POLITECHNIKA GDAŃSKA
WYDZIAŁ ELEKTRONIKI, TELEKOMUNIKACJI I INFORMATYKI

Imię i nazwisko studenta: Marcin Połajdowicz
Nr albumu: 184265
Poziom kształcenia: studia drugiego stopnia
Forma studiów: stacjonarne
Kierunek studiów: Informatyka
Specjalność: Algorytmy i technologie internetowe

Imię i nazwisko studenta: Maciej Sztramski
Nr albumu: 184779
Poziom kształcenia: studia drugiego stopnia
Forma studiów: stacjonarne
Kierunek studiów: Informatyka
Specjalność: Aplikacje rozproszone i systemy internetowe

PRACA DYPLOMOWA MAGISTERSKA

Tytuł pracy w języku polskim: Modelowanie optymalnych sposobów zakupu licencji oprogramowania w sieciach społecznościowych za pomocą dominowania w grafach

Tytuł pracy w języku angielskim: Modeling optimal ways to purchase software licenses in social networks via graph domination

Opiekun pracy: dr inż. Joanna Raczek

Streszczenie

Celem niniejszej pracy magisterskiej było opracowanie grafowego modelu optymalizacji zakupu licencji oprogramowania (np. Duolingo Super, Spotify Premium) w sieciach społecznościowych. Problem ten sformułowano jako rozszerzenie klasycznego dominowania (w tym dominowania rzymskiego) w grafach, przy czym uwzględniono praktyczne ograniczenia dotyczące rozmiaru grup licencyjnych oraz struktury kosztów licencji indywidualnych i grupowych.

W pracy przeprowadzono analizę złożoności obliczeniowej problemu, wykazując jego NP--trudność. Zaimplementowano i porównano podejścia algorytmiczne: metody dokładne (ILP/MIP) oraz kilka metod przybliżonych i metaheurystyk (algorytm zachłanny, algorytm genetyczny, przeszukiwanie tabu, symulowane wyżarzanie, algorytm mrówkowy). Badania eksperymentalne wykonano na rzeczywistych ego-sieciach Facebooka oraz na danych syntetycznych (Barabási-Albert, Erdős-Rényi, Watts-Strogatz).

Wyniki pokazują, że metaheurystyki (w szczególności algorytm genetyczny, przeszukiwanie tabu i symulowane wyżarzanie) pozwalają osiągać rozwiązania bliskie optymalnym przy znacznym skróceniu czasu obliczeń względem ILP na większych instancjach. Dodatkowo przeanalizowano wariant dynamiczny, w którym sieć ulega zmianom w czasie, oraz wpływ polityk cenowych i dodatkowych typów planów subskrypcyjnych na optymalne decyzje.

Słowa kluczowe: teoria grafów, optymalizacja kombinatoryczna, dominowanie rzymskie, sieci społecznościowe, heurystyki, metaheurystyki, programowanie całkowitoliczbowe.

Abstract

This thesis develops a graph-based model for optimizing software license purchases (e.g., Duolingo Super, Spotify Premium) in social networks. The problem is formulated as an extension of classical domination (including Roman domination) in graphs, incorporating realistic constraints on group sizes and pricing schemes for individual and group licenses.

We analyze the computational complexity and show NP-hardness. We implement and compare exact methods (integer linear programming) with heuristic and metaheuristic approaches (greedy, genetic algorithm, tabu search, simulated annealing, ant colony optimization). Experiments are conducted on real-world Facebook ego networks and on synthetic graphs (Barabási-Albert, Erdős-Rényi, Watts-Strogatz).

Results indicate that metaheuristics (notably genetic algorithm, tabu search and simulated annealing) achieve near-optimal solutions with significantly lower runtime than ILP for larger instances. We also study a dynamic variant in which the network evolves over time and evaluate the impact of pricing policies and additional subscription types on optimal purchasing decisions.

Keywords: graph theory, combinatorial optimization, Roman domination, social networks, heuristics, metaheuristics, integer programming.

SPIS TREŚCI

SPIS RYSUNKÓW

SPIS TABEL

LIST OF ALGORITHMS

WYKAZ WAŻNIEJSZYCH OZNACZEŃ I SKRÓTÓW

Oznaczenia

$G = (V, E)$	–	graf relacji społecznych.
V, E	–	zbiory wierzchołków i krawędzi.
$N(v),\ N[v]$	–	sąsiedztwo wierzchołka v oraz sąsiedztwo domknięte.
$\deg(v)$	–	stopień wierzchołka v.
$n = \|V\|,\ m = \|E\|$	–	liczba wierzchołków i krawędzi.
Δ	–	maksymalny stopień grafu.
$f : V \to \{0, 1, 2\}$	–	etykietowanie ról (0-odbiorca, 1-indywidualna, 2-grupowa).
$\mathrm{cost}(f)$	–	koszt rozwiązania (def. w rozdz. 2.2).
$\mathcal{L}$	–	rodzina typów licencji $\ell_t = (c_t,\, \ell_t^{\min},\, \ell_t^{\max})$, w skrócie $(c_t,\, l_t,\, u_t)$.
I, H, R	–	zbiory: licencji indywidualnych, grupowych i odbiorców.
p	–	względny koszt licencji grupowej $p = c_g/c_1$.

Skróty

SaaS	–	Software as a Service.
ILP/MIP	–	(Mixed) Integer Linear Programming.
GA	–	Genetic Algorithm.
SA	–	Simulated Annealing.
TS	–	Tabu Search.
ACO	–	Ant Colony Optimization.
MDS	–	Minimum Dominating Set.
RD	–	Roman Domination.

PODZIAŁ PRACY

Część implementacyjna obejmowała przygotowanie architektury projektu, modułów eksperymentów i analiz oraz zestawu algorytmów. W jej ramach Marcin Połajdowicz opracował ogólną strukturę systemu, moduły odpowiedzialne za wykonywanie eksperymentów i analiz, a także implementacje algorytmów dokładnych i konstrukcyjnych. Maciej Sztramski odpowiadał za logikę oraz implementację algorytmów metaheurystycznych.

Część teoretyczna, stanowiąca zasadniczą treść pracy magisterskiej, została podzielona według rozdziałów. Maciej Sztramski przygotował rozdziały 1–4, natomiast Marcin Połajdowicz odpowiadał za rozdziały 6–9. Rozdział 5 opracowano wspólnie, dzieląc obowiązki zgodnie z zakresem prac implementacyjnych nad poszczególnymi algorytmami.

1. WPROWADZENIE

1.1 Wstęp i motywacja

W ostatnich latach coraz większe znaczenie zyskują modele subskrypcyjne w sektorze oprogramowania i usług cyfrowych. Zgodnie z indeksem gospodarki subskrypcyjnej rynek ten zwiększył swoją wartość o ponad 400% od roku 2012 do roku 2021 [1], a w 2024 roku osiągnął przybliżoną wartość blisko 600 miliardów dolarów [2].

Konsumenci coraz częściej opłacają regularne abonamenty zamiast jednorazowych zakupów, co zapewnia firmom stałe przychody, a użytkownikom wygodny dostęp do usług. Wiele popularnych platform, w tym aplikacje edukacyjne, serwisy streamingowe czy oprogramowanie w modelu Software as a Service (SaaS), opiera się na modelu subskrypcyjnym. Serwisy streamingowe, takie jak Spotify czy Netflix, często oferują plany rodzinne, w których kilka osób może współdzielić jedną subskrypcję grupową, której koszt jest niższy niż zakup kilku subskrypcji indywidualnych. W podobny sposób platforma edukacyjna do nauki języków Duolingo udostępnia plan rodzinny Super Duolingo (ang. Duolingo Super Family) [3], który pozwala grupie znajomych lub osób spokrewnionych współdzielić korzyści subskrypcji premium. Zachęca to użytkowników do grupowego zakupu subskrypcji, redukując jednocześnie koszt przypadający na jedną osobę. Subskrypcja wiąże się z czasowym nabyciem licencji, dlatego w dalszej części pracy oba pojęcia będą używane zamiennie.

W przypadku formowania się takich grup kontekst społeczny odgrywa istotną rolę. Rozsądne i optymalne korzystanie z subskrypcji grupowych wymaga, aby główny posiadacz subskrypcji znał osoby zainteresowane wspólnym korzystaniem z usługi, czy to ze względu na chęć podziału kosztów, podobne zainteresowania lub zwyczajną bliskość relacji. Użytkownicy często łączą się w grupy z rodziną czy przyjaciółmi, ponieważ łatwiej wtedy o zaufanie w zakresie współdzielenia konta. Innym powodem może być też wygoda, ponieważ jedna wspólna subskrypcja upraszcza zarządzanie płatnościami i zapewnia wszystkim członkom grupy taki sam dostęp do usługi. Takie czynniki społeczne mają więc istotny wpływ na to, jak faktycznie kształtują się struktury współdzielenia w sieciach użytkowników. Rozwój mediów społecznościowych i komunikatorów ułatwia zawieranie takich porozumień w gronie znajomych lub osób o podobnych zainteresowaniach. W praktyce często dochodzi do sytuacji, w których użytkownicy umawiają się na wspólny zakup abonamentu. Sieć powiązań społecznych determinuje, kto z kim może skutecznie współdzielić licencję.

Analiza przedstawionych mechanizmów prowadzi do sformułowania problemu optymalizacyjnego: jak zaplanować zakup licencji w grupie powiązanych użytkowników, aby zminimalizować łączny koszt dostępu do usługi. Innymi słowy, mając daną sieć znajomości oraz dostępne opcje licencyjne, należy dobrać podzbiór użytkowników kupujących licencje (oraz rodzaj tych licencji)

tak, by wszyscy użytkownicy mieli dostęp do usługi przy możliwie najniższym koszcie. Intuicyjnie jest to problem pokrycia grafu zbiorem *właścicieli* (osób wykupujących licencje), w taki sposób, by każdy wierzchołek był albo objęty licencją, albo sąsiadował z kimś, kto licencję posiada.

Warto zauważyć, że opisana struktura problemu ma ścisłe powiązania z zagadnieniami teorii grafów. Jest ona bliska klasycznemu problemowi dominowania, ponieważ wybór użytkowników kupujących licencje grupowe pełni tę samą funkcję co wybór zbioru dominującego. W obu przypadkach chodzi o to, aby wybrany zestaw wierzchołków pokrywał całą sieć. W dalszej części pracy został również pokazany związek z wariantem znanym jako *dominowanie rzymskie*. Takie odniesienia do teorii grafów stanowią istotne uzupełnienie głównego celu badań, którym jest optymalizacja kosztów licencji w społeczności użytkowników.

1.2 Przegląd istniejących rozwiązań

Dotychczas w literaturze brak jest opracowań bezpośrednio analizujących problem optymalnego podziału licencji w sieciach społecznościowych. Innymi słowy, zagadnienie minimalizacji kosztów zakupu planów grupowych w sieciach powiązanych użytkowników nie zostało jeszcze opisane. Niniejsza praca podejmuje je po raz pierwszy, lokując je w ramach teorii grafów.

Najbliższą analogią jest klasyczne zagadnienie zbioru dominującego, które polega na znalezieniu minimalnego podzbioru wierzchołków, tak by każdy wierzchołek grafu należał do tego zbioru lub był jego sąsiadem. Problem ten jest NP-zupełny, co otwiera pole dla badań algorytmicznych i heurystycznych [4]. Szczególne znaczenie w kontekście analizowanego problemu ma dominowanie rzymskie, które zostało szczegółowo opisane w rozdziale 3. Koncepcja ta znalazła szerokie zastosowania i doczekała się wielu rozszerzeń, m.in. dominowania włoskiego czy dominowania zabezpieczonego [5].

Model ten jest szczególnie użyteczny dla odwzorowania problemu podziału licencji. Standardowa wersja dominowania rzymskiego zakłada brak ograniczeń co do liczby sąsiadów, których może dominować wierzchołek z etykietą 2. W rzeczywistości jednak plany subskrypcyjne narzucają limity (np. maksymalnie 6 osób w planie rodzinnym). Powoduje to naturalne odwołanie do dominowania z pojemnością, w której każdy wierzchołek dominujący ma przypisaną wartość $c(v)$ określającą, ilu sąsiadów może objąć swoim dominowaniem [6].

Podsumowując, problem podziału licencji stanowi nowy wariant zagadnienia dominowania w grafach, wykorzystujący idee dominowania rzymskiego oraz jego modyfikacji z ograniczeniami pojemności. Brak specjalistycznych opracowań wskazuje na wyraźną niszę badawczą. Jednocześnie bogata literatura dotycząca dominowania w grafach dostarcza ugruntowanych narzędzi teoretycznych i algorytmicznych, co pozwala formalnie wykazać NP-trudność badanego problemu oraz zastosować zarówno metody dokładne (np. programowanie całkowitoliczbowe), jak i przybliżone heurystyki, wzorując się na podejściach znanych z teorii dominowania [5, 6].

1.3 Cele i zakres pracy

Celem niniejszej pracy jest formalizacja i analiza problemu optymalnego zakupu licencji w sieciach społecznościowych, zaproponowanie metod jego rozwiązania oraz weryfikacja przyjętych rozwiązań. W pierwszej kolejności opracowany został model grafowy opisujący powiązania między użytkownikami oraz różne strategie zakupowe wraz z odpowiadającymi im kosztami. Taki model pozwolił zdefiniować problem minimalizacji kosztów jako zadanie optymalizacyjne na grafie. Następnie wykazano ścisły związek z problemem dominowania w grafach. W szczególności pokazano, że dla pewnej klasy modeli licencjonowania zadanie optymalnego doboru subskrypcji jest równoważne znalezieniu minimalnego zbioru dominującego lub rozwiązaniu pokrewnego problemu *dominowania rzymskiego*. Odwołanie do znanych wyników o dominowaniu obejmuje zarówno aspekty złożoności obliczeniowej, jak i badania nad algorytmami aproksymacyjnymi, co pozwala lepiej zrozumieć trudności badanego problemu oraz zaprojektować efektywne metody jego rozwiązywania.

