent, G', v')$. Przez $pos(t)$ będziemy oznaczać pozycję trójki t na tej liście.

Wykonanie algorytmu $Gaston$ (algorytm 3.8) rozpoczyna się od znalezienia wszystkich częstych grafów o jednym wierzchołku. Dla każdego takiego grafu G są następnie szukane jego częste przyrosty. Graf G wraz ze zbiorem jego częstych przyrostów jest następnie przekazywany do funkcji $gastonFindPaths$, której celem jest rozszerzanie łańcuchów o jedną krawędź. W zależności od tego, czy rozszerzenie prowadzi do powstania łańcucha, drzewa, czy grafu z cyklami wywoływana jest funkcja $gastonFindPaths$, $gastonFindTrees$, lub $gastonFindCyclicGraphs$. Każda z funkcji rozszerza podaną na wejściu strukturę o jedną krawędź pochodzącą z listy przyrostów, przy czym brane są pod uwagę tylko dozwolone

przyrosty. Do znajdowania nowych przyrostów służą dwie funkcje: $gastonJoin$ oraz $gastonExtend$. Funkcje te jednocześnie aktualizują strukturę zanurzeń.

Algorytm 3.8 gaston($\mathbb{D}, minSup$)

$F_0 \leftarrow$ zbiór częstych grafów o jednym wierzchołku;
for all $G \in F_0$ **do**
  $L \leftarrow$ zbiór częstych przyrostów grafu G;
  $R \leftarrow R \cup gastonFindPaths(G, L)$;
end for
return R;

Algorytm 3.9 gastonFindPaths(P,L)

for all dozwolony przyrost $l^P \in L$ **do**
  $G' \leftarrow l^P(P)$;
  if $l^P.v_j = null$ **then** $/* \; l^P$ jest przyrostem dodającym jeden nowy wierzchołek i jedną nową krawędź $*/$
    $L' \leftarrow gastonExtend(l^P)$;
    for all przyrost $l'^P \in L \wedge l'^P \neq l^P$ **do**
      $L' \leftarrow L' \cup gastonJoin(l^P, l'^P)$;
    end for
    $R \leftarrow R \cup \{G'\}$;
    if G' jest łańcuchem prostym **then**
      $R \leftarrow R \cup gastonFindPaths(G', L')$;
    else
      $R \leftarrow R \cup gastonFindTrees(G', L')$;
    end if
  else $/* \; l^P$ jest przyrostem dodającym jedną nową krawędź $*/$
    $R \leftarrow R \cup gastonFindCyclicGraphs(G', L')$;
  end if
end for
return R;

Funkcja $gastonJoin(l_1^G, l_2^G)$ łączy ze zobą dwa przyrosty l_1^G i l_2^G dając w efekcie (jeśli istnieje) częsty przyrost $l_2^{l_1^G(G)}$, czyli przyrost grafu $G_c = l_1^G(G)$ o krawędź wyspecyfikowaną w przyroście l_2^G. Ta funkcja jednocześnie generuje wszystkie zanurzenia grafu $l_2^{G_c}(G_c)$. Działanie funkcji jest następujące. Wynikowy przyrost l' jest kopią przyrostu l_2^G, czyli będzie rozszerzał graf G_c przez dodanie krawędzi w tym samym miejscu co l_2^G. Następnie znajdowane są wszystkie pary trójek t_1, t_2, które mają identyczną wartość pola $parent$, a t_1 należy do listy zanurzeń przyrostu l_1^G, t_2 należy do listy zanurzeń przyrostu l_2^G. Wspólny rodzic oznacza, że zanurzenia t_1 i t_2 są rozszerzeniami jednego wspólnego zanurzenia. Rozszerzenia można zatem połączyć tworząc nową trójkę $t = (pos(t_1), t_2.G', t_2.v')$, czyli zanurzenie, które jest

Algorytm 3.10 gastonFindTrees(T,L)

for all dozwolony przyrost $l^T \in L$ **do**
  $G' \leftarrow l^T(T)$;
  $R \leftarrow R \cup \{G'\}$;
  if $l^T.v_j = null$ **then** $/*$ l^T jest przyrostem dodającym jeden nowy wierzchołek i jedną nową krawędź $*/$
    $L' \leftarrow gastonExtend(l^T)$;
    for all dozwolony przyrost $l'^T \in L \wedge l'^T \neq l^T$ **do**
      $L' \leftarrow L' \cup gastonJoin(l^T, l'^T)$;
    end for
    $R \leftarrow R \cup gastonFindTrees(G', L')$;
  else $/*$ l^T jest przyrostem dodającym jedną nową krawędź $*/$
    for all przyrost $l'^T \in L \wedge l'^T \neq l^T$ **do**
      $L' \leftarrow L' \cup gastonJoin(l^T, l'^T)$;
    end for
    $R \leftarrow R \cup gastonFindCyclicGraphs(G', L')$;
  end if
end for
return R;

Algorytm 3.11 gastonFindCyclicGraphs(G,L)

for all dozwolony przyrost $l^G \in L$ **do**
  $G' \leftarrow l^G(G)$;
  $R \leftarrow R \cup \{G'\}$;
  for all przyrost $l'^G \in L$ taki, że $l'^G.v_i > l^G.v_i$ **do**
    $L' \leftarrow L' \cup gastonJoin(l^G, l'^G)$;
  end for
  $R \leftarrow R \cup gastonFindCyclicGraphs(G', L')$;
end for
return R;

Algorytm 3.12 gastonJoin(l_1^G, l_2^G)

$l' \leftarrow l_2^G$;
$embeddingList[l'] \leftarrow \emptyset$;
for all $(t_1 \in embeddingList[l_1^G], t_2 \in embeddingList[l_2^G]$ takie, że $t_1.parent = t_2.parent)$ **do**
  $embeddingList[l'] \leftarrow embeddingList[l'] \cup \{(pos(t_1), t_2.G', t_2.v')\}$;
end for
if l' jest częste **then**
  return $\{l'\}$;
else
  return $\emptyset$;
end if

Algorytm 3.13 gastonExtend(l^G)

$C \leftarrow \emptyset$;/$*$ zbiór kandydujących przyrostów $*$/
for all $t \in embeddingList[l^G]$ **do**
  for all $v' \in t.G'$ taki, że $\{t.v', v'\} \in t.G'.E$ **do** /$*$ wierzchołek v' jest połączony krawędzią z $t.v'$ w grafie $t.G'$ $*$/
    if v' należy do zanurzenia t **then**
      $v \leftarrow$ wierzchołek grafu G, który jest przyporządkowany wierzchołkowi $v' \in t.G'$ w zanurzeniu t;
      $l' = (l.v_i, v, lbl'(\{t.v', v'\}), -)$;
      $C \leftarrow C \cup \{l'\}$;
      $embeddingList[l'] \leftarrow embeddingList[l'] \cup \{(pos(t), -, -)\}$;
    else
      $l' = (l.v_i, -, lbl'(\{t.v', v'\}), -)$;
      $C \leftarrow C \cup \{l'\}$;
      $embeddingList[l'] \leftarrow embeddingList[l'] \cup \{(pos(t), t.G', v')\}$;
    end if
  end for
end for
usuń nieczęste przyrosty ze zbioru C;
return C;

rozszerzeniem zanurzenia t_1 o krawędź prowadzącą do $t_2.v'$. Zanurzenie t jest następnie dodawane do listy zanurzeń przyrostu wynikowego l'. Na koniec sprawdzane jest, czy przyrost jest częsty, co wykonuje się przez wyznaczenie liczby różnych grafów $t.G'$ na jego liście zanurzeń jes. Funkcja zwraca jednoelementowy zbiór składający się z przyrostu l', jeśli l jest częsty lub zbiór pusty w przeciwnym przypadku.

Funkcja $gastonExtned(l^G)$ zwraca listę częstych przyrostów grafu $G_c = l^G(G)$ wraz z zanurzeniami grafów, które są rozszerzeniami grafu G_c o te przyrosty. Działanie tej funkcji jest następujące. Dla każdego zanurzenia $t \in embeddingList[l^G]$ i każdego wierzchołka v', który jest połączony krawędzią z wierzchołkiem $t.v'$ w grafie $t.G'$, wykonuje się następujący test. Sprawdzane jest, czy wierzchołek v' należy do zanurzenia t, co można osiągnąć sprawdzając łańcuch wszystkich trójek połączonych przez pole $parent$ zaczynający się od t. Jeśli v' będzie występował w dowolnej z trójek tego łańcucha, wtedy v' należy do zanurzenia t. W tym przypadku tworzony jest odpowiedni przyrost l' dodający krawędź pomiędzy isniejące wierzchołki, a do jego listy zanurzeń dodawana jest trójka $(pos(t), -, -)$, co oznacza, że zanurzenie jest takie samo jak zanurzenie t. W przypadku, gdy wierzchołek v' nie należy do zanurzenia, tworzony jest odpowiedni przyrost l' dodający nową krawędź i nowy wierzchołek.

Do listy zanurzeń przyrostu l' dodawana jest trójka $(pos(t), t.G', v')$. Ostatecznie funkcja $gastonExtend$ zwraca zbiór wszystkich znalezionych w ten sposób częstych przyrostów.

Funkcje $gastonFindPaths$, $gastonFindTrees$ wykorzystują do rozszerzania zarówno operacje $gastonJoin$ i $gastonExtend$, natomiast $gastonFindCyclicGraphs$ - tylko operację $gastonJoin$.

3.5. Inne algorytmy

W tym rozdziale zostaną wymienione pozostałe ważne algorytmy odkrywania grafów częstych. Przedstawiony poniżej przegląd nie jest oczywiście kompletny; z dużo szerszym ujęciem można zapoznać się między innymi w pracach [17, 25, 72]. Algorytm $Subdue$ [16] jest powszechnie uznawany za pierwszy algorytm odkrywający częste grafy, choć w jego opisie nie używa się formalnej definicji grafu częstego. Ze względu na sposób działania, polegający na zachłannym rozszerzaniu grafów kandydujących, zbiór wynikowy generowany przez algorytm $Subdue$ jest niekompletny. Za pierwszy algorytm dający kompletny wynik uznaje się algorytm $WARMR$ [24], który jest systemem pozwalającym na odkrywanie różnego rodzaju wzorców częstych, nawet ogólnieszych od grafów, pod warunkiem, że dają się one zapisać w postaci predykatów logicznych. Odkrywanie częstych wzorców odbywa się za pomocą metod indukcji logicznej. Algorytm ten doczekał się rozwinięcia w postaci algorytmu $FARMAR$ [43]. Algorytm AGM [39, 40] odkrywa częste grafy spójne i niespójne, przy czym za definicję podgrafu przyjmuje się w nim podgraf indukowany wierzchołkowo, zatem liczba uzyskanych grafów częstych jest z tego powodu mniejsza niż przy klasycznej definicji podgrafu. Konstrukcja algorytmu AGM przypomina w budowie algorytm $Apriori$. Zaproponowany później algorytm $AcGM$ [41] jest modyfikacją algorytmu AGM, pozwalającą na odkrywania wyłącznie spójnych grafów częstych. Algorytm FSG [47, 48] jest kolejnym algorytmem bazującym na algorytmie $Apriori$. $ParSeMiS$[2] jest platformą rozszerzającą platformę $ParMol$, w której nacisk położono na możliwość zrównoleglenia algorytmów odkrywania grafów częstych. Platforma $ParSeMiS$ składa się trzech algorytmów zaimplementowanych z myślą o wykorzystaniu wielowątkowści: wymienionych wcześniej algorytmów $gSpan$ i $Gaston$ oraz algorytmu $DAG\text{-}Miner$ [73], który odkrywa częste skierowane grafy acykliczne. Do odkrywania grafów skierowanych służy także algorytm

[2] http://www2.informatik.uni-erlangen.de/EN/research/ParSeMiS/index.html

RCD-$Miner$ [67]. Istnieją też algorytmy zaprojektowane do odkrywania częstych grafów zamkniętych, np. $CloseGraph$ będący modyfikacją algorytmu $gSpan$ lub $LCGMiner$ [75], a także do odkrywania maksymalnych grafów częstych ($SPIN$ [38]).

4. Proponowane algorytmy odkrywania częstych grafów z uwzględnianiem niespójności

4.1. Algorytm odkrywający jednocześnie grafy spójne i niespójne

Algorytm UGM (Unconnected Graph Miner) [64, 65], który jest propozycją autora niniejszej rozprawy, pozwala na jednoczesne odkrywanie spójnych i niespójnych grafów częstych. Algorytm tworzy kandydatów na grafy częste poprzez rozszerzanie grafów częstych o jedną krawędź, niezależnie od tego, czy utworzone rozszerzenie jest spójne, czy nie. Wsparcie grafu jest wyznaczane za pomocą testów na izomorfizm tego grafu z podgrafami wejściowej bazy grafów, a nie z wykorzystaniem zanurzeń, co czyni algorytm znaczniej mniej wymagającym pamięciowo. Mniejsze zapotrzebowanie na pamięć wynika też z przeszukiwania przestrzeni grafów w głąb, a nie wszerz. Zapobieganie tworzeniu duplikatów kandydujących grafów jest częściowo realizowane przez ustaloną kolejność dodawania krawędzi - jeśli graf był rozszerzany (być może wielokrotnie) o krawędź o deskryptorze k, to nie będzie już rozszerzany o krawędzie o deskryptorze mniejszym niż k.

W algorytmie UGM zastosowano dodatkowo cztery techniki optymalizacyjne: metodę wykorzystującą *maksymalne częste wielozbiory deskryptorów* krawędzi grafów, metodę wykorzystującą *zbiór grafów nieczęstych*, metodę wykorzystującą *zbiór nieczęstych konstruktorów rozszerzeń* oraz metodę pozwalającą na *przerywanie wyznaczania wsparcia* grafu. Pierwsze trzy metody są autorskimi pomysłami w odniesieniu do odkrywania grafów częstych. W przypadku czwartej metody autorskie są heurystyki poprawiające jej wydajność.

4.1.1. Maksymalne częste wielozbiory deskryptorów krawędzi

Proponowana optymalizacja z wykorzystaniem maksymalnych wielozbiorów deskryptorów korzysta z pojęć *zbioru częstego* oraz *maksymalnego zbioru częstego*, które są zaczerpnięte z zagadnienia *odkrywania zbiorów częstych* [1].

Definicja 4.1. *(zbiór częsty)*
Niech $\mathbb{S}$ będzie rodziną zbiorów. Przy zadanym progu minimalnego wsparcia $minSup$ zbiór F jest zbiorem częstym, gdy F jest podzbiorem co najmniej $minSup$ zbiorów z rodziny $\mathbb{S}$, czyli gdy $|\{X \in \mathbb{S} \ : \ F \subseteq X\}| \geq minSup$.

Definicja 4.2. *(maksymalny zbiór częsty))*
Niech $\mathbb{S}$ będzie rodziną zbiorów. Przy zadanym progu minimalnego wsparcia $minSup$ zbiór F jest maksymalnym zbiorem częstym, gdy F jest zbiorem częstym oraz F nie jest podzbiorem żadnego innego zbioru częstego.

Analogicznie można podać definicję *wielozbioru częstego* oraz *maksymalnego wielozbioru częstego*.

Definicja 4.3. *(wielozbiór częsty)*
Niech $\mathbb{S}$ będzie rodziną wielozbiorów. Przy zadanym progu minimalnego wsparcia $minSup$ wielozbiór F jest wielozbiorem częstym, gdy F jest podzbiorem co najmniej $minSup$ wielozbiorów z rodziny $\mathbb{S}$, czyli gdy $|\{X \in \mathbb{S} \ : \ F \subseteq X\}| \geq minSup$.

Definicja 4.4. *(maksymalny wielozbiór częsty))*
Niech $\mathbb{S}$ będzie rodziną zbiorów. Przy zadanym progu minimalnego wsparcia $minSup$ wielozbiór F jest maksymalnym wielozbiorem częstym, gdy F jest wielozbiorem częstym oraz F nie jest podzbiorem żadnego innego wielozbioru częstego.

Pomysł wykorzystania *maksymalnych wielozbiorów deskryptorów krawędzi* polega na tym, aby przed rozpoczęciem odkrywania częstych grafów, wygenerować rodzinę wielozbiorów deskryptorów krawędzi na podstawie grafów z wejściowej bazy grafów $\mathbb{D}$, a następnie znaleźć wielozbiory częste w tej rodzinie. Z uzyskanych częstych wielozbiorów deskryptorów krawędzi buduje się następnie kandydatów na grafy częste.

Dla każdego grafu G z wejściowej bazy grafów $\mathbb{D}$ należy wyznaczyć jego wielozbiór deskryptorów $ES(G)$, według definicji 2.24. Uzyskaną w ten sposób rodzinę wielozbiorów oznaczamy przez $ES(\mathbb{D}) = \bigcup_{G \in \mathbb{D}} ES(G)$. Na zbiorze $ES(\mathbb{D})$ wykonuje się odkrywanie

wielozbiorów częstych przy minimalnym progu wsparcia takim samym jak minimalny próg wsparcia dla odkrywania grafów częstych.

Lemat 4.1. *Jeżeli graf G jest grafem częstym w zbiorze $\mathbb{D}$, to $ES(G)$ jest wielozbiorem częstym w rodzinie wielozbiorów $ES(\mathbb{D})$.*

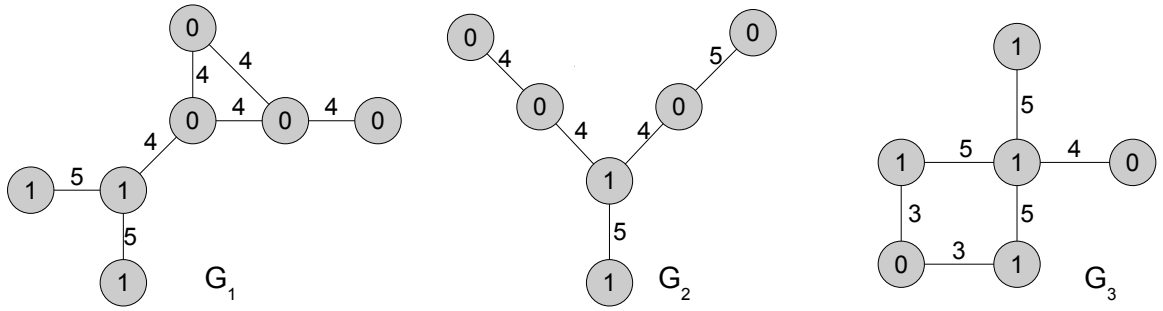
Lemat nie zachodzi w drugą stronę. Nie każdemu wielozbiorowi częstemu $ES(G)$ odpowiada częsty graf. Nie zachodzi również równość wsparcia grafu G i wsparcia wielozbioru $ES(G)$.

Lemat jest równoważny następującemu stwierdzeniu: jeśli dany wielozbiór deskryptorów es nie jest częsty, wtedy nie istnieje taki częsty graf G, że $es = ES(G)$. Własność tę można wykorzystać na etapie generowania kandydatów na grafy częste. Należy generować tylko takich kandydatów C, których wielozbiór deskryptorów $ES(C)$ jest wielozbiorem częstym.

Ponieważ istotne jest tylko pytanie, czy $ES(C)$ jest częsty, a nie jaka jest dokładna wartość jego wsparcia, więc dla oszczędności czasu i pamięci wystarczy odkrywanie tylko maksymalnych wielozbiorów częstych w zbiorze $ES(\mathbb{D})$, gdyż znajomość wszystkich częstych zbiorów maksymalnych pozwala wywnioskować dla dowolnego wielozbioru deskryptorów krawędzi, czy jest częsty, czy nie, a liczba częstych wielozbiorów maksymalnych jest dużo mniejsza od liczba wszystkich zbiorów częstych. Warunek generowania kandydatów zapisuje się wtedy w następującej postaci.

Należy generować tylko takich kandydatów na grafy częste C, których wielozbiór deskryptorów $ES(C)$ jest podzbiorem jednego z maksymalnych wielozbiorów częstych uzyskanych z $ES(\mathbb{D})$.

Istnieje wiele efektywnych algorytmów odkrywania zbiorów częstych oraz maksymalnych zbiorów częstych. Ich obszerny przegląd znajduje się w [33]. Algorytmy te można wykorzystać do odkrywania częstych wielozbiorów w rodzinie zbiorów $\mathbb{S}$ w następujący sposób. Najpierw każdy wielozbiór $X \in \mathbb{S}$ należy przekształcić w zbiór X' o tej samej liczności, zastępując elementy $e \in X$ przez elementy e_k, gdzie k jest numerem wystąpienia danego elementu w wielozbiorze. Na przykład wielozbiór $X = \{a, a, a, b, b\}$ zostanie zamieniony na zbiór $X' = \{a_1, a_2, a_3, b_1, b_2\}$. W tak utworzonej rodzinie zbiorów należy następnie wykonać odkrywanie zbiorów częstych. Każdy uzyskany zbiór częsty należy potem zamienić na



Rysunek 4.1. Wejściowy zbiór grafów $\mathbb{D}$.

wielozbiór częsty przez zignorowanie indeksów k. Na przykład zbiór częsty $F' = \{a_1, a_2, b_1\}$ zostanie zamieniony na wielozbiór częsty $F = \{a, a, b\}$.

Przykład 4.1. Celem jest znalezienie częstych grafów w wejściowym zbiorze trzech grafów przedstawionych na rysunku 4.1 przy założeniu progu minimalnego wsparcia $minSup = 2$. Wielobiory deskryptorów dla grafów z $\mathbb{D}$ to odpowiednio:

$ES(G_1) = \{(\{1,1\},5)(\{1,1\},5)(\{0,1\},4)(\{0,0\},4)(\{0,0\},4)(\{0,0\},4)(\{0,0\},4)\},$

$ES(G_2) = \{(\{0,0\},5)(\{1,1\},5)(\{0,1\},4)(\{0,1\},4)(\{0,0\},4)\},$

$ES(G_3) = \{(\{1,1\},5)(\{1,1\},5)(\{1,1\},5)(\{0,1\},4)(\{0,1\},3)(\{0,1\},3)\}.$

Zbiory maksymalne wygenerowane z tych zbiorów przy $minSup = 2$ to:

$ms_1 = \{(\{1,1\},5)(\{1,1\},5)(\{0,1\},4)\},$

$ms_2 = \{(\{1,1\},5)(\{0,1\},4)(\{0,0\},4)\}.$

Dla prostszego zapisu przyjmijmy następujące oznaczenia deskryptorów krawędzi: $a = (\{1,1\},5)$, $b = (\{0,1\},4)$, $c = (\{0,0\},4)$. Zbiory maksymalne mają wtedy postać: $ms_1 = \{aab\}$, $ms_2 = \{abc\}$. Kandydaci na grafy częste będą tworzeni tylko za pomocą krawędzi o deskryptorach a, b i c. Dokładny sposób tworzenia kandydatów jest opisany w rozdziale 4.1.5, a w tym przykładzie istotny jest tylko fakt, że wielozbiór deskryptorów krawędzi tworzonych kandydatów musi być podzbiorem zbioru ms_1 lub ms_2. Zatem będą generowani tylko kandydaci o następujących wielozbiorach deskryptorów krawędzi: $\{a\}$, $\{b\}$, $\{c\}$, $\{aa\}$, $\{ab\}$, $\{ac\}$, $\{bc\}$, $\{aab\}$, $\{abc\}$. Dla przykładu, kandydaci o wielozbiorze deskryptorów równym $\{bb\}$ nie zostaną wygenerowani, gdyż $\{bb\}$ nie jest podzbiorem żednego maksymalnego zbioru częstego. Zbiór $\{bb\}$ nie jest zatem zbiorem częstym, a co za tym idzie żaden graf o wielozbiorze deskryptorów równym $\{bb\}$ nie jest częsty.

4.1.2. Zbiór grafów nieczęstych

Wykorzystanie zbioru grafów nieczęstych w celu optymalizacji algorytmu UGM opiera się na własności 4.1.

Własność 4.1. *Jeżeli graf G jest nieczęsty oraz G jest izomorficzny z podgrafem grafu G', to graf G' jest nieczęsty.*

Podczas odkrywania grafów częstych algorytm UGM utrzymuje zbiór grafów nieczęstych $\mathbb{GN}$, w którym umieszczani są nieczęści kandydaci.

Jeżeli dowolny z grafów ze zbioru $\mathbb{GN}$ jest izomorficzny z podgrafem rozpatrywanego kandydata C, wtedy C jest nieczęsty i nie trzeba wykonywać kosztownej operacji wyznaczania wsparcia. Zamiast wykonywać testy na izomorfizm kandydata C z podgrafami grafów z wejściowej bazy grafów $\mathbb{D}$ wykonuje się testy na izomorfizm grafów ze zbioru $\mathbb{GN}$ z podgrafem kandydata C. Jak pokazują eksperymenty jest to operacja zdecydowanie szybsza, nawet w przypadku, gdy zbiór $\mathbb{GN}$ jest dużo liczniejszy niż zbiór $\mathbb{D}$. Wynika to z dwóch faktów. Po pierwsze grafy w zbiorze $\mathbb{GN}$ są dużo mniejsze niż w grafy z zbiorze $\mathbb{D}$, a po drugie zbiór $\mathbb{GN}$ jest przechowywany w strukturze wspierającej wykonywanie takich operacji (rozdział 4.1.7). Poza tym, do zbioru $\mathbb{GN}$ nie są dodawane wszystkie odnajdywane grafy nieczęste, a jedynie te, których nieczęstość została stwierdzona przez wyznaczenie wsparcia, gdyż wywnioskowanie nieczęstości grafu G w inny sposób (tzn. za pomocą częstych wielozbiorów deskryptorów, zbioru grafów nieczęstych lub zbioru nieczęstych konstruktorów rozszerzeń) umożliwia wywnioskowanie nieczęstości każdego nadgrafu grafu G w ten sam sposób. Dodanie takiego grafu do zbioru $\mathbb{GN}$ byłoby więc redundantne, powiększałoby zbiór $\mathbb{GN}$ i spowalniało wnioskowanie.

4.1.3. Zbiór nieczęstych konstruktorów rozszerzeń

Definicja 4.5. *(konstruktor rozszerzeń)*
Konstruktorem rozszerzeń nazywamy parę (G, ed), gdzie G jest grafem, a ed jest deskryptorem krawędzi.

Definicja 4.6. *(nieczęsty konstruktor rozszerzeń)*
Konstruktor rozszerzeń $k = (G, ed)$ jest nieczęsty, gdy każde rozszerzenie grafu G o krawędź o deskryptorze ed jest nieczęste. Rozszerzenie grafu o krawędź o deskryptorze ed polega

na dodaniu jednej krawędzi do grafu i może utworzyć zarówno graf spójny, jak i niespójny. Operacja rozszerzania jest szczegółowo opisana w rozdziale 4.1.6.

Podczas odkrywania grafów częstych algorytm UGM utrzymuje zbiór nieczęstych konstruktorów rozszerzeń $\mathbb{NKR}$. Wykorzystanie zbioru $\mathbb{NKR}$ w celu optymalizacji algorytmu UGM opiera się na własności 4.2.

Własność 4.2. *Jeżeli wszystkie rozszerzenia grafu G o krawędź o deskryptorze ed są nieczęste oraz G jest izomorficzny z podgrafem grafu G', to wszystkie rozszerzenia grafu G' o krawędź o deskryptorze ed są nieczęste.*

Wnioskowanie ze zbioru $\mathbb{NKR}$ ma miejsce na etapie tworzenia grafów kandydujących poprzez rozszerzanie grafu częstego. Jeżeli graf częsty G' ma być rozszerzony o krawędź ed, a w zbiorze $\mathbb{NKR}$ istnieje taki konstruktor rozszerzeń $k = (G, ed)$, że graf G jest izomorficzny z podgrafem grafu G', wtedy rozszerzanie grafu G' o krawędź ed nie jest wykonywane.

Można zauważyć, że, gdyby zrezygnować ze zbioru $\mathbb{NKR}$, do zbioru $\mathbb{GN}$ zostałyby dodane wszystkie grafy, które mogą powstać z nieczęstych konstruktorów rozszerzeń znajdujących się normalnie w $\mathbb{NKR}$. Wnioskowanie ze zbioru $\mathbb{GN}$ wykryłoby wtedy między innymi te same nieczęste grafy kandydujące, co wnioskowanie ze zbioru $\mathbb{NKR}$, ale byłoby ono wolniejsze, gdyż zbiór $\mathbb{GN}$ byłyby dużo liczniejszy niż zbiór $\mathbb{NKR}$.

4.1.4. Przerywanie wyznaczania wsparcia

Wyznaczanie wsparcia grafu kandydującego polega na wykonaniu testów na izomorfizm z podgrafem każdego grafu ze zbioru wspierającego $\mathbb{D}$ rodzica grafu kandydującego. Znajomość dokładnej wartości wsparcia jest istotna tylko w przypadku grafów częstych. W przypadku grafów nieczęstych wystarczy stwierdzić, że wsparcie jest poniżej progu $minSup$. W czasie wyznaczania wsparcia kandydata należy zliczać liczbę grafów w $\mathbb{D}$, które nie mają podgrafów izomorficznych z grafem kandydującym. Jeśli ta liczba przekroczy $|\mathbb{D}| - minSup$, wtedy kandydat ma na pewno wsparcie mniejsze niż $minSup$, czyli nie jest częsty.

Przykład 4.2. *Wyznaczane jest wsparcie kandydata C w zbiorze $\mathbb{D}$ o liczności 10. Próg minimalnego wsparcia jest równy $minSup = 5$. Wykonano już testy na izomorfizm grafu C z podgrafem siedmiu grafów zbioru $\mathbb{D}$. Jeden test zakończył się pozytywnie, sześć negatywnie. Liczba negatywnych testów przekracza wartość $|\mathbb{D}| - minSup = 5$, zatem wyznaczanie*

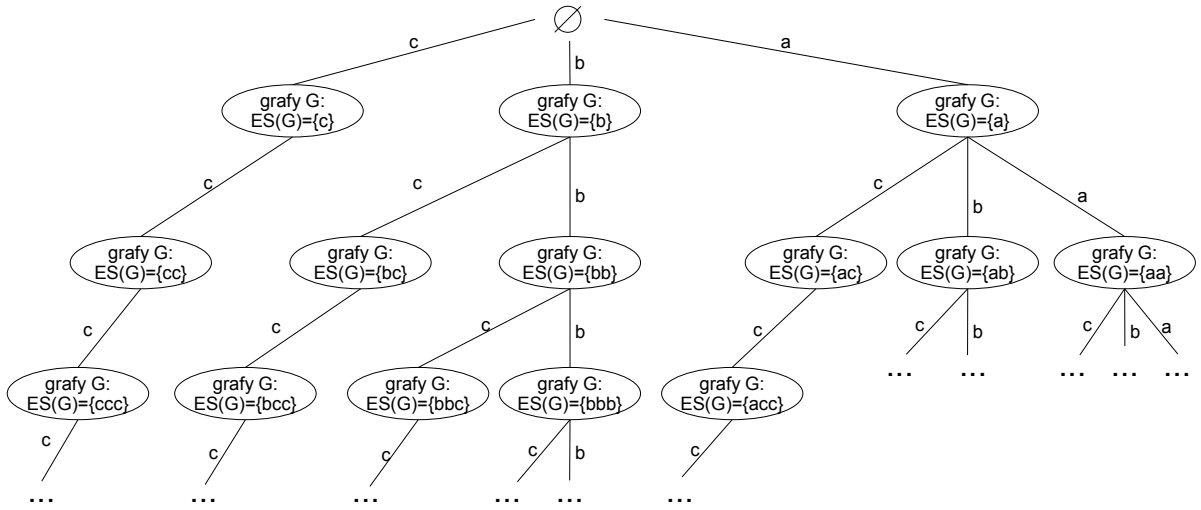
wsparcia można zakończyć wnioskiem, że C nie jest grafem częstym. Nawet, gdyby pozostałe trzy testy zakończyły się pozytywnie, wsparcie kandydata wynosiłoby 4, a więc byłoby poniżej progu minimalnego wsparcia.

Skuteczność tej metody zależy w dużej mierze od kolejności w jakiej pobierane są grafy ze zbioru $\mathbb{D}$. Teoretycznie najlepsze wyniki daje ustalenie takiego porządku, aby w pierwszej kolejności wykonywane były testy, które kończą się wynikiem negatywnym oraz wykonują się szybko. Pożądane cechy nie są znane z góry, dlatego w algorytmie UGM stosowane są heurystyczne oszacowania. Jedną z proponowanych heurystyk jest posortowanie zbioru $\mathbb{D}$ według stopnia skomplikowania grafu szacowanego na podstawie liczby wierzchołków i krawędzi. Zakłada się tu, że im więcej graf ma wierzchołków i krawędzi tym dłużej wykonywany będzie test na izomorfizm z podgrafem. Inną proponowaną heurystyką jest posortowanie zbioru $\mathbb{D}$ wg liczby operacji podstawowych algorytmu badającego izomorfizm rodzica kandydata z podgrafem grafów ze zbioru $\mathbb{D}$.

4.1.5. Generowanie kandydatów

Algorytm generuje kandydatów przez rozszerzanie grafów częstych o kolejne krawędzie z listy deskryptorów krawędzi częstych. Na początku algorytmu jedynym znanym grafem częstym jest graf bez wierzchołków, jeśli $|\mathbb{D}| \geq minSup$. Lista deskryptorów krawędzi częstych jest uporządkowana nierosnąco według wsparcia. Każdy graf częsty jest rozszerzany o kolejne krawędzie pobierane od końca listy, co nie ma znaczenia dla poprawności wyniku, ale jak będzie wyjaśnione dalej, przyspiesza wykonanie algorytmu. Dodatkowo, jeśli graf częsty zawiera krawędź o deskryptorze znajdującym się na k-tej pozycji we wspomnianej liście, wtedy nie jest rozszerzany o żadną krawędź o deskryptorze znajdującym się na pozycji mniejszej niż k na tej liście. Kandydaci są generowani w głąb.

Przykład 4.3. Ponownie rozpatrzmy odkrywanie grafów częstych wśród zbioru grafów przedstawionych na rysunku 4.1 przy założeniu progu minimalnego wsparcia $minSup = 2$. W podanym zbiorze grafów istnieją trzy częste krawędzie o deskryptorach $a = (\{1, 1\}, 5)$, $b = (\{0, 1\}, 4)$, $c = (\{0, 0\}, 4)$. Wsparcia krawędzi a, b, c wynoszą odpowiednio $3, 2, 2$. Lista deskryptorów uprządkowana nierosnąco wg wspracia ma zatem postać (a, b, c). Rysunek 4.2 przestawia drzewo przeszukiwania grafów częstych, przy założeniu, że nie są stosowane żadne techniki optymalizacyjne. Każdy węzeł drzewa oznacza zbiór grafów kandydujących o danym wielozbiorze deskryptorów krawędzi. Każda krawędź drzewa oznacza proces



Rysunek 4.2. Drzewo przeszukiwań algorytmu UGM bez uwzględniania optymalizacji.

rozszerzania jednego grafu o jedną krawędź o danym deskryptorze na wszystkie możliwe sposoby. Drzewo jest generowane w głąb. Na szczycie drzewa znajduje się graf bez wierzchołków. Jest on w pierwszej kolejności rozszerzany o krawędź c, gdyż krawędzie są wybierane od końca listy. Utworzony graf jest rozszerzany na wszystkie możliwe sposoby (patrz rozdział 4.1.6) o krawędź c tworząc zbiór grafów, których wielozbiór deskryptorów krawędzi jest równy $\{cc\}$. Każdy z grafów tego zbioru jest ponownie rozszerzany o krawędź c, tworząc zbiór grafów, których wielozbiór deskryptorów krawędzi jest równy $\{ccc\}$. Grafy będą rozszerzane o krawędź c dopóty, dopóki będą powstawały grafy częste. Następnie graf bez wierzchołków, znajdujący się w korzeniu drzewa, zostanie rozszerzony o krawędź b, a uzyskany graf będzie rozszerzany rekurencyjnie najpierw o krawędź c (tworząc grafy, których wielozbiór deskryptórów jest równy kolejno $\{bc\}, \{bcc\} \ldots$) , a następnie o krawędź b. Można zaobserwować, że w momencie, gdy graf zostanie rozszerzony o krawędź c, uzyskany zbiór grafów nie będzie już rozszerzany ani o krawędź krawędź b, ani o krawędź a. Podobnie, jeśli graf zostanie rozszerzony o krawędź b, uzyskany zbiór grafów nie będzie rozszerzany o krawędź a.

Wybieranie krawędzi z końca listy nie jest obowiązkowe dla uzyskania poprawnego wyniku, czyli wszystkich grafów częstych, ale ma tę zaletę, że zbiór grafów nieczęstych będzie się wypełniał grafami, które potencjalnie są izomorficzne z podgrafami kandydatów z później rozpatrywanych gałęzi drzewa przeszukiwań. W przykładzie powyżej najpierw generowane są grafy o krawędziach $c, c, \ldots$. Podczas generowania kandydatów o krawędziach $b, c, c, \ldots$

będzie można pominąć kandydatów, którzy są nadgrafami nieczęstych grafów o krawędziach $c, c, \ldots$.

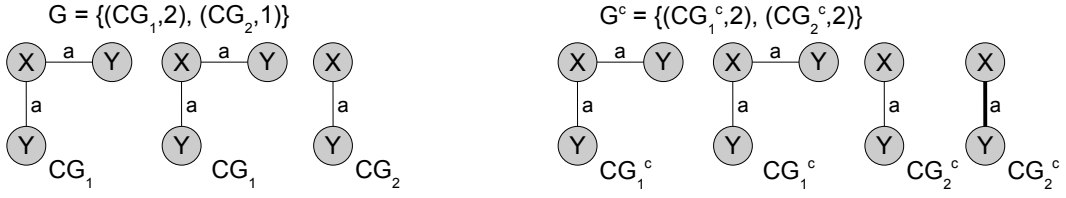
Operacja rozszerzania grafu częstego o krawędź o deskryptorze ed (patrz rozdział 4.1.6) zwraca zbiór grafów kandydujących, w których nie ma duplikatów. Operacja ta nie gwarantuje jednak, że wygenerowane grafy nie były wcześniej rozpatrywane w innej gałęzi drzewa przeszukiwań, co obrazuje podany wcześniej rysunek 2.8. Dlatego też wszystkie rozpatrzone grafy są zapamiętywane. W momencie powstania nowego kandydata sprawdzane jest, czy zbiór rozpatrzonych dotychczas grafów o tym samym wielozbiorze deskryptorów zawiera izomorficzny z nim graf. Jeśli zawiera, rozpatrywanie kandydata jest pomijane.

4.1.6. Rozszerzanie grafu

Graf G jest reprezentowany jako zbiór par (CG_i, c_i), gdzie CG_i jest i-tą składową spójną grafu G, c_i jest liczbą wystąpień tej składowej w grafie G, $i = 1 \ldots n$ i $\forall_{i \neq j}\ CG_i$ nie jest izomorficzny z CG_j. Graf $G = \{(CG_1, c_1), (CG_2, c_2), \ldots (CG_n, c_n)\}$ może zostać rozszerzony o krawędź o deskryptorze $ed = (\{lv_1, lv_2\}, le)$ na trzy sposoby poprzez:

1. utworzenie nowej składowej spójnej,
2. rozszerzenie istniejącej składowej spójnej,
3. połączenie dwóch istniejących składowych spójnych.

Utworzenie nowej składowej spójnej polega na utworzeniu składowej spójnej o dwóch wierzchołkach o etykietach lv_1, lv_2 oraz jednej krawędzi o etykiecie le, która łączy te wierzchołki. W wyniku może powstać nowa para $(CG_{n+1}, 1)$ w przypadku, gdy utworzona składowa nie jest izomorficzna z żadną inną składową spójną. W przeciwnym przypadku zwiększeniu ulegnie licznik c_i w odpowiedniej parze (CG_i, c_i).



Rysunek 4.3. Rozszerzanie grafu G o krawędź o deskryptorze (X, Y, a) poprzez utworzenie nowej składowej spójnej. Wynikiem rozszerzenia jest jeden graf G^c.

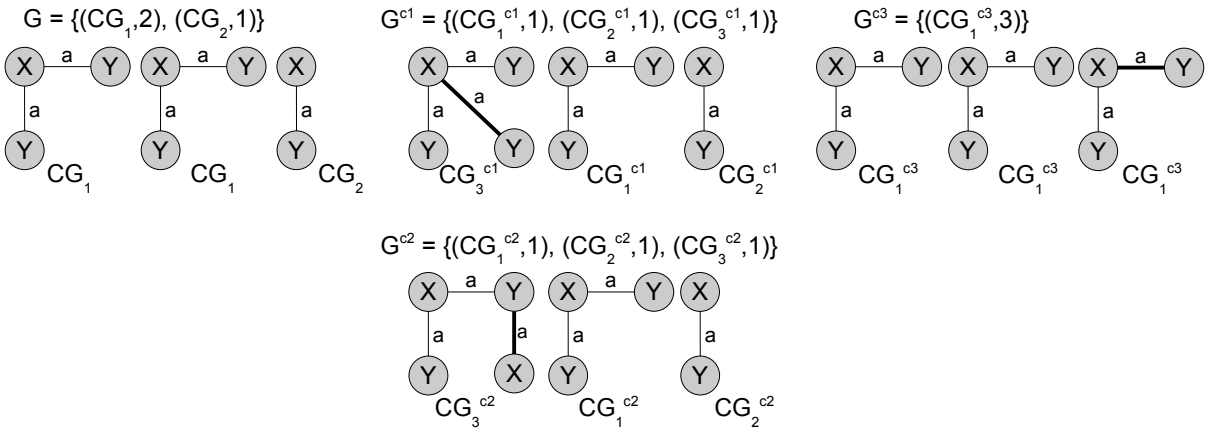
Przykład 4.4. *Na rysunku 4.3 pokazane jest rozszerzanie grafu G o krawędź o deskryptorze (X, Y, a) poprzez utworzenie nowej składowej spójnej. W tym przypadku nie powstaje nowa para $(CG, 1)$, gdyż graf G zawiera składową spójną składającą się z jednej krawędzi o deskryptorze (X, Y, a). Zwiększeniu ulega więc licznik przy składowej spójnej CG_2.*

Rozszerzanie istniejącej składowej spójnej polega na dodaniu nowej krawędzi do istniejącej składowej spójnej. Ponieważ jedną krawędź można dodać na wiele sposobów, w wyniku rozszerzania może powstać wiele nowych składowych spójnych. Schemat procedury jest następujący. Dla każdej pary (CG_i, c_i) wykonaj:

— Dodaj nowy wierzchołek o etykiecie lv_1 do grafu CG_i i połącz go kolejno krawędziami o etykiecie le z każdym wierzchołkiem grafu CG_i, które mają etykietę lv_2, tworząc za każdym razem nową składową. W efekcie może powstać wiele składowych spójnych, wśród których mogą znajdować się duplikaty.

— Jeżeli $lv_1 \neq lv_2$, dodaj nowy wierzchołek o etykiecie lv_2 do grafu CG_i i połącz go kolejno krawędziami o etykiecie le z każdym wierzchołkiem grafu CG_i, które mają etykietę lv_1, tworząc za każdym razem nową składową. W efekcie może powstać wiele składowych spójnych, wśród których mogą znajdować się duplikaty.

— Każdą parę wierzchołków z grafu CG_i o etykietach lv_1 i lv_2 połącz kolejno krawędzią o etykiecie le, tworząc za każdym razem nową składową spójną. W efekcie może powstać wiele składowych spójnych, wśród których mogą znajdować się duplikaty.



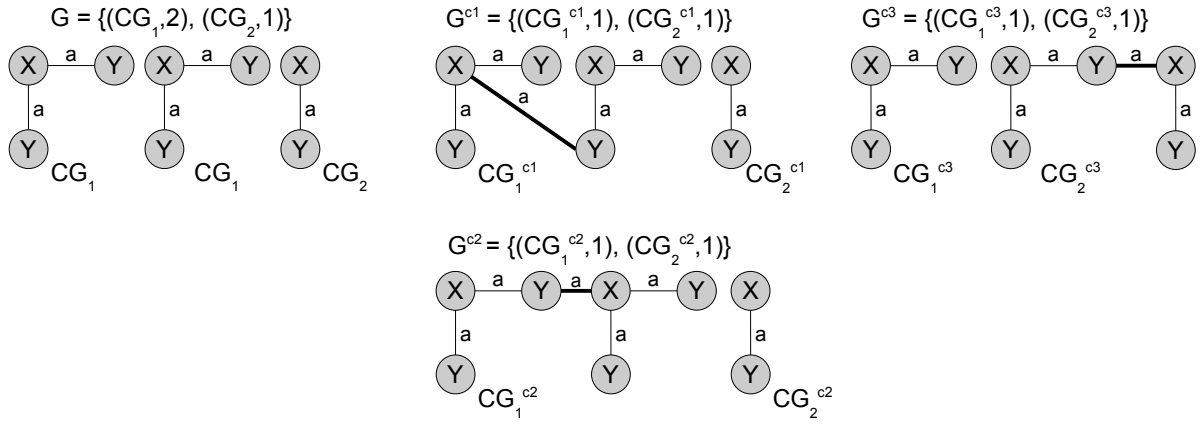
Rysunek 4.4. Rozszerzanie grafu G o krawędź o deskryptorze (X, Y, a) poprzez rozszerzanie składowych spójnych grafu G. Wynikiem rozszerzenia są trzy nieizomorficzne grafy G^{c1}, G^{c2}, G^{c3}.

Przykład 4.5. *Na rysunku 4.4 pokazane jest rozszerzanie grafu G o krawędź o deskryptorze (X, Y, a) poprzez rozszerzanie składowych spójnych grafu G. Do składowej CG_1 można dodać krawędź (X, Y, a) na trzy sposoby (dodając krawędź do każdego z trzech wierzchołków), ale powstaną tylko dwa nieizomorficzne grafy CG_3^{c1} i CG_3^{c2}. Do składowej CG_2 można dodać krawędź (X, Y, a) na dwa sposoby, ale z obu powstaną grafy izomorficzne z CG_1^{c3}. W wyniku rozszerzenia powstaną więc trzy nieizomorficzne grafy G^{c1}, G^{c2}, G_{c3}.*

Połączenie dwóch istniejących składowych spójnych polega na dodaniu krawędzi o etykiecie le pomiędzy wierzchołki o etykietach lv_1 i lv_2 z dwóch różnych składowych spójnych. Mogą wystąpić dwa przypadki:

— Połączenie dwóch składowych w ramach jednej pary (CG, c). W tym przypadku łączymy ze sobą dwa różne wystąpienia tej samej składowej CG. Aby to było możliwe muszą istnieć co najmniej dwa wystąpienia tej składowej, czyli $c \geq 2$. Ponieważ połączenie można wykonać na wiele sposobów, w wyniku może powstać wiele składowych spójnych, wśród których mogą znajdować się duplikaty.

— Połączenie dwóch składowych w ramach dwóch różnych par (CG_i, c_i) i $(CG_j, c_j), i \neq j$, $i, j = 1 \ldots n$. Ponieważ połączenie można wykonać na wiele sposobów, w wyniku może powstać wiele składowych spójnych, wśród których mogą znajdować się duplikaty.



Rysunek 4.5. Rozszerzanie grafu G o krawędź o deskryptorze (X, Y, a) poprzez utworzenie nowej składowej spójnej. Wynikiem rozszerzenia jest jeden graf G^c.

Przykład 4.6. *Na rysunku 4.5 pokazane jest rozszerzanie grafu G o krawędź o deskryptorze (X, Y, a) poprzez połączenie dwóch składowych spójnych grafu G. Dwie składowe spójne CG_1 można ze sobą połączyć na dwa sposoby, prowadzące do powstania składowych spójnych CG_1^{c1}*

oraz CG_1^{c2}. Składowe CG_1 i CG_2 można ze sobą połączyć na kilka sposobów, ale we wszystkich przypadkach powstaną grafy izomorficzne z CG_2^{c3}. W wyniku rozszerzania powstaną więc trzy nieizomorficzne grafy G^{c1}, G^{c2}, G^{c3}.

W utworzonym za pomocą tej procedury zbiorze nowych grafów wykonuje się następnie usuwanie grafów izomorficznych.

4.1.7. Struktury danych

Algorytm UGM utrzymuje cztery istotne zbiory: zbiór maksymalnych częstych wielozbiorów deskryptorów krawędzi, zbiór rozpatrzonych nieizomorficznych grafów, zbiór grafów nieczęstych, zbiór nieczęstych konstruktorów rozszerzeń. Zbiory te powinny być umieszczone w strukturach wspomagających szybkie wykonywanie operacji często wykonywanych w algorytmie. Poniżej podane są oczekiwane własności każdej ze struktur oraz sposób implementacji w algorytmie UGM.

Maksymalne częste wielozbiory deskryptorów krawędzi

Struktura przechowująca maksymalne częste wielozbiory deskryptorów krawędzi powinna umożliwiać szybkie odpowiadanie na pytanie: Czy dany zbiór $ES(G)$ jest podzbiorem dowolnego ze zbiorów znajdujących się w tej strukturze? Przegląd mechanizmów indeksowania zbiorów, które wspierają zapytania o podzbiory znajduje się w pracy [58]. Algorytm UGM wykorzystuje implementację opartą na drzewach przedrostkowych.

Zbiór rozpatrzonych nieizomorficznych grafów

Struktura przechowująca rozpatrzone nieizomorficzne grafy powinna umożliwiać szybkie odpowiadanie na pytanie: Czy dany graf G jest izomorficzny z dowolnym z grafów znajdujących się w tej strukturze? W algorytmie UGM, aby odpowiedzieć na to pytanie, wykonuje się testy na izomorfizm grafu G z takimi grafami $G_i \in \mathbb{D}$, że $ES(G) = ES(G_i)$. Struktura przechowująca zbiór rozpatrzonych grafów jest tablicą asocjacyjną opartą na tablicy mieszającej, która przyporządkowuje kluczowi es zbiór takich grafów G_i, że $es = ES(G_i)$.

Zbiór grafów nieczęstych

Struktura przechowująca grafy nieczęste powinna umożliwiać szybkie odpowiadanie na pytanie: Czy dany graf G posiada podgraf izomorficzny z dowolnym z grafów znajdujących się w tej strukturze? W algorytmie UGM struktura ta została zaimplementowana jako zestaw

tablic asocjacyjnych. Każda tablica przechowuje grafy nieczęste o innej liczbie krawędzi. W tablicy asocjacyjnej kluczem jest zbiór deskryptorów, a wartością zbiór grafów nieczęstych o danym wielozbiorze deskryptorów krawędzi. Szukanie grafu izomorficznego z podgrafem danego grafu G polega na przeszukiwaniu kolejnych tablic asocjacyjnych, zaczynając od tablicy zawierającej grafy o najmniejszej liczbie krawędzi. Przeszukiwanie kończy się w momencie znalezienia grafu izomorficznego z podgrafem grafu G lub przeszukaniu wszystkich tablic zawierających grafy o liczbie krawędzi nie większej niż liczba krawędzi grafu G. Przeszukiwanie tablicy asocjacyjnej polega na znalezieniu wszystkich kluczy, które są podzbiorami zbioru $ES(G)$, a następnie wykonaniu testów na izomorfizm grafów ze zbiorów wskazywanych przez te klucze z podgrafem grafu G.

Zbiór nieczęstych konstruktorów rozszerzeń

Struktura przechowująca nieczęste konstruktory rozszerzeń powinna umożliwiać szybkie odpowiadanie na pytanie: Czy dla danej pary (graf G, deskryptor krawędzi ed) istnieje w strukturze taki konstruktor $k = (G_1, ed)$, że graf G_1 jest izomorficzny z podgrafem grafu G. Struktura jest zaimplementowana analogicznie do struktury przechowującej grafy nieczęste, przy czym z każdym grafem znajdującym się w tej strukturze jest związany dodatkowy wielozbiór deskryptorów krawędzi.

Eksperymenty pokazują, że obsługa i przeszukiwanie zaimplementowanych w powyższy sposób struktur zajmuje niewielką część czasu działania algorytmu UGM. Jedynie obsługa zbioru grafów nieczęstych staje się znacząco istotna w przypadku niskich wartości wsparcia.

4.1.8. Algorytm UGM

W opisie algorytmu będzie używana następująca notacja:

$maxES$	zbiór maksymalnych częstych wielozbiorów deskryptorów krawędzi
$\mathbb{GN}$	zbiór grafów nieczęstych
$\mathbb{NKR}$	zbiór nieczęstych konstruktorów rozszerzeń
$allGraphs$	zbiór rozpatrzonych nieizomorficznych grafów

Algorytm UGM (algorytm 4.14) rozpoczyna się od wyznaczenia wielozbiorów deskrytptorów krawędzi $ES(G)$ dla każdego grafu z wejściowej bazy danych $\mathbb{D}$. Następnie

w kolekcji tych wielozbiorów szuka się maksymalnych wielozbiorów częstych przy progu minimalnego wsparcia równym $minSup$. Uzyskane maksymalne wielozbiory częste są przechowywane w zmiennej $maxES$. Deskryptory częstych krawędzi są umieszczane na liście $frequentEdges$, która jest następnie sortowana nierosnąco według wsparcia. Kryterium sortowania nie ma wpływu na uzyskany wynik algorytmu, a jedynie na szybkość jego działania. Z wszystkich grafów zbioru $\mathbb{D}$ usuwane są krawędzie, których deskryptory nie znajdują się na liście $frequentEdges$. Następnie inicjalizowane są struktury służące przechowywaniu wszystkich rozpatrywanych nieizomorficznych grafów $allGraphs$, zbioru grafów nieczęstych $\mathbb{GN}$ oraz zbioru nieczęstych konstruktorów rozszerzeń $\mathbb{NKR}$. Następnie kolejno pobierane są deskryptory krawędzi z listy $frequentEdges$ zaczynając od końca i dla każdego takiego deskryptora, powiedzmy $newEdge$, wywoływana jest procedura $UgmDfsSearch(\textit{graf bez wierzchołków}, newEdge, \mathbb{D})$, która rozpoczyna rekurencyjne poszukiwanie grafów częstych zaczynając od rozszerzenia grafu bez wierzchołków o krawędź o deskryptorze $newEdge$. Wynikiem algorytmu UGM są wszystkie grafy częste znajdujące się w strukturze $allGraphs$.

Procedura $UgmDfsSearch(G, newEdge, \mathbb{D})$ rozpoczyna się od sprawdzenia, czy zbiór nieczęstych konstruktorów rozszerzeń zawiera taki konstruktor $k = (G_1, newEdge)$, że G_1 jest grafem izomorficznym z podgrafem grafu G. Jeśli zawiera taki konstruktor, procedura się kończy, gdyż zagwarantowane jest, że wszystkie rozszerzenie grafu G o krawędź o deskryptorze $newEdge$ są nieczęste. W przeciwnym przypadku generowane są wszystkie rozszerzenia grafu G o krawędź $newEdge$ za pomocą funkcji $UGMgenerateCandidates(G, newEdge)$ tworząc zbiór kandydatów $\mathbb{C}$ oraz funkcja $UGMgenerateChildren(\mathbb{C}, newEdge, G)$, która zwraca zbiór takich par $(G', newEdge')$, że G' jest grafem częstym ze zbioru $\mathbb{C}$, a $newEdge'$ jest deskryptorem krawędzi, o którą będzie rozszerzany graf G' w następnej kolejności. Dla każdej pary $(G', newEdge')$ otrzymanej za pomocą funkcji $UGMgenerateChildren(\mathbb{C}, newEdge, G)$ wywoływana jest rekurencyjnie procedura $UgmDfsSearch(G', newEdge', G'.supporitngSet)$.

Celem procedury $UgmGenerateCandidates(G, newEdge)$ jest utworzenie wszystkich rozszerzeń grafu G o krawędź o deskryptorze $newEdge$. Każde rozszerzenie będzie miało wielozbiór deskryptorów równy $ES(G) \cup \{newEdge\}$, więc na początku sprawdzane jest, czy ten wielozbiór deskryptorów jest podzbiorem dowolnego ze zbiorów znajdujących się w $maxES$. Jeśli nie jest, procedura się kończy, gdyż zagwarantowane jest, że wszystkie

Algorytm 4.14 UGM($\mathbb{D}, minSup$)

$ESlist \leftarrow \emptyset$; $/*$ lista wielozbiorów deskryptorów krawędzi $*/$
$edges \leftarrow \emptyset$; $/*$ lista deskryptorów krawędzi wraz z ich wsparciami $*/$
for all graf $G \in \mathbb{D}$ **do**
  dodaj $ES(G)$ do listy $ESlist$;
  for all unikalny deskryptor krawędzi $ed \in ES(G)$ **do**
    if $ed \in edges$ **then**
      $ed.sup \leftarrow ed.sup + 1$;
    else
      dołącz ed do listy $edges$;
      $ed.sup \leftarrow 1$;
    end if
  end for
end for
znajdź maksymalne częste wielozbiory w liście $ESlist$ o wsparciu równym co najmniej $minSup$ i zapisz wynik w $maxES$;
$frequentEdges \leftarrow \{ed \in edges \mid ed.sup \geq minSup\}$;
posortuj nierosnąco $frequentEdges$ według wsparcia;
for all $G \in \mathbb{D}$ **do**
  usuń nieczęste krawędzie z G;
end for
$allGraphs \leftarrow \emptyset$; $/*$ zbiór wszystkich rozpatrzonych grafów $*/$
$\mathbb{GN} \leftarrow \emptyset$;
$\mathbb{NKR} \leftarrow \emptyset$;
for $i = |frequentEdges|$ DOWNTO 1 **do**
  $newEdge \leftarrow frequentEdges[i]$;
  $UgmDfsSearch$(graf bez wierzchołków, $newEdge, \mathbb{D}$);
end for
return częste grafy ze zbioru $allGraphs$;

Algorytm 4.15 UgmDfsSearch($G, newEdge, \mathbb{D}$)

if $unsuccessfulExtensions$ zawiera parę $(G_1, newEdge)$ taką, że G_1 jest izomorficzny z podgrafem grafu G **then**
  return
end if
$\mathbb{C} \leftarrow UGMgenerateCandidates(G, newEdge)$;
for all $(G', newEdge') \in generateChildren(\mathbb{C}, newEdge, G)$ **do**
  $UgmDfsSearch(G', newEdge', G'.supportingSet)$;
end for

rozszerzenia nie są częste. W przeciwnym przypadku następuje generowanie rozszerzeń grafu G reprezentowanego jako zbiór składowych spójnych $\{(CG_1, c_1), (CG_2, c_2), \ldots (CG_n, c_n)\}$. Generowanie rozszerzeń odbywa się w czterech etapach.

Algorytm 4.16 UgmGenerateCandidates($G, newEdge$)

$/*$ assert: G jest reprezentowany w postaci zbioru składowych spójnych wraz z ich liczbą wystąpień w grafie: $G = \{(CG_1, c_1), (CG_2, c_2), \ldots (CG_n, c_n)\}$ $*/$
if $ES(G) \cup \{newEdge\}$ nie jest podzbiorem żadnego ze zbiorów z $maxES$ **then**
  return $\emptyset$;
end if

$R \leftarrow \emptyset$;
for $i \leftarrow 1$ to n **do**
  if $c_i > 1$ **then**
    $R \leftarrow R \cup$ grafy powstałe z G w wyniku połączenia dwóch składowych CG_i krawędzią $newEdge$;
  end if
end for
for $i \leftarrow 1$ to $n - 1$ **do**
  for $j \leftarrow i + 1$ to n **do**
    $R \leftarrow R \cup$ grafy powstałe z G w wyniku połączenia dwóch składowych CG_i i CG_j krawędzią $newEdge$;
  end for
end for
for $i \leftarrow 1$ to n **do**
  $R \leftarrow R \cup$ grafy powstałe z G w wyniku rozszerzenia składowej CG_i o krawędź $newEdge$;
end for
$R \leftarrow R \cup G$ rozszerzony o nową składową spójną, która jest grafem o jednej krawędzi $newEdge$;
usuń duplikaty z R; $/*$ można pominąć, jeśli operacja $R \leftarrow R \cup \{G\}$ nie dodaje grafu G do zbioru R, gdy R zawiera graf izomorficzny z G $*/$
return R;

Na początku tworzone są rozszerzenia, które powstają przez połączenie dwóch składowych spójnych w ramach jednej pary (CG_i, c_i). Dla każdego $i = 1 \ldots n$ jeśli $c_i > 1$, dwie składowe spójne CG_i łączone są ze sobą na wszystkie możliwe sposoby za pomocą krawędzi o deskryptorze $newEdge$. Wszystkie utworzone grafy są dodawane do zbioru wynikowego R.

Na drugim etapie tworzone są rozszerzenia, które powstają przez połączenie dwóch składowych spójnych w ramach różnych par (CG_i, c_i). Dla każdej wartości i oraz j takich, że $1 \leq i < j \leq n$, składowe spójne CG_i i CG_j są łączone ze sobą na wszystkie możliwe

sposoby za pomocą krawędzi o deskryptorze $newEdge$. Utworzone grafy są dodawane do zbioru wynikowego R.

Na kolejnym etapie generowane są rozszerzenia, które powstają przez rozszerzenie jednej składowej spójnej (CG_i, c_i) o krawędź o deskryptorze $newEdge$. Rozszerzanie odbywa się dla każdej składowej spójnej CG_i, $1 \leq i \leq n$, i polega na dodaniu na wszystkie możliwe sposoby krawędzi o deskryptorze $newEdge$ do składowej CG_i przez dodanie krawędzi pomiędzy istniejące wierzchołki lub dodanie nowego wierzchołka i połączeniu go krawędzią do istniejącego wierzchołka. Utworzone grafy są dodawane do zbioru wynikowego R.

Na ostatnim etapie tworzone jest jedno rozszerzenie, które powstaje przez dodanie do grafu G nowej składowej spójnej, która jest grafem o jednej krawędzi o deskryptorze $newEdge$. To rozszerzenie również jest dodawane do zbioru wynikowego R.

W wyniku powyższych kroków zbiór R może zawierać duplikaty. Przed zwróceniem zbioru R i wyjściem z procedury należy więc usunąć z niego wszystkie duplikaty. Ten ostatni krok można pominąć, jeśli zapewni się, że operacja dodawania grafu G do zbioru R będzie sprawdzała, czy R zawiera graf izomorficzny z G i jeśli zawiera, graf G nie zostanie dodany do zbioru R.

Procedura $UgmGenerateChildren(\mathbb{C}, lastEdge, parent)$ znajduje grafy częste znajdujące się w zbiorze grafów kandydujących $\mathbb{C}$, które to powstały przez rozszerzenie grafu $parent$ o krawędź o deskryptorze $lastEdge$. Procedura zwraca listę par $(G, newEdge)$, gdzie $G \in \mathbb{C}$ jest grafem częstym, a $newEdge$ jest deskryptorem krawędzi o jaką należy rozszerzyć G w następnej kolejności, czyli deskryptorem znajdującym się na liście deskryptorów nie wcześniej niż desktyptor $lastEdge$. Procedura zaczyna się od przypisania wartości fałszu do zmiennej $anyFrequentCandidates$, która na końcu procedury będzie informowała o tym, czy w zbiorze $\mathbb{C}$ znajdowały się grafy częste. Następnie ze zbioru $\mathbb{C}$ usuwane są grafy, które są izomorficzne z grafami znajdujących się w zbiorze $allGraphs$, czyli zbiorze już rozpatrzonych grafów. Jednocześnie w przypadku, gdy ze zbioru $\mathbb{C}$ usuwany jest graf częsty, do zmiennej $anyFrequentCandidates$ przypisywana jest wartość prawdy. W dalszej kolejności dla każdego grafu G ze zbioru $\mathbb{C}$ wywoływana jest procedura $UgmIsFrequent(G, parent.supportingSet)$, która określa czy graf G jest częsty w zbiorze wspierającym rodzica grafu G. Jeśli G okaże się częsty, do zmiennej $anyFrequentCandidates$ przypisywana jest wartość prawdy i tworzone są wszystkie takie pary $(G, newEdge)$, że $newEdge$ pochodzi ze zbioru $frequentEdges$

i $newEdge \geq lastEdge$. Utworzone pary są dodawane do zbioru R, który jest zbiorem wynikowym procedury. W przypadku, gdy G nie jest częsty, jest on umieszczany w zbiorze grafów nieczęstych $\mathbb{GN}$. Niezależnie od tego, czy graf G jest częsty czy nie, jest on dodawany do zbioru rozpatrzonych grafów $allGraphs$. Po rozpatrzeniu wszystkich kandydatów ze zbioru $\mathbb{C}$ sprawdzana jest wartość zmiennej $anyFrequentCandidates$. Jeśli jest ona fałszem, wtedy para $(parent, lastEdge)$ dodawana jest do zbioru $\mathbb{NKR}$. Oznacza to, że żadne rozszerzenie grafu $parent$ o krawędź $lastEdge$ nie jest grafem częstym.

Algorytm 4.17 UgmGenerateChildren($\mathbb{C}, lastEdge, parent$)

$anyFrequentCandidates \leftarrow false$;
for all $G \in \mathbb{C}$ **do**
  if zbiór $allGraphs$ zawiera graf izomorficzny z G **then**
    usuń G z $\mathbb{C}$; $/*$ nie rozpatruj grafów, które są izomorficzne z grafami już rozpatrzonymi $*/$
    if G jest częsty **then**
      $anyFrequentCandidates \leftarrow true$;
    end if
  end if
end for
for all $G \in \mathbb{C}$ **do**
  if $UgmIsFrequent(G, minSup, parent.supportingSet)$ **then**
    $noFrequentCandidates \leftarrow false$;
    for $i = |freqentEdges|$ DOWNTO 1 **do** $/*$ $pos(ed)$ jest pozycją desktyptora krawędzi ed na liście $frequentEdges$ $*/$
      if $pos(newEdge) < pos(lastEdge)$ **then**
        break;
      end if
      $R \leftarrow R \cup (G, newEdge)$;
    end for
  else
    $\mathbb{GN} \leftarrow \mathbb{GN} \cup \{G\}$;
  end if
  $allGraphs \leftarrow allGraphs \cup \{G\}$;
end for
if $anyFrequentCandidates = false$ **then**
  $\mathbb{NKR} \leftarrow \mathbb{NKR} \cup \{k = (parent, lastEdge)\}$;
end if
return R;

Procedura $UgmIsFrequent(G, minSup, \mathbb{D})$ sprawdza, czy graf G jest częsty w zbiorze grafów $\mathbb{D}$. Na początku wykonywany jest test liczności zbioru $\mathbb{D}$ i jeżeli $|\mathbb{D}| < minSup$, wtedy zwracany jest wynik negatywny. Wynik negatywny zwracany jest też w przypadku, gdy zbiór grafów nieczęstych $\mathbb{GN}$ zawiera dowolny graf, który jest izomorficzny z podgrafem G. Jeśli

procedura nie skończyła się wynikiem negatywnym, wykonywane są testy na izomorfizm grafu G z podgrafem każdego grafu ze zbioru $\mathbb{D}$. Jeśli wynik testu jest pozytywny, to zwiększane jest wsparcie grafu G oraz dodawany jest odpowiedni graf ze zbioru $\mathbb{D}$ do zbioru wspierającego graf G. Liczba testów, które skończyły się wynikiem negatywnym, przechowywana jest w zmiennej $falseResults$. W przypadku, gdy wartość $falseResults$ przekroczy wartość $|\mathbb{D}| - minSup$ wykonywanie testów na izomorfizm jest przerywane i procedura zwraca wynik negatywny. Po wykonaniu wszystkich testów, wartość wsparcia grafu sup jest konfrontowana z wartością progu minimalnego wsparcia $minSup$ i jeśli $sup \geq minSup$ zwracany jest wynik pozytywny, a w przeciwnym przypadku G jest dodawany do zbioru $\mathbb{GN}$ i zwracany jest wynik negatywny.

Algorytm 4.18 UgmIsFrequent($G, minSup, \mathbb{D}$)

if $|\mathbb{D}| < minSup$ **then**
  return $false$;
end if
if $negativeBorder$ zawiera jakikolwiek graf izomorficzny z podgrafem G **then**
  return $false$;
end if
$sup \leftarrow 0$; $falseResults \leftarrow 0$;
for all $G_i \in D$ **do**
  if graf G jest izomorficzny z podgrafem grafu G_i **then**
    $sup \leftarrow sup + 1$;
    $G.supportingSet \leftarrow G.supportingSet \cup G_i$;
  else
    $falseResults \leftarrow falseResults + 1$;
    if $falseResults > |\mathbb{D}| - minSup$ **then**
      return $false$;
    end if
  end if
end for
if $sup < minSup$ **then**
  $negativeBorder \leftarrow negativeBorder \cup \{G\}$;
  return $false$;
else
  return $true$;
end if

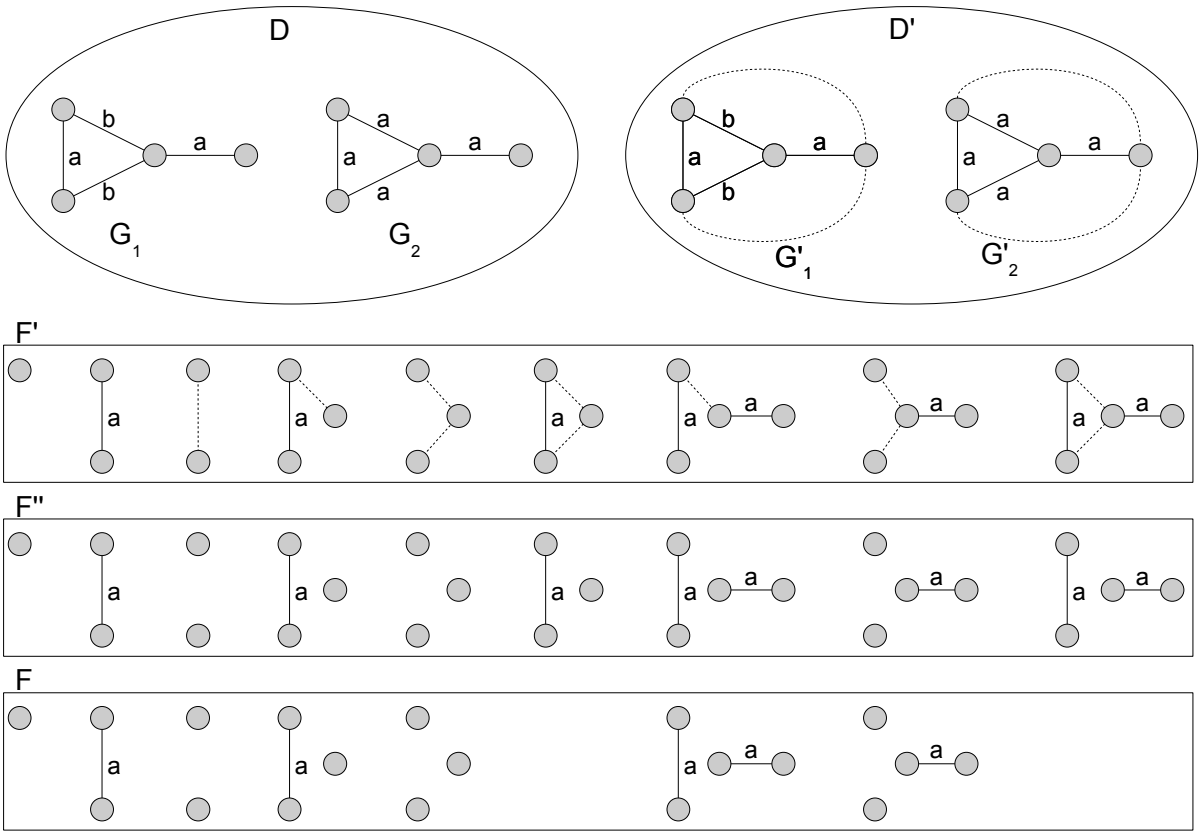
4.2. Algorytmy bazujące na wstępnym odkrywaniu grafów spójnych

W dziedzinie odkrywania spójnych grafów częstych zaproponowano wiele dobrych rozwiązań. Naturalne jest więc spróbowanie wykorzystania częstych grafów spójnych jako

punktu wyjścia do odkrywania grafów niespójnych. W tego typu podejściu wyznaczanie częstych grafów spójnych jest pierwszym etapem odkrywania grafów częstych. W niniejszej rozprawie są przedstawione dwie metody tego typu. Pierwsza, polegająca na wstępnym uzupełnieniu grafów z bazy wejściowej o sztuczne krawędzie, została zaczerpnięta z pracy [41]. Druga, bazująca na generowaniu niespójnych kandydatów na podstawie częstych grafów spójnych, jest propozycją autora niniejszej rozprawy. Eksperymenty pokazują, że pierwsza metoda jest praktycznie bezużyteczna, gdyż kończy się w sensownym czasie jedynie dla bardzo wysokich progów wparcia. Natomiast druga metoda, nazywana UFC, choć zdecydowanie szybsza, jest o rząd wielkości wolniejsza od algorytmu UGM. Okazuje się jednak, że zastosowanie w drugiej metodzie, optymalizacji przygotowanych dla algorytmu UGM (w szczególności zbioru grafów częstych) sprawia, że staje się ona konkurencyjna. Do odkrywania częstych grafów spójnych został wykorzystany algorytm $Gaston$, gdyż według pracy [74] jest on najwydajniejszy spośród algorytmów platformy $ParMol$, co potwierdzają również eksperymenty prowadzone na potrzeby tej rozprawy.