Zakres pracy obejmuje analizę teoretyczną badanego problemu oraz metody algorytmiczne jego rozwiązania, uzupełnione o eksperymenty obliczeniowe. Rozpatrzone zostały różne modele cenowe licencji, zarówno hipotetyczne, jak i rzeczywiste. Do pierwszej grupy należą warianty nawiązujące do dominowania rzymskiego, w których koszt licencji grupowej stanowi wielokrotność ceny licencji indywidualnej. W drugiej grupie znajdują się modele oparte na rzeczywistych ofertach usług, takich jak Spotify, Netflix czy Duolingo, co pozwala zweryfikować otrzymane wyniki w kontekście praktycznych scenariuszy.

Osobno przeanalizowane zostały warianty, w których decyzje zakupowe podejmowane są globalnie (jednocześnie dla całej społeczności), oraz scenariusze dynamiczne. W wersji dynamicznej zakupy realizowane są w kolejnych krokach czasowych, a struktura sieci społecznościowej może się zmieniać (rozszerzanie lub zmniejszanie liczby użytkowników, powstawanie lub zanik relacji). Ze względu na wysoką złożoność obliczeniową problemu przedstawione zostały zarówno metody dokładne, jak i heurystyczne. Metody dokładne gwarantują znalezienie rozwiązania optymalnego, lecz ich czas działania szybko rośnie wraz z rozmiarem grafu. Z kolei metody heurystyczne nie dają gwarancji optymalności, ale pozwalają uzyskać rozwiązania dobrej jakości w akceptowalnym czasie. Celem praktycznym jest wskazanie podejść skutecznych w optymalizacji kosztów subskrypcji w dużych sieciach społecznościowych oraz identyfikacja czynników najsilniej wpływających na wyniki, co zilustrowane jest wynikami eksperymentów.

Podsumowując, w ramach pracy zrealizowane zostały następujące zadania badawcze i implementacyjne:

- Sformalizowano model optymalizacji zakupu licencji w sieciach społecznościowych, obejmujący etykietowanie ról $f : V \to \{0, 1, 2\}$, warunki wykonalności (pokrycie, sąsiedztwo, pojemność) oraz funkcję kosztu.
- Wykazano powiązania z *dominowaniem rzymskim* i sformułowano twierdzenie o równoważności w szczególnym przypadku kosztów i pojemności; omówiono również konsekwencje

złożonościowe i aproksymacyjne.

- Zaimplementowano i porównano różne metody: ILP (PuLP/CBC), algorytm zachłanny, metaheurystyki (algorytm genetyczny, symulowane wyżarzanie, przeszukiwanie tabu, algorytm mrówkowy).
- Opracowano środowisko eksperymentalne dla grafów syntetycznych i ego-sieci Facebooka, wraz z pomiarem czasu działania i kosztu.
- Przeanalizowano wariant dynamiczny (mutacje grafu) oraz porównano podejścia cold-start i warm-start dla metaheurystyk.
- Zbadano wpływ polityk cenowych i typów planów (Individual/Duo/Family) na strukturę rozwiązań i koszt całkowity.

1.4 Struktura

Struktura pracy została zorganizowana w dziewięciu rozdziałach. Obejmują one część wprowadzającą, w której przedstawiono tło i motywację podjętego zagadnienia, część analityczno-badawczą, zawierającą opis zaproponowanych modeli oraz przeprowadzonych eksperymentów, a także część podsumowującą, w której sformułowano wnioski oraz wskazano możliwe kierunki dalszych badań. Poniżej zaprezentowano opis treści wszystkich rozdziałów, stanowiących kolejne etapy realizacji pracy.

Rozdział 1 – Wprowadzenie. Przedstawia tło problemu, motywację podjęcia tematu oraz znaczenie optymalizacji kosztów w kontekście współdzielenia licencji w sieciach społecznościowych. Określone są cele i zakres pracy oraz jej ogólna struktura.

Rozdział 2 – Model grafowy i analiza problemu. Definiuje reprezentację sieci społecznościowej w postaci grafu oraz formalny opis problemu optymalizacji kosztów. Uwzględnione są przyjęte założenia, definicje pojęć oraz warianty wynikające z odmiennych modeli cenowych i ograniczeń pojemnościowych. Rozdział stanowi podstawę do dalszych rozważań algorytmicznych.

Rozdział 3 – Związek z dominowaniem w grafach. Omawia pojęcie zbiorów dominujących i dominowania rzymskiego, pokazując, w jaki sposób badany problem można interpretować w tych kategoriach. Przedstawione są również aspekty złożoności obliczeniowej, w tym dowód NP-trudności, oraz wynikające z tego konsekwencje dla możliwości projektowania algorytmów.

Rozdział 4 – Dane testowe. Opisuje rodzaje grafów wykorzystanych w eksperymentach: syntetyczne, generowane przy pomocy popularnych modeli (Barabási–Albert, Watts–Strogatz oraz Erdős–Rényi), oraz grafy rzeczywiste pochodzące z repozytoriów badawczych. Uwzględniono także sposób przygotowania instancji testowych oraz narzędzia użyte do wizualizacji sieci.

Rozdział 5 – Metody algorytmiczne optymalizacji kosztów licencji. Rozdział skupia się na części obliczeniowej. Przedstawiona jest formalizacja problemu w postaci programu całkowitoliczbowego oraz opis metod dokładnych, które mogą znaleźć optymalne rozwiązania dla

mniejszych instancji. Omówione są także podejścia przybliżone i heurystyczne.

Rozdział 6 – Eksperymenty i analiza wyników. Przedstawia proces oceny algorytmów na przygotowanych danych testowych. Określone są kryteria porównawcze (m.in. czas działania, złożoność obliczeniowa, uzyskane koszty), a następnie zaprezentowane wyniki eksperymentów dla różnych typów grafów i ich skal. Analizowany jest wpływ parametrów algorytmów na efektywność i jakość uzyskiwanych rozwiązań.

Rozdział 7 – Analiza dynamicznej wersji problemu. Rozważany jest scenariusz, w którym zakupy licencji odbywają się sekwencyjnie, a struktura sieci może ulegać zmianie w czasie. Opisane są możliwe adaptacje algorytmów do takiej sytuacji oraz przeprowadzone eksperymenty badające ich skuteczność. Poruszona jest także kwestia stabilności i elastyczności strategii w środowisku dynamicznym.

Rozdział 8 – Rozszerzenia modelu. Omawia dodatkowe aspekty, które mogą wpływać na decyzje optymalizacyjne, takie jak polityki cenowe oraz zróżnicowanie typów licencji. Analizowane są również potencjalne ograniczenia modelu i możliwości jego dalszego uogólnienia.

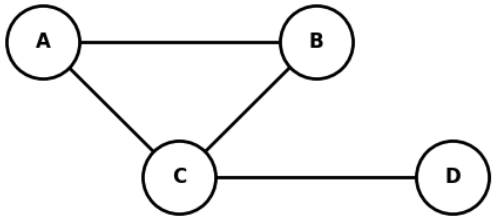
Rozdział 9 – Podsumowanie i wnioski. Zawiera syntetyczne zestawienie wyników pracy. Wskazuje, w jakim stopniu zrealizowane zostały założone cele, oraz proponuje kierunki dalszych badań, w tym rozwój modeli, udoskonalenia algorytmów oraz badania nad skalowalnością i praktycznymi zastosowaniami. Rozdział ten zamyka całość pracy, odpowiadając na pytania badawcze i wskazując, jak uzyskane rezultaty mogą zostać wykorzystane w praktyce.

2. MODEL GRAFOWY PROBLEMU ZAKUPU LICENCJI

2.1 Reprezentacja grafowa

Aby formalnie opisać zjawisko współdzielenia licencji, sieć relacji społecznych modelujemy jako graf nieskierowany $G = (V, E)$. Każdy wierzchołek $v \in V$ reprezentuje pojedynczego użytkownika, natomiast krawędź $\{u, v\} \in E$ oznacza, że użytkownicy u i v znajdują się w relacji umożliwiającej współdzielenie licencji grupowej. Graf jest nieskierowany, ponieważ zakładamy symetryczność tej relacji: jeśli u zna v, to również v zna u.

Przyjmujemy również, że graf G nie zawiera pętli, ani krawędzi wielokrotnych - każda para użytkowników może być powiązana co najwyżej jedną krawędzią. Przykład takiej struktury zilustrowano na Rysunku 2.1.



Rysunek 2.1: Przykładowy graf relacji społecznych między użytkownikami.

Opisana reprezentacja, w której wierzchołki odpowiadają jednostkom, a krawędzie bezpośrednim relacjom umożliwiającym interakcję, jest powszechnie stosowana w analizie sieci społecznościowych [7, 8]. Takie ujęcie pozwala formalnie modelować i badać zjawiska zachodzące w społecznościach użytkowników usług cyfrowych.

W przyjętym modelu zakłada się, że współdzielenie licencji może odbywać się wyłącznie między osobami połączonymi bezpośrednią krawędzią w grafie. Licencja grupowa oznacza w tym kontekście typ umożliwiający współdzielenie dostępu przez właściciela oraz jego bezpośrednich sąsiadów. Oznacza to, że użytkownicy muszą znać się bezpośrednio i mieć wzajemne zaufanie, co jest istotne na przykład w przypadku przekazywania danych logowania lub zapraszania do planu rodzinnego. Relacje pośrednie, w których użytkownicy są powiązani poprzez wspólnych znajomych (np. $A \sim B$ oraz $B \sim C$, lecz brak bezpośredniego powiązania $A \sim C$), nie są uwzględniane w analizie. Oznacza to, że dla danego grafu $G = (V, E)$, w którym V to zbiór użytkowników, a $E \subseteq \{\{u, v\} : u, v \in V, u \neq v\}$ to zbiór relacji znajomości, analizie podlegają wyłącznie relacje bezpośrednie, czyli pary $\{u, v\} \in E$. Na przykład w grafie przedstawionym na rysunku 2.1, użytkownicy A i D są wprawdzie połączeni za pośrednictwem ścieżki $A \to C \to D$, jednak ponieważ brakuje bezpośredniego połączenia $\{A, D\} \in E$, to taka relacja nie jest uznawana za podstawę do współdzielenia licencji w tym modelu. Mimo, że w praktyce relacje pośrednie, takie jak ścieżki

długości większej niż jeden, mogą sprzyjać tworzeniu grup subskrypcyjnych, w analizie zostały one pominięte w celu uproszczenia problemu.

Graf społecznościowy nie musi być pełny. Dopuszcza się dowolną strukturę odpowiadającą rzeczywistym relacjom społecznym. W analizie istotną rolę odgrywa stopień wierzchołków, ponieważ decyduje on o liczbie osób, którym dany użytkownik może udostępnić swoją licencję. Dla wierzchołka $v \in V$, jego stopień oznaczamy przez $\deg(v)$, co odpowiada liczbie sąsiadów użytkownika v w grafie $G = (V, E)$.

Należy jednak zauważyć, że nawet w przypadku wysokiego stopnia $\deg(v)$, użytkownik niekoniecznie może współdzielić licencję ze wszystkimi swoimi sąsiadami. Ograniczenia techniczne, takie jak limity liczby współużytkowników narzucane przez dostawcę usługi, sprawiają, że liczba osób objętych jedną licencją grupową pozostaje ograniczona. W analizowanym modelu wprowadzamy zatem parametr k, oznaczający pojemność licencji grupowej, czyli maksymalną liczbę osób, wliczając w to właściciela, które mogą korzystać z jednej licencji.

2.2 Definicja problemu

Optymalizacja kosztu dostępu do usługi wymaga przypisania wszystkim wierzchołkom grafu $G = (V, E)$ odpowiednich ról. Każdy użytkownik $v \in V$ może uzyskać dostęp do usługi na trzy sposoby:

1. Poprzez wykupienie licencji indywidualnej.
2. Poprzez wykupienie licencji grupowej.
3. Jako odbiorca, korzystający z licencji grupowej należącej do innego użytkownika.

Licencja indywidualna zapewnia dostęp wyłącznie jej właścicielowi, natomiast licencja grupowa umożliwia współdzielenie dostępu z maksymalnie $k - 1$ sąsiadami w grafie. Należy przy tym podkreślić, że pojedynczy użytkownik może posiadać najwyżej jedną licencję grupową, nawet jeśli liczba jego sąsiadów przekracza dopuszczalny limit. Ograniczenie to odzwierciedla praktyczne zasady zakładania kont, takie jak konieczność powiązania ich z jednym numerem telefonu, brak chęci zakładania wielu adresów e-mail czy inne regulacje usługodawców. Dla uproszczenia przyjmuje się, że zasada ta obowiązuje wszystkie typy licencji, niezależnie od rzeczywistych warunków oferowanych przez poszczególnych dostawców usług.

Dla przejrzystego i precyzyjnego zdefiniowania modelu wprowadzamy trzy zbiory reprezentujące użytkowników, czyli węzły posiadające własną licencję bądź korzystające z licencji innego węzła:

- I - posiadacze licencji indywidualnych;
- H - posiadacze licencji grupowych;
- R - odbiorcy, którzy sami nie kupują licencji, lecz korzystają z licencji innego użytkownika.

Aby formalnie opisać ten podział, wprowadzamy etykietowanie ról $f : V \to \{0, 1, 2\}$. Wartość 0 oznacza węzeł będący odbiorcą, 1 odpowiada licencji indywidualnej, a 2 licencji grupowej. Takie przypisanie nawiązuje do klasycznego problemu dominowania rzymskiego, w którym wierzchołki również otrzymują etykiety z tego samego zbioru wartości. Odpowiadające zbiory definiu-

jemy jako $I = \{v : f(v) = 1\}$, $H = \{v : f(v) = 2\}$ oraz $R = V \setminus (I \cup G)$. Przyjmujemy zbiór dostępnych typów licencji

$$\mathcal{L} = \{\ell_t = (c_t, m_t, k_t) \mid t = 1, 2, \ldots, T\}, \tag{2.1}$$

gdzie:

- c_t - koszt licencji typu t,
- m_t - minimalna liczba użytkowników wliczając właściciela,
- k_t - maksymalna liczba użytkowników wliczając właściciela,
- T - całkowita liczba dostępnych typów licencji.