4.2.1. Odkrywanie grafów niespójnych przez uzupełnienie grafów zbioru wejściowego o brakujące krawędzie

W tej metodzie należy dodać do grafów ze zbioru wejściowego krawędzie o specjalnej etykiecie VIRTUAL pomiędzy wierzchołkami, między którymi nie ma krawędzi. W ten sposób grafy stają się pełne. W tym zbiorze należy znaleźć wszystkie spójne podgrafy częste, a następnie usunąć w nich krawędzie o etykiecie VIRTUAL. W ten sposób otrzymamy zbiór wszystkich częstych grafów spójnych i niespójnych, w którym jednak mogą występować duplikaty, więc konieczne jest ich usunięcie. Działanie tej metody obrazuje rysunek 4.6. Szukamy grafów częstych w zbiorze $\mathbb{D}$ o wsparciu nie mniejszym niż 2. W tym celu dodajemy brakujące krawędzie do wszystkich grafów zbioru $\mathbb{D}$ tworząc w ten sposób zbiór $\mathbb{D}'$. Dodane krawędzie o etykiecie VIRTUAL są zaznaczone przerywaną linią. Spójne grafy częste ze zbioru $\mathbb{D}'$ znajdują się w zbiorze F'. Po usunięciu krawędzi VIRTUAL ze zbioru F' uzyskujemy zbiór F'' zawierający wszystkie spójne i niespójne grafy częste w zbiorze $\mathbb{D}$ łącznie z duplikatami. Po usunięciu duplikatów uzyskujemy ostateczny zbiór częstych grafów F. Aby uzyskać zgodność wyników z algorytmem UGM należałoby jeszcze usunąć ze zbioru F grafy, które zawierają izolowane wierzchołki. Omówioną procedurę ściślej



Rysunek 4.6. Odkrywanie częstych grafów spójnych i niespójnych przy $minSup = 2$ przez uzupełnienie grafów zbioru wejściowego o brakujące krawędzie. F' jest zbiorem częstych grafów spójnych w zbiorze $\mathbb{D}'$. F jest zbiorem częstych grafów spójnych i niespójnych w zbiorze $\mathbb{D}$.

przedstawia algorytm 4.19. Eksperymenty pokazują, że najbardziej czasochłonna jest pierwsza faza algorytmu, czyli wykrywanie grafów spójnych.

4.2.2. Odkrywanie częstych grafów niespójnych na podstawie częstych składowych spójnych

Zaproponowana metoda opiera się na następującej własności:

Własność 4.3. *Jeśli graf G składający się z k (niekoniecznie nieizomorficznych) składowych spójnych C_1, C_2, ... C_k jest częsty, to każda z jego składowych spójnych też jest grafem częstym.*

Własność ta wynika bezpośrednio z faktu, że każdy podgraf grafu częstego jest także częsty. Zatem jeśli najpierw odkryjemy wszystkie częste grafy spójne, to na ich podstawie można zbudować wszystkie częste grafy niespójne. W tej pracy proponuję metodę bazującą na algorytmie $Apriori$ [1], czyli tworzenie kandydatów o dwóch składowych spójnych z częstych grafów o jednej składowej spójnej, kandydatów o trzech składowych spójnych z częstych

Algorytm 4.19 UgmOnVirtual($\mathbb{D}, minSup$)

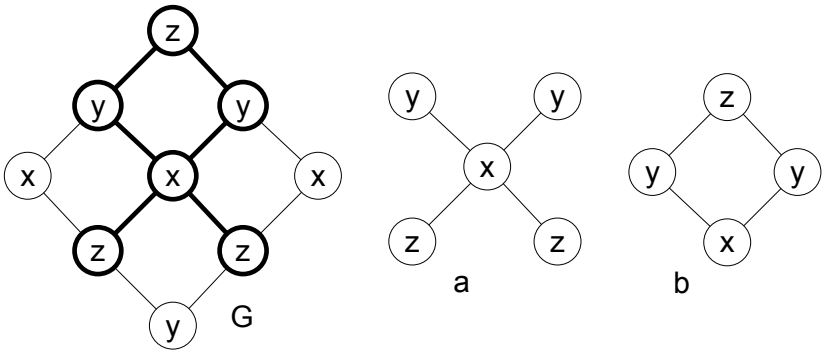
for all $G \in \mathbb{D}$ **do**
  for all $\{v_i \in G.V, v_j \in G.V\}$ **do**
    if graf G nie zawiera krawędzi pomiędzy wierzchołkami v_i i v_j **then**
      dodaj krawędź o etykiecie VIRTUAL pomiędzy wierzchołki v_i, v_j;
    end if
  end for
end for
$F \leftarrow ConnectedGraphMiner(\mathbb{D}, minSup)$; $/* ConnectedGraphMiner$ jest dowolnym algorytmem odkrywającym spójne grafy częste $*/$
for all $G \in F$ **do**
  w grafie G usuń wszystkie krawędzie o etykiecie VIRTUAL;
end for
usuń wszystkie duplikaty ze zbioru F;
return F;

grafów o dwóch składowych spójnych itd. Stosuję następującą notację:

F_k	lista grafów częstych o k spójnych składowych,
K_k	lista kandydatów na grafy częste o k spójnych składowych,
$G = (C_1, C_2, \ldots C_k)$	graf o k składowych spójnych,

Kolejne składowe spójne grafu G będą też oznaczane jako małe litery alfabetu $a, b, c, \ldots$, a grafy składające się kilku składowych - jako ciągi małych liter alfabetu. Np.: $aabd$ jest grafem o czterech składowych spójnych, w którym składowa a występuje dwukrotnie. Ciąg składowych musi być arbitralnie uporządkowany, gdyż ustalony początek będzie potem używany.

Do listy F_0 należy graf bez wierzchołków, gdy $|\mathbb{D}| \geq minSup$. Lista F_1 składa się ze wszystkich grafów częstych o jednej składowej spójnej, czyli grafów zwróconych przez algorytm odkrywania częstych grafów spójnych z pominięciem grafu bez wierzchołków. Np.: $F_1 = \{a, b, c, d, e\}$. Lista kandydatów o dwóch składowych powstaje przez łączenie składowych spójnych ze zbioru F_1. Zbiór kandydatów będzie miał postać: $K_2 = (aa, ab, ac, ad, ae, bb, bc, bd, be, cc, cd, ce, dd, de, ee)$. Dla każdego grafu kandydującego należy wyznaczyć jego wsparcie. Można to zrobić poprzez wykonanie testów na izomorfizm grafu kandydującego z podgrafem każdego grafu wejściowego należącego do przecięcia zbiorów wspierających grafy, z których powstał graf kandydujący. Na przykład, wsparcie grafu ab liczymy w zbiorze $a.supportingSet \cap b.supportingSet$. Wyznaczanie wsparcia grafu



Rysunek 4.7. Graf G wspiera grafy (posiada podraf izomorficzny z grafami) a oraz b, ale nie wspiera grafu ab. Zanurzenia grafów a i b w grafie G nie są rozłączne.

kandydującego jest konieczne, gdyż w odróżnieniu od częstych zbiorów, nie jest prawdziwa zasada, że jeśli zbiór z bazy danych wspiera a i jednocześnie wspiera b, to wspiera także ab. W przypadku grafów istnieje możliwość, że graf posiada podgraf izomorficzny z a oraz podgraf izomorficzny z b, lecz nie posiada podgrafu izomorficznego z ab, co obrazuje rys. 4.7. Załóżmy, że po wyznaczeniu wsparcia, lista częstych grafów o dwóch składowych ma postać $F_2 = (aa, ab, ae, bb, bd, cd, ce, de)$. Lista kandydatów o trzech składowych powstaje przez łączenie ze sobą par grafów o dwóch składowych, które mają identyczne dwie pierwsze składowe. Zatem $K_3 = (aaa, aab, aae, abb, abe, bbb, bbd, bdd, cdd, cde, cee, dee)$.

Tworzenie kandydatów o k składowych

Ogólną zasadę tworzenia kandydatów o k składowych z grafów częstych o $k-1$ składowych można sformułować następująco:
Do listy K_k należą grafy powstałe z połączenia par takich grafów G_i, G_j z listy F_{k-1} (przy czym $i \leq j$, i oraz j oznaczają pozycję występowania grafów na liście F_{k-1}), które są identyczne bez uwzględniania ostatniej składowej. Połączeniem grafów $G_i = (C_1, C_2, \ldots C_{k-2}, C_{k-1})$ i $G_j = (C_1, C_2, \ldots C_{k-2}, C'_{k-1})$ jest graf $G = (C_1, C_2, \ldots C_{k-2}, C_{k-1}, C'_{k-1})$. Na przykład, połączeniem grafów $abbcd$ i $abbcf$ jest graf $abbcdf$.

Relacje w zbiorze częstych grafów spójnych

Częste grafy spójne tworzą kratę opartą na relacji izomorfizmu podgrafu. Relację tę można wykorzystać do filtrowania zbioru kandydatów opierając się na następującej własności:

Własność 4.4. Jeśli rozszerzenie grafu $G = (C_1, \ldots C_n)$ o składową C_{n+1} poprzez połączenie grafu $(C_1, \ldots C_n)$ z grafem $(C_1, \ldots C_{n-1}, C_{n+1})$ nie jest grafem częstym, wtedy każde rozszerzenie grafu G o składową C będącą nadgrafem grafu C_{n+1} poprzez połączenie grafu $(C_1, \ldots C_n)$ z grafem $(C_1, \ldots C_{n-1}, C)$ również nie jest grafem częstym.

Przykład 4.7. Załóżmy, że w podanej powyżej własności graf G ma postać abc. Dla takiego grafu własność ta przyjmuje wtedy następującą postać. Jeśli rozszerzenie grafu abc o składową d (poprzez połączenie abc z abd) nie jest grafem częstym, wtedy każde rozszerzenie grafu abc o składową x będącą nadgrafem d (poprzez połączenie abc z abx) również nie jest grafem częstym. Operację łączenie grafów abc z abx można zatem pominąć, gdyż zagwarantowane jest, że wynik złączenia nie będzie częsty.

Tę własność można wykorzytać do przerywania operacji łączenia grafów, dzięki czemu powstanie mniej grafów kandydujących, co prowadzi do przyspieszenia algorytmu. Załóżmy, że dany graf G_1 ma być łączony kolejno z grafami $G_1, \ldots, G_n$, czyli te wszystkie grafy różnią się co najwyżej na ostatniej składowej spójnej. Dla takiej serii operacji łączenia należy przygotować zbiór neg, w którym będą sukcesywnie umieszczane ostatnie składowe grafów $G_1, \ldots, G_n$, których dodanie do grafu G_1 skutkuje uzyskaniem nieczęstego grafu. Przed każdym połączeniem grafu G_1 z grafami $G_i, i = 2 \ldots n$, sprawdzane jest, czy zbiór neg zawiera graf izomorficzny z podgrafem ostatniej składowej grafu G_i. Jeśli zawiera, operację połączenia można przerwać, gdyż uzyskany graf będzie nieczęsty. Jeśli nie zawiera, wtedy tworzone jest połączenie grafów G_1 i G_i, wyznaczane jest jego wsparcie i jeśli połączenie nie będzie częste, wtedy ostatnia skłądowa grafu G_i jest dodawana do zbioru neg.

Można zauważyć, że skuteczność tej metody zależy od kolejności w jakiej łączone są grafy. Do danego grafu powinny być w pierszej kolejności dodawane składowe spójne, które są izomorficzne z podgrafami składowych spójnych dodawanych później, gdyż w przeciwnym razie zbiór neg nie będzie wykorzystywany. Z tego względu proponuję, aby listę składowych spójnych F_1 uporządkować nierosnąco według klucza $f(C)$, gdzie $f(C)$ jest liczbą grafów z listy F_1, które posiadają podgraf izomorficzny z grafem C. W posortowanej liście F_1 składowa a powinna być zatem izomorficzna z podgrafami największej liczby grafów w porównaniu z kolejnymi składowymi $b, c, d, \ldots$.

Algorytm UFC

Zaproponowane idee opisuje algorytm 4.2.2 (UFC - Unconnected From Connected). Wykonanie algorytmu UFC rozpoczyna się od odkrycia wszystkich częstych grafów spójnych za pomocą dowolnego algorytmu przeznaczonego do tego celu i posortowania ich w kolejności wynikającej z liczby nadgrafów w tym zbiorze (na początku te z największą liczbą nadgrafów). W ten sposób powstaje lista F_1 (grafów częstych o jednej składowej). Graf bez wierzchołków należy zaś do listy F_0, jeśli $|\mathbb{D}| \geq minSup$. Grafy z F_0 i F_1 wchodzą automatycznie do wynikowego zbioru R. Następnie, na podstawie listy F_1 tworzona jest lista F_2 za pomocą funkcji $GenerateKComponentsFrequentGraphs$ i kolejno - dopóki uzyskane listy kandydujących grafów nie są puste - F_3 na podstawie F_2, F_4 na podstawie F_3, F_{k+1} na podstawieF_k. Grafy z kolejno uzyskiwanych listy F_i wchodzą do zbioru wynikowego R.

Algorytm 4.20 UFC($\mathbb{D}, minSup$)

---

if $|\mathbb{D}| \geq minSup$ **then**
  $F_0 \leftarrow$ {graf bez wierzchołków};
end if
$F_1 \leftarrow ConnectedGraphMiner(\mathbb{D}, minSup) -$ {graf bez wierzchołków}; /*
ConnectedGraphMiner jest dowolnym algorytmem odkrywającym spójne grafy częste */
posortuj nierosnąco F_1 według liczby grafów ze zbioru z F_1, które posiadają podgraf izomorficzny z danym grafem;
$R \leftarrow F_0 \cup F_1$;
$i \leftarrow 1$;
while $F_i \neq \emptyset$ **do**
  $i \leftarrow i + 1$;
  $F_i \leftarrow GenerateKComponentsFrequentGraphs(F_{i-1}, i)$;
  $R \leftarrow R \cup F_i$;
end while
return R;

Funkcja $GenerateKComponentsFrequentGraphs(F_{k-1}, k)$ generuje częste grafy o k składowych na podstawie listy F_{k-1} (częstych grafów o $k-1$ składowych). Funkcja posiada dwie pętle - zewnętrzną, iterującą po wszystkich grafach G_1 ze zbioru F_{k-1} oraz wewnętrzną, iterującą po wszystkich grafach G_2 ze zbioru F_{k-1} rozpoczynając od grafu G_1, a kończąc na ostatnim grafie, który da się połączyć z G_1. Grafy, które można kolejno łączyć z G_1 w celu wytworzenia nowych kandydujących grafów, występują na liście F_{k-1} obok siebie bezpośrednio po G_1. Dla każdego grafu G_1, przed rozpoczęciem wewnętrznej pętli, inicjalizowany jest zbiór neg, który będzie przechowywał składowe spójne grafów G_2, których dodanie do grafu G_1 skutkuje uzyskaniem grafu nieczęstego o k składowych spójnych.

Algorytm 4.21 GenerateKComponentsFrequentGraphs(F, k)

for $i \leftarrow 1$ to $|F|$ **do**
  $neg \leftarrow \emptyset$;
  for $j \leftarrow i$ to |F| **do**
    $G_1 \leftarrow F[i]$; $G_2 \leftarrow F[j]$;
    {G_i jest grafem reprezentowanym jako ciąg składowych spójnych}
    $\{G_1 = (C_1^1, C_2^1, \ldots C_{k-1}^1), G_2 = (C_1^2, C_2^2, \ldots C_{k-1}^2)\}$
    if $(C_1^1, C_2^1, \ldots C_{k-2}^1) = (C_1^2, C_2^2, \ldots C_{k-2}^2)$ **then** $/*$ grafy G_1 i G_2 są identyczne bez uwzględniania ostatnio dodanej składowej spójnej $*/$
      $contineInnerLoop \leftarrow false$;
      for all $G_{neg} \in neg$ **do**
        if G_{neg} jest podgrafem C_{k-1}^2 **then** $/*$ połączenie G_1 z C_{k-1}^2 nie będzie grafem częstym, gdyż połączenie G_1 z podgrafem C_{k-1}^2 nie było grafem częstym $*/$
          $continueInnerLoop \leftarrow true$;
          $break$;
        end if
      end for
      if $continueInnerLoop$ **then**
        $continue$;
      end if
      $/*$ Opcjonalne faza czyszczenia $*/$
      if zbiór składowych spójnych grafu G posiada podzbiory o $k-1$ składowych, które nie występują w F **then**
        $continue$;
      end if$/*$ Koniec fazy czyszczenia $*/$
      $G \leftarrow (C_1^1, C_2^1, \ldots C_{k-1}^1, C_{k-1}^2)$;
      if $isFrequent(G, minSup, G_1.supportingSet \cap G_2.supportingSet)$ **then**
        $R \leftarrow R \cup \{G\}$;
      else
        $neg \leftarrow \{C_{k-1}^2\}$;
      end if
    else
      break;
    end if
  end for
end for
return R;

Załóżmy, że wynik połączenia grafu G_1 z grafem G_2, jest grafem o k składowych spójnych, czyli że G_1 i G_2 różnią się co najwyżej na ostatniej składowej spójnej. Zanim nastąpi połączenie G_1 z G_2 sprawdzany jest warunek podany we własności 4.4. Jeśli ostatnia składowa grafu G_2 jest nadgrafem któregoś z grafów ze zbioru neg, wtedy połączenie G_1 z G_2 na pewno nie będzie częste, a więc nie jest wykonywane. Po połączeniu G_1 z G_2 wynikowy graf G składa się ze wszystkich składowych grafu G_1 i ostatniej składowej grafu G_2. Zanim dojdzie do wyznaczania wsparcia tego grafu, możliwe jest wykonanie opcjonalnej fazy czyszczenia. To znaczy, można sprawdzić, czy każdy o jeden mniejszy podzbiór składowych spójnych grafu G jest grafem częstym, czyli występuje w zbiorze F_{k-1}. Na przykład, jeśli graf G ma siedem składowych spójnych $aaabccd$ to należy sprawdzić, czy wszystkie grafy o sześciu składowych, z których graf G nie został utworzony, czyli $aabccd$ oraz $aaaccd$, występują w F_6. Nie trzeba sprawdzać $aaabcc$ ani $aaabcd$, gdyż z tych grafów powstał G więc muszą one występować w F_6. Wreszcie, ostateczny test na to, czy graf G jest częsty, przeprowadza funkcja $isFrequent$. Jeśli G jest częsty, zostaje umieszczony w zbiorze F_k, w przeciwnym razie ostatnia składowa grafu G jest dodawana do zbioru neg.

Funkcja $isFrequent$ sprawdza, czy graf G jest wspierany nie mniej niż $minSup$ razy przez grafy ze zbioru $\mathbb{D}$ będącego przecięciem zbiorów wspierających grafy G_1 i G_2. Na początku sprawdzane jest, czy liczność zbioru $\mathbb{D}$ jest nie mniejsza niż $minSup$ i jeśli tak, wyznaczane jest dokładne wsparcie przez wykonywanie testów na izomorfizm podgrafu G z każdym grafem zbioru $\mathbb{D}$.

Algorytm 4.22 isFrequent($G, minSup, \mathbb{D}$)

```
if |𝔻| < minSup then
  return false;
end if
sup ← 0;
for all G_i ∈ 𝔻 do
  if graf G jest izomorficzny z podgrafem grafu G_i then
    sup ← sup + 1;
  end if
end for
if sup < minSup then
  return false;
else
  return true;
end if
```

Wykorzystanie optymalizacji z algorytmu UGM

Niektóre z metod przygotowanych dla algorytmu UGM można wykorzystać do optymalizacji algorytmu UFC. Warto wykorzystać:

— wielozbiory maksymalnych częstych deskryptorów krawędzi (patrz rozdział 4.1.1)
Wielobiór deskryptorów każdego grafu częstego musi być podzbiorem któregoś z maksymalnych częstych wielozbiorów deskryptorów. Przed wyznaczaniem wsparcia grafów kandydujących (z listy K_i) należy sprawdzić, czy ich wielozbiory deskryptorów spełniają podany wyżej warunek. Jeśli nie - można uznać, że graf nie jest częsty bez potrzeby wyznaczania dokładnego wsparcia.

— zbiór grafów nieczęstych (patrz rozdział 4.1.2) Nadgraf grafu nieczęstego też nie jest częsty. Grafy, które okazują się nieczęste w czasie wyznaczania wsparcia powinny zostać umieszczone we wspomnianym zbiorze grafów nieczęstych $\mathbb{GN}$. Podczas rozpatrywania kandydatów na grafy częste można sprawdzić, czy nie zawierają one podgrafów izomorficznych z dowolnym z grafów z tego zbioru. Jeśli zawierają - można uznać, że nie są one częste bez potrzeby wyznaczania dokładnego wsparcia. Metoda ta jest uogólnieniem optymalizacji na podstawie własności 4.4, która jest stosowana w UFC, ale jest od niej wolniejsza, gdyż wymaga wykonywania testów na izomorfizm z podgrafem, zatem powinna być stosowana raczej jako uzupełnienie tej metody niż jej zastąpienie. Korzystanie ze zbioru grafów nieczęstych jest tym skuteczniejsze, im szybciej zbiór będzie wypełniany małymi grafami nieczęstymi. Stosowane w UFC sortowanie częstych grafów spójnych sprzyja tej własności.

— przerywanie wyznaczania wsparcia (patrz rozdział 4.1.4) Do poprawnego działania algorytmu nie jest potrzebne dokładne wyznaczenie wsparć grafów, które nie są częste. Jeśli zatem na etapie wyznaczania wsparcia dla danego grafu można stwierdzić, że graf nie jest częsty, wyznaczanie wsparcia można przerwać. Wsparcie danego grafu jest zawsze wyznaczane w zbiorze $\mathbb{D}$ stanowiącym przecięcie zbiorów wspierających grafów z których powstał. Jeśli liczba nieudanych testów na izomorfizm podgrafu przekroczy $|\mathbb{D}| - minSup$, wtedy graf ma na pewno wsparcie mniejsze niż $minSup$, czyli nie jest częsty.

Wszystkie te metody wystarczy włączyć do funkcji $isFrequent$; pozostałe funkcje nie wymagają wprowadzania zmian. Zmodyfikowana funkcja nosi nazwę $isFrequentEx$ (algorytm 4.23) Sprawdzanie, czy wielozbiór deskryptorów grafu jest zawarty w zbiorze

maksymalnych wielozbiorów deskryptorów, znajduje się na początku funkcji $isFrequentEx$, zaraz po sprawdzeniu, czy zbiór wspierający jest wystarczająco liczny. Następnie sprawdzany jest zbiór grafów nieczęstych. Wreszcie, na etapie wyznaczania wsparcia śledzona jest liczba nieudanych testów na izomorfizm podgrafu - $falseResults$. Jeśli $falseResult$ przekroczy $|\mathbb{D}| - minSup$, pętla wyznaczająca wsparcie jest przerywana.

Algorytm 4.23 $isFrequentEx(G, minSup, \mathbb{D})$

if $|\mathbb{D}| < minSup$ **then**
  return $false$;
end if
if $ES(G)$ nie jest podzbiorem żadnego z maksymalnych częstych wielozbiorów deskryptorów **then**
  return $false$;
end if
if $\mathbb{GN}$ zawiera jakikolwiek podgraf izomorficzny z G **then**
  return $false$;
end if
$sup \leftarrow 0$; $falseResults = 0$;
for all $G_i \in \mathbb{D}$ **do**
  if graf G jest izomorficzny z podgrafem grafu G_i **then**
    $sup \leftarrow sup + 1$;
  else
    $falseResults \leftarrow falseResults + 1$;
    if $falseResults > |\mathbb{D}| - minSup$ **then**
      return $false$;
    end if
  end if
end for
if $sup < minSup$ **then**
  $\mathbb{GN} \leftarrow \mathbb{GN} \cup \{G\}$;
  return $false$;
else
  return $true$;
end if

5. Problemy izomorfizmu grafów i izomorfizmu z podgrafem

5.1. Znane algorytmy

Problem izomorfizmu grafów należy klasy NP, ale nie jest wiadome, ani czy należy do klasy problemów NP-zupełnych, ani czy jest rozwiązywalny w czasie wielomianowym[1] [27, 42]. Naiwne rozwiązanie polega na generowaniu permutacji wierzchołków jednego grafu z pary porównywanych grafów i sprawdzaniu, czy po przenumerowaniu wierzchołków na podstawie permutacji oba grafy są identyczne. Takie rozwiązanie ma złożoność $O(n!)$, gdzie n jest liczbą wierzchołków grafu. W ulepszonej wersji wierzchołki grafu są najpierw partycjonowane na klasy równoważności, np. na podstawie etykiety i/lub stopnia. Permutowanie można wtedy przeprowadzić niezależnie w każdej klasie [21]. Obecnie teoretycznie najniższą złożoność równą $2^{O(\sqrt{nlogn})}$ posiada algorytm zaproponowany w [4]. W zastosowaniach praktycznych za wiodący algorytm rozwiązywania problemu izomorfizmu grafów uważa się program NAUTY[2] [26], bazujący na pracy [54]. W przypadku ograniczonej klasy grafów istnieją skuteczniejsze algorytmy: algorytmy o złożoności wielomianowej zaproponowano m. in. dla grafów planarnych [36], przedziałowych [50], grafów permutacji [15], grafów o ograniczonym stopniu wierzchołków [51].

Problem izomorfizmu z podgrafem należy natomiast do klasy NP-zupełnych [30], zatem na chwilę obecną nie istnieje algorytm rozwiązujący ten problem w czasie wielomianowym. Pierwsze rozwiązanie problemu izomorfizmu z podgrafem podał Ullmann [69]. Rozwiązanie wielomianowe zostało zaprezentowane w [56], przy założeniu, że zbiór potencjalnych nadgrafów dany jest z góry. W tej metodzie dla danego zbioru grafów $\mathbb{D}$ wykonuje

[1] Wyniki ostatnich badań sugerują, że problem należy jednak do klasy P (jest rozwiązywalny w czasie wielomianowym). W [22] pojawiła się propozycja algorytmu należącego do klasy P, która wymaga szerszej weryfikacji.

[2] NAUTY, http://www.cs.sunysb.edu/~algorith/implement/nauty/implement.shtml

się przetwarzanie wstępne, które tworzy drzewiastą strukturę wspomagającą późniejsze wykonywanie testów na izomorfizm podgrafu. Algorytm korzystający z tej struktury potrafi odpowiedzieć na pytanie czy dany graf G jest izomorficzny z podgrafem któregoś z grafów z zbioru $\mathbb{D}$ w czasie proporcjonalnym do kwadratu liczby wierzchołków grafu G. Z tego względu rozwiązanie to jest interesujące, gdyż w zagadnieniu odkrywania grafów częstych baza grafów jest znana. Niestety wspomagająca struktura rośnie wykładniczo ze względu na liczbę i rozmiar grafów w $\mathbb{D}$ i jak podają autorzy, realne zastosowanie kończy się na liczbie kilkunastu grafów o wielkości kilkunastu wierzchołków, co wyklucza jej wykorzystanie w odkrywaniu grafów częstych. Najbardziej popularnym narzędziem do badania problemu izomorfizmu z podgrafem jest biblioteka *vflib*[3] wykorzystująca algorytmy VF [19] i VF2 [18]. W niniejszej rozprawie proponuję nowy algorytm $SubgraphIsomorphism$ rozwiązujący problem izomorfizmu z podgrafem, zoptymalizowany pod kątem wykorzystania w algorytmach UGM i UFC. Algorytm ten opiera się na zdefiniowaniu izomorfizmu z podgrafem w postaci problemu spełniania ograniczeń (CSP, ang. Constraints Satisfaction Problem) w sposób podobny do [49, 63, 81, 79], użyciu optymalizacji opartej na symetrii grafów będącej modyfikacją pomysłu z [80], oraz wykorzystaniu faktu, że większość operacji izomorfizmu z podgrafem wykonuje się na grafach z tego samego zbioru, czyli grafach z wejściowej bazy grafów oraz kolejnych rozszerzeń grafów częstych. W przypadku zastosowań wewnątrz UGM i UFC proponowany algorytm okazuje się o kilka rzędów wielkości efektywniejszy od implementacji umieszczonej w platformie $ParMol$ oraz o rząd wielkości efektywniejszy od algorytmów biblioteki *vflib*.

5.2. Problem izomorfizmu grafów jako CSP

5.2.1. CSP - Problem Spełniania Ograniczeń

Definicja 5.1. *(Problem CSP)*

Problem CSP jest trójką (X, D, C), gdzie

$X = \{x_1, x_2, \ldots, x_n\}$ jest zbiorem n zmiennych.

$D = \{D_1, D_2, \ldots, D_n\}$ jest zbiorem dziedzin. Zmienne x_i przyjmują wartości z D_i.

C jest zbiorem zdań logicznych dotyczących zmiennych ze zbioru X.

[3] http://amalfi.dis.unina.it/graph/db/vflib-2.0/doc/vflib.html

Podczas definiowania problemu zmienne $x_1, \ldots, x_n$ mają nieokreślone wartości. Rozwiązanie problemu polega na znalezieniu takiego przyporządkowania wartości do zmiennych ($x_1 \leftarrow d_1 \in D_1, \ldots, x_n \leftarrow d_n \in D_n$), aby wszystkie zdania logiczne ze zbioru C były prawdziwe, lub stwierdzeniu, że takie przyporządkowanie nie istnieje.

Wiele problemów można zdefiniować w tej postaci, m.in. problem n hetmanów, kolorowanie map, układanie rozkładu zajęć, układanie harmonogramów.

Ogólne rozwiązanie problemu CSP polega na zastosowaniu algorytmu przeszukiwania z powrotami (ang. backtracking [20]). Ogólny schemat tej metody przedstawia algorytm 5.24. Algorytm z powrotami polega na przyporządkowywaniu dozwolonych wartości do kolejnych zmiennych, a w momencie, gdy do rozpatrywanej zmiennej nie da się przyporządkować wartości, gdyż ograniczenia zostałyby naruszone, wraca się do poprzedniej zmiennej. Do tego algorytmu można dodać wiele usprawnień poprawiających efektywność przeszukiwania [29], m.in. heurystyki wyboru zmiennej i wartości, sprawdzanie w przód, propagację ograniczeń, przeszukiwanie lokalne, powroty ze znakowaniem (ang. backmarking), powroty z przeskokami (ang. backjumping). Jak pokazano w [66] wykorzystanie tych technik pozwala na rozwiązywanie większości przypadków problemów CSP w prawie stałym czasie. Istnieje tylko wąski przedział kombinacji parametrów problemu, w którym problem ujawnia wykładniczy charakter.

Algorytm 5.24 CSPbactracking(X, D, C)

```
1:  if wszystkie zmienne z X mają przyporządkowane wartości then
2:     return true;
3:  end if
4:  cur ← Wybierz zmienną ze zbioru X, która nie ma przyporządkowanej wartości;
5:  for all d ∈ D_cur do
6:     if przyporządkowanie x_cur ← d nie narusza ograniczenia C then
7:        przyporządkuj x_cur ← d;
8:        usuń d z D_cur;
9:        if CSPbactracking(X, D, C) then
10:          return true;
11:       else
12:          Anuluj przyporządkowanie x_cur ← d;
13:          Dodaj d do D_cur;
14:       end if
15:    end if
16: end for
17: return false;
```

Przykład 5.1. *(przykładowy problem spełnialności)*

Należy sprawdzić, czy dla formuły logicznej

$$f = (x_2 \vee x_3 \vee x_4) \wedge (x_1 \vee x_2 \vee \overline{x_3}) \wedge (x_2 \vee x_3 \vee \overline{x_4}) \wedge (x_1 \vee \overline{x_2})$$

istnieje takie podstawienie zmiennych x_1, x_2, x_3, x_4, że formuła jest prawdziwa.

Zadanie jest przykładem problemu 3-SAT, gdyż formuła logiczna jest w koniunkcyjnej postaci normalnej, a wszystkie klauzule mają nie więcej niż 3 literały. Problem 3-SAT należy do klasy NP-zupełnych. Sprowadzamy podany problem do postaci $CSP = (X, D, C)$, gdzie:

$X = \{x_1, x_2, x_3, x_4\}$

$D_1 = D_2 = D_3 = D_4 = \{0, 1\}$

$C = \{$

$C_1 : (x_2 \vee x_3 \vee x_4) = 1$

$C_2 : (x_1 \vee x_2 \vee \overline{x_3}) = 1$

$C_3 : (x_2 \vee x_3 \vee \overline{x_4}) = 1$

$C_4 : (x_1 \vee \overline{x_2}) = 1$

$\}$

W określaniu zbioru ograniczeń wykorzystujemy wiedzę z problemu 3-SAT. Wprowadzamy cztery mniejsze ograniczenia, które zastępuje jedno duże $f = 1$. Jeśli przyjmiemy, że algorytm z powrotami przyporządkowuje zmienne w kolejności x_1, x_2, x_3, x_4, a wartości w kolejności $0, 1$, wtedy ślad wykonania tego algorytmu przedstawia tabela 5.1. Wiersz C tabeli 5.1 pokazuje ograniczenie, które nie jest spełnione. Po czternastu iteracjach algorytm znajduje rozwiązanie $x_1 = 1, x_2 = 0, x_3 = 1, x_4 = 0$ spełniające warunki zadania. Algorytm z powrotami można zmodyfikować tak, aby znajdował wszystkie poprawne rozwiązania, zamiast pierwszego napotkanego. W tym celu linie 1-3 algorytmu 5.24 należałoby zastąpić poniższym kodem.

---

if wszystkie zmienne z X mają przyporządkowanie **then**
  Wypisz lub zapamiętaj rozwiązanie;
  return false;
end if

5.2.2. Problem izomofizmu z podgrafem jako CSP

Problemu izomorfizmu grafu G z podgrafem grafu G' można zapisać w postaci CSP na dwa ogólne sposoby:

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14
x_1	0	0	0	0	0	0	0	1	1	1	1	1	1	1
x_2		0	0	0	0	0	1		0	0	0	0	0	0
x_3			0	0	0	1				0	0	0	1	1
x_4				0	1						0	1		0
C				C_1	C_3	C_2	C_4				C_1	C_3		

Tabela 5.1. Kolejne przyporządkowania w algorytmie z powrotami podczas rozwiązywania problemu z przykładu 5.1.

— bazując na wierzchołkach [53, 49] - w tym modelu zbiór X reprezentuje wierzchołki grafu G, a zbiór D reprezentuje wierzchołki grafu G';

— bazując na wierzchołkach i krawędziach [63] - w tym modelu zbiór X reprezentuje wierzchołki oraz krawędzie grafu G, a zbiór D reprezentuje wierzchołki i krawędzie grafu G'.

W tej pracy używany jest pierwszy model w postaci podanej w definicji 5.2. W przeważającej części pracy, pod pojęciem grafu rozumiany jest nieskierowany graf etykietowany. Z tego względu problem izomorfizmu rozpatrywany jest obszernie tylko w przypadku nieskierowanych grafów etykietowanych.

Definicja 5.2. *(izomorfizm z podgrafem w postaci CSP)*
Dane są dwa nieskierowane grafy etykietowane $G = (V = \{v_1, \ldots, v_n\}, E, lbl, L)$ oraz $G' = (V' = \{v'_1, \ldots, v'_N\}, E', lbl', L')$. Problem izomorfizmu grafu G z podgrafem grafu G' można zapisać w postaci $CSP(X, D, C)$, gdzie:

$X = \{x_{v_1}, x_{v_2}, \ldots, x_{v_n}\}$

$D = \{D_{v1} = V', D_{v2} = V', \ldots, D_{vn} = V'\}$

$C = \{$

$\forall_{e=\{v_1,v_2\}\in E} \quad \{x_{v_1}, x_{v_2}\} \in E',$

$\forall_{v\in V} \quad lbl(v) = lbl'(x_v),$

$\forall_{e=\{v_1,v_2\}\in E} \quad lbl(e) = lbl'(\{x_{v_1}, x_{v_2}\})$

$\forall_{i\neq j} \quad x_i \neq x_j$

$\}$

Można łatwo zauważyć, że definicja zbioru C jest tożsama z definicją izomorfizmu (patrz definicja 2.8) - przyporządkowania x_{v_i} odpowiadają funkcji $\phi(v_i)$.