W podstawowym wariancie rozważanym w tym rozdziale zakładamy, że zbiór $\mathcal{L}$ opisany we wzorze (2.1) obejmuje wyłącznie dwa typy licencji: indywidualną oraz jedną grupową, a zatem przyjęte jest $T = 2$. Model ten nie przewiduje sytuacji z wieloma rodzajami planów grupowych (np. Duo, Family), lecz ogranicza się do najprostszego przypadku odpowiadającego rozwiązaniom takim jak w Duolingo.

Warunki wykonalności. Spełnienie rozwiązania wymaga, aby dla etykietowania $f : V \to \{0, 1, 2\}$ zachodziły następujące warunki:

1. Pokrycie: Każdy użytkownik $v \in V$ musi mieć dostęp do usługi, tj. $f(v) \in \{1, 2\}$ lub istnieje sąsiad $u \in N(v)$ taki, że $f(u) = 2$ i v jest przypisany do grupy u.
2. Sąsiedztwo: Odbiorca $v \in R$ może być przypisany tylko do właściciela $u \in H$ z $\{u, v\} \in E$.
3. Pojemność: Liczba użytkowników przypisanych do właściciela $u \in H$ (wraz z nim samym) musi spełniać $m \leq 1 + |R_u| \leq k$, gdzie R_u oznacza zbiór odbiorców korzystających z licencji u.
4. Jednoznaczność licencji: Każdy wierzchołek $v \in V$ może być oznaczony jako właściciel grupowy ($f(v) = 2$) najwyżej raz, tj. $v \in H$ w co najwyżej jednym przypisaniu.

Funkcja kosztu. W wariancie podstawowym, w którym $T = 2$, całkowity koszt rozwiązania wyraża się zależnością:

$$\text{cost}(f) = |I| \cdot c_i + |H| \cdot c_g, \tag{2.2}$$

gdzie:

- I - zbiór użytkowników z licencją indywidualną,
- H - zbiór użytkowników z licencją grupową,
- c_i - koszt licencji indywidualnej,
- c_g - koszt licencji grupowej.

Cel optymalizacji. Celem problemu jest minimalizacja funkcji kosztu (2.2) dla danego grafu $G = (V, E)$, przy spełnieniu wszystkich istotnych warunków wykonalności.

2.3 Koszty i ograniczenia

2.3.1 Ograniczenia techniczne i społeczne współdzielenia licencji

Kluczowymi parametrami modelu są najmniejsza oraz największa liczba osób, które mogą współdzielić jedną licencję grupową. Największy rozmiar grupy oznaczamy przez k i wliczamy do niego także użytkownika nabywającego licencję. Parametr k jest zwykle narzucany przez dostawcę usługi. Przykładowo, plan rodzinny Spotify Premium pozwala na korzystanie co najwyżej sześciu osobom (właściciel + pięć członków rodziny), co odpowiada wartości $k = 6$.

Analogicznie wprowadzamy parametr m, który określa najmniejszą liczbę osób niezbędnych do utworzenia grupy. Oznacza to, że licencja grupowa jest ważna tylko wtedy, gdy zostanie wykorzystana przez co najmniej m osób (łącznie z właścicielem). Przykładem jest tzw. plan „Duo", w którym licencję mogą współdzielić dokładnie dwie osoby ($m = 2, k = 2$). W innych przypadkach m może przyjmować wartości mniejsze niż k, np. $m = 2, k = 6$ dla planów rodzinnych.

W analizie przyjmujemy m oraz k jako zmienne parametry. Nawet jeśli użytkownik posiada wielu znajomych (czyli ma wysoki stopień w grafie), ograniczenie k sprawia, że może objąć współdzieleniem tylko określoną najwyższą liczbę osób, natomiast ograniczenie m wymusza, by grupy nie były zbyt małe. Parametry te modelują zarówno ograniczenia techniczne narzucane przez dostawców usług, jak i czynniki społeczne, takie jak opłacalność czy gotowość do współdzielenia subskrypcji. W konsekwencji grupy współdzielenia odwzorowują typowe sytuacje, w których w praktyce licencje grupowe są wykorzystywane.

2.3.2 Struktura kosztów i modele cenowe

W analizowanym problemie istotnym elementem jest sposób odwzorowania polityki cenowej dostawców usług. Różne plany subskrypcyjne charakteryzują się nie tylko innym kosztem, ale również odmiennym zakresem liczby użytkowników m oraz k, którzy mogą korzystać z jednej licencji. Rodzinę typów licencji zdefiniowaną we wzorze (2.1) stosujemy tutaj do opisu struktury kosztów i ograniczeń dostępnych planów. W testach syntetycznych wartości parametrów licencji dobierane są eksperymentalnie, co pozwala badać zachowanie algorytmów w różnych wariantach cenowych i strukturalnych. W testach na danych rzeczywistych wykorzystywane są faktyczne ceny subskrypcji. Dla wariantu teoretycznego odpowiadającego dominowaniu rzymskiemu koszty normalizowane są względem licencji indywidualnej, co umożliwia powiązanie modelu z klasycznymi zagadnieniami teorii grafów. Przykłady zestawiono w tabeli 2.1.

W przypadku usług rzeczywistych koszty planów grupowych są znacznie niższe niż suma odpowiadających im licencji indywidualnych. Na przykład plan Spotify Family pozwala na współdzielenie subskrypcji przez co najwyżej sześć osób przy koszcie 37.99 PLN, co czyni go zdecydowanie bardziej opłacalnym od planu indywidualnego (23.99 PLN). Podobna sytuacja występuje w przypadku Duolingo Super Family.

W modelu teoretycznym opartym na dominowaniu rzymskim koszty są podane w jednostkach względnych. Licencja indywidualna ma koszt 1.0, a licencja grupowa koszt 2.0, co odpowiada

Tabela 2.1: Przykładowe modele licencji dla usług rzeczywistych i modelu teoretycznego

Typ licencji	**Koszt** c_t	**Min** m_t	**Max** k_t
Spotify (ceny w PLN)			
Individual	23.99	1	1
Duo	30.99	2	2
Family	37.99	2	6
Netflix (ceny w PLN)			
Basic	33.00	1	1
Standard	49.00	1	2
Premium	67.00	1	4
Duolingo Super (ceny w PLN)			
Individual	13.99	1	1
Family	29.17	2	6
Model teoretyczny (koszty znormalizowane)			
Solo	1.0	1	1
Group	2.0	2	999999

Źródło: opracowanie własne modelu teoretycznego oraz [9], [10], [11].

klasycznemu przypisaniu wag w problemie dominowania rzymskiego. W ramach przeprowadzanych eksperymentów etykieta „2” będzie zmieniała swoją wartość jako wielokrotność kosztu ceny indywidualnej, co pozwala na analizę różnych wariantów tego zagadnienia.

2.3.3 Zakup jednoczesny i sekwencyjny

W najprostszym wariancie zakłada się, że wszystkie decyzje zakupowe zapadają w tym samym momencie. Umożliwia to globalną optymalizację i stanowi punkt odniesienia dla analiz teoretycznych. W praktyce jednak proces współdzielenia licencji jest bardziej dynamiczny. Użytkownicy dołączają do planów w różnych chwilach, część początkowo wybiera licencje indywidualne, a dopiero później postanawia zakupić subskrypcję grupową. Takie zmiany licencji mogą również wynikać z równoczesnej zmiany i ewolucji sieć społecznościowej w czasie rzeczywistym. Relacje znajomości mogą zanikać, powstawać mogą nowe połączenia, a liczba aktywnych użytkowników zmienia się w czasie.

Takie sytuacje można modelować jako proces sekwencyjny, w którym w kolejnych krokach przydzielane są nowe licencje przy uwzględnieniu bieżącej struktury grafu. Prowadzi to do odmiennych trudności niż w przypadku wariantu uproszczonego, czyli jednoczesnego. Rozwiązanie optymalne globalnie może okazać się nieosiągalne, ponieważ wcześniejsze decyzje oraz zmiany w strukturze sieci ograniczają przestrzeń dostępnych opcji w późniejszych etapach.

W pracy przeanalizowane zostały konsekwencje tego typu dynamicznych zmian takie jak stabilność i trwałość powstałych grup, wpływ zmian w grafie na opłacalność wcześniejszych decyzji, a także mechanizmy sprzyjające koordynacji i adaptacji użytkowników. Mimo że główny nacisk położony jest na wariant jednoczesny, wariant sekwencyjny stanowi istotne rozszerzenie modelu i lepiej odzwierciedla rzeczywiste warunki funkcjonowania sieci społecznościowych.

3. ZWIĄZEK Z DOMINOWANIEM W GRAFACH

3.1 Dominowanie - podstawowe definicje

Problematyka dominowania w grafach jest dobrze zbadana i szeroko opisana w literaturze [12]. W szczególności znane są klasyczne definicje zbioru dominującego oraz liczby dominowania, które stanowią punkt odniesienia dla dalszych analiz. Niech $G = (V, E)$ będzie grafem nieskierowanym. Zbiorem dominującym nazywamy podzbiór $D \subseteq V$ taki, że każdy wierzchołek spoza D ma co najmniej jednego sąsiada w D:

$$\forall v \in V \setminus D \;\; \exists\, u \in D : \{u, v\} \in E, \tag{3.1}$$

gdzie:

- V - zbiór wierzchołków grafu,
- E - zbiór krawędzi grafu,
- $D \subseteq V$ - zbiór dominujący,
- $u, v \in V$ - wierzchołki grafu,
- $\{u, v\} \in E$ - krawędź łącząca wierzchołki u i v.

Najmniejszą moc zbioru dominującego oznaczamy symbolem $\gamma(G)$ i nazywamy liczbą dominowania:

$$\gamma(G) = \min\{\, |D| : D \subseteq V \,\}, \tag{3.2}$$

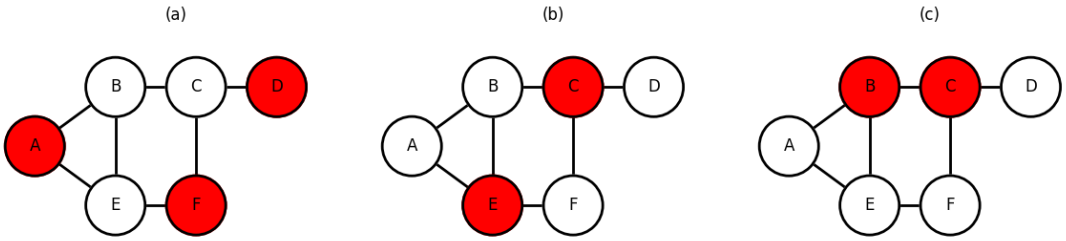
gdzie:

- $\gamma(G)$ - liczba dominowania grafu G,
- D - zbiór dominujący w grafie G,
- V - zbiór wierzchołków grafu,
- $|D|$ - moc zbioru D.

Z punktu widzenia rozważanego problemu zakupu licencji, podstawowe założenie jest następujące: jeśli potraktujemy osoby kupujące licencje jako zbiór D, a krawędzie grafu jako relacje umożliwiające udostępnianie licencji, wówczas warunek dominowania opisany równaniem (3.1) określa sytuację, w której każdy użytkownik spoza D ma znajomego w D, a zatem uzyskuje dostęp do usługi. Gdyby wszystkie licencje były identyczne i pozwalały obsłużyć dowolną liczbę sąsiadów, minimalizacja kosztu sprowadzałaby się do wyznaczenia liczby dominowania (3.2). W praktyce jednak występują różne typy licencji oraz limity współużytkowników, co czyni problem znacznie bardziej złożonym.

Należy zauważyć, że najmniejszy zbiór dominujący nie musi być jednoznaczny. W części przypadków można w grafie znaleźć kilka różnych najmniejszych zbiorów dominujących. Wynika to bezpośrednio z definicji liczby dominowania (3.2), która określa jedynie minimalną moc zbioru

dominującego, a nie jego unikalność. Często bywa tak, że graf ma wiele różnych zbiorów dominujących o rozmiarze równym $\gamma(G)$, tak jak przedstawiono na rysunku 3.1, gdzie trzy różne zbiory wierzchołków są minimalne ale tylko dwa są najmniejsze. Problem znajdowania $\gamma(G)$ jest jednak dobrze określony i należy do klasy problemów NP-trudnych. Jego wersja decyzyjna sformułowana jest następująco: dla zadanego grafu G oraz liczby całkowitej k pytamy, czy istnieje zbiór dominujący o rozmiarze co najwyżej k. Jest to klasyczny problem NP-zupełny [13, 14, 15]. Oznacza to, że najprawdopodobniej (przy założeniu $P \neq NP$) nie istnieje algorytm wielomianowy rozwiązujący ten problem w ogólności. W praktyce stosuje się zatem algorytmy przybliżone lub ogranicza analizę do specjalnych klas grafów, dla których problem staje się prostszy. Znane są między innymi efektywne algorytmy zachłanne, które zapewniają rozwiązanie przybliżone z gwarantowanym współczynnikiem. Przykładem jest prosty algorytm zachłanny wybierający kolejno wierzchołki do zbioru dominującego, osiągający przybliżenie rzędu $O(\ln n)$.



Rysunek 3.1: Przykładowe zbiory dominujące (wyróżnione na czerwono). **(a)** Minimalny zbiór dominujący o mocy 3. **(b)**-**(c)** Dwa różne minimalne i najmniejsze zbiory dominujące, gdzie $\gamma(G) = 2$. Każdy wierzchołek nieczerwony ma sąsiada czerwonego.

Wiadomo jednak, że nie da się w ogólności przekroczyć bariery logarytmicznej. Problem zbioru dominującego jest APX-trudny, a dokładniej log-APX-zupełny [14]. Co więcej, nawet na bardzo ograniczonych grafach, np. grafach o maksymalnym stopniu 3 (grafy kubiczne), problem pozostaje NP-trudny i APX-zupełny [16]. Berman i Fujito (1999) wykazali m.in. NP-trudność pewnych wariantów dominowania w grafach o ograniczonym stopniu [17], co potwierdza, że zasadnicza trudność problemu dominowania jest obecna już w stosunkowo prostych strukturach.