5.3. Metody optymalizacji algorytmu CSP dla problemu izomorfizmu z podgrafem

Istnieje wiele metod zwiększających efektywność podanego wcześniej algorytmu z powrotami. Do znanych metod należą przede wszystkim metody ograniczania dziedziny, heurystyki kolejności wyboru wartości i zmiennych oraz wykorzystanie symetrii. Kolejne sekcje przedstawiają opis tych metod wraz ze sposobem ich implementacji na potrzeby algorytmów UGM i UFC. Samodzielnym i oryginalnym dorobkiem autora niniejszej rozprawy jest zaproponowanie metody sprawdzania spójności dziedziny (rozdział 5.3.4), użycie symetrii jako mechanizmu ograniczania dziedziny (rozdział 5.3.8) oraz wykorzystanie zebranych informacji z kolejnych wykonań testów na izomorfizm w algorytmach UGM i UFC (rozdział 5.3.10).

5.3.1. Ograniczanie dziedziny

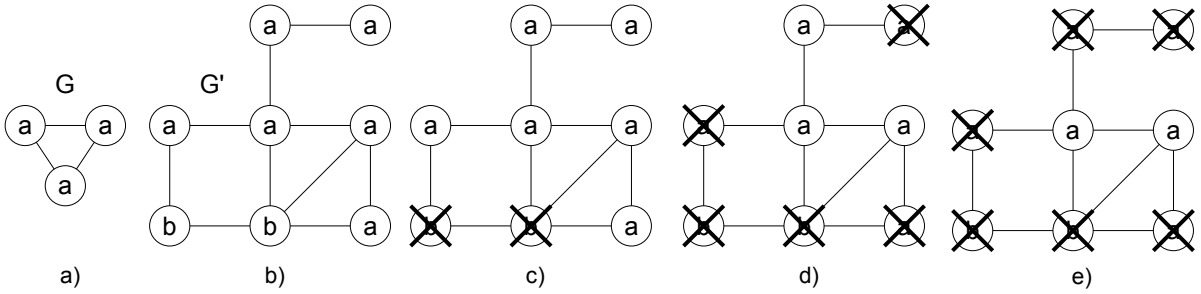
Zbiór D, czyli zbiór dozwolonych wartości określa przestrzeń przeszukiwań. Liczność przestrzeni przeszukiwań jest równa liczbie wszystkich możliwych przyporządkowań zmiennych do wartości, czyli $|X|^{|D|}$. W związku z tym nawet niewielkie zmniejszenie dziedziny powoduje znaczące zmniejszenie przestrzeni przeszukiwań. Metody ograniczania dziedziny dzielą się na te, które wykonuje się przed uruchomieniem algorytmu przeszukiwania oraz takie, które wykonuje się w trakcie przeszukiwania - najczęściej po każdym przyporządku wartości do zmiennej (tzw. sprawdzanie w przód). Ograniczanie dziedziny odbywa się przez interpretację ograniczeń ze zbioru C. Najprostszy przypadek stanowią ograniczenia unarne, czyli dotyczące jednej zmiennej. W przypadku izomorfizmu z podgrafem ograniczeniem unarnym jest $lbl'(x_v) = lbl(v)$, czyli warunek zgodności etykiet wierzchołków. Wierzchołek o danej etykiecie można przyporządkować tylko do wierzchołków o tej samej etykicie. Zatem dziedzinę zmiennej x_v można ograniczyć do wierzchołków o etykiecie $lbl(v)$. Do ograniczania dziedziny na podstawie ograniczeń binarnych służą algorytmy sprawdzania spójności łukowej (np. $AC3$, rozdział 5.3.2). Przykładem takiego ograniczenia w problemie izomorfizmu z podgrafej jest $lbl'(\{x_{v_1}, x_{v_2}\}) = lbl(\{v_1, v_2\})$, czyli warunek zgodności etykiet krawędzi.

Dla problemu izomorfizmu z podgrafem nie istnieją ograniczenia ternarne i wyższego stopnia. Do ograniczania dziedziny mogą też posłużyć ograniczenia nie występujące wprost w zbiorze C, ale wynikające z wiedzy o problemie. Dla przykładu rozpatrzmy problem izomorfizmu grafu G z podgrafem grafu G' znajdujący się na rysunku 5.1. Warunkiem koniecznym na to, aby przyporządkowanie wierzchołka grafu G do wierzchołka grafu G' było izomorfizmem, jest zgodność etykiet wierzchołków. Wierzchołki o etykiecie *a* można przyporządkować tylko do wierzchołków o etykiecie *a*. Zastosowanie tego warunku zmiejsza liczność dziedziny z ośmiu do sześciu (rys. 5.1c). Warunek zgodności etykiet wierzchołków występuje w zbiorze C, ale kolejny już nie: wielozbiór etykiet wierzchołków sąsiadujących z wierzchołkiem v musi być podzbiorem wielozbioru etykiet wierzchołków sąsiadujących z wierzchołkiem v'. Ten warunek ogranicza dziedzinę do liczności 3 (rys. 5.1d) - wierzchołki sąsiadujące z dwoma wierzchołkami o etykietach a można przyporządkować tylko do wierzchołków sąsiadujących z co najmniej dwoma wierzchołkami o etykiecie a. Podobny warunek można podać dla etykiet krawędzi sąsiadujących, lecz w rozważanym przykładzie nie przyniesie to żadnego efektu, gdyż krawędzie nie są etykietowane, czyli można uznać, że wszystkie krawędzie mają tę samą etykietę. Kolejne ograniczenie, które nie występuje bezpośrednio w zbiorze C, dotyczy stopnia wierzchołków sąsiadujących: stopnie wierzchołków sąsiadujących z wierzchołkiem v muszą być nie większe niż stopienie odpowiadających im wierzchołków sąsiadujących z wierzchołkiem v'. Po uwzględnieniu tego warunku liczność dziedziny spada do dwóch (rys. 5.1e) - wierzchołki sąsiadujące z dwoma wierzchołkami o stopniach równych 2 można przyporządkować tylko do wierzchołków sąsiadujących z co najmniej dwoma wierzchołkami o stopniach równych co najmniej 2. Możliwe jest tworzenie bardziej skomplikowanych warunków, np. uwzględniających większe otoczenie wierzchołka niż bezpośredni sąsiedzi, pod warunkiem, że możliwe jest efektywne ograniczanie dziedziny na ich podstawie. W pracy zostały przetestowane podane niżej warunki wstępnie ograniczające dziedzinę.

Do dziedziny zmiennej x_v należą tylko te wartości v', które spełniają następujące warunki:

— zgodność etykiet: $lbl(v) = lbl'(v')$; etykieta wierzchołka v jest taka sama jak etykieta wierzchołka v'.

— zgodność sąsiadujących krawędzi: $\exists$ funkcja różnowartościowa $f : N_v(v) \rightarrow N_v(v')$ taka, że $\forall_{v_i \in N_v(v)}\ lbl(v_i) = lbl'(f(v_i)) \wedge lbl(e = \{v, v_i\}) = lbl'(e' = \{v', f(v_i)\})$, gdzie $N_v(i)$ jest zbiorem wierzchołków sąsiadujących z wierzchołkiem i. Warunek ten jest równoważny



Rysunek 5.1. Ograniczenia dziedziny wierzchołków grafu G po uwzględnianiu warunków: c) zgodności etykiet, d) zgodności sąsiedztwa, e) zgodności stopnia wierzchołków sąsiadujących

następującemu: wielozbiór deskryptorów krawędzi zawierających wierzchołek v musi być podzbiorem wielozbioru deskryptorów krawędzi zawierających v'.

— zgodność stopnia wierzchołków sąsiadujących: $\exists$ funkcja różnowartościowa $f: \ N_v(v) \rightarrow N_v(v')$ taka, że $\forall_{v_i \in N_v(v)} \ lbl(v_i) = lbl'(f(v_i)) \wedge lbl(e = \{v, v_i\}) = lbl'(e' = \{v', f(v_i)\}) \wedge d(v) \leq d(v')$, gdzie $N_v(i)$ jest zbiorem wierzchołków sąsiadujących z wierzchołkiem i. Jest to zaostrzenie warunku poprzedniego o warunek stopnia wierzchołków. Stopnie wierzchołków z otoczenia wierzchołka v muszą być mniejsze niż lub równe stopniom odpowiadających im wierzchołków z otoczenia wierzchołka v'.

Algorytm ograniczający dziedzinę na podstawie wymienionych warunków będzie omówiony w rozdziale 5.3.4 przy okazji omawiania sprawdzania spójności dziedziny.

Algorytm 5.25 AC-3

if ac3 jest wykonywany przed uruchomieniem algorytmu z powrotami **then**
  $arcs \leftarrow (v_i, v_j)$ dla każdego $v_i, v_j \in V \,|\{v_i, v_j\} \in E$
else $/*$ ac3 jest wykonywany po każdym przyporządkowaniu w algorytmie z powrotami $*/$
  $arcs \leftarrow (v_i, v_j)$ dla każdego v_j, które zmieniło dziedzinę przy ostatnim przyporządkowaniu i $\{v_i, v_j\} \in E$
end if
while arcs jest niepusty **do**
  $a \leftarrow arcs.first$;
  usuń a z $arcs$;
  if removeInconsistentValues($a.v_1, a.v_2$) **then**
    $arcs \leftarrow arcs \cup (v_i, a.v_1)$ dla każdego v_i takiego, że $\{v_i, v_j\} \in E$
  end if
end while

Algorytm 5.26 removeInconsistentValues(v_i, v_j)

$removed \leftarrow false$;
for all $v'_i \in D_{vi}$ **do**
  if nie istnieje $v'_j \in D_{vj}$ spełniające ograniczenie (v_i, v_j) (tzn. $\{v'_i, v'_j\} \in E' \wedge lbl'(\{v'_i, v'_j\}) = lbl(\{v_i, v_j\})$) **then**
    usuń v'_i z D_{vi};
    $removed \leftarrow true$;
  end if
end for
return $removed$;

5.3.2. Ograniczanie dziedziny za pomocą badania spójności łuków

Sprawdzanie spójności łuków (ang. arc consistency) ma zastosowanie w problemach CSP z ograniczeniami binarnymi. Problem taki można wtedy reprezentować w postaci grafu, w którym wierzchołki reprezentują zmienne, a krawędzie (tu nazywane łukami) - ograniczenia między zmiennymi. W celu ograniczania dziedziny za pomocą ograniczeń binarnych wprowadza się pojęcie *spójności łukowej*. Każdemu ograniczeniu binarnemu $c \in C$, w którym biorą udział dwie zmienne x_1 i x_2, odpowiadają dwa łuki: $(x_1 \rightarrow x_2)$ oraz $(x_2 \rightarrow x_1)$. Łuk $(x_1 \rightarrow x_2)$ uznajemy za spójny, jeśli dla każdej wartości $v_1 \in D_{x_1}$ istnieje wartość $v'_2 \in D_{x_2}$, spełniająca ograniczenie c. Jeśli taka wartość nie istnieje, wtedy v_1 należy usunąć z dziedziny x_1. Spójność łukową należy sprawdzić dla każdego łuku, a ponieważ sprawdzenie jednego łuku może zmienić dziedzinę zmiennych, łuk wcześniej sprawdzony i uznany za spójny może się okazać niespójny. Z tego względu może istnieć konieczność wielokrotnego sprawdzania spójności danego łuku, zatem złożoność algorytmu może być większa niż liniowa ze względu na liczbę łuków. Zaproponowano wiele algorytmów sprawdzania spójności łuków (ac1 [52], ac2[52], ac3 [52], ac4[57], ac5[61, 35], ac6[7], ac7[8]), z których najbardziej znanym jest ac-3 (patrz algorytm 5.25).

Algorytm AC-3 wprowadza dość duży narzut czasowy i eksperymenty (patrz rozdział 6.6) pokazują, że raczej nie warto go stosować w przypadku problemu izomorfizmu grafów i podgrafów.

5.3.3. Sprawdzanie w przód

Mechanizm sprawdzania w przód polega na ograniczaniu dziedziny po każdym przyporządkowaniu wartości do zmiennej. Jeśli któraś z dziedzin nieprzyporządkowanych zmiennych okaże się pusta, algorytm z powrotami może natychmiast anulować to

przyporządkowanie i powrócić do kolejnych przyporządkowań. Sposób w jaki przyporządkowanie wpływa na zmianę dziedziny innych zmiennych zależy od problemu. W przypadku izomorfizmu podgrafu, przyporządkowanie $x_v \leftarrow v'$ można wykorzystać do ograniczenia dziedziny na następujące sposoby:

— usuń v' z wszystkich dziedzin różnych od D_v. Przyporządkowanie w izomorfizmie z podgrafem jest różnowartościowe, więc raz wykorzystana wartość nie może być przyporządkowana innej zmiennej.

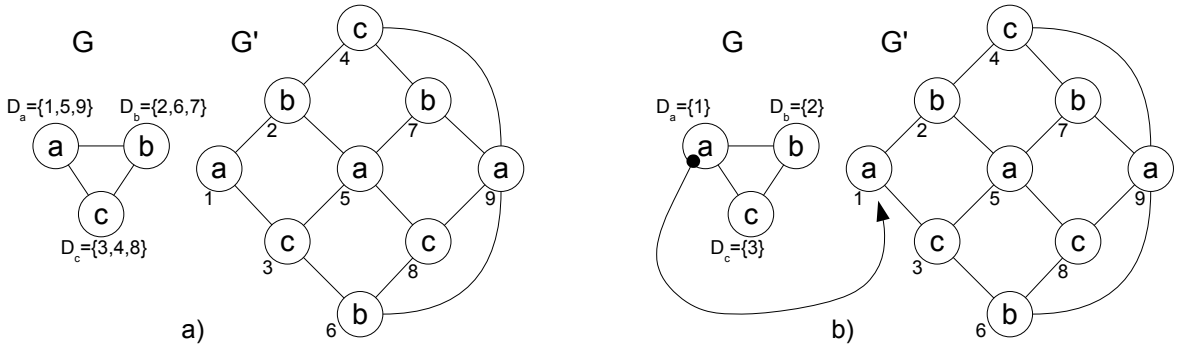
— w dziedzinach wierzchołków sąsiadujących z wierzchołkiem v, pozostaw tylko te wartości, które są wierzchołkami sąsiadującymi z x_v. Na rysunku 5.2 dziedzina każdego wierzchołka grafu G ma liczność 3. Po przyporządkowaniu wierzchołka a z grafu G do wierzchołka 1 z grafu G', w dziedzinach wierzchołków b i c pozostaną tylko wierzchołki sąsiadujące z wierzchołkiem 1, czyli wierzchołki 2 i 3. W dziedzinie wierzchołka b pozostanie zatem tylko wierzchołek 2, a w dziedzinie wierzchołka c - tylko wierzchołek 3.

Sprawdzanie w przód wymaga zapamiętywania usuniętych wartości z dziedzin w każdym kroku algorytmu z powrotami, aby w przypadku niepowodzenia przyporządkowania, można było przywrócić stan dziedziny sprzed nieudanego przyporządkowania. Wprowadza to dodatkowy koszt pamięciowy i czasowy, ale eksperymenty pokazują, że całkowity czas wykonania algorytmu jest zdecydowanie krótszy. Wynika to z faktu, że sprawdzanie w przód potrafi wykryć nieudane przyporządkowanie zdecydowanie wcześniej i co za tym idzie, liczba przyporządkowań i powrotów jest mniejsza.

Sprawdzanie w przód, wraz z przedstawionym wcześniej ograniczaniem dziedziny pozwala na całkowite zignorowanie fazy sprawdzania ograniczeń w algorytmie z powrotami, gdyż zapewnione jest, że dziedziny nie posiadają wartości, których przyporządkowanie naruszałoby ograniczenia.

5.3.4. Sprawdzanie spójności dziedziny

Po ograniczeniu dziedziny, ale jeszcze przed próbą znalezienia przyporządkowania, tzn. przed uruchomieniem algorytmu przeszukiwania (np. algorytmu z powrotami) istnieje możliwość sprawdzenia, czy uzyskana dziedzina daje w ogóle możliwość pozytywnego przyporządkowania zmiennych. Może się okazać, że dziedzina jednej zmiennej jest pusta, a więc z trywialnych powodów algorytm z powrotami nie zwróci pozytywnego rezultatu i nie



Rysunek 5.2. a) dziedziny wierzchołków grafu G b)dziedziny wierzchołków grafu G po przyporządkowaniu $a \leftarrow 1$ z wykorzystaniem sprawdzania w przód

ma sensu szukać przyporządkowań dla innych zmiennych. Autor tej rozprawy proponuje sprawdzanie spójności dziedziny na podstawie klas równoważności wierzchołków.

Podzielmy wierzchołki grafu G na klasy równoważności. Klasy równoważności są zdefiniowane i wyznaczane w sposób podobny do algorytmów badających izomorfizm z grafem, które wykorzystują partycjonowanie wierzchołków w klasy równoważności [21]. Do jednej klasy należą wierzchołki, które:

— mają tę samą etykietę,

— mają ten sam wielozbiór deskryptorów krawędzi wychodzących.

Jeśli algorytm ograniczania dziedziny korzystał z warunków *zgodności etykiet* oraz *zgodności sąsiadujących krawędzi* (i ewentualnie dodatkowo innych), to wierzchołki z tej samej klasy równoważności będą miały tę samą dziedzinę. Jeśli okaże się, że liczność tej dziedziny jest mniejsza niż liczność klasy równoważności, to dziedzina nie jest spójna i graf G na pewno nie jest podgrafem grafu G'. Na rysunku 5.1 wszystkie wierzchołki grafu G należą do tej samej klasy równoważności. Po zastosowaniu warunków zgodności etykiet i sąsiadujących krawędzi dziedzina tej klasy została zredukowana do liczności dwóch, co zostało omówione wcześniej. Istnieją trzy wierzchołki, które należy przyporządkować różnowartościowo do zbioru dwóch wierzchołków, co naturalnie jest niewykonalne. Zbadanie spójności dziedziny od razu odrzuci możliwość pozytywnego przyporządkowania, bez konieczności uruchamiania algorytmu z powrotami. Mówiąc ogólniej, dziedzina jest niespójna, gdy istnieje klasa równoważności wierzchołków, której liczność jest mniejsza niż liczność dziedziny wierzchołków należącej do tej klasy.

Badanie spójności dziedziny przedstawia algorytm 5.27, którego zadaniem jest także ograniczanie dziedziny według warunków podanych w rozdziale 5.3.1. Pierwszą fazą algorytmu jest wyznaczenie dziedziny. Dziedzina nie jest wyznaczana dla każdego wierzchołka oddzielnie, ale dla każdej klasy równoważności. W tej fazie algorytm opiera się na działaniu dwóch pętli - zewnętrzej, która iteruje po klasach równoważności wierzchołków grafu G oraz wewnętrznej, która iteruje po klasach równoważności wierzchołków grafu G'. Jeśli rozpatrywane w danym momencie klasy równoważności K i K' spełniają warunki zgodności etykiet, sąsiadujących krawędzi i stopnia sąsiadujących wierzchołków, wtedy do dziedziny każdego wierzchołka klasy K jest dodawany każdy wierzchołek klasy K'. Po wyznaczeniu całej dziedziny dla wierzchołków klasy K sprawdzany jest warunek spójności: jeśli liczność dziedziny wierzchołków z klasy K jest mniejsza niż liczność klasy K, wtedy algorytm kończy się podając wynik fałszu, co oznacza, że dziedzina jest niespójna i graf G nie jest izomorficzny z podgrafem grafu G'. Jeżeli zostaną wyznaczone dziedziny wierzchołków z wszystkich klas i wszystkie one będą spójne, wtedy algorytm kończy się wynikiem prawdy, co oznacza, że dziedzina jest spójna i aby stwierdzić, czy graf G jest izomorficzny z podgrafem grafu G' należy uruchomić algorytm z powrotami.

Istnieje możliwość wyznaczania klas równoważności nie na podstawie podanych na początku tej sekcji warunków, lecz na podstawie dziedzin uzyskanych po wykonaniu algorytmu ograniczania dziedziny. To znaczy, że wierzchołki trafią do tej samej klasy, jeśli mają tę samą dziedzinę. W ten sposób można potencjalnie uzyskać więcej klas równoważności, a więc więcej możliwości na wykrycie niespójności, ale dodatkowo traci się dwie możliwości optymalizacji.

— algorytmy UGM i UFC wykonują serie testów na izomorfizm grafów kandydujących z podgrafami grafów z bazy danych. Zaproponowane klasy równoważności wystarczy wyznaczyć raz i używać w każdym teście na izomorfizm. Natomiast klasy równoważności bazujące na dziedzinie trzeba liczyć oddzielnie dla każdego testu na izomorfizm.

— włączenie badania spójności do algorytmu ograniczania dziedziny pozwala na szybsze wykrycie niespójności i przerwanie wykonywania ograniczania dziedziny oszczędzając czas wykonania.

Algorytm 5.27 InitiateLegalValues(G, G')

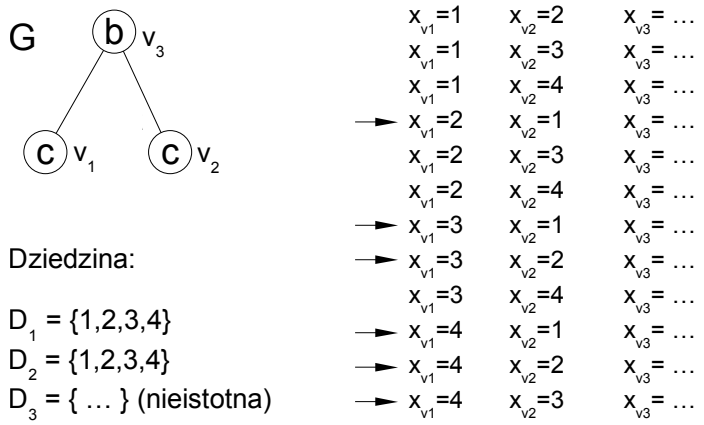
```
1: for all K ∈ K(G) do /* dla każdej wierzchołkowej klasy równoważności grafu G */
2:    v ← K.first;/* dowolny (np pierwszy) wierzchołek z klasy K */
3:    ne[] ← G.edges(v); /* tablica krawędzi zawierających v posortowana wg etykiet
      krawędzi (w pierwszej kolejności) i etykiet wierzchołków (w drugiej kolejności) */
4:    for all K' ∈ K(G') do
5:       v' ← K'.first;
6:       if lbl(v)! = lbl(v') then /* zgodność etykiet */
7:          continue;
8:       end if
9:       ne'[] ← G'.edges(v');
10:      found ← true;
11:      i ← 0; i' ← 0;
12:      loop:
13:      while i < ne.length do
14:         e ← ne[i]; i++;
15:         while i' < ne'.length do
16:            e' ← ne'[i']; i'++;
17:            if lbl(e'.e) = lbl(e.e) then /* zgodność sąsiadujących krawędzi */
18:               if lbl(e'.v) = lbl(e.v) then
19:                  if d(e'.v) ≥ d(e.v) then /* zgodność st. sąsiadujących wierzchołków */
20:                     continue loop;
21:                  end if
22:               else
23:                  if lbl(e'.v) > lbl(e.v) then
24:                     found ← false;
25:                     break loop;
26:                  end if
27:               end if
28:            else
29:               if lbl(e'.e) > lbl(e.e) then
30:                  found ← false;
31:                  break loop;
32:               end if
33:            end if
34:         end while
35:         found ← false;
36:         break loop;
37:      end while
38:      if found = true then
39:         do dziedziny każdego wierzchołka klasy K dodaj każdy wierzchołek klasy K';
40:      end if
41:   end for
42:   if |D_v| < |K| then /* warunek spójności dziedziny */
43:      return false;
44:   end if
45: end for
46: return true;
```

5.3.5. Kolejność wyboru zmiennej

Algorytm z powrotami w każdym kroku wybiera nieprzyporządkowaną zmienną, dla której szukana będzie wartość spełniająca ograniczenia. Kolejność wyboru zmiennych ma bardzo duże znaczenie dla szybkości działania algorytmu z powrotami. Heurystyki wyboru kolejności zmiennej dzielą się na statyczne i dynamiczne. Heurystyki statyczne ([23, 28, 71]) określają kolejność przed uruchomieniem algorytmu z powrotami, natomiast heurystyki dynamiczne ([34, 29]) określają kolejność po każdym przyporządkowaniu. Heurystyki dynamiczne są potencjalnie bardziej efektywne i wykorzystuje się je w większości przypadków. Heurystyki statyczne mają zastosowanie w przypadku różnych modyfikacji algorytmu z powrotami, w których kolejność wyboru zmiennych musi być założona z góry (np. pierwsze wersje algorytmu backjumping [29]). Oczekiwaną własnością heurystyk jest takie wybieranie zmiennych, aby zminimalizować rozmiar drzewa przeszukiwań przez jak najszybsze ograniczanie dziedziny oraz jak najszybsze znajdowanie nieudanych przyporządkowań, gdyż powrót z głębi drzewa jest bardziej kosztowny niż powrót z początku drzewa. W pracy [34] wykazano, że wybór zmiennej o najmniej licznej dziedzinie minimalizuje średnią łączną długość gałęzi drzewa. Skuteczność tej heurystyki w dużej mierze zależy od skuteczności ograniczania dziedziny przez metodę sprawdzania w przód. Eksperymenty prowadzone w ramach tej pracy, potwierdzają najwyższą efektywność tej metody (rozdział 6.6.2).

5.3.6. Kolejność wyboru wartości

W algorytmie z powrotami każdej zmiennej próbuje się przyporządkować wartości z jej dziedziny aż do momentu znalezienia rozwiązania lub wyczerpania dziedziny. Zatem jeśli problem CSP ma rozwiązanie, wybór właściwych wartości do przyporządkowania pozwoli na szybsze zakończenie algorytmu. Celem heurystyki wyboru kolejności wartości jest wybieranie takich wartości z dziedziny danej zmiennej, aby maksymalizować prawdopodobieństwo, że przyporządkowanie jest częścią rozwiązania. Podobnie jak dla heurystyk wyboru zmiennych, heurystyki wyboru wartości dzielą się na statyczne i dynamiczne. Najczęściej spotykaną heurystyką jest wybór najmniej ograniczającej wartości, czyli takiej, która wyklucza jak najmniej wartości innych zmiennych.



Rysunek 5.3. Izomorfizm podgrafu w przypadku pary symetrycznych wierzchołków. Niepotrzebne przyporządkowania zaznaczono strzałką.

5.3.7. Symetria w grafie

Rozpatrzmy przypadek znajdujący się na rysunku 5.3. Badamy izomorfizm grafu G z podgrafem pewnego grafu G'. Graf G' nie jest przedstawiony na rysunku, znane są natomiast dziedziny wierzchołków grafu G. W grafie G wierzchołki 1 i 2 są symetryczne, co oznacza, że jeśli je zamienimy ze sobą, uzyskany graf będzie identyczny z G, czyli zamiana miejscami wierchołków 1 i 2 jest automorfizmem grafu G. Klasyczny algorytm z powrotami (bez sprawdzania w przód) będzie przyporządkowywał wartości w kolejności podanej na rysunku 5.3. Jeśli G nie jest izomorficzny z podgrafem G' to wszystkie wymienione przeporządkowania zostaną sprawdzone. Niepowodzenie połowy z nich można wywnioskować od razu, korzystając z własności symetrii wierzchołków 1 i 2. Są to przyporządkowania zaznaczone na rysunku strzałką. Jeśli bowiem wierzchołek 1 ma już przyporządkowanie, to wierzchołkowi drugiemu nie ma sensu przyporządkowywać wartości, które były już sprawdzane dla wierzchołka pierwszego, gdyż gdyby to przyporządkowanie było poprawne, zostałoby znalezione wcześniej. Na przykład przyporządkowanie $x_{v1} \leftarrow 2$, $x_{v2} \leftarrow 1$ jest redundantne, gdyż gdyby było poprawne, powodzeniem zakończyłoby się przyporządkowanie $x_{v1} \leftarrow 1$, $x_{v2} \leftarrow 2$.

Podany przykład stanowi najprostszy przypadek symetrii w problemie CSP. Symetria w problemie CSP objawia się występowaniem wielu symetrycznych poddrzew w drzewie przeszukiwań. Jeśli problem CSP nie ma rozwiązania, symetryczne poddrzewa są

niepotrzebnie przeszukiwane[4]. Metody wykrywania i wykorzystywania symetrii w ogólnym problemie CSP były studiowane od wielu lat, m.in. w pracach [62, 5, 6]. Symetria w problemie CSP jest permutacją σ przekształcającą przyporządkowania w przyporządkowania symetryczne, to znaczy takie, które nie zmieniają kształtu zdań logicznych w zbiorze ograniczeń C. Możemy wyróżnić dwie klasy symetrii: symetrię zmiennych i symetrię wartości. Symetria zmiennych jest permutacją $\sigma : X \to X$ przekształcającą przyporządkowanie $s = (x_1 \leftarrow d_1, \ldots, x_n \leftarrow d_n)$ w przyporządkowanie $s' = (\sigma(x_1) \leftarrow d_1, \ldots, \sigma(x_n) \leftarrow d_n)$. Symetria wartości jest permutacją $\sigma : D \to D$ przekształcającą przyporządkowanie $s = (x_1 \leftarrow d_1, \ldots, x_n \leftarrow d_n)$ w przyporządkowanie $s' = (x_1 \leftarrow \sigma(d_1), \ldots, \sigma(x_n) \leftarrow \sigma(d_n))$. Jeśli w problemie CSP istnieje pewna symetria σ, wtedy dla danego przyporządkowania s istnieje przyporządkowania symetryczne $\sigma(s)$. Jedno z tych przyporządkowań można uznać za redundantne i rozwiązując problem CSP można pominąć sprawdzanie jego poprawności, gdyż jeśli s jest poprawne, czyli nie narusza ograniczeń ze zbioru C, wtedy $\sigma(s)$ też nie narusza tych ograniczeń. Podobnie, jeśli s narusza ograniczenia ze zbioru C, wtedy $\sigma(s)$ również narusza te ograniczenia. Spośród przyporządkowań s i $\sigma(s)$ za rendundante można uznać arbitralnie dowolne z nich. W niniejszej rozprawie za redundantne przyporządkowanie będziemy uznawali większe z symetrycznych przyporządkowań.

Przykład 5.2. *(przykład symetrii w problemie CSP)*

Dany jest problem $CSP = (X, D, C)$, gdzie:

$X = \{x_1, x_2\}$

$D_1 = D_2 = \{-3, -2, -1, 0, 1, 2, 3\}$

$C = \{x_1^2 \cdot x_2^2 = 36\}$

W podanym problemie występuje jedna symetria zmiennych σ_x oraz trzy symetrie wartości $\sigma_1, \sigma_2, \sigma_3$.

$\sigma_x = \binom{x_1 \;\; x_2}{x_2 \;\; x_1}$

$s = (x_1 \leftarrow d_1, x_2 \leftarrow d_2), \qquad C = \{d_1^2 \cdot d_2^2 = 36\}$

$\sigma_x(s) = (x_2 \leftarrow d_1, x_1 \leftarrow d_2), \; C = \{d_2^2 \cdot d_1^2 = 36\}$

[4] Nie należy w tym miejscu wyciągać wniosku, że wykorzystanie symetrii przynosi efekty tylko w przypadku problemów, które nie mają rozwiązań. Nawet jeśli rozwiązanie istnieje, symetria może wskazać fragmenty drzewa przeszukiwań, które mogą być pominięte przy szukaniu rozwiązania

$\sigma_1 = \binom{-1\ \ 1}{1\ -1}$

$s = (x_1 \leftarrow 1), \qquad C = \{1^2 \cdot x_2^2 = 36\}$

$\sigma_1(s) = (x_1 \leftarrow -1), C = \{(-1)^2 \cdot x_2^2 = 36\}$

$\sigma_2 = \binom{-2\ \ 2}{2\ -2}$

$s = (x_1 \leftarrow 2), \qquad C = \{2^2 \cdot x_2^2 = 36\}$

$\sigma_1(s) = (x_1 \leftarrow -2), C = \{(-2)^2 \cdot x_2^2 = 36\}$

$\sigma_3 = \binom{-3\ \ 3}{3\ -3}$

$s = (x_1 \leftarrow 3), \qquad C = \{3^2 \cdot x_2^2 = 36\}$

$\sigma_1(s) = (x_1 \leftarrow -3), C = \{(-3)^2 \cdot x_2^2 = 36\}$

W pracy [80] pokazano, że w przypadku problemu izomorfizmu grafu G z podgrafem grafu G', symetrie zmiennych są automorfizmami grafu G i nie zależą od G', natomiast symetrie wartości są automorfizmami grafu G' i nie zależą od G. Autorzy tej pracy proponują, aby symetrie włączyć do ograniczeń problemu CSP, dzięki czemu odcięcia drzewa przeszukiwań będą następowały szybciej. W przypadku symetrii zmiennych procedura jest następująca:

1. Znajdź wszystkie nietrywialne automorfizmy grafu G.
2. Dla każdego automorfizmu σ utwórz ograniczenie $s \leq \sigma(s)$ i dodaj je do zbioru C.

Odnajdywanie wszystkich automorfizmów grafu G może być czasochłonne, zatem proponowane przez autora niniejszej rozprawy rozwiązanie korzysta jedynie ze szczególnego przypadku automorfizmu - symetrii, która ze względu na sposób działania algorytmów UGM i UFC jest łatwa do zidentyfikowania (szczegóły w rozdziale 5.3.10).

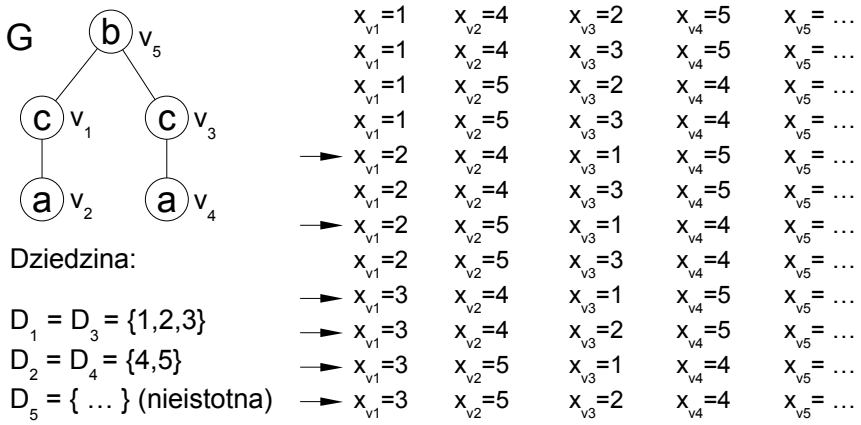
Definicja 5.3. *(wierzchołki parami symetryczne)*
W grafie $G(V = \{v_1, \ldots, v_n\}, E)$ wierzchołki
L=$(v_{i1}, v_{i2}, \ldots v_{ik})$, $v_{i1} < v_{i2} < \ldots < v_{ik}$, są parami symetryczne do wierzchołków
R=$(v_{j1}, v_{j2}, \ldots v_{jk})$, $v_{j1} > v_{i1}, v_{j2} > v_{i2}, \ldots, v_{jk} > v_{ik}$
wtedy i tylko wtedy, gdy istnieje automorfizm σ grafu G taki, że

$\sigma(v_{i1}) = v_{j1} \wedge$

$\sigma(v_{i2}) = v_{j2} \ \wedge \ldots \wedge$

$\sigma(v_{ik}) = v_{jk} \wedge$

$\sigma(v_{j1}) = v_{i1} \wedge$



Rysunek 5.4. Izomorfizm podgrafu w przypadku symetrycznych wierzchołków. Niepotrzebne przyporządkowania zaznaczono strzałką.

$\sigma(v_{j2}) = v_{i2} \ \wedge \ldots \wedge$

$\sigma(v_{jk}) = v_{ik} \wedge$

$\sigma(v_i) = v_i \ \forall_{i \notin \{i_1,i_2,\ldots,i_k,j_1,j_2,\ldots,j_k\}}$.