3.2 Dominowanie rzymskie a licencje grupowe

Dominowanie rzymskie to wariant problemu dominowania, w którym każdemu wierzchołkowi przypisuje się jedną z trzech wartości: 0, 1 lub 2. Wierzchołek o wartości 1 dominuje wyłącznie samego siebie, natomiast wierzchołek o wartości 2 dominuje zarówno siebie, jak i wszystkich swoich sąsiadów. Wymaga się przy tym, aby każdy wierzchołek o wartości 0 był sąsiadem co najmniej jednego wierzchołka oznaczonego wartością 2. Formalnie, funkcja dominowania rzymskiego na grafie $G = (V, E)$ to funkcja $f : V \to \{0, 1, 2\}$, spełniająca warunek, że dla każdego wierzchołka v z $f(v) = 0$ istnieje sąsiad $u \in V$ taki, że $f(u) = 2$ [18]. Minimalizacja sumy wartości $f(v)$ po wszystkich $v \in V$ prowadzi do zdefiniowania tzw. liczby dominowania rzymskiego grafu, oznaczanej $\gamma_R(G)$.

Terminologia i metafora związana z dominowaniem rzymskim wywodzą się z legendy o obronie granic imperium rzymskiego. Zakładano w niej, że w każdej osadzie można umieścić pewną liczbę jednostek wojskowych. Osada z dwiema jednostkami była w stanie bronić się samodzielnie i jednocześnie wysłać wsparcie do sąsiedniej osady. Lokacja z jedną jednostką potrafiła bronić tylko siebie, a miejscowości pozbawione jednostek militarnych wymagała ochrony z zewnątrz. W tej metaforze graf reprezentuje system osad i połączeń między nimi, a etykiety 0, 1 i 2 odpowiadają decyzjom o rozmieszczeniu wojsk. Koncepcję dominowania rzymskiego wprowadzili do teorii grafów Cockayne i współpracownicy w 2004 roku [19].

W kontekście problemu optymalnego zakupu licencji w grafach reprezentujących sieci społecznościowe interpretacja jest bezpośrednia. Wartość $f(v) = 2$ odpowiada użytkownikowi v, który nabywa licencję grupową i zapewnia dostęp zarówno sobie, jak i co najmniej jednemu ze swoich sąsiadów. Wartość $f(v) = 1$ oznacza użytkownika posiadającego licencję indywidualną, pokrywającą wyłącznie jego samego. Natomiast $f(v) = 0$ reprezentuje użytkownika bez własnej licencji, który musi polegać na pokryciu przez sąsiada z wartością 2. Warunek dominowania rzymskiego, zgodnie z którym każdy wierzchołek z etykietą 0 ma sąsiada z etykietą 2, gwarantuje dokładnie to, co w naszym modelu jest wymagane: każdy użytkownik bez licencji ma znajomego z licencją grupową, który może podzielić się z nim dostępem. W ten sposób każda funkcja $f : V \to \{0, 1, 2\}$ spełniająca warunki dominowania rzymskiego wyznacza dopuszczalną strategię licencyjną w rozważanej sieci.

Waga funkcji dominowania rzymskiego definiowana jest jako $w(f) = \sum_{v \in V} f(v)$, czyli suma przypisanych wartości. Liczba dominowania rzymskiego $\gamma_R(G)$ to najmniejsza możliwa waga funkcji dominowania rzymskiego dla grafu G. Jeśli przyjmiemy, że koszt licencji indywidualnej wynosi 1, a grupowej 2, to minimalizacja kosztu w naszym problemie pokrywa się z zagadnieniem znalezienia funkcji dominowania rzymskiego o najmniejszej wadze. Cel znalezienia jak największego sumarycznego kosztu $C = |I| + 2|H|$ jest zatem równoważny minimalizacji $w(f)$, gdy utożsamimy $|I|$ z liczbą wierzchołków o etykiecie 1, a $|H|$ o etykiecie 2. Należy jednak podkreślić, że pełna równoważność z dominowaniem rzymskim zachodzi tylko wtedy, gdy liczba sąsiadów dowolnego wierzchołka nie przekracza maksymalnej liczby użytkowników dopuszczonych w planie grupowym. Jak wspomniano wcześniej przy omawianiu parametrów m i k, nasze ujęcie wprowadza dodatkowe ograniczenie pojemności - wierzchołek dla którego funkcja f przyjmuje wartość 2 może pokrywać jedynie ograniczoną liczbę sąsiadów. Problem optymalnego zakupu licencji jest więc uogólnieniem dominowania rzymskiego, dostosowanym do praktycznych limitów występujących w planach subskrypcyjnych.

W przypadku innych modeli cenowych, dominowanie rzymskie stanowi nadal użyteczną metaforę, choć nie oddaje w sposób wystarczający struktury kosztów. Gdy $p \neq 2$, możemy rozważyć ogólniejsze przypisania wag, w których etykieta odpowiada rzeczywistemu kosztowi planu. Przykładowo, gdy $p = 3$, to licencja grupowa ma koszt równy trzem jednostkom, co nie mieści się w klasycznym schemacie $\{0, 1, 2\}$. W literaturze zaproponowano rozszerzenia pozwalające modelować takie sytuacje, m.in. k-dominowanie rzymskie, gdzie dopuszcza się wartości $\{0, 1, \ldots, k\}$

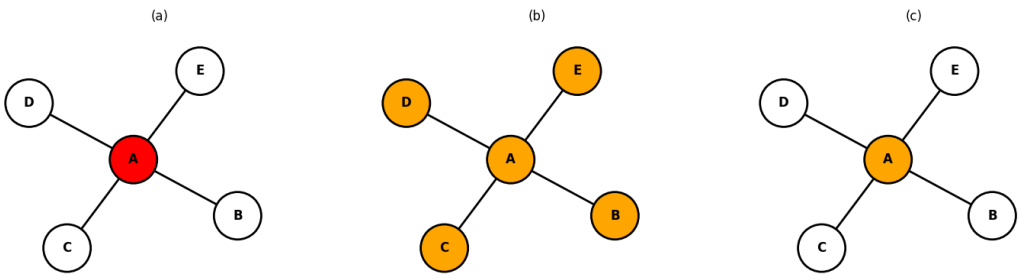
[20], czy też dominowanie rzymskie z wagami [21]. Warianty te pozwalają odwzorować przypadki, w których dostępne są plany o różnych kosztach i różnej pojemności, np. licencja droższa, ale umożliwiająca współdzielenie w większej grupie. W tym sensie uogólnienia dominowania rzymskiego są bliższe rzeczywistemu problemowi optymalizacji kosztów licencji niż klasyczna wersja ograniczona do wartości 0, 1 i 2.

Na potrzeby tej pracy nie jest jednak konieczne wchodzenie w szczegółowe odmiany dominowania rzymskiego. Wystarczy zauważyć, że w analizowanym modelu schemat pozostaje taki sam jak w klasycznej wersji, a jedyną różnicą jest koszt przypisywany wierzchołkom o etykiecie 2. W zależności od przyjętego modelu cenowego wartość ta może wynosić $p = 2$, ale równie dobrze $p = 1.5$ czy $p = 3$. Konstrukcja funkcji dominowania rzymskiego oraz sam podział na etykiety $\{0, 1, 2\}$ nie ulega zmianie. Innymi słowy, dominowanie rzymskie dostarcza prostego i intuicyjnego opisu sytuacji.

Prosty przykład zastosowania dominowania rzymskiego można przedstawić na grafie typu gwiazda. Centralny wierzchołek A jest połączony z liśćmi B, C, D, E. Najlepszą strategią jest, aby A wykupił licencję grupową i udostępnił ją wszystkim swoim sąsiadom. W modelu oznacza to $f(A) = 2$, a dla każdego liścia $f(B) = f(C) = f(D) = f(E) = 0$. Warunek dominowania rzymskiego jest spełniony, ponieważ każdy liść, gdzie funkcja f przyjmuje wartość 0 ma sąsiada A, który dla tej samej funkcji przejmuje wartość 2. Waga funkcji wynosi wówczas $w(f) = 2$ - odpowiada to kosztowi jednej licencji grupowej obsługującej całą pięcioosobową grupę.

Dla porównania można rozważyć inne strategie. Jeśli każdy wierzchołek kupiłby licencję indywidualną, wówczas $w(f) = 5$ co jest równoznaczne kosztowi pięciu licencji indywidualnych. Jeśli centralny wierzchołek kupiłby tylko licencję indywidualną ($f(A) = 1$), a liście pozostałyby bez dostępu ($f(B) = f(C) = f(D) = f(E) = 0$), warunek nie byłby spełniony - żaden liść nie miałby sąsiada przyjmującego dla funckji f wartość 2.

Rysunek 3.2 ilustruje ten przykład i pokazuje, jak dominowanie rzymskie wskazuje optymalny wybór użytkownika pełniącego rolę lidera w centrum gwiazdy. W większych grafach kandydatów do roli właścicieli licencji grupowych jest więcej - ich odpowiedni dobór prowadzi już do właściwego problemu optymalizacji funkcji dominowania rzymskiego.



Rysunek 3.2: Kolory: biały = 0, pomarańczowy = 1, czerwony = 2. **(a)** Przypisanie optymalne (koszt 2). **(b)** Wszyscy z $f = 1$ (koszt 5). **(c)** $A = 1$, liście $f = 0$ (niespełniony warunek).

3.3 Złożoność obliczeniowa problemu

Dokładne dowody z zakresu złożoności obliczeniowej znajdują się w literaturze - np. pokazano NP-zupełność problemu dominowania rzymskiego poprzez redukcję z problemu pokrycia wierzchołków lub zbioru dominującego [22]. W kontekście naszego modelu oznacza to, że optymalne wyznaczenie użytkowników kupujących poszczególne licencje jest obliczeniowo trudne dla dużych sieci. Innymi słowy, nie istnieje znany algorytm wielomianowy, który gwarantowałby znalezienie rozwiązania minimalnego kosztu dla dowolnej struktury grafu znajomości - w najgorszym przypadku liczba kombinacji rośnie wykładniczo wraz z liczbą wierzchołków.

Konsekwencją NP-trudności jest także brak prostego schematu aproksymacyjnego dla problemu minimalizacji kosztów licencji. Ponieważ problem dominowania (minimalnego zbioru dominującego) jest APX-zupełny (dla konkretnych rodzajów grafów) [14], nie ma wielomianowego schematu aproksymacji (PTAS) gwarantującego dobre przybliżenie. Dla naszego problemu, który jest uogólnieniem dominowania, można oczekiwać podobnych ograniczeń - prawdopodobnie nie da się znaleźć w czasie wielomianowym rozwiązań bliższych optimum niż o czynnik $O(\ln n)$ w najgorszym przypadku. Proste heurystyki zachłanne mogą jednak dawać przyzwoite wyniki. Na przykład, heurystyka wybierająca iteracyjnie wierzchołek, który pokrywa najwięcej jeszcze niepokrytych sąsiadów (i dająca mu licencję grupową), jest jedną z implementacji algorytmu zachłannego dla zbioru dominującego i osiąga współczynnik $\approx (2 + \ln \Delta)$, gdzie Δ to maksymalny stopień grafu [23]. W najgorszym razie jest to $O(\ln n)$, ale dla wielu grafów rzeczywiste wyniki są lepsze niż ta pesymistyczna granica z ograniczeniem. Ważnym faktem jest też to, że problem pozostaje trudny nawet dla grafów o niewielkich stopniach - np. wykazano, że dla grafów o stopniu nie większym niż cztery wyznaczenie najmniejszego zbioru dominującego jest problemem APX-zupełnym [16, 14]. To implikuje, że ograniczenie maksymalnej liczby znajomych nie czyni problemu trywialnym. W kontekście subskrypcji oznacza to, że nawet w sieci gdzie każdy zna tylko kilka osób, optymalny dobór kto z kim powinien się połączyć we wspólnej licencji nadal może wymagać złożonych obliczeń.

Biorąc powyższe pod uwagę, w dalszej części badania główny nacisk zostanie położony na podejścia algorytmiczne, które biorą pod uwagę tę złożoność. Ponieważ nie istnieje wydajny algorytm dokładny dla ogólnego przypadku, rozważymy dwutorowo: (a) zastosowanie metod dokładnych dla umiarkowanych rozmiarów sieci oraz (b) zaprojektowanie i analizę algorytmów heurystycznych zdolnych dostarczać dobre rozwiązania dla większych sieci. W literaturze można już znaleźć próby wykorzystania technik optymalizacyjnych do problemu dominowania; przykładem jest praca Parra Inza i współautorów [24], w której zagadnienie sformułowano jako ILP i uzupełniono heurystykami naprawczymi. Tego typu konstrukcję można zaadaptować do omawianego problemu, wprowadzając zmienne decyzyjne wskazujące wybór typu licencji dla każdego użytkownika oraz odpowiednie ograniczenia. Z drugiej strony, heurystyki takie jak algorytm zachłanny, metody lokalnego przeszukiwania czy metaheurystyki - np. algorytmy genetyczne lub symulowane wyżarzanie - pozwalają szybko eksplorować przestrzeń możliwych konfiguracji licencji.

4. DANE TESTOWE

W celu przeprowadzenia szczegółowej analizy efektywności algorytmów optymalizujących zakup licencji w sieciach społecznościowych niezbędne było wykorzystanie różnorodnych danych testowych. Posłużyły do tego syntetycznie generowane grafy losowe oraz rzeczywiste fragmenty sieci społecznościowej. Pierwsza grupa stanowi kontrolowany zbiór danych sztucznych, pozwalający na symulowanie różnych scenariuszy topologicznych i analizę wpływu struktury sieci na działanie algorytmów. Druga grupa to rzeczywiste ego-sieci z platformy Facebook, umożliwiające weryfikację metod na prawdziwych danych społecznościowych.

Niniejszy rozdział koncentruje się na charakterystyce wykorzystanych danych, natomiast ich praktyczne zastosowanie zostanie szczegółowo przedstawione w części poświęconej opisowi przeprowadzonych eksperymentów.