Innymi słowy, jeśli zamienimy parami wierzchołki v_{i1} z v_{j1}, v_{i2} z v_{j2}, ..., v_{ik} z v_{jk}, uzyskany graf będzie identyczny z G. Ciąg wierzchołków L nazywamy lewą stroną symetrii, a ciąg wierzchołków R - prawą stroną symetrii. Symetrię oznaczamy jako $S = (L, R)$

Na rysunku 5.3 wierzchołek 1 jest parami symetryczny do wierzchołka 2, a na rysunku 5.4 wierzchołki (1,2) są parami symetryczne do wierzchołków (3,4). Dla przykładu z rysunku 5.4 można przeprowadzić podobne rozumowanie, jakie pokazano wcześniej dla prostszego przypadku pary symetrycznych wierzchołków. Jeśli graf G nie jest izomorficzny z żadnym podgrafem grafu G', to wszystkie przyporządkowania pokazane na rysunku zostaną sprawdzone. Połowa z nich (zaznaczona na rysunku strzałką) jest jednak redundantna, gdyż dla każdej z nich istnieje przyporządkowanie symetryczne. Na przykład, gdyby przyporządkowanie $x_{v1} \leftarrow 2, \ x_{v2} \leftarrow 4, \ x_{v3} \leftarrow 1, \ x_{v4} \leftarrow 5$ było poprawne, to poprawne było by też przyporządkowanie symetryczne $x_{v1} \leftarrow 1, \ x_{v2} \leftarrow 5, \ x_{v3} \leftarrow 2, \ x_{v4} \leftarrow 4$. Zatem algorytm z powrotami powinien rozpatrywać tylko jedno wyróżnione w pewien sposób przyporządkowanie i omijać wszystkie przyporządkowania symetryczne, na przykład przyporządkowania większe. Na rysunku 5.4 mamy jedną symetrię σ: $\sigma(v_1) = v_3$, $\sigma(v_2) = v_4$, $\sigma(v_3) = v_1$, $\sigma(v_4) = v_2$, $\sigma(v_5) = v_5$. Zatem, jeśli $s = (x_{v1} \leftarrow 2, x_{v2} \leftarrow 4, x_{v3} \leftarrow 1, x_{v4} \leftarrow 5)$, wtedy $\sigma(s) = (x_{v3} \leftarrow 2, \ x_{v4} \leftarrow 4, \ x_{v1} \leftarrow 1, \ x_{v2} \leftarrow 5)$ = $(x_{v1} \leftarrow 1, \ x_{v2} \leftarrow 5, \ x_{v3} \leftarrow 2, \ x_{v4} \leftarrow 4)$. W tym przypadku okazuje się, że $s > \sigma(s)$, zatem

przyporządkowanie s można odrzucić. Autor tej rozprawy proponuje inne rozwiązanie, które jest szczegółowo opisane w następnym rozdziale. Warunki $s \leq \sigma(s)$ zostały przekształcone do innej postaci i służą nie tylko do określania redundantego przyporządkowania, ale zostają wykorzystane do ograniczania dziedziny.

5.3.8. Wykorzystanie symetrii jako mechanizmu ograniczania dziedziny

Pomysł wykorzystania symetrii jako mechanizmu ograniczania dziedziny opiera się na zaproponowanym przez autora tej rozprawy twierdzeniu 5.1.

Twierdzenie 5.1. *(warunek wystarczający nieredundancji przyporządkowania w przypadku symetrii zmiennych)*
Jeśli wierzchołki $L = (v_{i1}, v_{i2}, \ldots v_{ik})$ grafu G są parami symetryczne do wierzchołków $R = (v_{j1}, v_{j2}, \ldots v_{jk})$ grafu G, wtedy przyporządkowania zawierające zmienne $x_{v_{i1}}$ i $x_{v_{j1}}$, spełniające warunek

$$x_{v_{i1}} < x_{v_{j1}} \tag{5.1}$$

są nieredundantne z odpowiadającymi im przyporządkowaniami symetrycznymi.

Dowód. Wg definicji 5.3 symetria $S = (L, R)$ w grafie G o n wierzchołkach $v_1 \ldots v_n$ jest permutacją σ :

$\sigma(v_{i1}) = v_{j1} \wedge$
$\sigma(v_{i2}) = v_{j2} \wedge \ldots \wedge$
$\sigma(v_{ik}) = v_{jk} \wedge$
$\sigma(v_{j1}) = v_{i1} \wedge$
$\sigma(v_{j2}) = v_{i2} \wedge \ldots \wedge$
$\sigma(v_{jk}) = v_{ik} \wedge$
$\sigma(v_i) = v_i \; \forall_{i \notin \{i_1, i_2, \ldots, i_k, j_1, j_2, \ldots, j_k\}}$,
$v_{i1} < v_{i2} < \ldots < v_{ik}$,
$v_{i1} < v_{j1}, v_{i2} < v_{j2}, \ldots, v_{ik} < v_{jk}$.

Na mocy tej samej definicji permutacja σ jest też automorfizmem grafu G, zatem jeśli s jest dowolnym przyporządkowaniem, w którym biorą udział zmienne $x_{v_{i1}}$ oraz $x_{v_{j1}}$, to $\sigma(s)$ jest przyporządkowaniem symetrycznym. Na początek przyjmijmy, że s jest przyporządkowaniem zawierającym wszystkie zmienne, czyli

$$s = (x_{v1}, x_{v2}, \ldots x_{v_{i1}}, x_{v_{i2}}, \ldots, x_{v_{ik}}, \ldots x_{v_{j1}}, \ldots x_{vn}),$$
$$\sigma(s) = (x_{v1}, x_{v2}, \ldots x_{v_{j1}}, x_{v_{j2}}, \ldots, x_{v_{jk}}, \ldots x_{v_{i1}}, \ldots x_{vn}),$$
$$\sigma(\sigma(s)) = s.$$

Dla danej symetrii σ istnieją tylko 2 symetryczne przyporządkowania: s i $\sigma(s)$. Jedno z nich jest redundantne. Załóżmy, że przyporządkowanie większe jest redundantne. Mamy więc warunek nieredundancji $s < \sigma(s)$. Po podstawieniu

$$(x_{v1}, x_{v2}, \ldots x_{v_{i1}}, x_{v_{i2}}, \ldots, x_{v_{ik}}, \ldots x_{v_{j1}}, \ldots x_{vn}) <$$
$$(x_{v1}, x_{v2}, \ldots x_{v_{j1}}, x_{v_{j2}}, \ldots, x_{v_{jk}}, \ldots x_{v_{i1}}, \ldots x_{vn}),$$
$$\text{stąd } x_{v_{i1}} < x_{v_{j1}}.$$

Przedstawione powyżej rozumowanie można przeprowadzić dla przyporządowań zawierających dowolny podzbiór zmiennych, zawierający $x_{v_{i1}}$ oraz $x_{v_{j1}}$. □

Twierdzenie 5.1 może być wykorzystane do ograniczania dziedziny w następujący sposób. W chwili, gdy zmienna, powiedzmy $x_{v_{cur}}$, zostaje wybrana do przyporządkowania, rozpatrywane są wszystkie symetrie $S = (L, R)$, w których wierzchołek v_{cur} występuje jako pierwszy element ciągu L lub ciągu R, czyli $v_{cur} = v_{i1}$ lub $v_{cur} = v_{j1}$. W zależności od tego, czy v_{cur} należy do L czy do R warunkiem nieredudancji przyprządkowania jest albo $x_{v_{cur}} < x_{v_{j1}}$ albo $x_{v_{i1}} < x_{v_{cur}}$. Dziedzinę zmiennej $x_{v_{cur}}$ można zatem ograniczyć do wartości, które spełniają dany warunek nieredundancji. Dla każdej symetrii, w której wierzchołek v_{cur} występuje jako pierwszy element ciągu L lub ciągu R wyznacza się wartość minimalną pierwszego elementu lewej strony $minL$, oraz wartość maksymalną pierwszego elementu prawej strony $maxR$ w następujący sposób. Jeśli zmienna $x_{v_{i1}}$ ma przyporządkowanie, wtedy $minL = x_{v_{i1}}$. W przeciwnym przypadku za $minL$ przyjmuje się najmniejszą wartość z dziedziny zmiennej $x_{v_{i1}}$, czyli $minL = min(D_{v_{i1}})$. Analogicznie postępujemy dla prawej strony symetrii. Jeśli zmienna $x_{v_{j1}}$ ma przyporządkowanie, wtedy $maxR = x_{v_{j1}}$. W przeciwnym przypadku za $maxR$ przyjmuje się największą wartość z dziedziny zmiennej $x_{v_{j1}}$, czyli $maxR = max(D_{v_{j1}})$. Po wyznaczeniu $minL$ i $maxR$ dla danej symetrii S następuje ograniczenie dziedziny zmiennej $x_{v_{cur}}$ (czyli $D_{v_{cur}}$). Dziedzina zostaje ograniczona w trzech przypadkach:

— $minL \geq maxR \wedge (v_{cur} \in L \vee v_{cur} \in R)$:

Warunek z twierdzenia 5.1 nie może zostać spełniony. Dziedzina zostaje ograniczona do

zbioru pustego. Pozostałe symetrie nie muszą być sprawdzane.

$D_{v_{cur}} \leftarrow \emptyset$

— $minL < maxR \wedge v_{cur} \in L$:

Warunek z twierdzenia 5.1 będzie spełniony tylko w przypadku, gdy $x_{v_{cur}} < maxR$. Dziedzina $D_{v_{cur}}$ zostaje ograniczona z góry do wartości $maxR - 1$.

$D_{v_{cur}} \leftarrow [min(D_{v_{cur}}), min(max(D_{v_{cur}}), maxR - 1)]$

— $minL < maxR \wedge v_{cur} \in R$:

Warunek z twierdzenia 5.1 będzie spełniony tylko w przypadku, gdy $x_{v_{cur}} > minL$. Dziedzina $D_{v_{cur}}$ zostaje ograniczona z dołu do wartości $minL + 1$.

$D_{v_{cur}} \leftarrow [max(min(D_{v_{cur}}), minL + 1), max(D_{v_{cur}})]$

W pozostałych przypadkach D_{cur} nie ulega zmianie. Przedstawiony sposób wykorzystania symetrii do ograniczania dziedziny szczegółowo przedstawia algorytm 5.28.

Powyżej rozpatrywane były symetrie zmiennych, czyli symetrie korzystające z automorfizmów grafu G. Symetrie wartości korzystają natomiast z automorfizmów grafu G'. Jeśli w grafie G' wierzchołki v'_1 i v'_2, $v'_1 < v'_2$ są symetryczne i nie zostały jeszcze użyte jako wartość przyporządkowania, to danej zmiennej x nie trzeba przyporządkowywać wartości v'_2, gdyż przyporządkowanie $x \leftarrow v'_2$ jest symetryczne z rozpatrywanym wcześniej przyporządkowaniem $x \leftarrow v'_1$.

Twierdzenie 5.2. *(warunek wystarczający redundancji przyporządkowania w przypadku symetrii wartości)*
Jeśli do dziedziny danej zmiennej x należą (między innymi) dwa symetryczne do siebie wierzchołki v'_1 i v'_2, $v'_1 < v'_2$ to zakładając, że algorytm rozwiązujący CSP przyporządkowuje wartości w kolejności rosnącej, przyporządkowanie $x \leftarrow v'_2$ jest redundantne z odpowiadającym mu przyporządkowaniem symetrycznym $x \leftarrow v'_1$.

5.3.9. Powroty z przeskokami

Algorytm z przeskokami (ang. backjumping) jest rozwinięciem algorytmu z powrotami, w którym po sprawdzeniu wszystkich możliwych przyporządkowań danej zmiennej, następuje powrót nie do poprzednio rozpatrywanej zmiennej, ale bezpośrednio do zmiennej, która była przyczyną niepowodzenia przyporządkowania. Pozwala to na uniknięcie serii przyporządkowań, o których z góry wiadomo, że nie będą poprawne. Na przykład, załóżmy, że znaleziono już przyporządkowanie dla zmiennych kolejno x_1, x_2, ..., x_9, a bieżąco

Algorytm 5.28 variableSymmetryDomainLimitation(v_{cur})

/* assert: $L = (v_{i1}, v_{i2}, \ldots v_{ik})R = (v_{j1}, v_{j2}, \ldots v_{jk})$ */
for all $S = (L, R) \in Symmetries(G)$ **do**
  if $v_{cur} \neq v_{i1} \wedge v_{cur} \neq v_{j1}$ **then**
    continue;
  end if
  if $x_{v_{i1}}$ ma przyporządkowanie **then**
    $minL = x_{v_{i1}}$;
  else
    $minL = min(D_{v_{i1}}))$;
  end if
  if $x_{v_{j1}}$ ma przyporządkowanie **then**
    $maxR = x_{v_{j1}}$;
  else
    $maxR = max(D_{v_{j1}}))$
  end if
  if $minL \geq maxR$ **then**
    $D_{v_{cur}} \leftarrow \emptyset$;
    return ;
  end if
  if $minL < maxR \wedge v_{cur} \in L$ **then**
    $D_{v_{cur}} \leftarrow [min(D_{v_{cur}}), min(max(D_{v_{cur}}), maxR - 1)]$
  end if
  if $minL < maxR \wedge v_{cur} \in R$ **then**
    $D_{v_{cur}} \leftarrow [max(min(D_{v_{cur}}), minL + 1), max(D_{v_{cur}})]$
  end if
  if $D = \emptyset$ **then**
    return
  end if
end for

Algorytm 5.29 valueSymmetryRedundancyCheck(v_{cur}, v'_{cur})

for all $v' \in D_{v_{cur}} : v' < v'_{cur}$ **do**
  if v' jest symetryczny z v'_{cur} **then**
    return true; /* $x_{v_{cur}} \leftarrow v'_{cur}$ jest rozwiązaniem redundantnym */
  end if
end for
return false; /* $x_{v_{cur}} \leftarrow v'_{cur}$ nie jest rozwiązaniem redundantnym */

rozpatrywaną zmienną jest x_{10}. Okazuje się, że żadne przyporządkowanie zmiennej x_{10} nie jest poprawne, ale w zbiorze C nie ma żadnych ograniczeń wiążących zmienną x_9 i x_{10}. Zatem powrót do zmiennej x_9 i przyporządkowywanie jej innych wartości nie pozwoli na znalezienie poprawnego przyporządkowania zmiennej x_{10}. Jedyną szansą na poprawne przyporządkowanie zmiennej x_{10} jest zmiana ostanio modyfikowanej zmiennej, która występuje razem z x_{10} w zbiorze ograniczeń. Algorytm z przeskokami dosyć trudno jest połączyć z dynamiczną kolejnością wyboru zmiennych oraz ze sprawdzaniem w przód. Metody łączenia algorytmu z przeskokami z innymi technikami zostały szeroko omówione w [29]. Eksperymenty prowadzone w ramach tej pracy pokazują, że w przypadku problemu izomorfizmu z podgrafem nie wykazuje on przewagi nad algorytmem z powrotami.

5.3.10. Izomorfizm z podgrafem w kontekście algorytmów UGM i UFC

W algorymach UGM i UFC wykonuje się wiele testów na izomorfizm grafu G z podgrafem grafu G'. W przypadku wyznaczania wsparcia, graf G jest grafem kandydującym, a graf G' jest grafem z wejściowej bazy danych. W przypadku optymalizacji z wykorzystaniem zbioru grafów nieczęstych graf G jest grafem pochodzącym ze zbioru $\mathbb{GN}$, a graf G' jest grafem kandydującym. Zarówno dany graf G jak i dany graf G' biorą udział w testach na izomorfizm wielokrotnie. Warto więc utrzymywać razem z danym grafem pomocnicze informacje wspierające wykonywanie testów na izomorfizm, gdyż będą one używane wiele razy. W proponowanej implementacji przechowywane są dwie tego rodzaju struktury, dzięki czemu nie jest konieczna ich każdorazowa generacja:

— klasy równoważności wierzchołków - zgodnie z opisem w rozdziale 5.3.4 do jednej klasy należą wierzchołki o tej samej etykiecie i tym samym wielozbiorze deskryptorów krawędzi wychodzących;

— symetrie wierzchołków - graf G jest reprezentowany jako zbiór składowych spójnych wraz z krotnościami występowania poszczególnych składowych $G = \{(CG_1, c_1), (CG_2, c_2), \ldots, (CG_n, c_n)\}$. Jeśli składowa spójna CG_i występuje w grafie więcej niż jeden raz ($c_i > 1$), wtedy kolejne wystąpienia tej składowej są symetryczne. Symetrie występujące w ramach jednej składowej spójnej również bezpośrednio przenoszą się na wszystkie wystąpienia tej składowej.

Eksperymenty pokazują, że przechowywanie wymienionych struktur danych, zamiast ich każdorazowego wyznaczenia zwiększa wydajność algorytmu nawet o rząd wielkości.

Kolejną informacją jaką niesie kontekst jest to, że dany graf kandydujący G został utworzony przez dodanie jednej krawędzi do grafu częstego, czyli grafu, który przeszedł wystarczająco dużo testów na izomorfizm z podgrafem. Pozwala to na wykorzystanie informacji pochodzących z tych testów podczas rozpatrywania grafu G. Autor tej rozprawy proponuje dwie niezależne techniki działające na tej zasadzie: przenoszenie poprawnego rozwiązania i przenoszenie niepoprawnych przyporządkowań. Obie te techniki wiążą się z dodaniem informacji do każdej pary grafów (G, G'), dla której test na izomorfizm grafu G z podgrafem grafu G' zakończył się sukcesem i G' jest grafem z wejściowej bazy grafów.

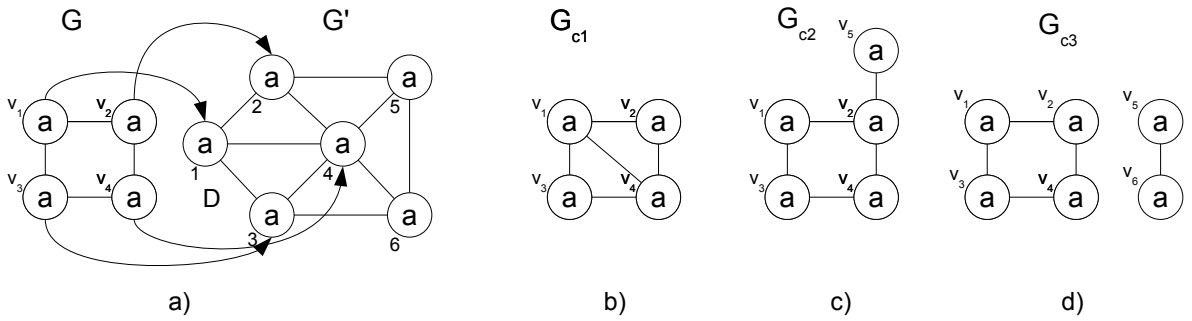
Przenoszenie poprawnego rozwiązania

Załóżmy, że G jest izomorficzny z pewnym podgrafem grafu G'. Odpowiedni test podał rozwiązanie w postaci $s = (x_{v1} \leftarrow v'_1, \ldots, x_{vn} \leftarrow v'_n)$. Rozwiązanie s jest zgodnie z definicją 2.16 jednym z zanurzeń grafu G w G'. Jeśli G jest grafem częstym, to powstanie graf kandydujący G_c poprzez dodanie jednej krawędzi do grafu G. Jeśli w zanurzeniu s możliwe jest dodanie tej krawędzi, wtedy można stwierdzić, że graf G_c jest izomorficzny z podgrafem grafu G' bez potrzeby wykonywania testu na izomorfizm. Graf G_c może powstać z grafu G na trzy sposoby:

— dodanie krawędzi o etykiecie le pomiędzy istniejące wierzchołki v_i i v_j (rys 5.5b);
Jeżeli w grafie G' istnieje krawędź pomiędzy wierzchołkami x_{vi} i x_{vj} o etykiecie le, wtedy graf G_c jest izomorficzny z podgrafem grafu G. Rozwiązaniem tego izomorfizmu jest $s_c = s$. W przeciwnym przypadku należy przeprowadzić osobny test.

— dodanie nowego wierzchołka v_{n+1} o etykiecie lv i krawędzi o etykiecie le pomiędzy istniejącym wierzchołkiem v_i a dodanym wierzchołkiem (rys 5.5c);
Jeżeli w grafie G' istnieje krawędź o etykiecie le wychodząca z x_{vi} do pewnego wierzchołka $v' \neq x_{vk}, k = 1 \ldots n$ o etykiecie lv, wtedy graf G_c jest izomorficzny z podgrafem grafu G. Rozwiązaniem tego izomorfizmu jest $s_c = s \cup (x_{vn+1} \leftarrow v')$. W przeciwnym przypadku należy przeprowadzić osobny test.

— dodanie dwóch nowych wierzchołków (v_{n+1} o etykiecie lv_1 i v_{n+2} o etykiecie lv_2) i krawędzi o etykiecie le pomiędzy nimi (rys 5.5d);
Jeżeli w grafie G' istnieją krawędź o etykiecie le łącząca dwa wierzchołki $v'_a \neq x_{vk}, k = 1 \ldots n$ i $v'_b \neq x_{vk}, k = 1 \ldots n$ o etykietach odpowiednio lv_1 i lv_2, wtedy graf G_c jest izomorficzny z podgrafem grafu G. Rozwiązaniem tego izomorfizmu jest



Rysunek 5.5. Izomorfizm grafu G i jego rozszerzeń G_{c1}, G_{c2}, G_{c3} z podgrafem grafu G'. Zanurzenie zostało zobrazowane za pomocą strzałek.

$s_c = s \cup (x_{vn+1} \leftarrow v'_a, x_{vn+2} \leftarrow v'_b)$. W przeciwnym przypadku należy przeprowadzić osobny test.

Przenoszenie poprawnego rozwiązania jest okrojonym przypadkiem przechowywania listy zanurzeń. Jak zostało wspomniane, przechowywanie wszystkich zanurzeń pozwala na uniknięcie wykonywanie testów na izomorfizm podgrafu, ale jest wymagające pamięciowo i obliczeniowo w przypadku grafów niespójnych. W przypadku przenoszenia poprawnego rozwiązania przechowywane jest tylko jedno zanurzenie, ale eksperymenty pokazują, że prawdopodobieństwo, że jest ono zawarte w rozszerzeniach jest w granicach 50%, a więc redukuje liczbę wykonywanych testów na izomorfizm z podgrafem o połowę.

Przenoszenie niepoprawnych przyporządkowań

W czasie wykonywania testów na izomorfizm z podgrafem może się okazać, że niektóre przyporządkowania $x_{vi} \leftarrow v'$ nie mogą być częścią poprawnego pełnego rozwiązania. Jeśli uda się stwierdzić, że nie istnieje zanurzenie G w G' w którym $x_{vi} \leftarrow v'$ mimo, że $v' \in D_{vi}$, to takie przyporządkowanie będzie niepoprawne we wszystkich rozszerzeniach grafu G oraz rozszerzeniach ich rozszerzeń itd. Wartość v' można więc usunąć z dziedziny D_{vi} w każdym z takich grafów. Wyszukiwanie wszystkich takich przyporządkowań byłoby czasochłonne, ale niektóre z nich można łatwo odnaleźć w trakcie standardowego algorytmu z powrotami. Załóżmy, że algorytm z powrotami wybiera do przyporządkowań kolejno zmienne $x_{v1}, x_{v2}, \ldots, x_{vn}$. Wszystkie nieudane przyporządkowania pierwszej zmiennej (x_{v1}) na pewno nie są częścią pełnego poprawnego rozwiązania, gdyż dla x_{v1} zostały sprawdzone wszystkie kombinacje przyporządkowań pozostałych zmiennych. Poza tym, jeśli $|D_{v1}| = 1$, wtedy wszystkie nieudane przyporządkowania drugiej zmiennej (x_{v2}) również nie są częścią

pełnego poprawnego rozwiązania itd. Jako przykłady niepoprawnych przyporządkowań przenoszone są niepoprawne przyporządkowania pierwszej wybranej zmiennej oraz niepoprawne przyporządkowania kolejnych wybranych zmiennych pod warunkiem, że dziedziny wcześniej wybieranych zmiennych miały liczność jeden, a jest to bardzo prawdopodobne, gdyż najlepszą heurystyką wyboru zmiennej jest ta wybierająca zmienną o najmniej licznej dziedzinie. Eksperymenty pokazują, że liczba znalezionych niepoprawnych przyporządkowań jest bardzo duża, ale niekoniecznie wpływają one na efektywność algorytmu, dlatego warto ograniczyć przenoszenie niepoprawnych przyporządkowań tylko do nietrywialnych przypadków. Za nietrywialne niepoprawne przyporządkowania zostają uznane te niepoprawne przyporządkowania, których znalezienie wymagało liczby powrotów algorytmu przekraczającej określony próg $min\_wrong\_backtracks$.

5.4. Algorytm izomorfizmu z podgrafem

Algorytm 5.30 przedstawia pełne rozwiązanie problemu izomorfizmu grafu G z podgrafem grafu G' wraz z przedstawionymi optymalizacjami. Na początku algorytmu sprawdzane są trzy proste warunki, pozwalające natychmiast podać rozwiązanie:

— jeśli graf G nie posiada wierzchołków, wtedy G jest izomorficzny z podgrafem grafu G' i algorytm się kończy,

— jeśli graf G zawiera więcej wierzchołków niż graf G', wtedy graf G nie jest izomorficzny z podgrafem graf G' i algorytm się kończy,

— jeśli rozmiar największej składowej spójnej grafu G jest większy pod względem liczby wierzchołków niż rozmiar największej składowej spójnej grafu G', wtedy G nie jest izomorficzny z podgrafem grafu G' i algorytm się kończy.

Następnie algorytm wykonuje procedurę $initializeLegalValues$, której celem jest ograniczenie dziedziny oraz jednocześnie sprawdzenie jej spójności, zgodnie z opisem w rozdziałach 5.3.1 i 5.3.4. Jeśli dziedzina okaże się niespójna, wtedy graf G nie jest izomorficzny z podgrafem grafu G' i algorytm się kończy.

Następnym krokiem jest ograniczanie dziedziny z pomocą algorytmu $AC3$. Potem wykorzystywany jest mechanizm przenoszenia poprawnego rozwiązania opisany w rozdziale 5.3.10: jeśli graf G powstał z grafu G_{prev} przez dodanie jednej krawędzi i w zanurzeniu

Algorytm 5.30 subgraphIsomorphism(G,G')

```
/* assert: V jest zbiorem wierzchołków grafu G; V' jest zbiorem wierzchołków grafu G' */
if |V|=0 then
   return true;
end if
if |V| > |V'| then
   return false;
end if
if rozmiar największej składowej spójnej grafu G jest większy niż rozmiar największej
składowej spójnej grafu G' then
   return false;
end if
if initializeLegalValues(G, G') = false then
   return false;
end if
AC_3(); /* opcjonalnie */
if G powstał z grafu G_prev przez dodanie jednej krawędzi i w zanurzeniu grafu G_prev
w grafie G' możliwe jest dodanie tej krawędzi (patrz rozdział 5.3.10) then
   return true;
end if;
X ← zbiór o liczności |V| zawierący zmienne bez przyporządkowania;
return recursiveBactracking(X, G, G'); /* lub backjumping(X, G, G'), dla wersji
z przeskokami */
```

grafu G_{prev} w grafie G' możliwe jest dodanie tej krawędzi, wtedy graf G jest izomorficzny z podgrafem grafu G' i algorytm się kończy.

Dopiero po tych krokach następuje przeszukiwanie przestrzeni rozwiązań za pomocą algorytmu z powrotami (funkcja $recursiveBacktracking$) lub z przeskokami (funkcja $backjumping$). Procedura $recursiveBactracking(X, G, G')$ rozpoczyna się sprawdzeniem warunku zakończenia rekurencji: jeżeli wszystkie zmienne ze zbioru X mają przyporządkowanie, wtedy funkcja zwraca wartość prawdy, co oznacza, że graf G jest izomorficzny z podgrafem grafu G'. W przeciwnym wypadku wybierana jest zmienna $x_{v_{cur}}$ do przyporządkowania według ustalonej heurystyki. Dziedzina wybranej zmiennej jest następnie ograniczana za pomocą mechanizmu wykorzystania symetrii zmiennych (funkcja $variableSymmetryDomainLimitation$). Do zmiennej $x_{v_{cur}}$ przyporządkowuje się następnie kolejne wartości z tak ograniczonej dziedziny. Po każdym przyporządkowaniu następuje ograniczenie dziedziny za pomocą sprawdzania w przód i ponownie wywołuje się procedurę $recursiveBactracking(X, G, G')$. Jeśli wywołanie zakończy się wartością prawdy, wtedy bieżące wywołanie także zwraca wartość prawdy. W przeciwnym przypadku

przyporządkowanie zmiennej $x_{v_{cur}}$ zostaje anulowane. Jeżeli przyporządkowanie nie powiedzie się dla każdej wartości z dziedziny, wtedy dziedzina zmiennej $x_{v_{cur}}$ zostaje przywrócona do stanu sprzed ograniczania dziedziny i funkcja zwraca wartość fałszu.

Procedura $backjumping(X, G, G')$ jest podobna do $recursiveBactracking(X, G, G')$ z tą różnicą, że nie jest wykonywana rekurencyjnie. Dodatkowo dla każdej zmiennej x_v utrzymywany jest ciąg P_v zawierający zmienne, których przyporządkowanie powoduje zmianę dziedziny zmiennej x_v. Ciąg P_v uaktualniany jest po każdym ograniczaniu dziedziny i używany jest później do wykonywania przeskoków - w przypadku niepowodzenia przyporządkowania bieżącej zmiennej $x_{v_{cur}}$ następuje powrót do ostatniej zmiennej dodanej do $P_{v_{cur}}$, a nie do ostatnio rozpatrywanej zmiennej, jak to ma miejsce w algorytmie z powrotami.

Algorytm 5.31 recursiveBactracking(X, G, G')

```
/* assert: V jest zbiorm wierzchołków grafu G */
if wszystkie zmienne z X mają przyporządkowanie then
   return true;
end if
v_cur ← Wybierz wierzchołek z V taki, że x_{v_cur} nie ma przyporządkowania;
variableSymmetryDomainLimitation(v_cur);
for all v' ∈ D_{v_cur} do
   if valueSymmetryRedundancyCheck(v_cur, v') then
      continue;
   end if
   przyporządkuj x_{v_cur} ← v';
   ogranicz wszystkie dziedziny za pomocą sprawdzania w przód (patrz rozdział 5.3.3)
   i opcjonalnie AC_3;
   if recursiveBactracking() then
      return true;
   else
      Anuluj przyporządkowanie x_{v_cur} ← v';
      Przywróć stan wszystkich dziedzin do stanu sprzed sprawdzania w przód;
      Dodaj przyporządkowanie x_{v_cur} ← v' do listy niepoprawnych przyporządkowań, pod
      warunkiem, że spełnione są warunki z rozdziału 5.3.10
   end if
end for
Przywróć stan dziedziny D_{v_cur} do stanu sprzed wywołania
variableSymmetryDomainLimitation;
return false;
```

Algorytm 5.32 backjumping(X, G, G')

$v_{cur} \leftarrow -1$;
$v' \leftarrow -1$;
$P_v \leftarrow \emptyset$ dla każdego $v \in V$
while $true$ **do**
  if $v_{cur} = -1$ **then**
    if wszystkie zmienne z X mają przyporządkowanie **then**
      return true;
    end if
    $v_{cur} \leftarrow$ Wybierz wierzchołek z V taki, że $x_{v_{cur}}$ nie ma przyporządkowania;
    $variableSymmetryDomainLimitation(v_{cur})$;
  end if
  if $|D_{v_{cur}}| > 0$ **then**
    $v' \leftarrow$ pierwsza wartość v'_i z $D_{v_{cur}}$ większa od v' i spełniająca warunek
    $valueSymmetryRedundancyCheck(v_{cur}, v'_i)$;
  else
    $v' \leftarrow -1$;
  end if
  if $v' \geq 0$ **then**
    przyporządkuj $x_{v_{cur}} \leftarrow v'$;
    ogranicz wszystkie dziedziny za pomocą sprawdzania w przód (patrz rozdział 5.3.3)
    i opcjonalnie AC_3;
    dla każdej zmodyfikowanej dziedziny $D_v \neq D_{v_{cur}}$ wykonaj $P_v \leftarrow P_v \cup v_{cur}$;
    $v_{cur} \leftarrow -1$;
    $continue$;
  else
    if $P_{v_{cur}} = \emptyset$ **then**
      return $false$;
    end if
    $P_{tmp} \leftarrow P_{v_{cur}}$;
    $v_{cur} \leftarrow$ ostatnia wartość dodana do $P_{v_{cur}}$;
    $P_{v_{cur}} \leftarrow P_{v_{cur}} \cup P_{tmp} - v_{cur}$;
    anuluj wszystkie przyporządkowania zmiennych, które nastąpiły po
    przyporządkowaniu $x_{v_{cur}}$ włącznie;
    przywróć stan wszystkich dziedzin do stanu sprzed przyporządkowania zmiennej $x_{v_{cur}}$;
  end if
end while

5.5. Algorytm izomorfizmu grafów

Rozwiązanie problemu izomorfizmu grafów G i G' można rozwiązań analogicznymi metodami jak w przypadku izomorfizmu podgrafu. Należy jedynie pamiętać, że w przypadku izomorfizmu rozwiązanie musi być bijekcją. Modyfikacji musi ulec tylko część odpowiedzialna za ograniczanie dziedziny. W 5.3.1 zostały przedstawione trzy warunki jakie muszą spełniać wierzchołki grafu G', aby mogły należeć do dziedziny zmiennych reprezentujacych wierzchołki grafu G. W przypadku izomorfizmu grafu warunek zgodności etykiet pozostaje taki sam, natomiast warunki zgodności sąsiadujących krawędzi oraz stopnia sąsiadujących wierzchołków wymagają zmiany. Dla problemu izomorfizmu grafów G i G' wszystkie warunki wyglądają następująco:

Do dziedziny zmiennej x_v należą tylko te wartości v', które spełniają następujące warunki:

— zgodność etykiet: $lbl(v) = lbl'(v')$; etykieta wierzchołka v jest taka sama jak etykieta wierzchołka v'.

— zgodność sąsiadujących krawędzi: $\exists$ bijekcja $f: \quad N_v(v) \to N_v(v')$ taka, że $\forall_{v_i \in N_v(v)}\ lbl(v_i) = lbl(f(v_i)) \wedge lbl(e = \{v, v_i\}) = lbl(e' = \{v', f(v_i)\}$, gdzie $N_v(i)$ jest zbiorem wierzchołków sąsiadujących z wierzchołkiem i. Warunek ten jest równoważny następującemu: wielozbiór deskryptorów krawędzi zawierających wierzchołek v musi być równy wielozbiorowi deskryptorów krawędzi zawierających wierzchołek v'.

— zgodność stopnia wierzchołków sąsiadujących: $\exists$ bijekcja $f: \ N_v(v) \to N_v(v')$ taka, że $\forall_{v_i \in N_v(v)}\ lbl(v_i) = lbl(f(v_i)) \wedge lbl(e = \{v, v_i\}) = lbl(e' = \{v', f(v_i)\}) \wedge d(v) = d(v')$, gdzie $N_v(i)$ jest zbiorem wierzchołków sąsiadujących z wierzchołkiem i. Jest to zaostrzenie warunku poprzedniego o warunek stopnia wierzchołków. Stopnie wierzchołków z otoczenia wierzchołka v muszą być równe stopniom odpowiadającym im wierzchołkom z otoczenia wierzchołka v'.

W związku ze zmianą warunków odpowiedniej modyfikacji musi ulec także algorytm 5.27. Linie 17-33 w algorytmie 5.27 muszą być zastąpione kodem pokazanym w algorytmie 5.33.

Dodatkowo, w przypadku problemu izomorfizmu grafu, istnieje więcej warunków koniecznych na istnienie izomorfizmu niż w przypadku izomorfizmu podgrafu. W grafach G i G' musi się zgadzać liczba wierzchołków, krawędzi i składowych spójnych. Ponieważ dla każdego grafu mamy reprezentację krotnościową jego składowych spójnych $G = \{(CG_1, c_1), (CG_2, c_2), \ldots, (CG_n, c_n)\}$, problem izomorfizmu grafów można wykonywać

Algorytm 5.33 Linie 17-33 funkcji InitiateLegalValues w przypadku izomorfizmu grafu

```
if lbl(e'.e) = lbl(e.e) then /* zgodność sąsiadujących krawędzi */
  if lbl(e'.v) = lbl(e.v) then
    if d(e'.v) = d(e.v) then /* zgodność stopnia sąsiadujących wierzchołków */
      continue loop;
    end if
  else
    found ← false;
    break loop;
  end if
else
  found ← false;
  break loop;
end if
```

niezależnie dla składowych spójnych. Grafy $G = \{(CG_1, c_1), (CG_2, c_2), \ldots, (CG_n, c_n)\}$ i $G' = \{(CG'_1, c'_1), (CG'_2, c'_2), \ldots, (CG'_n, c'_n)\}$ są izomorficzne wtedy i tylko wtedy gdy istnieje permutacja $f : N \to N$ taka, że $c_i = c'_{f(i)}$ i graf CG_i jest izomorficzny z grafem $CG_{f(i)}$. Permutację f (lub stwierdzenie jej braku) można znaleźć w czasie liniowym ze względu na liczbę składowych spójnych[5]. Eksperymenty pokazują, że wykonanie kilku oddzielnych testów na izomorfizm grafów spójnych jest szybsze niż jeden test na izomorfizm grafów niespójnych.