4.1 Grafy syntetyczne

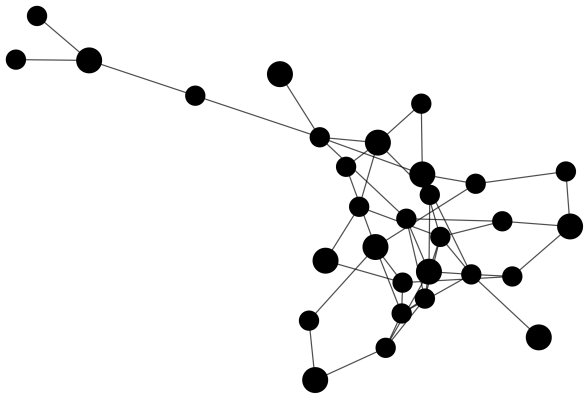
Do generowania danych syntetycznych wykorzystano trzy standardowe modele grafów: Erdős–Rényi (ER), Barabási–Albert (BA) oraz Watts–Strogatz (WS). W eksperymentach wykorzystywano kilka odrębnych konfiguracji rozmiaru w zależności od scenariusza eksperymentalnego:

- **Benchmark statyczny** obejmował grafy o $n \in \{20, 40, 60, 80, 100, 120, 140, 160, 180, 200, 300, 600, 1000\}$, generowane po trzy próbki na rozmiar. Rozdział 6 zawiera szczegółowe omówienie testów przeprowadzonych na tych grafach.
- **Symulacje dynamiczne** korzystały z mniejszego wachlarza wielkości: dla mutacji syntetycznych $n \in \{20, 40, 80, 160, 320, 640\}$, natomiast warianty realistyczne (pref_triadic, pref_pref, rand_rewire) operowały odpowiednio na zestawach $\{40, 80, 160, 320\}$ oraz $\{40, 80, 160, 320, 640\}$. Szczegółowe definicje tych mutacji oraz uzyskane wyniki znajdują się w rozdziale 7.
- **Rozszerzenia taryfowe** w wariancie statycznym analizowano na rozmiarach $n \in \{20, 50, 100, 200\}$, natomiast część dynamiczna korzystała z $n \in \{40, 80, 160\}$ przy krótszym horyzoncie czasowym. Szczegóły eksperymentów z rozszerzeniami taryfowymi opisano w rozdziale **??**.

4.1.1 Model Erdős–Rényi - klasyczne grafy losowe

Pierwszym rozważanym modelem jest klasyczny losowy graf Erdős–Rényi (ER) zaproponowany przez Erdős'a i Rényi'ego w 1959 roku [25]. W modelu tym rozpatruje się zbiór n wierzchołków, a każda z $\binom{n}{2}$ potencjalnych krawędzi pojawia się niezależnie z prawdopodobieństwem p. Parametrami modelu są więc n oraz p. W używanej implementacji zaadaptowano właśnie ten wariant $G(n, p)$, który prezentuje się w sposób przedstawiony na rys. 4.1.

Model ER stanowi istotny punkt odniesienia jako najprostszy model sieci pozbawiony struktury społecznościowej. Motywacją uwzględnienia go w testach jest możliwość porównania działania algorytmów na zupełnie przypadkowych sieciach z ich działaniem na bardziej uporząd-



Rysunek 4.1: Przykładowa realizacja grafu Erdős–Rényi

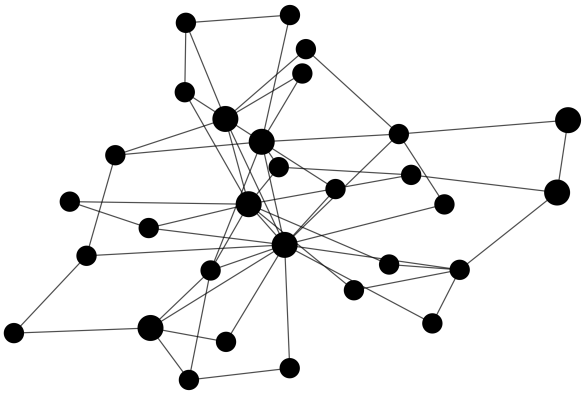
kowanych grafach takich jak grafy skalowane, małego świata oraz rzeczywiste. Choć prawdziwe sieci społecznościowe odbiegają od założeń pełnej losowości, mają zwykle wyższy poziom klasteryzacji węzłów i nierównomierny rozkład stopni wierzchołków, to jednak model $G(n, p)$ stanowi istotną podstawę porównawczą.

Z punktu widzenia właściwości, grafy ER cechują się stosunkowo niskim średnim współczynnikiem klasteryzacji. Współczynnik klasteryzacji mierzy, w jakim stopniu sąsiedzi danego wierzchołka są ze sobą połączeni. Dokładniej, oblicza się go jako stosunek liczby krawędzi faktycznie istniejących między sąsiadami wierzchołka do liczby wszystkich możliwych krawędzi między tymi sąsiadami. Średni współczynnik klasteryzacji dla całego grafu definiuje się jako średnią arytmetyczną wartości współczynników klasteryzacji po wszystkich wierzchołkach. Oczekiwana wartość współczynnika klasteryzacji jest równa p dla wystarczająco dużego n, a dla dostatecznie dużego p istnieje możliwość powstania jednej gigantycznej składowej spójności. Istnieje znana granica perkolacji, gdy $p > \frac{\ln n}{n}$, graf ER jest z dużym prawdopodobieństwem spójny - poniżej tego progu sieć rozpada się na wiele składowych [25]. Gęstość grafu, rozumiana jako odsetek istniejących krawędzi w stosunku do wszystkich możliwych, wynosi w tym modelu w przybliżeniu p. Przykładowo, dla $p = 0.1$ graf będzie miał ok. 10% maksymalnej liczby krawędzi. Rozkład stopni w modelu ER ma charakter dwumianowy, przy czym zbiega do rozkładu Poissona przy $n \to \infty$. Oznacza to, że w grafach tych nie występują węzły o niezwykle wysokich stopniach (tzw. huby), które są charakterystyczne dla wielu rzeczywistych sieci społecznościowych. W konsekwencji model ER nie oddaje wielu kluczowych właściwości takich sieci - stanowi jednak użyteczny model kontrolny, pozbawiony zjawisk typu mały świat czy skalowość, dzięki czemu można wyraźnie uwypuklić wpływ tych cech w innych modelach.

4.1.2 Model Barabási–Albert - sieci bezskalowe

Drugim wykorzystanym generatorem jest model Barabási–Albert (BA), wprowadzony przez Barabási'ego i Alberta w 1999 roku [26]. Model BA pozwala generować grafy o strukturze bezskalowej, których rozkład stopni wierzchołków ma rozkład wykładniczy. Tego typu sieci charakteryzują się istnieniem niewielkiej liczby wierzchołków o bardzo wysokim stopniu (tzw. hubów) oraz wie-

lu wierzchołków o małym stopniu - jest to cecha obserwowana w wielu rzeczywistych sieciach, w tym społecznościowych. Dla ilustracji, na rys. 4.2 przedstawiono przykładową realizację grafu BA, w której wyraźnie widoczne są huby. W przypadku sieci społecznościowych odnosi się to do tego, że niektórzy użytkownicy mogą mieć tysiące znajomych/obserwujących, podczas gdy większość ma ich kilkadziesiąt lub mniej.



Rysunek 4.2: Przykładowa realizacja grafu Barabási–Albert z widocznymi hubami

Parametrem wejściowym modelu Barabási–Albert jest przede wszystkim n - docelowa liczba węzłów w grafie - oraz m - liczba krawędzi, jakie dodaje każdy nowy węzeł. Procedura generowania rozpoczyna się od małego grafu startowego (np. klika złożona z m wierzchołków, aby zapewnić początkową spójność). Następnie dodaje się kolejno nowe wierzchołki; każdy nowy węzeł łączy się z m już istniejącymi wierzchołkami, przy czym prawdopodobieństwo połączenia z danym istniejącym węzłem jest proporcjonalne do jego bieżącego stopnia (tzw. reguła preferencyjnego łączenia, ang. preferential attachment). Mechanizm preferencyjnego przyłączania zwiększa prawdopodobieństwo dalszego wzrostu stopnia wierzchołków o wysokiej liczbie połączeń, czyli wierzchołki, które zyskały wiele połączeń na wcześniejszych etapach, mają większą szansę zdobyć kolejne połączenia, co prowadzi do wykładniczego rozkładu stopni wierzchołków.

Motywacją użycia modelu BA było odzwierciedlenie w danych testowych właściwości często spotykanej w sieciach społecznościowych i sieciach informacji - silnego zróżnicowania w stopniach węzłów. Dzięki grafom BA można przetestować algorytmy pod kątem radzenia sobie z obecnością hubów oraz z rozkładem stopni o ciężkim ogonie (ang. heavy-tailed distribution). Pojęcie ciężkiego ogona oznacza, że prawdopodobieństwo wystąpienia wierzchołków o bardzo dużym stopniu maleje stosunkowo wolno - w efekcie w sieci, obok wielu węzłów o niskim stopniu, pojawia się również pewna liczba hubów o ekstremalnie wysokim stopniu. Zjawisko to odróżnia sieci bezskalowe od np. grafów ER, w których prawdopodobieństwo pojawienia się węzłów o bardzo dużej liczbie sąsiadów jest znikome.

Grafy generowane modelem BA mają z reguły jedną składową spójności. Przy założeniu, że graf początkowy jest spójny i $m \geq 1$, każdy nowy wierzchołek dołącza do istniejącej struktury, więc sieć pozostaje spójna, a średni stopień w takim grafie wynosi około $2m$. Stąd gęstość gra-

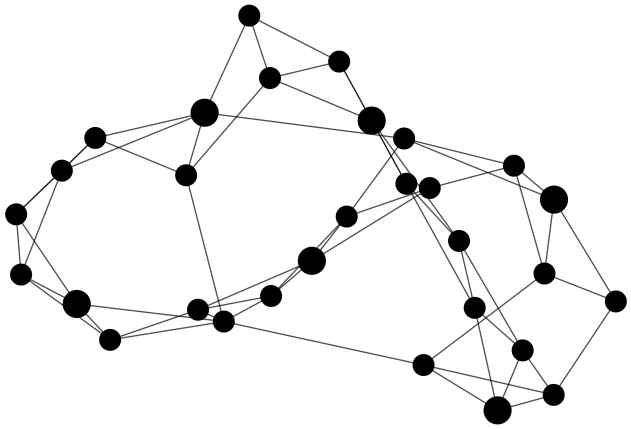
fu BA maleje wraz ze wzrostem n. Dla dużych grafów jest ona rzędu $\frac{2m}{n}$, co oznacza, że grafy te są rzadkie. W przeciwieństwie do modelu ER, współczynnik klasteryzacji grafów BA nie jest determinowany przez pojedynczy parametr w oczywisty sposób - klasyczny model BA generuje sieci o stosunkowo niskim średnim współczynnikiem klasteryzacji (niższym niż obserwowany w rzeczywistych sieciach społecznościowych), ponieważ nowe połączenia tworzone są głównie z hubami, co sprzyja tworzeniu gwiazd zamiast trójkątów. Istnieją modyfikacje modelu BA dodające mechanizmy triadycznego dosłączania, które zwiększają współczynnik klasteryzacji - jednak w czystej postaci model BA zazwyczaj skutkuje średnim współczynnikiem klasteryzacji malejącym wraz z rozmiarem grafu. Niemniej jednak, nawet przy relatywnie niskim współczynniku klasteryzacji, grafy BA zachowują własność małych średnich odległości. Powyższe cechy sprawiają, że grafy BA stanowią przydatny model testowy - oddają one istnienie hubów i krótkich odległości jak w wielu sieciach społecznościowych, choć nie odwzorowują silnego grupowania lokalnego.

4.1.3 Model Watts–Strogatz - graf małego świata

Trzecim fundamentalnym modelem wykorzystanym w pracy jest model Watts–Strogatz (WS), opisany przez Wattsa i Strogatza w 1998 roku [27]. Umożliwia on generowanie grafów małego świata (small-world networks), które łączą w sobie dwie istotne cechy: wysoki współczynnik klasteryzacji (podobny do obserwowanego w sieciach regularnych) oraz niską średnią odległość (podobnie jak w grafach losowych). Model ten odzwierciedla fakt, że w sieciach społecznościowych często występują silnie zżyte grupy znajomych, a jednocześnie dowolne dwie osoby są połączone relatywnie krótką ścieżką znajomości.

Generacja grafu WS wymaga dostarczenia mu trzech parametrów: liczby wierzchołków, stopnia każdego wierzchołka w początkowej regularnej strukturze oraz prawdopodobieństwa istnienia krawędzi. Procedura rozpoczyna się od utworzenia grafu regularnego, gdzie każdy wierzchołek jest połączony z k najbliższymi sąsiadami w cyklu (tj. tworzymy cykl z n węzłów, a następnie każdy węzeł łączymy z $\frac{k}{2}$ następnymi i $\frac{k}{2}$ poprzednimi na cyklu, zakładając dla uproszczenia, że k jest parzyste). Tak powstała sieć charakteryzuje się wysokim współczynnikiem klasteryzacji - węzły sąsiadujące na cyklu tworzą małe kliki. Jednocześnie początkowa średnia odległość jest stosunkowo duża, bo graf tuż przed modyfikacją krawędzi ma strukturę cyklu, więc dystans między węzłami oddalonymi na cyklu jest znaczny. Na rys. 4.3 zilustrowano przykładową realizację grafu Watts–Strogatz, w której widoczne są opisane wyżej właściwości.

Następnie, w modelu WS wprowadza się losowe przepięcia. Dla każdej krawędzi łączącej węzeł z jednym z $\frac{k}{2}$ najbliższych sąsiadów w cyklu. Dokonuje się, z prawdopodobieństwem p, przepięcia jednego końca tej krawędzi do losowo wybranego innego wierzchołka. Przepięcie polega na usunięciu oryginalnej krawędzi i dodaniu nowej krawędzi łączącej dany węzeł z innym losowym węzłem. W wyniku tych losowych przepięć, przy zachowaniu większości lokalnych połączeń cyklu, otrzymujemy graf, który dla małych p wciąż charakteryzuje się wysokim współczynnikiem klasteryzacji, ale jednocześnie kilka losowych połączeń długodystansowych znacząco skraca średnie odległości w sieci. Dla umiarkowanych wartości p (np. $p \approx 0.01$ czy 0.1) sieć uzyskuje bardzo



Rysunek 4.3: Przykładowa realizacja grafu Watts–Strogatz, w której widoczne są lokalne kliki oraz losowe połączenia długodystansowe skracające średnie odległości

małą średnią odległość - zbliżoną do grafów losowych - podczas gdy współczynnik klasteryzacji pozostaje o rząd wielkości wyższy niż w grafie Erdős–Rényi o porównywalnej gęstości.