[5] traktując problem izomorfizmu grafów CG i CG' jako operację jednostkową

6. Eksperymenty

6.1. Opis eksperymentów

Wszystkie eksperymenty zostały wykonane z wykorzystaniem platformy *ParMol* [55]. Zaproponowane w pracy algorytmy UGM oraz UFC zostały zaimplementowane i włączone do platformy *ParMol*. Autor niniejszej rozprawy rozszerzył platformę $ParMol$ także o opisane w [77] rozszerzenie $gSpanUnconnected$ algorytmu $gSpan$, pozwalające na odkrywanie częstych grafów z uwzględnianiem niespójności oraz opisany w rozdziale 4.2.1 algorytm $UgmOnVirtual$. Łącznie zostało przetestowanych osiem algorytmów: cztery algorytmy odkrywające częste grafy spójne ($gSpan$, $MoFa$, $Gaston$, $FFSM$) oraz cztery algorytmy odkrywające wszystkie częste grafy (UGM, UFC, $gSpanUnconnected$, $UgmOnVirtual$). Za wejściowe zbiory grafów posłużyły chemiczne zbiory danych NCI[1], PTC[2] oraz $MUTAG$[3]. Tabele 6.1-6.4 przedstawiają charakterystykę danych wejściowych, na którą składa się liczba grafów w zbiorze, średnia liczba wierzchołków w grafie, średnia liczba krawędzi w grafie, liczba unikalnych deskryptorów krawędzi w całym zbiorze grafów oraz średnia liczba unikalnych deskryptorów krawędzi w grafie. Zbiór MUTAG składa się ze 188 związków chemicznych o działaniu mutagennym na bakterie z grupy *Salmonella typhimurium*. Kolekcja NCI składa się ze związków chemicznych mających działanie antynowotworowe, pogrupowanych według rodzaju komórek na 73 zbiory . Kolekcja PTC składa się z czterech zbiorów zawierających związki o działaniu rakotwórczym dla szczurów (zbiory FR i MR) oraz myszy (zbiory FM i MM).

Każdy algorytm został uruchomiony na każdym zbiorze danych z kilkoma wartościami progu minimalnego wsparcia. Zakres wsparcia był wybierany tak, aby czas wykonania jednego algorytmu zawierał się w przedziale od kilku sekund do kilku godzin. Dla każdego wykonania algorytmu zbierane były następujące wskaźniki: łączny czas wykonania oraz zbiór odkrytych grafów częstych, a dla algorytmów UGM i UFC dodatkowo: czasy działania poszczególnych

[1] http://cactus.nci.nih.gov/ncidb2/download.html

[2] http://www.predictive-toxicology.org/ptc/

[3] ftp://ftp.ics.uci.edu/pub/baldig/learning/mutag/INFO.txt

faz algorytmów, liczba wykonanych testów na izomorfizm grafu i podgrafu w różnych fazach algorytmu a także rozmiar zbiorów $\mathbb{GN}$ i $\mathbb{NKR}$.

6.2. Liczba spójnych grafów częstych i liczba wszystkich grafów częstych

Tabele 6.5-6.8 przedstawiają liczbę częstych grafów spójnych (kolumna *częste grafy spójne*) oraz liczbę wszystkich częstych grafów (kolumna *wszystkie częste grafy*) dla różnych wartości progu minimalnego wsparcia. Fragment tych danych został też zobrazowany na wykresie 6.1. Z przedstawionych danych wynikają dwa podstawowe wnioski:

— Liczba wszystkich częstych grafów jest od jednego do kilku rzędów wielkości większa niż liczba częstych grafów spójnych.

— Stosunek liczby częstych grafów spójnych do liczby wszystkich częstych grafów maleje wraz ze zmniejszającym się progiem minimalnego wsparcia. Na przykład w zbiorach z kolekcji NCI dla wsparcia na poziomie 30% częste grafy spójne stanowią około 10% wszystkich grafów częstych, a dla wsparcia 10% - już tylko 1%.

— Przedział wartości progu minimalnego wsparcia, dla których proces odkrywania grafów częstych kończy się w założonym czasie, zależy w dużym stopniu od charakteru danych wejściowych. Dla zbiorów z kolekcji PTC wartość minimalnego wsparcia zawierała się w przedziale 2%-10%, dla zbiorów z kolekcji NCI - 10%-30%, zaś dla zbioru $MUTAG$ - 30%-50%. W każdym z trzech przypadków liczba odkrytych grafów częstych była zbliżona.

zbiór grafów	liczba grafów	średnia liczba wierzchołków	średnia liczba krawędzi	liczba różnych deskryptorów	średnia liczba deskryptorów
mutag_188	188	17,93	19,79	18	5,04

Tabela 6.1. Własności zbioru grafów MUTAG.

zbiór grafów	liczba grafów	średnia liczba wierzchołków	średnia liczba krawędzi	liczba różnych deskryptorów	średnia liczba deskryptorów
786_0	3506	22,95	24,74	131	5,24
A498	3480	23,15	24,96	129	5,25
A549_ATCC	3734	23,02	24,83	137	5,25
ACHN	3531	22,97	24,76	130	5,24
BT_549	2778	23,27	25,1	109	5,28
CAKI_1	3580	22,98	24,79	133	5,25
CCRF_CEM	3480	23,18	24,99	129	5,24
COLO_205	3645	23,15	24,95	130	5,24
DLD_1	1222	24,62	26,54	97	5,26
DMS_114	1243	24,77	26,71	96	5,26
DMS_273	1181	24,89	26,85	95	5,26
DU_145	2945	23,13	24,95	113	5,26
EKVX	3681	23,11	24,91	137	5,25
HCC_2998	3177	23,5	25,36	123	5,2
HCT_116	3723	23,02	24,81	137	5,24
HCT_15	3731	23,0	24,79	136	5,25
HL_60_TB	3386	23,2	25,02	133	5,25
HOP_18	1041	25,19	27,16	92	5,27
HOP_62	3628	23,02	24,79	135	5,26
HOP_92	3503	23,23	25,06	132	5,27
HS_578T	2870	23,11	24,93	112	5,27
HT29	3712	23,07	24,86	137	5,25
IGROV1	3690	23,13	24,93	136	5,25
KM12	3705	23,08	24,89	134	5,25
KM20L2	1209	24,84	26,79	97	5,27
K_562	3618	23,02	24,8	133	5,25
LOX_IMVI	3603	22,9	24,68	134	5,22
LXFL_529	863	24,54	26,49	85	5,26
M14	3551	22,96	24,76	132	5,24
M19_MEL	1234	24,87	26,83	97	5,27
MALME_3M	3507	23,0	24,78	132	5,24
MCF7	3039	23,15	24,95	114	5,25

Tabela 6.2. Własności zbiorów grafów z kolekcji NCI. Część 1 z 2.

zbiór grafów	liczba grafów	średnia liczba wierzchołków	średnia liczba krawędzi	liczba różnych deskryptorów	średnia liczba deskryptorów
MDA_MB_231	2948	23,11	24,92	112	5,26
MDA_MB_435	2981	23,07	24,88	109	5,26
MDA_N	2962	23,06	24,87	112	5,26
MOLT_4	3534	23,23	25,04	130	5,24
NCI_ADR_RES	3111	22,96	24,74	116	5,27
NCI_H226	3464	23,05	24,86	132	5,24
NCI_H23	3719	23,0	24,8	137	5,25
NCI_H322M	3690	23,02	24,81	135	5,24
NCI_H460	3599	22,98	24,77	136	5,22
NCI_H522	3573	23,17	24,98	135	5,25
OVCAR_3	3691	23,07	24,87	136	5,26
OVCAR_4	3582	23,07	24,88	133	5,25
OVCAR_5	3670	23,17	24,98	136	5,25
OVCAR_8	3714	23,05	24,85	137	5,25
P388_ADR	455	26,41	28,42	54	5,34
P388	463	27,08	29,13	54	5,37
PC_3	2982	23,04	24,85	112	5,27
RPMI_8226	3564	23,07	24,87	133	5,27
RXF_393	3401	22,87	24,64	134	5,27
RXF_631	665	25,42	27,47	65	5,23
SF_268	3721	23,11	24,91	136	5,24
SF_295	3745	23,04	24,84	137	5,25
SF_539	3384	23,36	25,21	129	5,23
SK_MEL_28	3724	23,02	24,81	132	5,25
SK_MEL_2	3600	23,13	24,94	134	5,26
SK_MEL_5	3685	23,07	24,87	132	5,25
SK_OV_3	3503	23,2	25,01	131	5,24
SN12C	3682	23,01	24,81	137	5,24
SN12K1	468	26,93	28,98	54	5,37
SNB_19	3725	23,05	24,86	137	5,25
SNB_75	3490	23,23	25,04	134	5,24
SNB_78	1175	25,09	27,05	93	5,25
SR	3006	23,28	25,08	118	5,24
SW_620	3753	23,0	24,8	134	5,25
TK_10	3490	22,93	24,73	130	5,24
T_47D	2909	23,07	24,9	112	5,26
U251	3755	23,01	24,81	137	5,25
UACC_257	3681	23,14	24,95	132	5,24
UACC_62	3684	23,11	24,92	136	5,26
UO_31	3615	23,11	24,93	131	5,24
XF_498	1093	24,92	26,87	95	5,28

Tabela 6.3. Własności zbiorów grafów z kolekcji NCI. Część 2 z 2.

zbiór grafów	liczba grafów	średnia liczba wierzchołków	średnia liczba krawędzi	liczba różnych deskryptorów	średnia liczba deskryptorów
ptc_FM	349	14,09	14,46	49	4,13
ptc_FR	351	14,54	14,98	52	4,12
ptc_MM	336	13,95	14,3	52	4,08
ptc_MR	344	14,27	14,67	49	4,05

Tabela 6.4. Własności zbiorów grafów z kolekcji PTC.

	częste grafy spójne				wszystkie częste grafy		
	wsparcie [%]				wsparcie [%]		
zbiór grafów	50	40	30	20	50	40	30
mutag_188	432	661	2517	17172	3966	4893	31758

Tabela 6.5. Liczba częstych grafów spójnych oraz liczba wszystkich grafów częstych w zbiorze *MUTAG* dla różnych progów minimalnego wsparcia.

	częste grafy spójne						wszystkie częste grafy				
	wsparcie [%]						wsparcie [%]				
zbiór grafów	30	25	20	15	10	5	30	25	20	15	10
786_0	44	77	124	215	552	6115	388	895	2106	7340	54895
A498	45	79	121	224	554	6635	418	963	2238	8067	62778
A549_ATCC	43	76	121	213	547		375	871	2083	7332	56375
ACHN	44	75	122	213	545	5731	399	905	2104	7218	
BT_549	44	80	128	223	617	7270	423	998	2333	8649	67316
CAKI_1	45	78	125	223	579	7455	385	913	2202	7896	
CCRF_CEM	44	77	123	224	608	7084	383	907	2243	8309	69441
COLO_205	44	78	124	220	585	6832	385	922	2200	7949	
DLD_1	49	91	126	289	838		463	1279	4230	26555	
DMS_114	51	93	127	309	852		499	1323	4471	29029	
DMS_273	52	93	134	314	922		498	1383	4984	33443	
DU_145	45	79	127	223	586	6523	425	990	2293	8181	
EKVX	44	77	122	218	561	6362	384	918	2163	7644	
HCC_2998	45	81	125	242	675		401	994	2524	10610	105014
HCT_116	44	76	122	218	552		376	880	2100	7429	56490
HCT_15	43	77	118	215	545	6354	381	899	2098	7341	55114
HL_60_TB	44	77	121	226	586	7741	380	903	2210	8217	
HOP_18	54	95	140	314	915		558	1565	5529	34426	
HOP_62	43	76	123	206	530		372	870	2030	6941	51013
HOP_92	45	79	126	227	596		390	933	2260	8285	66645
HS_578T	46	79	126	222	579	5909	434	1020	2305	8111	56828
HT29	44	77	122	218	551		382	905	2132	7542	
IGROV1	44	78	120	220	561	6605	380	910	2164	7650	
KM12	44	78	117	218	558	6733	387	920	2124	7669	60584
KM20L2	54	94	134	311	892	12582	529	1410	4980	32381	
K_562	44	76	124	214	538		376	878	2083	7140	54010
LOX_IMVI	44	77	119	221	558		380	901	2099	7609	59570
LXFL_529	56	87	137	314	894		576	1502	4686		
M14	44	76	124	213	546	5571	397	918	2104	7228	52577
M19_MEL	50	95	128	312	872	11453	508	1401	4710	29990	
MALME_3M	43	77	122	213	547		357	835	2044	7088	55412
MCF7	47	79	128	225	569		419	979	2300	8347	61827
MDA_MB_231	44	79	127	221	584	6291	418	976	2247	8077	58342
MDA_MB_435	44	80	126	220	584	6207	413	969	2232	7953	56843
MDA_N	45	79	127	219	582		408	959	2235	7950	57671
MOLT_4	45	77	125	222	595		389	919	2242	8258	67841
NCI_ADR_RES	44	76	126	217	549	5475	378	870	2101	7024	46203
NCI_H226	44	77	124	224	579		383	886	2176	7888	65001

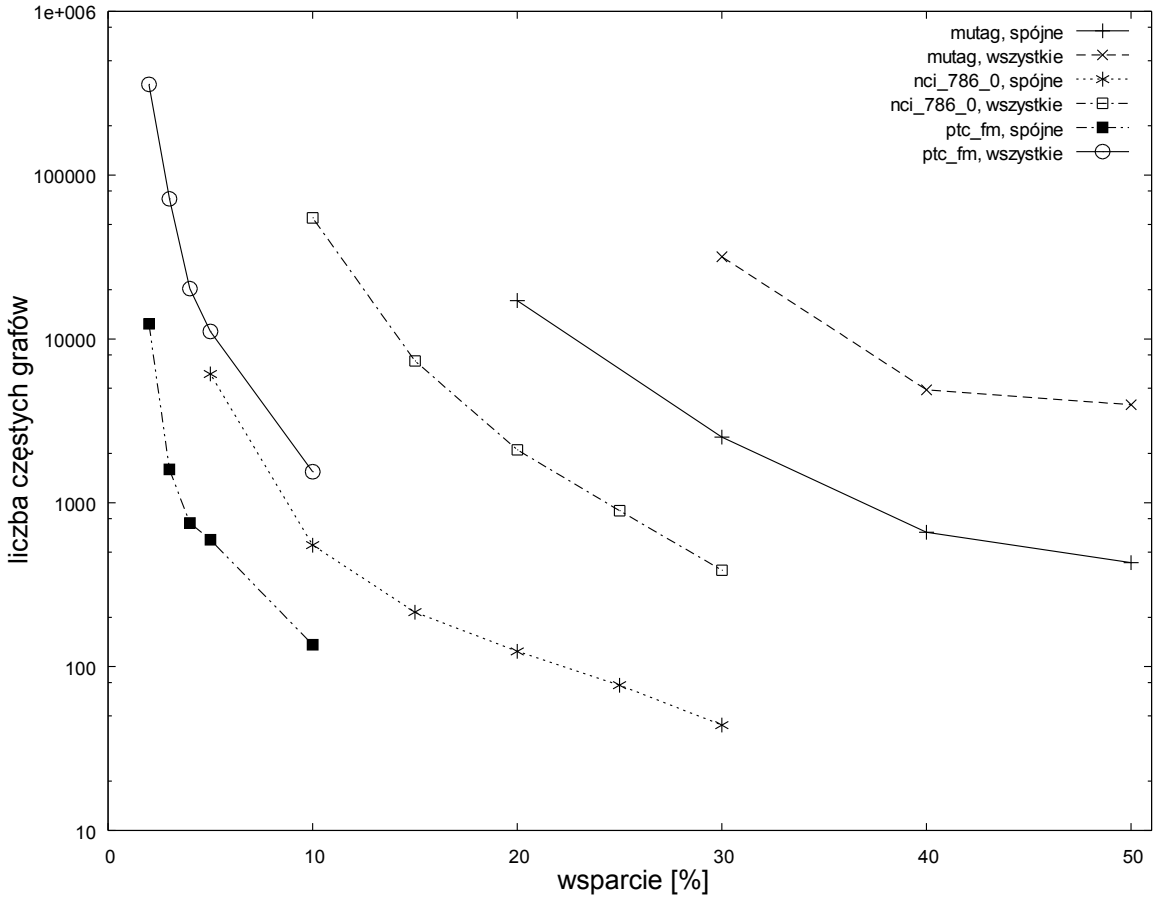
Tabela 6.6. Liczba częstych grafów spójnych oraz liczba wszystkich grafów częstych w zbiorach z kolekcji NCI dla różnych progów minimalnego wsparcia. Część 1 z 2.

	częste grafy spójne						wszystkie częste grafy				
	wsparcie [%]						wsparcie [%]				
zbiór grafów	30	25	20	15	10	5	30	25	20	15	10
NCI_H23	44	76	121	217	544		377	883	2085	7354	56305
NCI_H322M	44	76	122	218	558	6151	384	905	2129	7463	56254
NCI_H460	44	77	116	215	538		397	917	2070	7474	58346
NCI_H522	36	53	100	154	363		255	535	1351	3746	18788
OVCAR_3	43	77	122	214	555	6088	379	890	2102	7357	55592
OVCAR_4	44	77	125	217	569		379	910	2149	7633	58237
OVCAR_5	45	78	123	220	560	6550	384	921	2198	7871	61579
OVCAR_8	44	77	120	216	543		386	916	2118	7402	55664
P388_ADR	78	115	197	429	1508	26945	1337	4365	21991	162342	
P388	77	114	196	441	1561		1271	4364	21548	166316	
PC_3	45	79	127	221	587		417	971	2263	8028	58612
RPMI_8226	44	77	122	219	559		391	918	2169	7677	59761
RXF_393	42	75	122	183	518	4622	372	849	1970	6443	43802
RXF_631	72	97	173	404	1241	18793	778	2158	9211	65101	
SF_268	44	77	122	218	565	6512	389	915	2157	7685	59350
SF_295	43	77	122	218	558	6290	377	887	2112	7324	55707
SF_539	45	80	124	240	650		420	994	2396	9537	
SK_MEL_28	44	77	121	216	551		380	898	2108	7360	56880
SK_MEL_2	45	78	124	223	567	6623	388	919	2212	8016	64584
SK_MEL_5	44	77	121	217	564	6513	385	914	2143	7563	58280
SK_OV_3	45	79	120	227	593		390	929	2185	8100	65085
SN12C	44	75	121	214	538		381	899	2079	7309	54142
SN12K1	77	114	193	421	1498		1233	4058	19100	143787	
SNB_19	43	77	122	215	553	6362	381	902	2122	7386	56869
SNB_75	44	78	119	223	586	7129	394	934	2240	8263	71223
SNB_78	53	94	132	312	900	12102	555	1488	5171	34366	
SR	46	77	127	223	583	7253	419	986	2369	8751	72620
SW_620	44	76	121	216	544		382	899	2095	7352	55764
TK_10	44	77	122	214	550	6067	387	898	2082	7116	
T_47D	45	78	128	224	587		425	999	2306	8221	58745
U251	43	77	121	217	548	6195	381	896	2104	7369	55033
UACC_257	44	77	124	218	567	6387	391	933	2195	7793	59956
UACC_62	44	77	121	218	563		386	918	2174	7681	59739
UO_31	44	77	122	217	565	7073	382	899	2127	7523	57230
XF_498	54	96	137	321	993		536	1499	5555	37885	

Tabela 6.7. Liczba częstych grafów spójnych oraz liczba wszystkich grafów częstych w zbiorach z kolekcji NCI dla różnych progów minimalnego wsparcia. Część 2 z 2.

	częste grafy spójne					wszystkie częste grafy				
	wsparcie [%]					wsparcie [%]				
zbiór grafów	10	5	4	3	2	10	5	4	3	2
ptc_FM	136	595	753	1599	12379	1548	11120	20290	71671	358325
ptc_FR	149	564	897	1471	9695	1764	10407	24087	60913	
ptc_MM	117	492	763	1344	6465	1433	9571	23579	64913	325435
ptc_MR	129	602	875	2020	10838	1707	11475	21182	88256	

Tabela 6.8. Liczba częstych grafów spójnych oraz liczba wszystkich grafów częstych w zbiorach z kolekcji PTC dla różnych progów minimalnego wsparcia.



Rysunek 6.1. Liczba grafów spójnych oraz liczba wszystkich częstych grafów w zbiorach grafów PTC_FM, NCI_786_0 oraz MUTAG dla różnych wartości minimalnego wsparcia.

6.3. Porównianie wydajnościowe algorytmów odkrywania grafów częstych z uwzględnianiem niespójności

Tabele 6.9-6.12 oraz wykres 6.2 przedstawiają czasy wykonania algorytmów UGM, UFC oraz $gSpanUnconnected$ dla różnych wartości progu minimalnego wsparcia. Algorytm $UgmOnVirtual$ zakończył się błędem wyczerpania pamięci dla wszystkich badanych wartości progu minimalnego wsparcia, dlatego nie został umieszczony w tabelach. Późniejsze testy wykazały, że algorytm $UgmOnVirtual$ kończy się powodzeniem jedynie dla bardzo wysokich wartości $minSup$, często przekraczających nawet 90%, co czyni go bezużytecznym w większości zastosowań.

Tabele 6.13-6.16 zawierają procentowe zestawienie czasów wykonania algorytmów UFC oraz $gSpanUnconnected$ do czasów wykonania algorytmu UGM. Analiza wyników pozwala na wyciągnięcie następujących wniosków:

— Czas wykonania każdego algorytmu rośnie bardzo szybko (szybciej niż wykładniczo) wraz ze zmniejszaniem progu minimalnego wsparcia.

— Algorytm UFC jest w większości przypadków najszybszy. Algorytm UGM jest do kilkudziesięciu procent wolniejszy. Algorytm $gSpanUnconnected$ jest o rząd wolniejszy od algorytmów UGM i UFC.

— Im mniejsza wartość progu minimalnego wsparcia, tym większa przewaga algorytmów UGM i UFC nad algorytmem $gSpanUnconnected$. Dla dużych progów wsparcia ta przewaga jest kilkukrotna, a dla małych - kilkunastokrotna.

	czas wykonania [s]								
	ugm			ufc			gspan unconnected		
	wsparcie [%]			wsparcie [%]			wsparcie [%]		
zbiór grafów	50	40	30	50	40	30	50	40	30
mutag_188	16	21	481	19	27	465	242	335	

Tabela 6.9. Czas odkrywania częstych grafów spójnych i niespójnych w zbiorze MUTAG za pomocą algorytmów UGM, UFC oraz $gSpanUnconnected$.

	czas wykonania [s]												
	ugm					ufc					gspan unconnected		
	wsparcie [%]					wsparcie [%]					wsparcie [%]		
zbiór grafów	30	25	20	15	10	30	25	20	15	10	30	25	20
786_0	19	35	83	310	4135	19	35	73	250	2679	132	301	768
A498	19	37	88	340	5080	19	36	76	265	3310	149	333	841
A549_ATCC	20	36	85	323	4523	19	35	77	259	2992	137	311	806
ACHN	18	35	82	297		18	35	73	235		139	306	770
BT_549	16	31	72	297	4741	15	29	61	219	2766	120	274	663
CAKI_1	18	36	87	343		18	36	77	265		133	319	848
CCRF_CEM	20	47	89	361	5995	21	34	76	282	3980	131	307	796
COLO_205	19	38	89	356		19	36	79	281		137	330	843
DLD_1	12	22	79	961		8	31	52	594		64	181	733
DMS_114	11	23	87	1097		8	16	55	647		70	189	786
DMS_273	9	23	99	1346		8	16	62	827		69	208	889
DU_145	19	34	78	293		18	31	64	217		126	294	715
EKVX	19	37	88	336		19	36	79	272		139	326	838
HCC_2998	18	36	95	477	11032	16	34	77	344	6618	128	320	858
HCT_116	18	35	86	326	4451	18	36	78	259	3014	136	317	807
HCT_15	19	36	86	325	4331	18	36	78	256	2875	138	323	821
HL_60_TB	18	34	84	349		16	33	72	258		128	301	790
HOP_18	9	24	90	1182		7	16	59	689		69	201	855
HOP_62	18	32	73	274	3588	18	35	74	232	2548	130	302	759
HOP_92	19	37	90	357	5569	18	36	79	280	3580	137	333	851
HS_578T	16	31	69	256	3248	16	31	64	203	2097	126	292	700
HT29	19	36	89	334		19	37	79	266		141	323	841
IGROV1	19	37	87	337		18	36	78	265		139	325	811
KM12	21	40	87	339	5090	19	37	78	266	3203	140	336	830
KM20L2	10	24	96	1298		8	17	61	798		72	208	873
K_562	18	35	82	301	4152	18	35	74	236	2645	133	310	768
LOX_IMVI	19	36	84	326	4871	18	35	74	252	3197	137	312	808
LXFL_529	8	17	58			6	12	35			57	158	
M14	19	35	83	305	3815	18	36	74	235	2354	138	319	1350
M19_MEL	10	24	92	1126		8	17	59	694		71	204	1666
MALME_3M	17	33	78	295	4350	16	32	71	230	2844	121	274	1308
MCF7	17	33	79	311	4238	16	31	66	233	2628	125	294	1326
MDA_MB_231	17	32	76	284	3855	15	30	63	213	2335	127	288	1217
MDA_MB_435	16	32	75	285	3747	16	31	64	213	2126	125	284	1217
MDA_N	16	32	75	284	3721	15	30	63	212	2314	120	280	1202
MOLT_4	18	36	90	360	5765	18	36	78	278	3720	134	319	1472
NCI_ADR_RES	15	28	66	236	2408	15	29	62	195	1656	115	255	1172
NCI_H226	18	34	85	334	5473	17	33	73	259	3657	130	292	1392

Tabela 6.10. Czas odkrywania częstych grafów spójnych i niespójnych w kolekcji NCI za pomocą algorytmów UGM, UFC oraz $gSpanUnconnected$. Część 1 z 2.

	czas wykonania [s]												
	ugm					ufc					gspan unconnected		
	wsparcie [%]					wsparcie [%]					wsparcie [%]		
zbiór grafów	30	25	20	15	10	30	25	20	15	10	30	25	20
NCI_H23	18	36	86	328	4541	18	36	77	252	3005	139	318	1429
NCI_H322M	19	37	86	327	4399	18	37	77	261	2871	138	322	1423
NCI_H460	19	36	85	312	4729	19	36	74	242	3065	140	321	1406
NCI_H522	11	19	42	115	657	12	20	43	107	506	79	165	713
OVCAR_3	19	36	86	322	4391	19	36	77	255	2850	139	315	1404
OVCAR_5	19	37	90	348	5212	18	37	80	274	3375	139	326	1503
OVCAR_8	20	49	88	322	4314	20	38	78	251	2725	141	332	1430
P388_ADR	13	48	483	10223		9	31	264	3376		107	396	
P388	18	51	480	10953		9	31	264	3561		105	401	
PC_3	17	33	76	286	3910	16	31	65	214	2277	125	287	1232
RPMI_8226	18	36	84	325	4675	18	36	76	255	2910	138	321	1414
RXF_393	16	30	66	228	2629	17	31	65	194	1816	124	270	1177
RXF_631	9	23	121	2123		6	14	73	1157		62	191	
SF_268	20	38	90	344	4815	19	38	80	275	3223	141	336	1477
SF_295	18	37	88	326	4452	19	37	78	253	2950	139	318	1443
SF_539	19	38	95	426		18	36	79	312		144	341	1609
SK_MEL_28	19	36	86	323	4683	18	37	78	253	3080	140	320	1434
SK_MEL_2	19	37	89	344	5402	18	36	77	270	3544	136	319	1449
SK_MEL_5	20	37	87	323	4718	18	36	78	261	3124	140	325	1465
SK_OV_3	20	38	87	353	5329	20	36	75	262	3381	136	315	1447
SN12C	18	36	85	321	4171	18	36	76	243	2609	135	321	1402
SN12K1	13	45	388	8792		8	28	232	3076		102	371	
SNB_19	40	40	89	322	4543	19	37	78	262	3008	140	322	1431
SNB_75	39	36	88	355	6250	19	36	76	271	3969	134	319	1404
SNB_78	10	24	96	1329		8	17	62	809		73	217	
SR	17	33	82	330	5904	16	32	67	241	3731	128	299	1348
SW_620	20	37	87	324	4448	19	37	79	255	3010	140	323	1436
TK_10	18	35	81	291		17	34	72	230		133	298	1310
T_47D	18	32	76	291	3771	16	31	63	213	2269	126	283	1242
U251	19	37	87	318	4324	19	37	79	257	2918	141	322	1435
UACC_257	19	38	90	344	4922	19	38	80	270	3268	141	338	1489
UACC_62	23	41	93	339	4821	20	38	79	265	3125	141	324	1476
UO_31	19	36	85	328	4655	18	35	76	259	3078	137	314	1400
XF_498	11	24	109	1574		9	19	67	923		67	206	1988

Tabela 6.11. Czas odkrywania częstych grafów spójnych i niespójnych w kolekcji NCI za pomocą algorytmów UGM, UFC oraz $gSpanUnconnected$. Część 2 z 2.

	czas wykonania [s]												
	ugm					ufc					gspan unconnected		
	wsparcie [%]					wsparcie [%]					wsparcie [%]		
zbiór grafów	10	5	4	3	2	10	5	4	3	2	10	5	4
ptc_FM	5	28	62	510	7158	5	24	43	237	14603	52	366	
ptc_FR	6	29	89	417		5	24	59	200		52	379	
ptc_MM	4	24	77	444	5743	5	19	51	197	4757	46	337	
ptc_MR	6	31	76	726		5	26	52	446		57	414	

Tabela 6.12. Czas odkrywania częstych grafów spójnych i niespójnych w kolekcji PTC za pomocą algorytmów UGM, UFC oraz $gSpanUnconnected$.

	stosunek czasów wykonania					
	ufc/ugm			gspan unconnected/ugm		
	wsparcie [%]			wsparcie [%]		
zbiór grafów	50	40	30	50	40	30
mutag_188	1,15	1,31	0,97	14,70	16,25	

Tabela 6.13. Stosunek czasu wykonania algorytmu UFC oraz $gSpanUnconnected$ do czasu wykonania algorytmu UGM dla zbioru MUTAG.

	stosunek czasów wykonania							
	ufc/ugm					gspan unconnected/ugm		
	wsparcie [%]					wsparcie [%]		
zbiór grafów	30	25	20	15	10	30	25	20
786_0	0,96	0,99	0,88	0,81	0,65	6,81	8,50	9,22
A498	0,98	0,98	0,87	0,78	0,65	7,72	9,05	9,56
A549_ATCC	0,94	0,98	0,91	0,80	0,66	6,94	8,62	9,48
ACHN	0,99	1,00	0,89	0,79		7,55	8,69	9,40
BT_549	0,97	0,95	0,84	0,74	0,58	7,68	8,88	9,14
CAKI_1	0,97	1,00	0,89	0,77		7,27	8,93	9,77
CCRF_CEM	1,10	0,73	0,86	0,78	0,66	6,68	6,50	8,96
COLO_205	0,97	0,97	0,89	0,79		7,13	8,75	9,46
DLD_1	0,70	1,42	0,65	0,62		5,32	8,36	9,30
DMS_114	0,70	0,71	0,63	0,59		6,52	8,31	9,06
DMS_273	0,82	0,72	0,62	0,61		7,52	9,04	8,97
DU_145	0,96	0,91	0,82	0,74		6,72	8,75	9,11
EKVX	0,99	0,99	0,89	0,81		7,38	8,82	9,49
HCC_2998	0,89	0,93	0,81	0,72	0,60	7,07	8,86	9,02
HCT_116	0,98	1,03	0,90	0,79	0,68	7,38	9,00	9,36
HCT_15	0,94	0,99	0,90	0,79	0,66	7,08	8,85	9,51
HL_60_TB	0,91	0,95	0,86	0,74		7,14	8,77	9,39
HOP_18	0,77	0,68	0,65	0,58		7,28	8,51	9,47
HOP_62	1,00	1,07	1,01	0,85	0,71	7,34	9,31	10,41
HOP_92	0,95	0,98	0,87	0,79	0,64	7,13	9,12	9,42
HS_578T	0,97	1,00	0,93	0,80	0,65	7,89	9,47	10,15
HT29	0,99	1,01	0,89	0,80		7,33	8,90	9,51
IGROV1	0,96	0,98	0,90	0,78		7,29	8,78	9,30
KM12	0,91	0,92	0,89	0,79	0,63	6,73	8,32	9,51
KM20L2	0,80	0,70	0,64	0,61		7,36	8,74	9,06
K_562	0,99	1,00	0,91	0,78	0,64	7,31	8,88	9,37
LOX_IMVI	0,97	0,97	0,89	0,78	0,66	7,42	8,67	9,66
LXFL_529	0,82	0,69	0,61			7,45	9,07	
M14	0,98	1,02	0,90	0,77	0,62	7,32	8,99	16,35
M19_MEL	0,80	0,70	0,63	0,62		7,06	8,42	18,04
MALME_3M	0,98	0,97	0,91	0,78	0,65	7,27	8,42	16,79
MCF7	0,95	0,93	0,83	0,75	0,62	7,47	8,84	16,75
MDA_MB_231	0,93	0,94	0,84	0,75	0,61	7,70	9,02	16,10
MDA_MB_435	0,95	0,96	0,85	0,75	0,57	7,57	8,89	16,13
MDA_N	0,95	0,95	0,84	0,75	0,62	7,44	8,84	15,93
MOLT_4	0,96	0,99	0,86	0,77	0,65	7,27	8,86	16,29
NCI_ADR_RES	0,99	1,02	0,94	0,82	0,69	7,53	9,02	17,73
NCI_H226	0,94	0,97	0,85	0,77	0,67	7,20	8,59	16,34

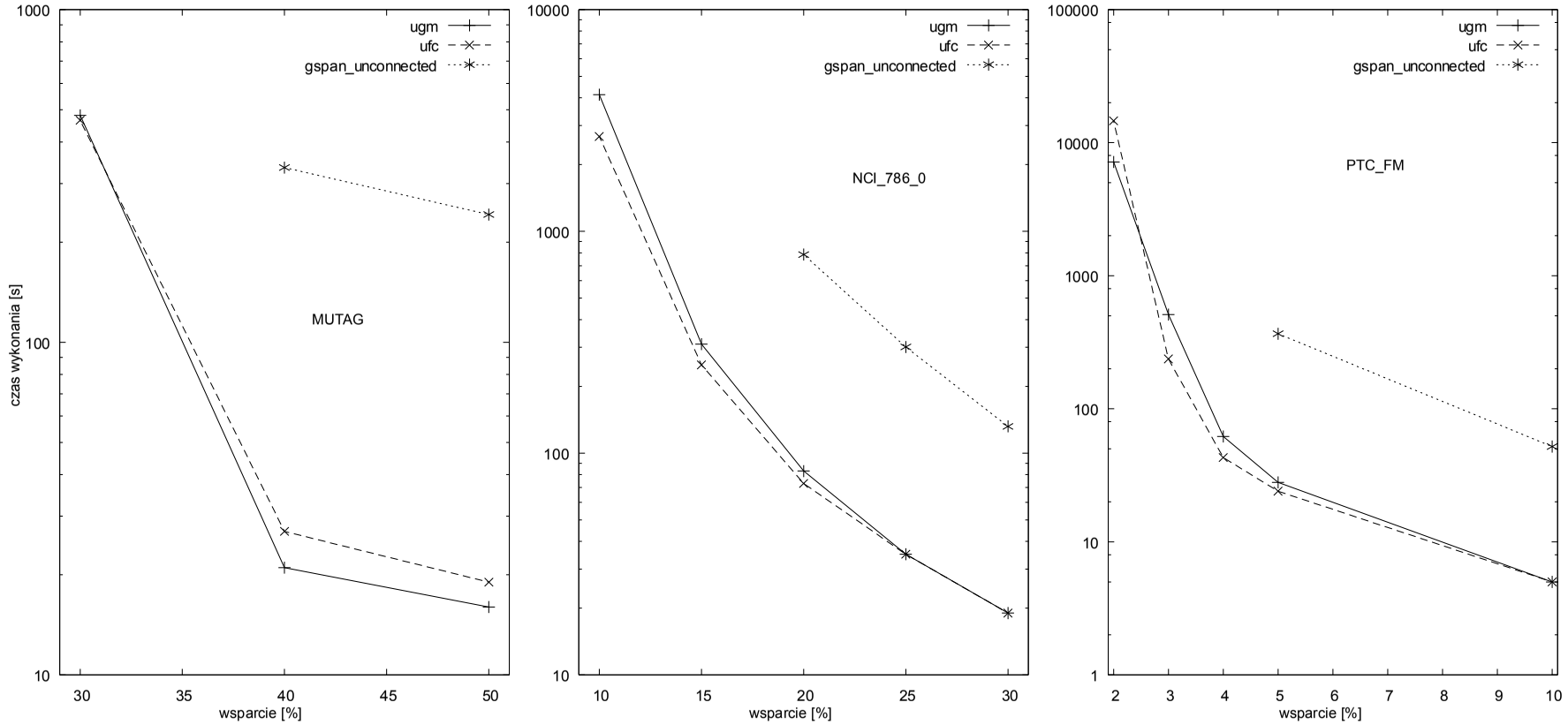
Tabela 6.14. Stosunek czasu wykonania algorytmu UFC oraz $gSpanUnconnected$ do czasu wykonania algorytmu UGM dla kolekcji NCI. Część 1 z 2.

	stosunek czasów wykonania							
	ufc/ugm					gspan unconnected/ugm		
	wsparcie [%]					wsparcie [%]		
zbiór grafów	30	25	20	15	10	30	25	20
NCI_H23	0,98	1,00	0,89	0,77	0,66	7,61	8,88	16,56
NCI_H322M	0,97	1,00	0,90	0,80	0,65	7,29	8,81	16,55
NCI_H460	1,02	0,99	0,87	0,78	0,65	7,54	8,90	16,59
NCI_H522	1,07	1,04	1,02	0,93	0,77	7,11	8,52	17,12
OVCAR_3	1,00	1,00	0,89	0,79	0,65	7,48	8,64	16,28
OVCAR_5	0,97	0,99	0,89	0,79	0,65	7,37	8,74	16,70
OVCAR_8	0,98	0,79	0,88	0,78	0,63	7,07	6,84	16,17
P388_ADR	0,64	0,64	0,55	0,33		8,02	8,26	
P388	0,48	0,61	0,55	0,33		5,76	7,88	
PC_3	0,92	0,94	0,85	0,75	0,58	7,41	8,80	16,20
RPMI_8226	0,97	0,99	0,90	0,79	0,62	7,49	8,93	16,75
RXF_393	1,03	1,03	0,99	0,85	0,69	7,63	8,97	17,88
RXF_631	0,74	0,63	0,61	0,54		7,14	8,43	
SF_268	0,95	0,99	0,89	0,80	0,67	7,07	8,88	16,49
SF_295	1,00	1,01	0,88	0,78	0,66	7,53	8,69	16,44
SF_539	0,94	0,95	0,84	0,73		7,49	8,95	17,02
SK_MEL_28	0,98	1,01	0,91	0,78	0,66	7,48	8,86	16,62
SK_MEL_2	0,96	0,99	0,87	0,78	0,66	7,24	8,73	16,22
SK_MEL_5	0,91	0,98	0,90	0,81	0,66	6,91	8,77	16,90
SK_OV_3	1,01	0,95	0,86	0,74	0,63	6,88	8,22	16,64
SN12C	1,00	0,99	0,89	0,76	0,63	7,38	8,81	16,52
SN12K1	0,64	0,63	0,60	0,35		7,99	8,31	
SNB_19	0,48	0,92	0,88	0,81	0,66	3,52	7,99	16,04
SNB_75	0,48	0,98	0,86	0,76	0,64	3,45	8,81	15,88
SNB_78	0,78	0,71	0,64	0,61		7,29	9,00	
SR	0,96	0,96	0,82	0,73	0,63	7,72	8,99	16,46
SW_620	0,93	1,01	0,90	0,79	0,68	6,86	8,81	16,49
TK_10	0,97	0,99	0,88	0,79		7,40	8,61	16,18
T_47D	0,87	0,96	0,83	0,73	0,60	6,83	8,87	16,31
U251	0,97	1,00	0,90	0,81	0,67	7,38	8,76	16,52
UACC_257	0,99	0,99	0,89	0,79	0,66	7,43	8,91	16,51
UACC_62	0,87	0,93	0,85	0,78	0,65	6,19	7,92	15,82
UO_31	0,95	1,00	0,89	0,79	0,66	7,24	8,82	16,47
XF_498	0,86	0,78	0,61	0,59		6,40	8,53	18,16

Tabela 6.15. Stosunek czasu wykonania algorytmu UFC oraz $gSpanUnconnected$ do czasu wykonania algorytmu UGM dla kolekcji NCI. Część 2 z 2.

	stosunek czasów wykonania							
	ufc/ugm					gspan unconnected/ugm		
	wsparcie [%]					wsparcie [%]		
zbiór grafów	10	5	4	3	2	10	5	4
ptc_FM	0,99	0,86	0,70	0,47	2,04	10,88	12,97	
ptc_FR	0,84	0,84	0,66	0,48		8,64	13,15	
ptc_MM	1,12	0,82	0,66	0,44	0,83	10,45	14,25	
ptc_MR	0,90	0,82	0,68	0,62		10,07	13,26	

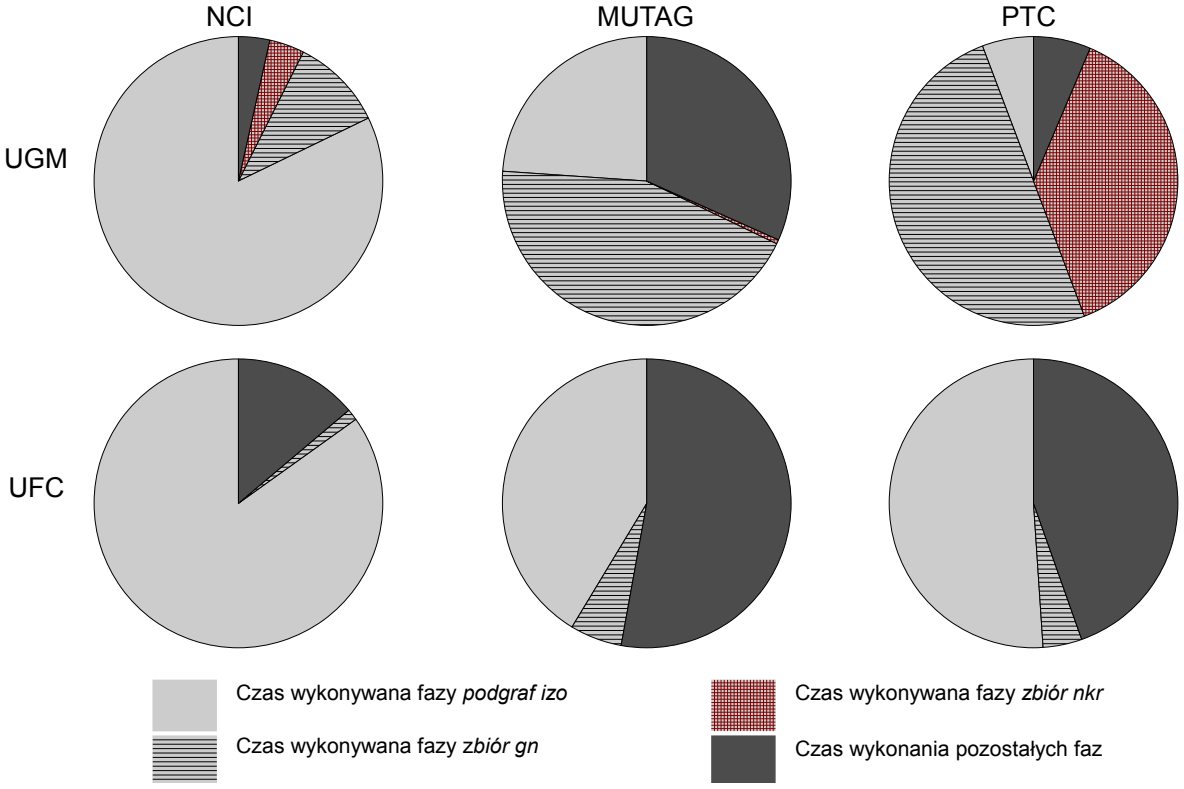
Tabela 6.16. Stosunek czasu wykonania algorytmu UFC oraz $gSpanUnconnected$ do czasu wykonania algorytmu UGM dla kolekcji PTC.



Rysunek 6.2. Czas odkrywania wszystkich częstych grafów w zbiorach grafów PTC_FM, NCI_786_0 oraz MUTAG przez algorytmy UGM, UFC oraz $gSpanUnconnected$

6.4. Analiza operacji składowych zaproponowanych algorytmów

Tabele 6.19-6.22 zawierają czasy wykonania najważniejszych faz algorytmów UGM i UFC dla wybranych zbiorów danych oraz wartości minimalnego wsparcia. Wyróżnione są następujące fazy:



Rysunek 6.3. Średni udział czasu wykonywania operacji związanych z badaniem izomorfizmu podgrafu (wyznaczanie wsparcia, obsługa zbioru grafów nieczęstych, obsługa zbioru nieczęstych konstruktorów rozszerzeń) w łącznym czasie wykonania algorytmów UGM i UFC w zbiorach grafów z kolekcji PTC, NCI oraz MUTAG.

— *całość* - łączny czas wykonania algorytmu,

— *podgraf izo* - czas wykonywania testów na izomorfizm z podgrafem podczas wykonywania operacji wyznaczania wsparcia grafów kandydujących,

— *graf izo* - czas wykonywania testów na izomorfizm z grafem,

— *zbiór gn* - łączny czas wszystkich operacji wykonywanych na strukturze przechowującej zbiór grafów nieczęstych (łącznie z niezbędnymi do tego celu operacjami testów na izomorfizm z podgrafem),

— *zbiór nkr* - łączny czas wszystkich operacji wykonywanych na strukturze przechowującej zbiór nieczęstych konstruktorów rozszerzeń (łącznie z niezbędnymi do tego celu operacjami testów na izomorfizm z podgrafem),
— *wielozbiory* - czas wyznaczania maksymalnych częstych wielozbiorów deskryptorów krawędzi,
— *podzbiór* - czas operacji sprawdzania, czy wielozbiór deskryptorów danego grafu, jest podzbiorem maksymalnego częstego wielozbioru deskryptorów krawędzi,
— *inne* - łączny czas wykonania innych, niewymienionych wyżej faz algorytmu.

zbiór grafów	całość	podgraf izo	graf izo	zbiór gn	zbiór knr	wielozbiory	podzbiór	inne
786_0	4135,0 s	3398,1 s	15,7 s	430,2 s	162,7 s	2,5 s	75,0 s	50,8 s
		82,2%	0,4%	10,4%	3,9%	0,1%	1,8%	1,2%
A498	5080,3 s	4074,0 s	15,7 s	610,4 s	217,1 s	2,1 s	103,5 s	57,4 s
		80,2%	0,3%	12,0%	4,3%	0,0%	2,0%	1,1%
A549_ATCC	4522,9 s	3733,0 s	13,8 s	472,0 s	169,8 s	2,1 s	87,5 s	44,7 s
		82,5%	0,3%	10,4%	3,8%	0,0%	1,9%	1,0%

Tabela 6.17. Czasy wykonania poszczególnych faz algorytmu UGM dla wybranych zbiorów kolekcji NCI przy progu minimalnego wsparcia 10%.

zbiór grafów	całość	podgraf izo	graf izo	zbiór gn	zbiór knr	wielozbiory	podzbiór	inne
786_0	2679,2 s	2274,5 s	0,9 s	33,7 s	0,0 s	2,5 s	8,1 s	359,5 s
		84,9%	0,0%	1,3%	0,0%	0,1%	0,3%	13,4%
A498	3309,7 s	2840,2 s	0,9 s	45,0 s	0,0 s	2,1 s	10,2 s	411,3 s
		85,8%	0,0%	1,4%	0,0%	0,1%	0,3%	12,4%
A549_ATCC	2991,6 s	2534,9 s	0,9 s	37,8 s	0,0 s	2,1 s	9,1 s	406,9 s
		84,7%	0,0%	1,3%	0,0%	0,1%	0,3%	13,6%

Tabela 6.18. Czasy wykonania poszczególnych faz algorytmu UFC dla wybranych zbiorów kolekcji NCI przy progu minimalnego wsparcia 10%.

zbiór grafów	całość	podgraf izo	graf izo	zbiór gn	zbiór knr	wielozbiory	podzbiór	inne
786_0	83,3 s	73,3 s	0,5 s	1,9 s	0,6 s	2,2 s	0,4 s	4,4 s
		88,0%	0,7%	2,3%	0,7%	2,6%	0,4%	5,3%
A498	88,0 s	78,6 s	0,4 s	2,1 s	0,5 s	1,9 s	0,4 s	4,1 s
		89,2%	0,5%	2,4%	0,5%	2,2%	0,4%	4,7%
A549_ATCC	84,9 s	75,7 s	0,5 s	1,9 s	0,3 s	2,0 s	0,4 s	4,1 s
		89,1%	0,6%	2,3%	0,4%	2,4%	0,4%	4,8%

Tabela 6.19. Czasy wykonania poszczególnych faz algorytmu UGM dla wybranych zbiorów kolekcji NCI przy progu minimalnego wsparcia 20%.

Analizę danych zawartych w tabelach ułatwia wykres 6.3. Analiza tych danych prowadzi do następujących obserwacji i wniosków:

zbiór grafów	całość	podgraf izo	graf izo	zbiór gn	zbiór knr	wielozbiory	podzbiór	inne
786_0	73,4 s	42,7 s	0,1 s	0,1 s	0,0 s	2,2 s	0,0 s	28,4 s
		58,1%	0,1%	0,1%	0,0%	3,0%	0,0%	38,6%
A498	76,4 s	44,7 s	0,0 s	0,1 s	0,0 s	1,9 s	0,1 s	29,5 s
		58,6%	0,1%	0,2%	0,0%	2,5%	0,1%	38,7%
A549_ATCC	77,1 s	43,9 s	0,0 s	0,1 s	0,0 s	2,0 s	0,1 s	31,0 s
		56,9%	0,1%	0,2%	0,0%	2,6%	0,1%	40,2%

Tabela 6.20. Czasy wykonania poszczególnych faz algorytmu UFC dla wybranych zbiorów kolekcji NCI przy progu minimalnego wsparcia 20%.

algorytm	całość	podgraf izo	graf izo	zbiór gn	zbiór knr	wielozbiory	podzbiór	inne
UGM	480,5 s	123,6 s	53,7 s	226,9 s	2,5 s	0,3 s	1,8 s	71,7s
		25,7%	11,2%	47,2%	0,5%	0,1%	0,4%	14,9%
UFC	465,6 s	192,5 s	10,9 s	27,1 s	0,0 s	0,3 s	0,4 s	234,3 s
		41,3%	2,3%	5,8%	0,0%	0,1%	0,1%	50,3%

Tabela 6.21. Czasy wykonania poszczególnych faz algorytmu UGM i UFC dla zbioru $MUTAG$ przy progu wsparcia 30%.

algorytm	całość	podgraf izo	graf izo	zbiór gn	zbiór knr	wielozbiory	podzbiór	inne
UGM	20,6 s	14,9 s	0,6 s	2,6 s	0,1 s	0,3 s	0,1 s	2,0 s
		72,2%	3,1%	12,6%	0,4%	1,5%	0,3%	9,9%
UFC	27,5 s	15,8 s	0,2 s	0,6 s	0,0 s	0,3 s	0,0 s	10,6 s
		57,5%	0,7%	2,2%	0,0%	1,1%	0,0%	38,5%

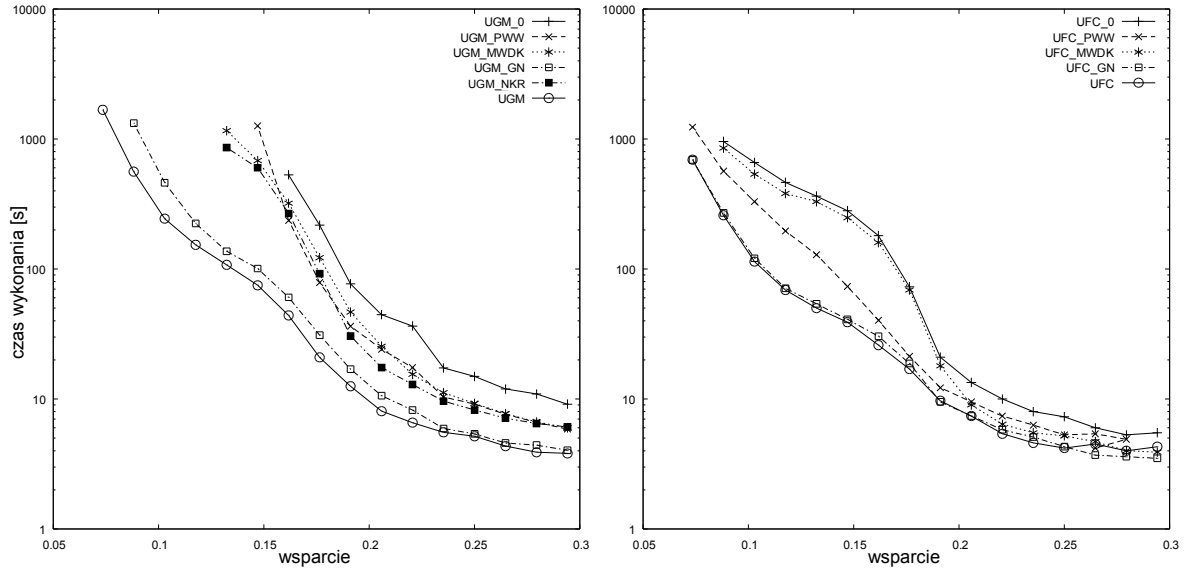
Tabela 6.22. Czasy wykonania poszczególnych faz algorytmu UGM i UFC dla zbioru $MUTAG$ przy progu wsparcia 40%.

— Wykonywanie testów na izomorfizm z podgrafem na etapie wyznaczania wsparć grafów jest w większości przypadków najbardziej czasochłonną operacją w przypadku obu algorytmów UGM i UFC i stanowi do 90% łącznego czasu wykonania algorytmów.
— Operacje związane z obsługą zbioru $\mathbb{GN}$ są czasochłonne w przypadku algorytmu UGM. Dla małych wartości wsparć czas wykonywania tych operacji może nawet stać się dominujący, co pokazują wyniki dla kolekcji $MUTAG$ i PTC.
— Obsługa zbioru $\mathbb{NKR}$ jest czasochłonna jedynie w przypadku kolekcji PTC.
— Wszystkie trzy wymienione wyżej fazy są związane z wykonywaniem testów na izomorfizm z podgrafem.
— Pozostałe wyróżnione operacje czyli testy na izomorfizm z grafem oraz wyznaczanie i zastosowanie wielozbiorów deskryptorów krawędzi zajmują marginalną część całkowitego czasu wykonania algorytmów.
— Operacje oznaczone jako *inne* zajmują dość dużą część całkowitego czasu wykonania algorytmów, co jest zauważalne szczególnie w przypadku algorytmu UFC. Na ten czas składają się m.in. operacje budowania grafów kandydujących, zbierania statystyk, a w przypadku algorytmu UFC także uwzględnianie optymalizacji związanej z własnością 4.4.

6.5. Analiza skuteczności poszczególnych optymalizacji algorytmu UGM

Eksperyment ma zadanie zbadania skuteczności optymalizacji zastosowanych w algorytmach UGM i UFC. Na potrzeby algorytmów zostały zaproponowane cztery techniki optymalizacyjne:
— optymalizacja z wykorzystaniem zbioru $\mathbb{GN}$,
— optymalizacja z wykorzystaniem zbioru $\mathbb{NKR}$ (dotyczy tylko algorytmu UGM),
— optymalizacja z wykorzystaniem maksymalnych częstych wielozbiorów deskryptorów krawędzi,
— optymalizacja z wykorzystaniem przerywania wyznaczania wsparcia.

Wykres 6.4 przedstawia czasy wykonania sześciu wersji algorytmów UFC i UGM dla różnych progów wsparcia. Wyróżniono następujące wersje algorytmów:
— UGM oraz UFC - algorytmy ze wszystkimi optymalizacjami,



Rysunek 6.4. Czasy działania algorytmów UGM i UFC z różnymi optymalizacjami w zbiorze grafów Chemical_340 z kolekcji NCI.

— UGM_0 oraz UFC_0 - algorytmy bez żadnych optymalizacji,

— UGM_GN oraz UFC_GN - algorytmy z optymalizacją opartą na zbiorze $\mathbb{GN}$,

— UGM_NKR - algorytm z optymalizacją opartą na zbiorze $\mathbb{NKR}$,

— UGM_MWDK oraz UFC_MWDK - algorytmy z optymalizacją opartą na maksymalnych częstych wielozbiorach deskryptorów krawędzi,

— UGM_PWW oraz UFC_PWW - algorytmy z optymalizacją opartą na przerywaniu wyznaczania wsparcia,

Z wykresów można odczytać następujące wnioski:

— Wszystkie optymalizacje dają zauważalny przyrost prędkości działania algorytmów UGM i UFC.

— Algorytmy z wszystkimi optymalizacjami są około rząd wielkości szybsze od algorytmów bez optymalizacji.

— Optymalizacja oparta na zbiorze grafów nieczęstych jest najbardziej skuteczna.

— Algorytm UGM jest bardziej podatny na optymalizacje niż algorytm UFC.

Dodatkowych danych dotyczących skuteczności poszczególnych optymalizacji dostarcza tabela 6.23. W tabeli zawarte są dane dotyczące liczby i czasów wykonania testów na izomorfizm z podgrafem wykonanych w celu wyznaczenia wsparcia kandydatów wraz z wyróżnieniem liczby i czasów wykonania testów, które zakończyły się wynikiem

pozytywnym (kolumny liczba$^+$ oraz czas$^+$). Im większy jest stosunek liczba$^+$/liczba tym skuteczniejsza optymalizacja. Obserwacja tabeli prowadzi do następujących wniosków:

— Wszystkie optymalizacje zmniejszają liczbę wykonanych testów na izomorfizm z podgrafem.

— Algorytm UGM bez optymalizacji jest mało efektywny, gdyż średnio tylko 18% wykonanych testów na izomorfizm z podgrafem kończy się wynikiem pozytywnym. Ponadto liczba wykonanych testów na izomorfizm z podgrafem jest o rząd wielkości większa niż w algorytmie UFC bez optymalizacji.

— Zastosowanie optymalizacji zmniejsza liczbę wykonanych testów na izomorfizm z podgrafem średnio dwudziestokrotnie w przypadku algorytmu UGM i średnio dwukrotnie w przypadku algorytmu UFC.

— Optymalizacja oparta na zbiorze grafów nieczęstych jest najbardziej skuteczna.

— Zarówno w algorytmie UGM i UFC, optymalizacje prowadzą do uzyskania podobnej liczby wykonanych testów na izomorfizm z podgrafem oraz stosunku liczba$^+$/liczba na poziomie około 75%.

Warto zauważyć, że optymalizacja oparta na zbiorze grafów nieczęstych jest najbardziej skuteczną z zaproponowanych, ale jednocześnie obsługa zbioru grafów nieczęstych jest jedną z najbardziej czasochłonnych operacji w algorytmie UFC, a zwłaszcza w algorytmie UGM. Tabele 6.24 oraz 6.25 przedstawiają podstawowe własności zbioru grafów nieczęstych uzyskanego w czasie wykonania algorytmów UGM i UFC na przykładowych danych. Tabele zawierają następujące dane:

— liczba grafów częstych - liczba wszystkich grafów częstych w danym zbiorze danych przy danym wsparciu,

— rozmiar zbioru GN - liczba grafów znajdujących się w zbiorze $\mathbb{GN}$ po zakończeniu algorytmu,

— liczba testów w zbiorze GN - liczba wykonanych testów na izomorfizm z podgrafem, związanych z obsługą zbioru $\mathbb{GN}$,

— liczba testów$^+$ w zbiorze GN - jak wyżej, ale dotyczy tylko testów, które zakończyły się wynikiem pozytywnym,

— liczba grafów$^+$ w zbiorze GN - liczba grafów ze zbioru $\mathbb{GN}$, które w czasie działanie algorytmu okazały się pografami kandydatów.

Z uzyskanych wyników można wyciągnąć następujące wnioski:

— W przypadku algorytmu UGM rozmiar zbioru $\mathbb{GN}$ jest zbliżony do liczby wszystkich grafów częstych.

— W przypadku algorytmu UFC rozmiar zbioru $\mathbb{GN}$ jest o rząd wielkości mniejszy od liczby wszystkich grafów częstych.

— Tylko kilka procent testów na izomorfizm z podgrafem wykonywanych w zbiorze $\mathbb{GN}$ kończy się wynikiem pozytywnym, czyli przyczynia się do optymalizacji algorytmów UGM i UFC.

— Około kilkadziesiąt procent grafów ze zbioru $\mathbb{GN}$ jest istotnych z punktu widzenia optymalizacji to znaczy okazuje się podgrafami grafów kandydujących.

	testy na izomorfizm z podgrafem					
	czas [s]	czas$^+$ [s]	czas$^+$/czas [%]	liczba	liczba$^+$	liczba$^+$/liczba [%]
UGM_0	377	106	28	19757430	3586105	18
UGM_PWW	81	40	49	5057789	1589601	31
UGM_MWDK	212	39	18	12170991	1694625	13
UGM_GN	23	17	74	968144	722299	74
UGM_NKR	169	24	14	9008562	992862	11
UGM	18	15	83	912196	687104	75
UFC_0	173	22	12	1427212	819834	57
UFC_PWW	33	15	45	977496	680090	69
UFC_MWDK	153	18	11	1054147	722982	68
UFC_GN	22	14	63	793693	615498	78
UFC	17	13	76	777195	603356	78

Tabela 6.23. Średnia liczba i czas wykonanych testów na izomorfizm podgrafu w różnych wersjach algorytmów UGM i UFC.

zbiór/ wsparcie	liczba grafów częstych	rozmiar zbioru GN	liczba testów w zbiorze GN	liczba testów$^+$ w zbiorze GN	liczba grafów$^+$ w zbiorze GN
mutag/50%	3966	1379	513489	7998	272
mutag/40%	4893	1416	606387	9139	315
mutag/30%	31758	13865	37635351	182048	4657
ptc_fm/10%	1548	755	58084	5552	249
ptc_fm/5%	11120	6360	1634520	61697	1668
ptc_fm/4%	20290	12649	5253084	122041	2895
nci_786/25%	895	1004	102722	8436	338
nci_786/20%	2106	2761	715887	22304	803

Tabela 6.24. Własności zbioru grafów nieczęstych utworzonego przez UGM.

zbiór/ wsparcie	liczba grafów częstych	rozmiar zbioru GN	liczba testów w zbiorze GN	liczba testów$^+$ w zbiorze GN	liczba grafów$^+$ w zbiorze GN
mutag/50%	3966	123	62125	1356	91
mutag/40%	4893	202	91400	1580	131
mutag/30%	31758	1486	3232512	23385	842
ptc_fm/10%	1548	256	5085	412	95
ptc_fm/5%	11120	1325	182099	3193	541
ptc_fm/4%	20290	2022	427963	5812	867
nci_786/25%	895	142	1559	156	42
nci_786/20%	2106	302	15095	384	90

Tabela 6.25. Własności zbioru grafów nieczęstych utworzonego przez UFC.

6.6. Analiza skuteczności poszczególnych heurystyk i optymalizacji algorytmu testu na izomorfizm z podgrafem

Algorytmy UGM i UFC wykorzystują zaproponowaną przez autora tej pracy implementację algorytmu testu na izomorfizm z podgrafem. Problem izomorfizmu z podgrafem został sprowadzony do problemu spełniania ograniczeń, którego rozwiązaniem zajmuje się algorytm z powrotami. Przeprowadzone eksperymenty miały za zadanie zbadać skuteczność różnych technik optymalizacyjnych zawartych w algorytmie z powrotami, takich jak: metody wybory zmiennej, spójność łukowa oraz zaproponowane przez autora tej pracy ograniczanie dziedziny za pomocą symetrii grafów oraz wykorzystanie kontekstu algorytmu UGM. Dodatkowym celem eksperymentu było porównanie algorytmu z algorytmem $VF2$ pochodzącym z biblioteki *vflib*. Wszystkie eksperymenty zostały przeprowadzone pod kątem przydatności w algorytmach UGM i UFC.

6.6.1. Wykorzystanie symetrii

Algorytm z powrotami wykorzystuje dwa rodzaje symetrii: symetrię zmiennych oraz symetrię wartości. Dla problemu izomorfizmu grafu G z podgrafem grafu G' symetrią zmiennych są automorfizmy grafu G, a symetrią wartości - automorfizmy grafu G'. Tabela 6.26 oraz wykres 6.5 przedstawiają łączny czas działania wszystkich testów na izomorfizm z podgrafem, które były wykonane w ramach algorytmu UGM. Dla celów informacyjnych przedstawiona jest także łączna liczba powrotów, które musiał wykonać algorytm z powrotami. Kolumna $czas^+$ oznacza czas wykonania testów zakończonych wynikiem pozytywnym, natomiast kolumna $czas$ oznacza czas wykonania wszystkich testów. Wersja algorytmu oznaczona jako *bez sym 1* różni się od wersji normalnej tym, że nie korzysta z symetrii

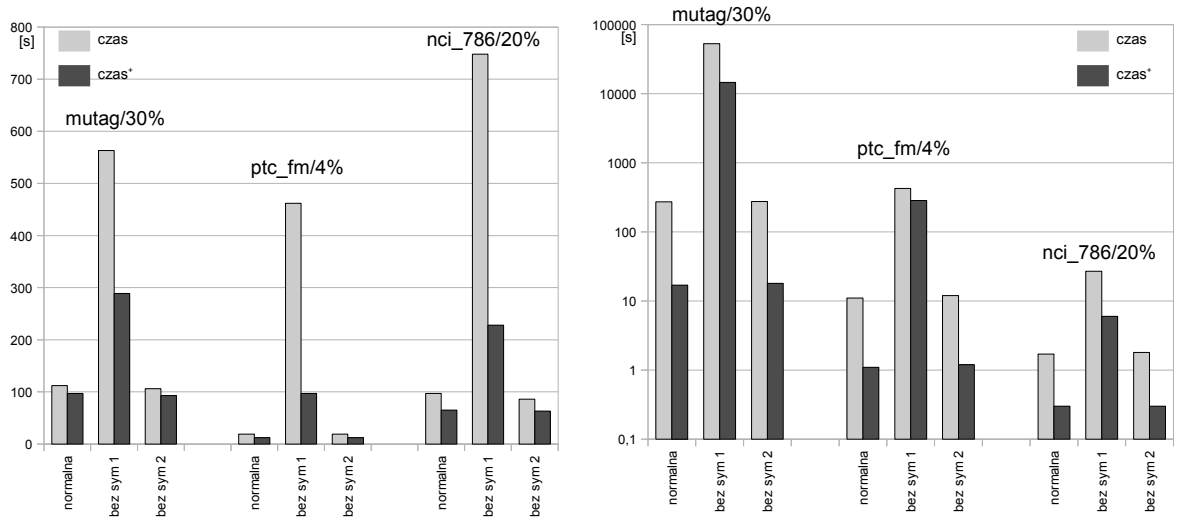
zmiennych, a wersja oznaczona jako *bez sym 2* różni się od wersji normalnej tym, że nie korzysta z symetrii wartości.

		testy na izomorfizm z podgrafem				
		w celu obliczania wsparcia kandydatów		w celu obsługi zbioru grafów nieczęstych		
zbiór/ wsparcie	wersja algorytmu	czas [s]	czas$^+$ [s]	czas [s]	czas$^+$ [s]	liczba powrotów
mutag/50%	normalna	12.9	11.7	2.7	0.2	11896177
	bez sym 1	24.9	13.9	47.6	6.6	132865206
	bez sym 2	12.7	11.4	2.8	0.2	12139556
mutag/40%	normalna	16.5	13.75	3.2	0.2	14698151
	bez sym 1	32.9	15.54	46.2	3.6	139769040
	bez sym 2	15.6	13.34	3.22	0.18	14904096
mutag/30%	normalna	112.1	97.1	272.6	16.7	261865481
	bez sym 1	563.0	289.0	53015	14569	103171439225
	bez sym 2	106.8	93.0	276.3	17.9	278973111
ptc_fm/10%	normalna	2.3	1.5	0.18	0.04	1642036
	bez sym 1	18.6	4.3	2.6	0.36	38681460
	bez sym 2	2.37	1.56	0.19	0.038	1664053
ptc_fm/5%	normalna	11.4	6.99	3.99	0.63	10067775
	bez sym 1	109.6	19.7	407.66	287.9	948423601
	bez sym 2	11.7	7.2	4.25	0.68	10517385
ptc_fm/4%	normalna	19.18	11.9	10.85	1.1	18716182
	bez sym 1	462.5	97.4	424.5	284.9	1664561380
	bez sym 2	19.24	11.9	11.63	1.18	19708233
nci_786/25%	normalna	35.7	26.8	0.33	0.07	18734960
	bez sym 1	188.2	74.2	4.0	0.98	320479586
	bez sym 2	34.5	25.7	0.31	0.06	18835538
nci_786/20%	normalna	87.49	64.86	1.69	0.29	46279141
	bez sym 1	747.5	228.3	26.6	5.9	1329932216
	bez sym 2	86.0	63.1	1.71	0.29	46964528

Tabela 6.26. Wpływ uwzględniania symetrii w algorytmie izomorfizmu z podgrafem na jego czas działania.

Wnioski:

— Wykorzystanie symetrii zmiennych znacząco poprawia efektywność algorytmu. Algorytm nie korzystający z tej optymalizacji był od dwóch do dziesięcu razy wolniejszy w fazie wyznaczania wsparcia kandydatów oraz od dziesięciu do stu razy wolniejszy w fazie obsługi zbioru $\mathbb{GN}$. Wykorzystanie symetrii zmiennych zmniejsza liczbę powrotów nawet do trzech rzędów wielkości. Dużo większa skuteczność wykorzystania symetrii w fazie obsługi zbioru $\mathbb{GN}$ wynika z faktu, że do zbióru $\mathbb{GN}$ należą przede wszystkim grafy niespójne o kilku składowych, a zatem o licznych symetriach. Im więcej symetrii w grafie tym skuteczniejsza jest optymalizacja.



Rysunek 6.5. Wpływ uwzględniania symetrii na czas działania algorytmu izmorfizmu z podgrafem. Wykres po lewej stronie dotyczy wyznaczania wsparcia kandydatów. Wykres po prawej stronie dotyczy obsługi zbioru $\mathbb{GN}$.

— Wykorzystanie symetrii wartości ma niewielki wpływ na efektywność algorytmu. Liczba powrotów w algorytmie bez symetrii wartości jest tylko o ułamek procenta mniejsza niż w algorytmie korzystającym z symetrii wartości.

6.6.2. Kolejność wyboru zmiennej

Przebadano trzy strategie wyboru zmiennej:

— $wz1$ - wybór pierwszej nieprzyporządkowanej zmiennej,

— $wz2$ - wybór najbardziej ograniczającej nieprzyporządkowanej zmiennej, to znaczy wierzchołka o największym stopniu (przy czym stopnie wierzchołków nie zmieniają się w trakcie działania algorytmu z powrotami),

— $wz3$ - wybór najbardziej ograniczonej nieprzyporządkowanej zmiennej, to znaczy zmiennej o najmniej licznej dziedzinie w danym momencie (przy czym dziedziny zmiennych zmieniają się w trakcie działania algorytmu z powrotami).