W kontekście modelowania sieci społecznościowych, generator dla modelu WS dodano w celu odzwierciedlenia właściwości, których brakuje modelowi BA - mianowicie wysokiego lokalnego współczynnika klasteryzacji. Sieci społecznościowe cechują się tym, że znajomi często znają się nawzajem, tworząc kliki znajomych. Model WS pozwala symulować taką sytuację i sprawdzić, jak algorytmy radzą sobie np. z wykrywaniem społeczności czy zjawisk rozprzestrzeniania się informacji w warunkach silnego grupowania. Parametr k decyduje o początkowej gęstości połączeń lokalnych - większe k to więcej krawędzi lokalnych (każdy węzeł ma początkowo k sąsiadów), a zatem wyjściowo wyższy współczynnik klasteryzacji i gęstość. Parametr p kontroluje losowość grafu. Dla $p = 0$ otrzymujemy graf regularny, natomiast dla $p = 1$ graf staje się w dużej mierze losowy. W praktycznych zastosowaniach szczególnie interesujący jest zakres wartości p pomiędzy 0 a 1, w którym sieć łączy wysoki współczynnik klasteryzacji z niewielką średnią odległością między wierzchołkami.

Grafy WS generowane do testów miały parametry dobrane w taki sposób, aby możliwie dobrze odwzorowywać cechy typowe dla niedużych sieci społecznościowych. Uzyskiwane w ten sposób sieci charakteryzowały się relatywnie niską gęstością, ale jednocześnie wysokim współczynnikiem klasteryzacji, znacznie przewyższającym wartości obserwowane w losowych grafach ER o podobnej gęstości. Dzięki temu w grafach WS obecne są realistyczne zgrupowania lokalne, odpowiadające typowym kręgom znajomych w sieciach społecznościowych. Co istotne, sieci te z reguły pozostają spójne - niemal wszystkie wierzchołki należą do jednej dużej składowej spójności, a ewentualne izolowane węzły pojawiają się jedynie sporadycznie przy skrajnych ustawieniach parametrów. Taka struktura sprawia, że model WS stanowi dobre środowisko testowe.

4.2 Grafy rzeczywiste

Drugim zestawem danych testowych są rzeczywiste grafy pochodzące z sieci społecznościowej Facebook, a dokładniej zbiór Facebook Ego Network udostępniony w ramach Stanford Network Analysis Project (SNAP) [28]. Dane te zostały zebrane w 2012 roku przez J. McAuley i J. Leskovca z Uniwersytetu Stanforda w ramach badań nad automatycznym wykrywaniem kręgów społecznościowych [29]. Zbiór zawiera dziesięć tzw. ego-sieci, czyli sieci ego-centrycznych poszczególnych użytkowników Facebooka, pozyskane za zgodą uczestników przy użyciu dedykowanej aplikacji badawczej działającej w ramach platformy Facebook. Ego-sieć to sieć społecznościowa zbudowana z perspektywy pojedynczego użytkownika zwanego ego - węzłami są ego oraz wszystkie jego bezpośrednie znajome osoby, zaś krawędzie reprezentują relacje znajomości pomiędzy tymi znajomymi. W udostępnionych danych każda z dziesięciu sieci odpowiada innemu użytkownikowi i zawiera wyłącznie jego znajomych oraz powiązania między nimi. Węzeł ego nie jest jawnie ujęty jako wierzchołek w grafie i można go traktować jako ukrytą centralną jednostkę łączącą wszystkich znajomych. Innymi słowy, graf zapisany w pliku `X.edges` dotyczy tylko znajomych użytkownika X i relacji między nimi - sam X nie pojawia się w pliku jako węzeł.

4.2.1 Struktura danych

Każda ego-sieć udostępnione przez SNAP zapisana jest w osobnych plikach tekstowych, których nazwa odpowiada identyfikatorowi *ego* (np. `0.edges`, `0.circles`, `0.feat`, `0.egofeat` dla *ego* o ID=0). Struktura danych jest następująca:

Plik .edges - lista krawędzi w grafie znajomych danego ego. Każdy wiersz zawiera dwie liczby - identyfikatory dwóch różnych znajomych ego, między którymi istnieje relacja koleżeńska. Krawędzie te są nieskierowane. Ważną cechą jest to, że plik `.edges` nie zawiera połączeń od ego do jego znajomych - wierzchołek ego w ogóle nie występuje w tym pliku. Oznacza to, że rzeczywista sieć ego (gdyby uwzględnić w niej węzeł ego) miałaby dodatkowo krawędź łączącą ego z każdym z pojawiających się znajomych, jednak tych połączeń tutaj nie zapisano (są one domyślne - zakładamy, że ego jest połączone ze wszystkimi swoimi znajomymi). Pominięcie węzła ego jest zabiegiem celowym, pozwalającym skupić się na relacjach wewnątrz kręgów znajomych. Konsekwencją tego jest często podział grafu znajomych na kilka składowych - jeśli ego ma różne grupy znajomych wzajemnie się nieznających, to w pliku `.edges` każda taka grupa stanowi osobną składową spójności. Przykładowo, w ego-sieci `0.edges` znajomi tworzą 5 odrębnych składowych spójności. Oznacza to, że użytkownik o ID 0 miał około pięć niezależnych grup znajomych niepowiązanych ze sobą - dopiero poprzez jego osobę stawały się one pośrednio połączone.

Plik .circles - zestaw kręgów znajomych zdefiniowanych przez użytkownika. Każdy wiersz pliku reprezentuje jeden krąg towarzyski. Wiersz rozpoczyna się od nazwy kręgu - jednak w udostępnionych danych nazwy te zostały zanonimizowane lub pominięte, więc w praktyce każdy wiersz zaczyna się od identyfikatora kręgu albo pustej nazwy, po czym następuje lista ID użytkowników należących do tego kręgu. Kręgi mogą częściowo się pokrywać i nie muszą stanowić rozłącznych społeczności w sensie grafu - są to raczej dodatkowe metadane od ego, opisujące jak

kategoryzuje on swoich znajomych. Informacje te mogą być cenne pomocniczo, np. w oryginalnej pracy McAuley'ego i Leskovca posłużyły do oceny algorytmów automatycznie wykrywających społeczności [29].

Plik .feat - macierz cech atrybutów przypisanych do znajomych ego. Każdy wiersz odpowiada jednemu znajomemu i zawiera wektor wartości cech tej osoby. Cechy te mogą obejmować informacje z profilu Facebooka (np. miejsce pracy, szkoła, zainteresowania itp.). W udostępnionym zbiorze wartości atrybutów zostały zanonimizowane - nie znamy dokładnego znaczenia poszczególnych cech, jedynie ich binarne wartości (1 - użytkownik posiada daną cechę, 0 - nie posiada). Istnieje także plik `.featnames` zawierający oryginalne nazwy cech, ale w przypadku Facebooka nazwy te również zostały zanonimizowane (np. zamiast "szkoła: Uniwersytet Stanford" pojawia się anonimowa cecha 57). W niniejszej pracy dane atrybutów nie były wykorzystywane przez algorytmy i skupiono się tylko na strukturze grafów.

Plik .egofeat - wektor cech centralnego użytkownika, w tym samym formacie co pojedynczy wiersz pliku `.feat`, odnoszący się jednak do ego. Pozwala to porównać cechy ego z cechami jego znajomych. W kontekście naszych badań plik ten również nie był bezpośrednio wykorzystywany, poza podstawową walidacją danych.

W eksperymentach wykorzystano wszystkie dziesięć ego-sieci dostępnych w zbiorze SNAP. Liczba węzłów (po pominięciu centralnego ego) wynosiła odpowiednio 53, 62, 151, 169, 225, 334, 535, 748, 787 oraz 1035. Tak szeroki zestaw umożliwił ocenę algorytmów zarówno na niewielkich, jak i ponadtysięcznych grafach, w których różnice w gęstości i strukturze społeczności są wyraźnie widoczne.

4.2.2 Szczegółowy opis danych ze zbioru SNAP

W badanym zbiorze występują zarówno niewielkie sieci liczące poniżej setki węzłów (np. ID 3980: 53 wierzchołki, 252 krawędzie), jak i bardzo duże instancje przekraczające tysiąc węzłów (ID 0: 1035 wierzchołków, ponad 26 000 krawędzi). Ogólnie rzecz biorąc, większa liczba znajomych oznacza większe zróżnicowanie strukturalne: część ego-sieci składa się z kilku gęstych społeczności, podczas gdy inne są rozproszone i tworzą liczne składowe. Sumarycznie wszystkie dziesięć ego-sieci obejmuje 4039 unikalnych wierzchołków oraz 88 234 krawędzie [29], co dobrze oddaje skalę i złożoność sieci osobistych kontaktów.

Jak wspomniano, z powodu braku węzła ego w grafie znajomych większość ego-sieci dzieli się na więcej niż jedną składową spójności. W praktyce zazwyczaj istnieje jedna dominująca składowa, zawierająca największą grupę wzajemnie powiązanych znajomych, oraz kilka mniejszych składowych (np. dwu-lub kilkuosobowych grup) odpowiadających odizolowanym kręgom towarzyskim. Dla przykładu, sieć `107.edges` okazała się całkowicie spójna - wszyscy znajomi użytkownika 107 tworzyli jeden graf spójny. Natomiast sieć `0.edges` miała 5 składowych - co już sygnalizowano wcześniej. Wśród naszych analizowanych sieci: sieć `686.edges` jest spójna, sieć `414.edges` dzieli się na 2 składowe, sieć `698.edges` na 3, a sieć `3980.edges` na 4 składowe. Zwykle największa składowa obejmuje zdecydowaną większość wierzchołków. Taka struktura wskazuje na obecność

jednego głównego kręgu znajomych, uzupełnionego kilkoma mniejszymi grupami znajomości niepowiązanych z resztą.

Ego-sieci Facebooka cechują się na ogół wysokim współczynnikiem klasteryzacji, co zgodne jest z intuicją - znajomi konkretnej osoby często znają się nawzajem. W literaturze podaje się, że globalny współczynnik klasteryzacji dla całego grafu Facebooka jest stosunkowo niski [30]. W obrębie pojedynczej ego-sieci, gdzie wszyscy rozważani ludzie są znajomymi jednego ego, współczynnik klasteryzacji jest znacznie wyższy. Dla połączonej sieci 10 ego (4039 węzłów) średni współczynnik klasteryzacji wynosił aż 0.6055, co oznacza, że dwaj losowo wybrani znajomi danego ego mieli ponad 60% szans, by również być znajomymi między sobą. W naszych mniejszych sieciach wartości te różnią się w zależności od sieci, ale zwykle mieszczą się w przedziale 0.5-0.6 dla największej składowej (mniejsze składowe, np. dwuosobowe, mają współczynnik klasteryzacji równy 0 lub nieokreślony). Wysoki średni współczynnik klasteryzacji potwierdza istnienie silnych lokalnych powiązań - w grafie występuje wiele trójkątów (grup znajomych, z których każdy zna pozostałych). Przykładowo, jeśli ego posiada grupę bliskich przyjaciół ze szkoły, to prawdopodobne jest, że większość z nich zna się nawzajem, tworząc pełne podgrafy (kliki) o dużym współczynniku klasteryzacji.

W ego-sieciach Facebooka obserwuje się niskie wartości gęstości, co odzwierciedla ogólną rzadkość połączeń charakterystyczną dla sieci społecznościowych. Typowe wartości w badanym zbiorze mieszczą się od kilku do kilkunastu procent. Największe sieci (powyżej 700 węzłów) osiągają gęstości około 5%, natomiast mniejsze instancje (53 i 62 węzły) przekraczają 15% dzięki bardziej lokalnym połączeniom. W porównaniu do grafów losowych o podobnej skali ego-sieci mają wyższy współczynnik klasteryzacji (krawędzie nie są rozłożone przypadkowo, lecz skoncentrowane wewnątrz podzbiorów wierzchołków). Ta rzadka natura sieci społecznościowych jest istotna z punktu widzenia testowanych algorytmów, gdyż wiele z nich ma złożoności silnie zależne od liczby krawędzi (np. operacje przeszukiwania grafu lub znajdowania struktur klikowych mogą być szybsze w grafach rzadszych).

5. METODY ALGORYTMICZNE

W tym rozdziale przedstawiono wszystkie algorytmy zastosowane w pracy. Każdy algorytm opisano wraz z jego główną ideą, parametrami oraz analizą złożoności obliczeniowej.

Algorytmy zastosowane w pracy można podzielić na trzy grupy:

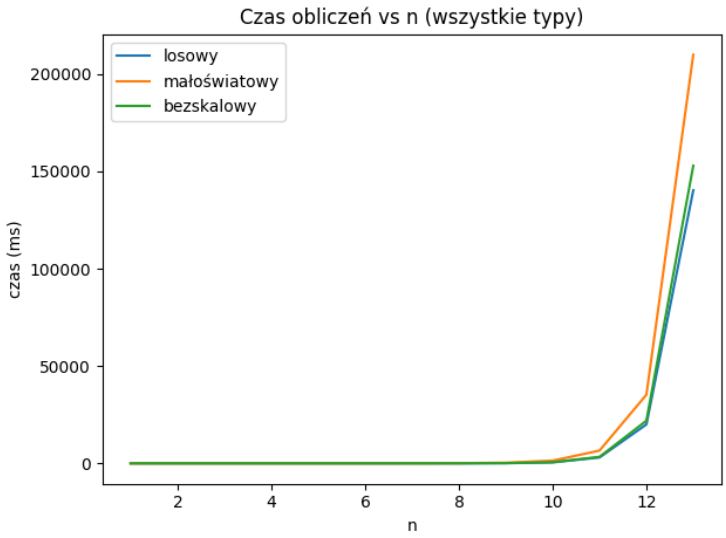
1. **Metody dokładne** – gwarantują znalezienie optymalnego rozwiązania: algorytm naiwny i programowanie całkowitoliczbowe (ILP).
2. **Heurystyki konstrukcyjne** – szybkie metody budujące rozwiązanie krok po kroku: algorytm zachłanny, zbiór dominujący, algorytm losowy.
3. **Metaheurystyki** – zaawansowane metody przeszukujące przestrzeń rozwiązań: algorytm genetyczny, przeszukiwanie tabu, algorytm mrówkowy, symulowane wyżarzanie.

5.1 Metody dokładne

5.1.1 Algorytm naiwny

Algorytm naiwny jest metodą dokładną. Przegląda wszystkie podziały zbioru wierzchołków na dopuszczalne grupy oraz wszystkie przypisania licencji. Wybierane jest rozwiązanie o najniższym koszcie. Liczba rozważanych konfiguracji rośnie wykładniczo, dlatego metoda ma znaczenie praktyczne tylko dla bardzo małych instancji.

Na rysunku 5.1 przedstawiono czasy obliczeń w funkcji liczby wierzchołków dla grafów losowych, małoświatowych i bezskalowych. Do 12 wierzchołków czas wykonania pozostaje akceptowalny dla celów eksperymentalnych. Z uwagi na wykładniczy wzrost czasu działania algorytmu metoda ta staje się niepraktyczna dla większych instancji, zwłaszcza w kontekście dostępności bardziej efektywnych algorytmów, takich jak programowanie całkowitoliczbowe (ILP).



Rysunek 5.1: Czas obliczeń algorytmu naiwnego w funkcji liczby wierzchołków n dla trzech typów grafów

Idea metody

1. Wygenerować wszystkie partycje zbioru V na niepuste bloki (każdy blok odpowiada kandydatowi na grupę licencyjną).
2. Dla każdego bloku w danej partycji wypisać wszystkie dopuszczalne pary (właściciel, licencja), czyli takie, które spełniają ograniczenia pojemności oraz sąsiedztwa.
3. Łączyć wybory z poszczególnych bloków w pełne przypisania i odrzucać te, które nie pokrywają wierzchołków lub łamią ograniczenia.
4. Obliczyć koszt na podstawie funkcji kosztu cost(f) (2.2) i zapamiętać najlepsze rozwiązanie.

Algorithm 1 Algorytm naiwny: pełny przegląd rozwiązań

Require: graf $G = (V, E)$, zbiory licencji L

if $|V| > 12$ **then**
    return przerwanie z powodu ograniczenia eksperymentalnego
end if
$best\_cost \leftarrow \infty$, $best \leftarrow \emptyset$
for każda partycja $P = \{P_1, \ldots, P_k\}$ zbioru V **do**
    for każde przypisanie $l \in L$ i właściciela w każdym P_i **do**
        if spełnione pojemności, sąsiedztwo i pełne pokrycie **then**
            policz koszt; zaktualizuj $best$, jeśli lepszy
        end if
    end for
end for
return $best$

Złożoność Algorytm generuje wszystkie partycje zbioru V na niepuste bloki, czyli wszystkie rozbicia odpowiadające potencjalnym zestawom grup licencyjnych. Takich partycji jest $B_{|V|}$, gdzie B_m oznacza liczbę Bella dla m elementów [31]. W implementacji nie ograniczamy z góry liczby bloków: niektóre typy licencji mogą pozostać nieużyte, dlatego dopuszczamy zarówno rozbicia na jedną grupę, jak i na $|V|$ grup jednowierzchołkowych. Gdyby liczba bloków była stała i wynosiła k, odpowiadałyby jej liczby Stirlinga drugiego rodzaju $\left\{{|V| \atop k}\right\}$; tutaj sumujemy je po wszystkich k, stąd pojawiają się liczby Bella. Samo wygenerowanie partycji kosztuje więc $\Theta(B_{|V|})$.

Dla partycji o k blokach o rozmiarach $s_1, s_2, \ldots, s_k$ rozpatrujemy każdą kombinację wyboru właściciela (co najwyżej s_i opcji na blok i) i dopuszczalnego typu licencji (co najwyżej T opcji, gdzie $T = |L|$). Prowadzi to do górnego oszacowania

$$\prod_{i=1}^{k} (s_i \cdot T) = T^k \cdot \prod_{i=1}^{k} s_i$$

konfiguracji dla pojedynczej partycji. Całkowity koszt można oszacować przez $O\big(B_{|V|} \cdot T^{|V|} \cdot 3^{|V|/3}\big)$, ponieważ $\prod s_i \le 3^{|V|/3}$. Wzrost jest superwykładniczy, gdzie asymptotycznie $\log B_n = n \log n - n \log\log n - n + O(n/\log n)$. W praktyce ograniczenia licencyjne i warunek sąsiedztwa znacząco zmniejszają liczbę rozważanych konfiguracji, lecz nawet wtedy pełny przegląd jest użyteczny wyłącznie dla bardzo małych grafów (rzędu kilku do kilkunastu wierzchołków).

Liczby Bella można zapisać wzorami

$$B_n = \sum_{k=0}^{n} \left\{ {n \atop k} \right\} \quad \text{oraz} \quad B_n = \frac{1}{e} \sum_{k=0}^{\infty} \frac{k^n}{k!},$$

a zliczają one wszystkie rozbiory zbioru na dowolną liczbę niepustych bloków (bloki są nierozróżnialne). Dla porównania liczba wszystkich podzbiorów zbioru n-elementowego wynosi jedynie 2^n, więc nawet tak uproszczone oszacowanie rośnie wolniej niż pełna liczba partycji.

5.1.2 Programowanie całkowitoliczbowe (ILP)

Formułujemy problem z rozdziału 2.2 jako minimalizację kosztu przy ograniczeniach liniowych z binarnymi zmiennymi. Model jest dokładny; solver zwraca rozwiązanie optymalne, jeśli zakończy działanie poprawnie.

Parametry i zmienne Korzystamy z rodziny licencji z (2.1), $\mathcal{L} = \{\ell_t = (c_t, m_t, k_t) : t = 1, \ldots, T\}$. Oznaczamy $N[i] = N(i) \cup \{i\}$. Dla $i \in V$, $t \in \{1, \ldots, T\}$ definiujemy:

- $a_{i,t} \in \{0, 1\}$: $a_{i,t} = 1$ gdy węzeł i kupuje licencję typu t i aktywuje grupę,
- $x_{i,j,t} \in \{0, 1\}$ dla $j \in N[i]$: $x_{i,j,t} = 1$ gdy j należy do grupy właściciela i typu t.

Zmienna $x_{i,j,t}$ odwzorowuje przypisanie odbiorcy do właściciela; zbiory ról są indukowane przez wartości (a, x).

Model

$$\text{min} \quad \sum_{i \in V} \sum_{t=1}^{T} c_t \, a_{i,t} \tag{5.1}$$

$$\text{pod warunkami:} \quad \sum_{i \in N[j]} \sum_{t=1}^{T} x_{i,j,t} = 1 \qquad \forall j \in V \tag{5.2}$$

$$m_t \, a_{i,t} \leq \sum_{j \in N[i]} x_{i,j,t} \leq k_t \, a_{i,t} \qquad \forall i \in V, \; \forall t \tag{5.3}$$

$$x_{i,i,t} = a_{i,t} \qquad \forall i \in V, \; \forall t \tag{5.4}$$

$$\sum_{t=1}^{T} a_{i,t} \leq 1 \qquad \forall i \in V \tag{5.5}$$

Znaczenie ograniczeń: (5.2) pokrywa każdy węzeł dokładnie raz; (5.3) wymusza przedziały wielkości grup; (5.4) zapewnia udział właściciela; (5.5) ogranicza do jednej licencji na węzeł (dowolnego typu). Ograniczenia pokrywają warunki wykonalności z rozdziału 2.2.

Złożoność Liczba zmiennych: $|V|T$ dla $a_{i,t}$ oraz $T \sum_i |N[i]|$ dla $x_{i,j,t}$, łącznie $O\big(T(|E| + |V|)\big)$. Liczba ograniczeń: $|V|$ dla (5.2), $2|V|T$ dla (5.3), $|V|T$ dla (5.4), $|V|$ dla (5.5). Model służy jako punkt odniesienia dla małych i średnich instancji.

Implementacja Implementacja w Python (PuLP + CBC) odzwierciedla powyższy model. Stosuje się eliminację niemożliwych par (i, t), gdy $m_t > |N[i]|$ lub $k_t < 1$, oraz dla $k_t = 1$ tworzy się wyłącznie zmienne $x_{i,i,t}$. Przy rekonstrukcji rozwiązania wartości binarne interpretowane są progowo $> 0{,}5$. W razie statusu innego niż rozwiązanie całkowite zastosowano awaryjny wariant „same licencje indywidualne” (procedura poza modelem).

5.2 Heurystyki konstrukcyjne

5.2.1 Algorytm zachłanny

Algorytm zachłanny to szybka heurystyka, która buduje rozwiązanie krok po kroku, w każdym kroku wybierając lokalnie najlepszą opcję. Algorytm nie gwarantuje znalezienia optymalnego rozwiązania, ale jest bardzo szybki i daje zazwyczaj w miarę dobre wyniki.

Idea metody Algorytm działa według następującej strategii:

1. Sortuje wierzchołki nierosnąco według liczby sąsiadów (stopnia wierzchołka).
2. Dla każdego wierzchołka sprawdza, czy może być właścicielem grupy.
3. Wybiera typ licencji i rozmiar grupy tak, aby zminimalizować stosunek kosztu do rozmiaru grupy.
4. Dodaje członków do grupy wybierając wierzchołki o największej liczbie sąsiadów.
5. Powtarza proces dla wszystkich niepokrytych wierzchołków.

Algorytm nie ma parametrów do dostrajania. Wszystkie decyzje są podejmowane deterministycznie na podstawie struktury grafu i kosztów licencji, choć w przypadku wierzchołków o tym samym stopniu kolejność wyboru może wpływać na wynik końcowy.

Sortowanie według stopnia wierzchołka (liczby sąsiadów) sprawdza się dobrze w praktyce, ponieważ wierzchołki o wysokim stopniu mogą tworzyć większe, bardziej efektywne grupy licencyjne.

Algorithm 2 Algorytm zachłanny

Require: graf $G = (V, E)$, typy licencji L

posortuj wierzchołki nierosnąco według stopnia
$niepokryte \leftarrow V$
for każdy wierzchołek v w posortowanej kolejności **do**
    if v już pokryty **then continue**
    end if
    znajdź dostępnych sąsiadów v wśród niepokrytych
    for każdy typ licencji $l_t \in L$ **do**
        oblicz efektywność: $koszt_{l_t}/rozmiar\_grupy$
    end for
    wybierz licencję i członków grupy o najlepszej efektywności
    utwórz grupę z v jako właścicielem
    usuń członków grupy z $niepokryte$
end for
return utworzone grupy

Złożoność i zastosowanie Algorytm ma złożoność czasową $O(|V|\,T + |E| \log |V|)$, gdzie $|V|$ to liczba wierzchołków, $|E|$ to liczba krawędzi, a T to liczba typów licencji. Algorytm zachłanny jest bardzo szybki i daje stabilne wyniki. Z tego powodu często używany był jako:

- podstawowa metoda do porównywania z innymi algorytmami,
- źródło rozwiązania początkowego dla bardziej zaawansowanych metod,
- szybka metoda dla dużych grafów, gdzie inne algorytmy są zbyt wolne.

Wadą algorytmu jest to, że podejmując lokalnie najlepsze decyzje, może przegapić lepsze rozwiązania globalne.

5.2.2 Heurystyka zbioru dominującego

Heurystyka korzysta z konstrukcji *minimalnych względem inkluzji* zbiorów dominujących. Zbiór dominujący jest minimalny, gdy usunięcie dowolnego jego wierzchołka powoduje utratę własności dominowania [12]. W kontekście licencjonowania dominatorem nazywa się wierzchołek wybrany do zbioru dominującego, który stanie się właścicielem grupy licencyjnej i pokryje siebie oraz swoich sąsiadów. Dla przejrzystości zapisu wprowadzamy zbiór F oznaczający wierzchołki, które nie zostały jeszcze przypisane ani do zbioru dominującego, ani do żadnej grupy licencyjnej.

Wskaźnik wyboru kandydata opiera się na coverage(v) oraz min_cpn(v).

- U to zbiór wierzchołków jeszcze niepokrytych, który jest bezpośrednią kopią V na początku działania algorytmu. W miarę tworzenia grup wierzchołki są usuwane z U.
- coverage$(v) = |N[v] \cap U|$ to liczba jeszcze niepokrytych węzłów, które może pokryć v.
- min_cpn(v) to minimalny koszt na węzeł dla v, liczony po wszystkich licencjach i dopuszczalnych rozmiarach grupy:

$$\text{min\_cpn}(v) = \min_{l_t \in L} \; \min_{s \in [m_t,\, \min\{k_t,\, \text{coverage}(v)\}]} \frac{c_t}{s}.$$

Interpretacja: wybieramy dla v najkorzystniejszą licencję i rozmiar grupy, które dają najniższy koszt jednostkowy.