Tabela 6.27 oraz wykres 6.6 przedstawiają czasy działania algorytmu dla różnych metod wyboru zmiennej. Strategie $wz3$ i $wz2$ powodują nawet dwukrotny przyrost prędkości algorytmu nawet w stosunku do naiwnej strategii $wz1$. Strategia $wz3$ daje rezultaty o kilkanaście procent lepsze niż strategia $wz2$. Okazuje się, że strategia wyboru zmiennej ma zauważalny wpływ na efektywność algorytmu, aczkolwiek nie tak duży, jak by wynikało

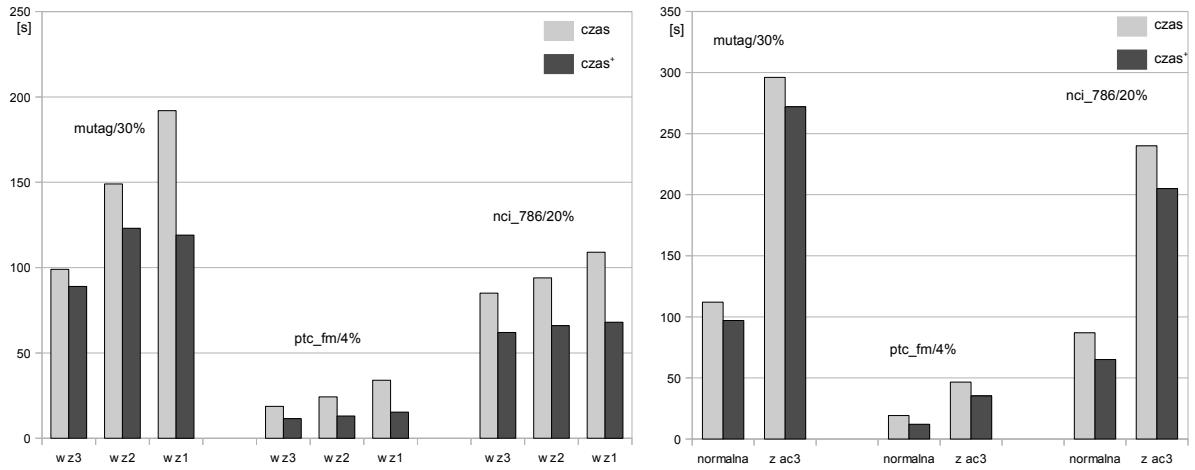
z literatury. Ten sam eksperyment przeprowadzono ponownie dla algorytmu bez optymalizacji wykorzystującej symetrie. Wyniki znajdują się w tabeli 6.28. W tym przypadku strategia $wz1$ okazuje się nawet do 20 razy wolniejsza od pozostałych strategii. Zaskakująca jest obserwacja, że strategia wyboru zmiennej nie ma prawie zupełnie wpływu na czas wykonywania testów na izomorfizm z podgrafem na potrzeby obsługi zbioru $\mathbb{GN}$. Można to tłumaczyć faktem, że do zbióru $\mathbb{GN}$ należą przede wszystkim grafy niespójne o kilku składowych, a wybór zmiennej pochodzącej z jednej składowej nie ma prawie wpływu na dziedziny zmiennych pochodzących z innych składowych.

		testy na izomorfizm z podgrafem				
		w celu obliczania wsparcia kandydatów		w celu obsługi zbioru grafów nieczęstych		
zbiór/ wsparcie	metoda wyboru zmiennej	czas [s]	czas$^+$ [s]	czas [s]	czas$^+$ [s]	liczba powrotów
mutag/50%	wz3	11.9	10.8	2.7	0.2	11399572
	wz2	13.5	12.1	2.9	0.2	13804234
	wz1	14.8	13.3	2.8	0.2	19119073
mutag/40%	wz3	14.97	12.9	3.0	0.18	13633140
	wz2	17.1	14.7	3.3	0.18	17010606
	wz1	19.1	16.1	3.5	0.16	25033120
mutag/30%	wz3	99.1	89.1	260.5	15.6	252068790
	wz2	148.6	122.7	283.6	16.3	335312933
	wz1	191.8	118.8	299.9	15.4	511398951
ptc_fm/10%	wz3	2.4	1.55	0.18	0.06	1669270
	wz2	2.49	1.59	0.19	0.05	2032994
	wz1	2.57	1.58	0.16	0.04	2300451
ptc_fm/5%	wz3	11.6	7.2	3.98	0.59	10116808
	wz2	14.57	8.1	4.57	0.65	14567822
	wz1	17.27	8.17	3.9	0.6	19693838
ptc_fm/4%	wz3	18.7	11.46	10.98	1.52	18583192
	wz2	24.35	12.99	11.3	1.3	26884581
	wz1	34.03	15.33	11.16	1.1	42671550
nci_786/25%	wz3	35.36	26.17	0.29	0.06	19066707
	wz2	37.6	27.5	0.3	0.08	22439979
	wz1	42.6	27.5	0.29	0.06	31326677
nci_786/20%	wz3	85.2	62.05	1.59	0.28	46613383
	wz2	93.8	66.2	1.63	0.24	57034348
	wz1	109.1	67.7	1.7	0.28	84725693

Tabela 6.27. Czas działania algorytmu izomormizmu z podgrafem dla różnych metod wyboru zmiennej.

6.6.3. Spójność łukowa

Tabela 6.29 oraz wykres 6.6 przedstawiają wpływ optymalizacji *ac-3* na czas wykonania algorytmu izomorfizmu z podgrafem. Okazuje się, że algorytm w wersji z ac3 wykonuje się do kilku razy wolniej niż algorytm bez *ac-3*. Liczba powrotów w przypadku algortmu z *ac-3* jest tylko o ułamek procenta mniejsza niż w przypadku algorytmu bez *ac-3* i czas



Rysunek 6.6. Czas działania algorytmu izomorfizmu z podgrafem w zależności od metody wyboru zmiennej (po lewej) oraz z wykorzystaniem ac-3 (po prawej).

		testy na izomorfizm z podgrafem				
		w celu obliczania wsparcia kandydatów		w celu obsługi zbioru grafów nieczęstych		
zbiór/ wsparcie	metoda wyboru zmiennej	czas [s]	$czas^+$ [s]	czas [s]	$czas^+$ [s]	liczba powrotów
mutag/50%	wz3	23.2	12.8	66.7	10.2	194295665
	wz2	24.5	14.3	66.1	9.7	197738282
	wz1	25.5	14.6	64.8	9.9	205125524
mutag/40%	wz3	29.8	14.3	61.4	5.5	190990709
	wz2	33.8	16.7	63.5	5.5	203662265
	wz1	37.8	17.2	62.7	5.2	229088552
ptc_fm/10%	wz3	16.2	3.8	3.8	0.77	44286569
	wz2	17.3	3.9	3.8	0.81	47018804
	wz1	27	5.3	3.6	0.75	74203308
ptc_fm/5%	wz3	97	18	968	592	2321302509
	wz2	110	20	963	590	2350568260
	wz1	1600	72.5	977	601	5670157228
ptc_fm/4%	wz3	437	91	923	543	3243669583
	wz2	467	95	993	584	3338492754
	wz1	8496	4273	936	552	19952683469
nci_786/25%	wz3	172	66	8.0	1.8	366256651
	wz2	175	66	10.7	1.8	369711485
	wz1	533	86	10.3	1.8	1183982984
nci_786/20%	wz3	754	202	85	10	1721578267
	wz2	778	204	57	9	1699536149
	wz1	5450	844	80	9.8	12577908486

Tabela 6.28. Czas działania algorytmu izomormizmu z podgrafem w wersji bez symetrii dla różnych metod wyboru zmiennej.

zaoszczędzony z tego powodu nie niweluje narzutu wprowadzonego przez *ac-3*. Niska skuteczność *ac-3* w ograniczaniu dziedziny zmiennych wynika z dwóch faktów. Po pierwsze dziedzina zmiennych jest silnie ograniczona przez warunki zgodności etykiet, sąsiadujących krawędzi i stopni sąsiadujących krawędzi. Po drugie, większość ograniczeń binarnych, na których opiera się *ac-3* jest spełniona, gdyż w algorytmie UGM wykonywany jest test na izomorfizm grafu kandydującego G z podgrafem pewnego grafu G', a graf kandydujący jest rozszerzeniem grafu, który jest izomorficzny z podgrafem grafu G'. W związku z tym jedynym ograniczeniem binarnym, które może być niespełnione, jest ograniczenie pochodzące od nowo dodanej krawędzi w grafie.

		testy na izomorfizm z podgrafem				
		w celu obliczania wsparcia kandydatów		w celu obsługi zbioru grafów nieczęstych		
zbiór/ wsparcie	wersja algorytmu	czas [s]	czas$^+$ [s]	czas [s]	czas$^+$ [s]	liczba powrotów
mutag/50%	normalna	12.9	11.7	2.7	0.2	11896177
	z ac3	34.6	34.6	4.34	0.28	11771943
mutag/40%	normalna	16.5	13.75	3.2	0.2	14698151
	z ac3	46.17	41.9	5.2	0.32	14530757
mutag/30%	normalna	112.1	97.1	272.6	16.7	261865481
	z ac3	296.4	271.9	523.4	24.1	254856370
ptc_fm/10%	normalna	2.3	1.5	0.18	0.04	1642036
	z ac3	5.78	4.7	0.26	0.07	1625715
ptc_fm/5%	normalna	11.4	6.99	3.99	0.63	10067775
	z ac3	29.0	22.1	6.3	1.1	9825516
ptc_fm/4%	normalna	19.18	11.9	10.85	1.1	18716182
	z ac3	46.53	35.29	17.1	1.99	18317763
nci_786/25%	normalna	35.7	26.8	0.33	0.07	18734960
	z ac3	96.53	83.07	0.47	0.13	18513751
nci_786/20%	normalna	87.49	64.86	1.69	0.29	46279141
	z ac3	240.3	204.8	2.53	0.51	45524974

Tabela 6.29. Wpływ ac-3 na czas działania algorytmu izomormizmu z podgrafem.

6.6.4. Wykorzystanie kontekstu algorytmu UGM

Optymalizacje z wykorzystaniem kontekstu algorytmu UGM można wykorzystać w przypadku testów na izomorfizm grafu kandydującego z podgrafem grafu z wejściowej bazy danych, gdyż wiadomo, że rodzic grafu kandydującego spełnił ten test. W pracy zapropnowano dwie techniki wykorzystania poprzednio wykonywanych testów na izomorfizm z podgrafem: przenoszenie poprawnego rozwiązania (oznaczane jako ppr) oraz przenoszenie niepoprawnych przyporządkowań (oznaczane przez pnp). Obie techniki dodają informację do każdej pary grafów (G, G'), dla której test na izomorfizm podgrafu G w G' zakończył się sukcesem

oraz G' jest grafem z wejściowej bazy grafów. Przenoszenie poprawnego rozwiązania jest podobne do metody przechowywania zanurzeń, z tą różnicą, że przechowywane jest tylko jedno zanurzenie. Przenoszenie niepoprawnych rozwiązań jest nową propozycją, która polega wykluczeniu pewnych wartości z dziedziny zmiennych grafu kandydującego na podstawie wyników testu jego rodzica. Tabela 6.30 przedstawia efekty użycia zaproponowanych optymalizacji. Optymalizacja pnp występuje w dwóch wersjach: $pnp5$ i $pnp50$, gdzie 5 i 50 są wartością parametru $min\_wrong\_backtracks$ (patrz rozdział 5.3.10), który oznacza minimalną liczbę powrotów którą musi wykonać algorytm z powrotami, aby niepoprawne przyporządkowane było uznane za nietrywialne.

		testy na izomorfizm z podgrafem				
		w celu obliczania wsparcia kandydatów		w celu obsługi zbioru grafów nieczęstych		
zbiór/ wsparcie	wersja algorytmu	czas [s]	czas$^+$ [s]	czas [s]	czas$^+$ [s]	liczba powrotów
mutag/50%	normalna	12.9	11.7	2.7	0.2	11896177
	ppr	9.01	7.20	x	x	3698446
	pnp 50	14.17	12.6	x	x	11824862
	pnp 5	14.1	12.61	x	x	11717919
mutag/40%	normalna	16.5	13.75	3.2	0.2	14698151
	ppr	12.5	9.04	x	x	5624427
	pnp 50	18.51	15.14	x	x	14605305
	pnp 5	18.80	15.47	x	x	14455761
mutag/30%	normalna	112.1	97.1	272.6	16.7	261865481
	ppr	62.03	49.1	x	x	43205239
	pnp 50	119.9	101.44	x	x	260154805
	pnp 5	Brak pamięci				
ptc_fm/10%	normalna	2.3	1.5	0.18	0.04	1642036
	ppr	2.49	1.42	x	x	1385977
	pnp 50	2.61	1.77	x	x	1641532
	pnp 5	2.66	1.83	x	x	1635075
ptc_fm/5%	normalna	11.4	6.99	3.99	0.63	10067775
	ppr	12.57	7.16	x	x	7411770
	pnp 50	12.6	7.97	x	x	10311502
	pnp 5	14.1	8.97	x	x	10172484
ptc_fm/4%	normalna	19.18	11.9	10.85	1.1	18716182
	ppr	18.9	10.8	x	x	13261022
	pnp 50	24.5	13.7	x	x	18639504
	pnp 5	26.7	16.2	x	x	18585988
nci_786/25%	normalna	35.7	26.8	0.33	0.07	18734960
	ppr	32.8	23.2	x	x	11629112
	pnp 50	40.3	30.1	x	x	18712098
	pnp 5	40.9	31.6	x	x	18576109
nci_786/20%	normalna	87.49	64.86	1.69	0.29	46279141
	ppr	72.6	49.8	x	x	27370597
	pnp 50	100.2	75.0	x	x	46110143
	pnp 5	107.09	81.3	x	x	45765273

Tabela 6.30. Czas działania algorytmu izomormizmu z podgrafem w wersjach z przenoszeniem rozwiązania i niepoprawnych przyporządkowań.

Wnioski:

— Optymalizacja ppr jest skuteczna - liczba powrotów ulega nawet wielokrotnemu zmniejszeniu, a czas wykonania zmniejsza się nawet o kilkanaście procent.
— Optymalizacja pnp jest nieskuteczna - liczba powrotów zmniejsza się tylko o ułamek procenta nie rekompensując narzutów wprowadzonych przez dodatkowe operacje; czas wykonania zwiększa się nawet o kilkanaście procent.
— Obie optymalizacje wymagają dodatkowej pamięci, a największe zapotrzebowanie na pamięć ma optymalizacja pnp z niskim parametrem $min\_wrong\_backtracks$, co w jednym przypadku spowodowało błąd wyczerpania pamięci.

6.6.5. Porównanie z algorytmem vf2

W eksperymentach skorzystano z implementacji algorytmu $VF2$ pochodzącej z biblioteki *vflib* napisanej w języku *C++*. Ze względu na różnice technologiczne niemożliwe było efektywne włączenie biblioteki $vflib$ do platformy $ParMol$. Z tego względu obiektywne porównanie algorytmów zostało wykonane poza platformą $ParMol$, ale dla danych pochodzących z wyników algorytmu UGM. Dla danego wykonania algorytmu UGM zostały zebrane wszystkie pary grafów (G, G'), dla których wykonywany był test na izomorfizm grafu G z podgrafem grafu G'. Pary te zostały następnie zapisane do pliku zewnętrzego wczytywanego potem przez oddzielny program, którego celem było wykonanie testów na izomorfizm z podgrafem za pomocą algorytmu $VF2$. Zmierzono tylko i wyłącznie czas wykonania testów na izomorfizm z podgrafem - czas wczytywania pliku i budowy grafów został pominięty. Tabela 6.31 przedstawia wyniki tego eksperymentu dla kilku zbiorów danych. W tabeli znajdują się wyniki działania trzech algorytmów badania izomorfizmu z podgrafem:

— normalna - wersja znajdująca się w algorytmach UGM i UFC,
— niezależna - jak wyżej, ale wykonana poza algorytmem UGM, zatem wszystkie pomocnicze struktury, takie jak informacje o symetriach, wielozbiorach deskryptorów oraz klasach wierzchołków są wyznaczane od początku dla każdego grafu,
— vf2 - wersja z biblioteki *vlflib*.

Przeprowadzono także eksperyment, w którym jako algorytm izomorfizmu z podgrafem posłużyła implementacja zawarta w platformie $ParMol$. Dla żadnego z zawartych w tabeli przypadków ten eksperyment nie zakończył się w czasie poniżej kilku godzin.

Wnioski:

		testy na izomorfizm z podgrafem	
		w celu obliczania wsparcia kandydatów	w celu obsługi zbioru grafów nieczęstych
zbiór/ wsparcie	implementacja	czas [s]	czas [s]
mutag/50%	normalna	12.9	11.7
	niezależna	86	44
	vf2	372	48
mutag/40%	normalna	16.5	13.75
	niezależna	110	51
	vf2	488	83
ptc_fm/10%	normalna	2.3	1.5
	niezależna	29	3.4
	vf2	64902	1.6
ptc_fm/5%	normalna	11.4	6.99
	niezależna	140	120
	vf2		
nci_786/30%	normalna	15.7	0.07
	niezależna	259	1.5
	vf2	213642	1.1
nci_786/25%	normalna	35.7	26.8
	niezależna	531	6.6
	vf2		8.3

Tabela 6.31. Czasy działania algorytmu izomorfizmu z podgrafem w trzech implementacjach: normalna (działająca wewnątrz algorymu UGM i UFC), niezależna (do wykorzystania poza algorytmami UGM i UFC), VF2 (pochodząca z biblioteki vflib)

— Algorytm $VF2$ jest nawet do kilku rzędów wielkości wolniejszy od zaproponowanego algorytmu, zarówno w wersji normalnej, jak i niezależnej.

— Algorytm w wersji niezależniej jest o rząd wielkości wolniejszy od algorytmu w wersji normalnej.

7. Podsumowanie i dalsze kierunki badań

Praca jest poświęcona zagadnieniu odkrywania częstych grafów spójnych i niespójnych. W pracy zaproponowano dwa nowe algorytmy UGM i UFC, które odkrywają wszystkie spójne i niespójne grafy częste. Oba algorytmy wykonują testy na izomorfizm z podgrafem względem wejściowej bazy danych w celu wyznaczenia wsparcia grafów, w odróżnieniu od typowo wykorzystywanych w tym celu zanurzeń. Oba algorytmy zostały zaimplementowane w ramach platformy $ParMol$ [55] i porównane z istniejącymi w platformie czterema algorytmami służącymi do odkrywania częstych grafów spójnych ($Gaston$, $MoFa$, $gSpan$, $FFSM$). Do platformy zostały także włączona opisana w [77] modyfikacja algorytmu $gSpan$ (nazywana w niniejszej rozprawie $gSpanUnconnected$), pozwalająca na odkrywanie częstych grafów niespójnych, a także zaczerpnięta z [41] metoda (nazywana w ninijeszej rozprawie $UgmOnVirtual$) odkrywania częstych grafów niespójnych za pomocą algorytmów odkrywania częstych grafów spójnych poprzez uzupełnianie grafów z wejściowej bazy danych o brakujące krawędzie. W celu przyspieszenia algorytmów UGM i UFC zaproponowano cztery potencjalnie uniwersalne techniki optymalizacyjne: metodę wykorzystującą *maksymalne częste wielozbiory deskryptorów krawędzi*, metodę wykorzystującą *zbiór grafów nieczęstych*, metodę wykorzystującą *zbiór nieczęstych konstruktorów rozszerzeń* oraz metodę pozwalającą na *przerywanie wyznaczania wsparcia* grafu, w momencie, gdy z bieżącej wartości wsparcia można wywnioskować, że będzie ono poniżej minimalnego progu wsparcia. Eksperymenty pokazują, że wszystkie te techniki dają znaczący przyrost wydajności algorytmów UGM i UFC i sprawiają, że algorytmy uzyskują czasy wykonania o rząd wielkości lepsze niż algorytm $gSpanUnconnected$. Eksperymenty pozwalają wywnioskować, że optymalizacją przynoszącą najwyższy wzrost wydajność jest technika wykorzystująca zbiór grafów nieczęstych. Z drugiej jednak strony, obsługa zbioru grafów nieczęstych staje się najbardziej czasochłonną operacją w algorytmie w przypadku niskich wartości minimalnego wsparcia. W pracy zaproponowano ograniczanie maksymalnej wielkości zbioru grafów nieczęstych za pomocą parametru wejściowego algorytmu, a metody automatycznego ustalania tej wielkości oraz propozycje

wyspecjalizowanych struktur przechowujących zbiór grafów nieczęstych będą stanowić przedmiot dalszych badań. Na potrzeby algorytmów UFC i UGM zaimplementowano algorytm badania izomorfizmu grafu z podgrafem, który opiera się na rozwiązywaniu problemu spełniania ograniczeń. W algorytmie zawarto nową metodę ograniczania dziedziny zmiennych na podstawie występujących w grafach symetrii. Opracowany algorytm badania izomorfizmu z podgrafem został eksperymentalnie porównany ze znanym algorytmem $VF2$. Eksperymenty pokazują, że propozycja autora jest o kilka rzędów wielkości szybsza od algorytmu z platformy $ParMol$ oraz kilka razy szybsza od algorytmu $VF2$ w zakresie stosowania w algorytmach UGM i UFC. Poza pojęciem grafu częstego w dziedzinie odkrywania wiedzy z grafów w literaturze zaproponowano też pojęcia zamkniętego grafu częstego oraz maksymalnego grafu częstego, które są analogiczne do pojęć zamkniętego zbioru częstego i maksymalnego zbioru częstego. Istnieją algorytmy odkrywające zamknięte grafy częste (np. $CloseGraph$ [77]) oraz maksymalne grafy częste (np. SPIN [38]), są to jednak algorytmy odkrywające tylko grafy spójne. Zaproponowanie algorytmu odkrywającego częste lub maksymalne grafy częste z uwzględnianiem niespójności, na przykład przez modyfikacje algorytmu UGM, będzie kierunkiem dalszych badań.

Bibliografia

[1] R. Agrawal, T. Imieliński, and A. Swami. Mining association rules between sets of items in large databases. In *SIGMOD '93: Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, pages 207–216, New York, NY, USA, 1993. ACM.

[2] R. Agrawal and R. Srikant. Mining sequential patterns. In *ICDE '95: Proceedings of the Eleventh International Conference on Data Engineering*, pages 3–14, Washington, DC, USA, 1995. IEEE Computer Society.

[3] Tatsuya Asai, Hiroki Arimura, Takeaki Uno, and Shin ichi Nakano. Discovering frequent substructures in large unordered trees. In *Discovery Science*, volume 2843 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2003.

[4] L. Babai and Eugene M. Luks. Canonical labeling of graphs. In *STOC '83: Proceedings of the fifteenth annual ACM symposium on Theory of computing*, pages 171–183, New York, NY, USA, 1983. ACM.

[5] B. Benhamou. Study of symmetry in constraint satisfaction problems. In *In Proceedings of CP-94*, pages 246–254, 1994.

[6] B Benhamou and M. R. Saïdi. Local symmetry breaking during search in CSPs. In *Principles and Practice of Constraint Programming – CP 2007*, volume 4741 of *Lecture Notes in Computer Science*, pages 195–209. Springer Berlin / Heidelberg, 2007.

[7] C. Bessière. Arc-consistency and arc-consistency again. *Artif. Intell.*, 65(1):179–190, 1994.

[8] C. Bessière, E.C. Freuder, and J.C. Régin. Using constraint metaknowledge to reduce arc consistency computation. *Artif. Intell.*, 107(1):125–148, 1999.

[9] C. Borgelt. Canonical forms for frequent graph mining. In *Advances in Data Analysis*, Studies in Classification, Data Analysis, and Knowledge Organization, pages 337–349. Springer Berlin Heidelberg, 2007.

[10] C. Borgelt and M. Fiedler. Graph mining: Repository vs. canonical form. In *Data Analysis, Machine Learning and Applications*, Studies in Classification, Data Analysis, and Knowledge Organization, pages 229–236. Springer Berlin Heidelberg, 2008.

[11] C. Borgelt and T. Meinl. Full perfect extension pruning for frequent graph mining. In *ICDMW '06: Proceedings of the Sixth IEEE International Conference on Data Mining - Workshops*, pages 263–268, Washington, DC, USA, 2006. IEEE Computer Society.

[12] C.B. Borgelt, T. Meinl, and M. Berthold. Moss: a program for molecular substructure mining. In *OSDM '05: Proceedings of the 1st international workshop on open source data mining*, pages 6–15, New York, NY, USA, 2005. ACM.

[13] Michael R. Borgelt, C.B. Mining molecular fragments: Finding relevant substructures of molecules. In *ICDM '02: Proceedings of the 2002 IEEE International Conference on Data Mining*, page 51, Washington, DC, USA, 2002. IEEE Computer Society.

[14] V. Chaoji, M. Al Hasan, S. Salem, and M.J. Zaki. An integrated, generic approach to pattern mining: data mining template library. *Data Mining and Knowledge Discovery*, 17(3):457–495, December 2008.

[15] C.J. Colbourn. On testing isomorphism of permutation graphs. *Networks*, 11(2):13–21, 1981.

[16] D.J. Cook and L.B. Holder. Graph-based data mining. *IEEE Intelligent Systems*, 15(2):32–41, 2000.

[17] D.J. Cook and L.B. Holder. *Mining Graph Data*. John Wiley & Sons, 2006.

[18] L.P. Cordella, P. Foggia, C. Sansone, and M. Vento. An improved algorithm for matching large graphs. In *In: 3rd IAPR-TC15 Workshop on Graph-based Representations in Pattern Recognition, Cuen*, pages 149–159, 2001.

[19] L.P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(10):1367–1372, 2004.

[20] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*. The MIT Press and McGraw-Hill Book Company, 2001.

[21] D.G. Corneil and C.C. Gotlieb. An efficient algorithm for graph isomorphism. *J. ACM*, 17(1):51–64, 1970.

[22] R. Czerwinski. A polynomial time algorithm for graph isomorphism. *CoRR*, abs/0711.2010, Nov 2007.

[23] R. Dechter and I Meiri. Experimental evaluation of preprocessing algorithms for constraint satisfaction problems. *Artif. Intell.*, 68(2):211–241, 1994.

[24] L. Dehaspe, H. Toivonen, S. Dzeroski, and N. Lavrač. Discovery of frequent datalog patterns. *Data Min. Knowl. Discov.*, 3(1):7–36, 1999.

[25] I. Fischer and T. Meinl. Graph based molecular data mining - an overview. In *In Wil Thissen, Peter Wieringa, Maja Pantic, and Marcel Ludema, editors, IEEE SMC 2004 Conference Proceedings*, pages 4578–4582, 2004.

[26] P. Foggia, C. Sansone, and M. Vento. A performance comparison of five algorithms for graph isomorphism. *Proc. of the 3rd IAPR TC-15 Workshop on Graph-based Representations in Pattern Recognition*, pages 188–199, 2001.

[27] S. Fortin. The graph isomorphism problem. Technical report, Dept. of Computing Science, Univ. Alberta, Edmonton, Alberta, Canada, 1996.

[28] E.C. Freuder. A sufficient condition for backtrack-free search. *J. ACM*, 29(1):24–32, 1982.

[29] D.H. Frost. *Algorithms and heuristics for constraint satisfaction problems*. PhD thesis, University of California, Irvine, 1997. Chair-Rina Dechter.

[30] M.R. Garey and D.S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[31] P. Gawrysiak. *Automatyczna kategoryzacja dokumentów*. PhD thesis, Politechnika Warszawska, Wydział Elektroniki i Technik Informacyjnych, 2001.

[32] P. Gawrysiak, H. Rybinski, L. Skonieczny, and P. Wiech. Ami-sme: An exploratory approach to knowledge retrieval for sme's. *Autonomic and Autonomous Systems, 2007. ICAS07. Third International Conference on Autonomic and Autonomous Systems*, pages 65–65, June 2007.

[33] B. Goethals and M.J. Zaki, editors. *Proceedings of the ICDM 2003 Workshop on Frequent Itemset Mining Implementations (FIMI'03)*, volume 90 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2003.

[34] R.M. Haralick and G.L. Elliott. Increasing tree search efficiency for constrain satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.

[35] V. Hentenryck, Y. Deville, and C.M. Teng. A generic arc-consistency algorithm and its specializations. *Artif. Intell.*, 57(2-3):291–321, 1992.

[36] J. E. Hopcroft and J. K. Wong. Linear time algorithm for isomorphism of planar graphs (preliminary report). In *STOC '74: Proceedings of the sixth annual ACM symposium on Theory of computing*, pages 172–184, New York, NY, USA, 1974. ACM.

[37] J. Huan, W. Wang, and J. Prins. Efficient mining of frequent subgraphs in the presence of isomorphism. In *Data Mining, 2003. ICDM 2003. Third IEEE International Conference on*, pages 549–552, Nov. 2003.

[38] J. Huan, W. Wang, J. Prins, and J. Yang. Spin: mining maximal frequent subgraphs from graph databases. In *KDD '04: Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 581–586, New York, NY, USA, 2004. ACM.

[39] A. Inokuchi, T. Washio, and H. Motoda. An apriori-based algorithm for mining frequent substructures from graph data. In *Principles of Data Mining and Knowledge Discovery*, pages 13–23, 2000.

[40] A. Inokuchi, T. Washio, and H. Motoda. Complete mining of frequent patterns from graphs: Mining graph data. *Machine Learning*, 50(3):321–354, March 2003.

[41] A. Inokuchi, T. Washio, K Nishimura, and H. Motoda. A fast algorithm for mining frequent connected subgraphs. *Research Report RT0448*, 2002.

[42] D.S. Johnson. The np-completeness column. *ACM Trans. Algorithms*, 1(1):160–176, 2005.

[43] J. Kok and S. Nijssen. Faster association rules for multiple relations. In *IJCAI'01: Proceedings of the 17th international joint conference on Artificial intelligence*, pages 891–896, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[44] S. Kramer, L. De Raedt, and C. Helma. Molecular feature mining in hiv data. In *KDD '01: Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 136–143, New York, NY, USA, 2001. ACM.

[45] M. Kryszkiewicz and L. Skonieczny. Faster clustering with DBSCAN. In *Intelligent Information Processing and Web Mining*, volume 2005 of *Advances in Soft Computing*, pages 605–614. Springer Berlin / Heidelberg, 2005.

[46] M. Kryszkiewicz and L Skonieczny. Hierarchical document clustering using frequent closed sets. In *Intelligent Information Processing and Web Mining*, volume 35/2006 of *Advances in Soft Computing*, pages 489–498. Springer Berlin / Heidelberg, 2006.

[47] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *ICDM '01: Proceedings of the 2001 IEEE International Conference on Data Mining*, pages 313–320, Washington, DC, USA, 2001. IEEE Computer Society.

[48] M. Kuramochi and G. Karypis. An efficient algorithm for discovering frequent subgraphs, 2004.

[49] J. Larrosa and G. Valiente. Constraint satisfaction algorithms for graph pattern matching. *Mathematical. Structures in Comp. Sci.*, 12(4):403–422, 2002.

[50] G.S. Lueker and K.S. Booth. A linear time algorithm for deciding interval graph isomorphism. *J. ACM*, 26(2):183–195, 1979.

[51] E.M. Luks. Isomorphism of graphs of bounded valence can be tested in polynomial time. In *SFCS '80: Proceedings of the 21st Annual Symposium on Foundations of Computer Science*, pages 42–49, Washington, DC, USA, 1980. IEEE Computer Society.

[52] A.K. Mackworth. Consistency in networks of relations. Technical report, University of British Columbia, Vancouver, BC, Canada, Canada, 1975.

[53] J.J. McGregor. Relational consistency algorithms and their application in finding subgraph and graph isomorphisms. *Information Sciences*, page 9(3):229–250, 1979.

[54] B.D. McKay. Practical graph isomorphism. *10th. Manitoba Conference on Numerical Mathematics and Computing, Congressus Numerantium*, pages 45–87, 1980.

[55] T. Meinl, M. Wörlein, O. Urzova, I. Fischer, and M. Philippsen. The parmol package for frequent subgraph mining. *ECEASST*, 1, 2006.

[56] B.T. Messmer and H. Bunke. Subgraph isomorphism in polynomial time. Technical report, Institut fur Informatik und angew andte Mathematik, University of Bern, Neubruckstr. 10, Bern, Switzerland, 1995.

[57] R. Mohr and T.C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.

[58] M. Morzy and Z. Królikowski. Metody indeksowania atrybutów zawierających zbiory. *ProDialog: A Scientific and Educational Journal of the Polish Information Processing Society*, 3(15):87–106, 2003.

[59] S. Nijssen and J.N. Kok. Efficient discovery of frequent unordered trees. In *In First International Workshop on Mining Graphs, Trees and Sequences*, pages 55–64, 2003.

[60] S. Nijssen and J.N. Kok. A quickstart in frequent structure mining can make a difference. In *KDD '04: Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 647–652, New York, NY, USA, 2004. ACM.

[61] M. Perlin. Arc consistency for factorable relations. *Artif. Intell.*, 53(2-3):329–342, 1992.

[62] J.F. Puget. On the satisfiability of symmetrical constrained satisfaction problems. In *ISMIS '93: Proceedings of the 7th International Symposium on Methodologies for Intelligent Systems*, pages 350–361, London, UK, 1993. Springer-Verlag.

[63] M. Rudolf and T. Berlin. Utilizing constraint satisfaction techniques for efficient graph pattern matching. In *In 6th International Workshop on Theory an Application of Graph Transformations (TAGT*, pages 238–251. Springer, 1998.

[64] L. Skonieczny. Mining of unconnected frequent graphs with direct subgraph isomorphism tests. In *ICS Research Report 12/08*, 2008.

[65] L Skonieczny. Mining of unconnected frequent graphs with direct subgraph isomorphism tests. In *ICMMI '09: Proceedings of the International Conference on Man-Machine Interactions*, page 523–531. Springer-Verlag, 2009.

[66] B.M. Smith. Locating the phase transition in constraint satisfaction problems. In *Artificial Intelligence*, 1994.

[67] G.J. Sreenivasa and V.S. Ananthanarayana. Efficient mining of frequent rooted continuous directed subgraphs. *Advanced Computing and Communications, 2006. ADCOM 2006. International Conference on*, pages 553–558, 20-23 Dec. 2006.

[68] R. Ting and J. Bailey. Mining minimal contrast subgraph patterns. In *Sixth SIAM International Conference on Data Mining, SDM 2006*, pages 639–643, Maryland USA, April 2006.

[69] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, 1976.

[70] N. Vanetik, E. Gudes, and S. E. Shimony. Computing frequent graph patterns from semistructured data. In *ICDM '02: Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM'02)*, page 458, Washington, DC, USA, 2002. IEEE Computer Society.

[71] D Waltz. Understanding line drawings of scenes with shadows. In *The Psychology of Computer Vision*, page pages. McGraw-Hill, 1975.

[72] T. Washio and T. Motoda. State of the art of graph-based data mining. *SIGKDD Explor. Newsl.*, 5(1):59–68, 2003.

[73] T. Werth. *Desing and Implementation of a DAG-Miner*. PhD thesis, Institut fur Informatik, Friedrich-Alexander-Universitat Erlangen–Nurnberg, 2007.

[74] M. Wörlein, T. Meinl, I. Fischer, and M. Philippsen. A quantitative comparison of the subgraph miners MoFa, gspan, FFSM, and gaston. In *Knowledge Discovery in Databases: PKDD 2005*, volume Volume 3721/2005, pages 392–403. Springer Berlin / Heidelberg, 2005.

[75] A. Xu and H. Lei. Lcgminer: Levelwise closed graph pattern mining from large databases. In *SSDBM '04: Proceedings of the 16th International Conference on Scientific and Statistical Database Management*, page 421, Washington, DC, USA, 2004. IEEE Computer Society.

[76] X. Yan and J. Han. gspan: Graph-based substructure pattern mining. *icdm*, 00:721, 2002.

[77] X Yan and J. Han. Closegraph: mining closed frequent graph patterns. In *KDD '03: Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 286–295, New York, NY, USA, 2003. ACM.

[78] M.J. Zaki. Efficiently mining frequent trees in a forest. In *KDD '02: Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 71–80, New York, NY, USA, 2002. ACM.

[79] S. Zampelli. *A constraint programming approach to subgraph isomorphism*. PhD thesis, UCLouvain, Department of Computing Science and Engineering, June 2008.

[80] S. Zampelli, Y. Deville, and P. Dupont. Symmetry breaking in subgraph pattern matching. In *Proccedings of the Sixth International Workshop on Symmetry in Constraint Satisfaction Problems (SymCon'06)*, 2006.

[81] S Zampelli, Y. Deville, and C. Solnon. Solving subgraph isomorphism problems with constraint programming. *Constraints*, 2010.