Algorithm 3 Zbiór dominujący z budowaniem grup

Require: graf $G = (V, E)$, zbiory licencji L

$U \leftarrow V$, $D \leftarrow \emptyset$, $R \leftarrow V$
while $U \neq \emptyset$ **do**
    dla każdego $v \in V$ policz coverage$(v) = |(N[v] \cap U|$ oraz min_cpn(v)
    wybierz u maksymalizujące coverage$(v)/$ min_cpn(v); jeśli brak rozstrzygnięcia wybierz dowolne $u \in U$
    $D \leftarrow D \cup \{u\}$, $U \leftarrow U \setminus N[u]$
end while
posortuj D nierosnąco według stopni wierzchołków
for każde $u \in D$ **do**
    $S \leftarrow N[u] \cap F$
    wybierz najtańszą dopuszczalną licencję dla u i wyznacz grupę $P \subseteq S$ o największym dopuszczalnym rozmiarze; w ostateczności przydziel licencję indywidualną
    $F \leftarrow F \setminus P$
end for
return utworzone grupy

Złożoność i zastosowanie Faza wyboru dominatorów w każdej rundzie przechodzi po wszystkich wierzchołkach, ich sąsiadach i typach licencji, co daje koszt rzędu $O(|V||E|T)$. Faza budowania grup dla każdego dominatora sortuje kandydatów i sprawdza warianty licencji. W gęstych grafach rośnie to do $O(|V|^3 T \log |V|)$, w rzadkich pozostaje bliżej $O(|V||E|T)$. Heurystyka w krótkim czasie wyznacza pełne pokrycie i dostarcza jakościowe rozwiązanie początkowe dla metod ulepszających.

5.2.3 Algorytm losowy

Algorytm losowy pełni rolę metody odniesienia do oceny jakości rozwiązań generowanych przez inne algorytmy. Weryfikuje poprawność implementacji i stanowi stochastyczny punkt

odniesienia. Wierzchołki są przetwarzane w losowej kolejności, a wybór licencji i składu grupy jest losowy w granicach ograniczeń pojemności i sąsiedztwa.

Idea metody

1. Losowana jest kolejność przetwarzania wierzchołków.
2. Dla bieżącego wierzchołka wyznaczany jest zbiór kandydatów obejmujący jego oraz nieprzydzielonych sąsiadów.
3. Jeżeli istnieje dopuszczalna licencja, losowany jest typ licencji, rozmiar grupy oraz członkowie grupy z dostępnych kandydatów. Rozważano wariant wyboru zgodnie z najlepszym kosztem jednostkowym $\min_{l,s} c_l/s$, analogicznie do heurystyki zbioru dominującego. W tej pracy przyjęto jednak minimalną deterministyczność i maksymalną losowość do wyznaczenia górnego ograniczenia efektywności metaheurystyk.
4. W przeciwnym razie przydzielana jest najtańsza dostępna licencja indywidualna.
5. Kroki są powtarzane do pełnego pokrycia grafu.

Uwagi o losowości Celem było uzyskanie szerokiego spektrum wyników, aby zobaczyć, jak zachowuje się heurystyka w losowych warunkach spełniających ograniczenia problemu. Nie stosowano wariantu wyboru licencji według najlepszego kosztu na węzeł, ponieważ prowadziłoby to do wyniku deterministycznego i zawężenia rozkładu rezultatów. Rozważano także uruchamianie algorytmu wielokrotnie i wybieranie najlepszego otrzymanego wyniku spośród dużej liczby uruchomień. Ze względu na dużą losowość w doborze licencji i składzie grup okazało się jednak, że średnia wartość jakości rozwiązania z wielu uruchomień nie różni się istotnie od wyniku pojedynczego najlepszego przebiegu. Z tego powodu algorytm uruchamiany jest tylko raz z możliwością ustawienia ziarna generatora liczb losowych, co pozwala odtwarzać eksperymenty.

Algorithm 4 Losowy dobór licencji i składu grupy

Require: graf $G = (V, E)$, zbiory licencji L

$U \leftarrow V$, $\pi \leftarrow$ losowa permutacja V
for node w kolejności π **do**
    if $node \notin U$ **then continue**
    end if
    $S \leftarrow N[node] \cap U$
    if istnieje licencja l_t z $m_t \leq |S|$ **then**
        losuj l_t oraz rozmiar $s \in \left[m_t, \min\{|S|, k_t\}\right]$, następnie losuj członków grupy z S ▷ wariant kierowany: można zastąpić wyborem $\arg\min_{l_t, s} c_t/s$
    else
        przydziel najtańszą licencję indywidualną
    end if
    dodaj grupę, usuń jej członków z U
end for
while $U \neq \emptyset$ **do** przydziel najtańszą licencję indywidualną i usuń węzeł z U
end while

Złożoność i zastosowanie Każdy wierzchołek i jego sąsiedzi są przeglądani co najwyżej raz, a przy każdej próbie losowania licencji przeglądane są wszystkie typy licencji, co daje koszt rzędu $O\big(T(|E|+|V|)\big)$. W gęstych grafach upraszcza się to do $O(T|V|^2)$. Algorytm służy jako benchmark stochastyczny oraz kontrola jakości innych metod. Wariant kierowany $\min c_t/s$ nie zmienia rzędu złożoności i może być użyty pomocniczo do analizy wrażliwości.

5.3 Metaheurystyki

Metaheurystyki to zaawansowane algorytmy przeszukujące przestrzeń rozwiązań w sposób inteligentny. W przeciwieństwie do heurystyk konstrukcyjnych, które budują rozwiązanie od zera, metaheurystyki zaczynają od pewnego rozwiązania i systematycznie je poprawiają.

Dobór parametrów Parametry metaheurystyk zostały dobrane eksperymentalnie na podstawie testów na grafach różnych rozmiarów.

Operacje modyfikacji rozwiązania Metaheurystyki poprawiają rozwiązanie stosując następujące operacje:

- zmiana typu licencji używanej przez właściciela grupy
- przeniesienie członka z jednej grupy do drugiej
- zamiana miejscami dwóch członków z różnych grup
- scalanie dwóch grup w jedną lub rozdzielanie na dwie dopuszczalne.

5.3.1 Algorytm genetyczny

Algorytm genetyczny utrzymuje populację pełnych przydziałów licencyjnych i z pokolenia na pokolenie ulepsza je, korzystając z losowych mutacji i krzyżowania par rodziców [32, 33]. Zaczyna od kilku rozwiązań zachłannych i losowych, a następnie w każdej generacji wybiera najlepsze osobniki (elita), losuje rodziców metodą turniejową i tworzy potomstwo przez krzyżowanie lub mutację. Słabsze rozwiązania są stopniowo zastępowane lepszymi, a algorytm zapamiętuje najlepszy znaleziony koszt.

Parametry

- **Wielkość populacji** $P_{GA} = 30$ - liczba rozwiązań utrzymywanych w każdej generacji.
- **Liczba pokoleń** $N_{GA} = 40$ - maksymalna liczba iteracji ewolucji.
- **Udział elity** $\alpha_{GA} = 20\%$ - część najlepszych osobników kopiowana bez zmian do kolejnego pokolenia.
- **Prawdopodobieństwo krzyżowania** $p_{c,GA} = 60\%$ - przy tej szansie dziecko powstaje przez połączenie dwóch rodziców; w przeciwnym razie wykonywana jest mutacja.

Algorithm 5 Algorytm genetyczny

Require: graf $G = (V, E)$, typy licencji L

```
utwórz populację początkową (zachłanny + losowe rozwiązania)
for każde pokolenie do
    oceń wszystkie rozwiązania (funkcja kosztu)
    zachowaj elitę (najlepsze rozwiązania)
    while populacja niepełna do
        if losowanie krzyżowania then
            wybierz dwóch rodziców (selekcja turniejowa)
            skrzyżuj rodziców (połącz efektywne grupy)
        else
            wybierz rozwiązanie i zmutuj (operacje sąsiedztwa)
        end if
        dodaj potomka do nowej populacji
    end while
    zaktualizuj najlepsze znalezione rozwiązanie
end for
return najlepsze rozwiązanie
```

Złożoność i zastosowanie Inicjalizacja populacji korzysta z jednego osobnika otrzymanego za pomocą algorytmu zachłannego i $P - 1$ losowych rozwiązań, co kosztuje około $O\big(P \cdot (|V|T + |E| \log |V|)\big)$. Każda generacja sortuje populację ($O(P \log P)$), a następnie tworzy nowe pokolenie. Mutacje wywołują ograniczoną liczbę operatorów sąsiedztwa (zmiana typu licencji, przeniesienie

członka, zamiana miejscami, scalanie lub podział grup), a krzyżowanie w razie potrzeby uruchamia heurystykę zachłanną na podgrafie, co razem daje koszt rzędu $O(|V|T + |E| \log |V|)$ na potomka. Łącznie otrzymujemy $O\big(G \cdot P \cdot (|V|T + |E| \log |V|)\big)$ w najgorszym przypadku. Algorytm działa wolniej od prostych heurystyk, ale potrafi znacząco poprawić ich wyniki i służy jako główna metoda poszukiwania wysokiej jakości rozwiązań, gdy możemy poświęcić na optymalizację więcej czasu.

5.3.2 Przeszukiwanie tabu

Algorytm tabu rozpoczyna działanie od rozwiązania uzyskanego heurystyką zachłanną, a następnie iteracyjnie przeszukuje lokalne sąsiedztwo. Sąsiadem nazywa się rozwiązanie otrzymane przez pojedynczą operację mutacji, na przykład zmianę typu licencji właściciela, przeniesienie członka między grupami, zamianę członków lub scalenie i podział grup. Lista tabu przechowuje podpisy ostatnich rozwiązań lub ruchów i zakazuje wyboru kandydatów, którzy prowadzą do niedawno odwiedzonych stanów. Kryterium aspiracji pozwala pominąć zakaz, gdy kandydat poprawia najlepszy dotąd koszt. Mechanizm ten ogranicza krótkie cykle i równoważy lokalne doskonalenie z eksploracją nowych przydziałów licencji [34].

Parametry

- **Maksymalna liczba iteracji** $I_{tabu} = 1000$.
- **Długość listy tabu** $L_{tabu} = 20$ elementów.
- **Liczba sąsiadów na iterację** $k_{tabu} = 10$.

Algorithm 6 Przeszukiwanie tabu

Require: graf $G = (V, E)$, typy licencji L

$aktualne \leftarrow$ rozwiązanie początkowe wyznaczone algorytmem zachłannym
$najlepsze \leftarrow aktualne$
$lista\_tabu \leftarrow$ pusta kolejka o stałej długości L; wstaw podpis $aktualne$
for każdą iterację **do**
    wygeneruj sąsiedztwo $aktualne$ przez operacje mutacji
    wybierz najlepszego kandydata, którego podpis nie znajduje się na $lista\_tabu$ lub który poprawia $najlepsze$ (aspiracja)
    if wybrano kandydata **then**
        $aktualne \leftarrow$ kandydat; zaktualizuj $lista\_tabu$
        if $aktualne$ lepsze niż $najlepsze$ **then**
            $najlepsze \leftarrow aktualne$
        end if
    else
        break
    end if
end for
return $najlepsze$

Złożoność i zastosowanie Inicjalizacja początkowego stanu grafu obejmuje jedno uruchomienie heurystyki zachłannej $O(|V|T+|E|\log|V|)$. W każdej z I iteracji generowanych jest do k sąsiadów. Ocena kandydata obejmuje sprawdzenie pojemności, pokrycia i zgodności z sąsiedztwem w grafie, co kosztuje około $O(|V|T+|E|)$. Łączna złożoność obliczeniowa wynosi w przybliżeniu $O(I \cdot k \cdot (|V|T+|E|))$. Przeszukiwanie tabu dobrze sprawdza się jako metoda ulepszająca: poprawia rozwiązania wyjściowe przy umiarkowanym czasie obliczeń i ogranicza powroty do niedawno odwiedzonych stanów dzięki liście tabu.

5.3.3 Algorytm mrówkowy

Algorytm mrówkowy buduje wiele rozwiązań równolegle. Każda mrówka konstruuje przydział licencji, kierując się siłą śladów feromonowych (informacja o dotychczas dobrych wyborach) oraz heurystyką preferującą wierzchołki o dużym stopniu i licencje o dobrym stosunku pojemności do ceny [35]. Po każdej iteracji, czyli po zakończeniu budowy rozwiązań przez wszystkie mrówki w danym kroku, feromony parują, a najlepsze dotąd rozwiązanie wzmacnia ścieżki feromonowe, czyli w przypadku naszej implementacji prawdopodobieństwo wyboru konkretnych właścicieli i typów licencji. Dzięki temu kolejne mrówki chętniej eksplorują obiecujące fragmenty przestrzeni rozwiązań (tj. wierzchołków i typów licencji).

Parametry

- **Waga feromonu** $\alpha = 1.0$ – określa, jak mocno mrówki ufają dotychczasowym śladom.
- **Waga heurystyki** $\beta = 2.0$ – wzmacnia lokalnie korzystne decyzje, czyli takie, które przy danym wierzchołku pozwalają dobrać licencję z wysoką pojemnością i niskim kosztem jednostkowym. Koszt jednostkowy to cena licencji l_t podzielona przez liczbę użytkowników w grupie lub cenę licencji indywidualnej.
- **Tempo parowania** $\rho = 0.5$ – część feromonu usuwana po każdej iteracji.
- **Prawdopodobieństwo wyboru zachłannego** $q_0 = 0.9$ – z tą szansą mrówka wybiera najlepszą dostępną opcję, w przeciwnym razie losuje wierchołek i licencje proporcjonalnie do wag wyliczonych w trakcie budowy rozwiązania.
- **Liczba mrówek** $A = 20$ – ile rozwiązań konstruujemy równolegle w jednej iteracji.
- **Maksymalna liczba iteracji** $N_{ACO} = 100$ – ile razy aktualizujemy feromony.
- **Losowe ziarno** (opcjonalne) – pozwala odtworzyć przebieg eksperymentu.

Algorithm 7 Algorytm mrówkowy

Require: graf $G = (V, E)$, typy licencji L

zainicjalizuj feromony τ (dla par wierzchołek-licencja)
zainicjalizuj heurystyki η (na podstawie stopni wierzchołków i kosztów licencji)
$najlepsze \leftarrow$ rozwiązanie początkowe (zachłanne)
for każdą iterację **do**
    for każdą mrówkę **do**
        $niepokryte \leftarrow V$
        while $niepokryte \neq \emptyset$ **do**
            wybierz właściciela na podstawie τ i η (reguła wyboru lub ruletka)
            wybierz typ licencji na podstawie τ i η
            utwórz grupę, usuń członków z $niepokryte$
        end while
        if mrówka znalazła lepsze rozwiązanie **then**
            $najlepsze \leftarrow$ rozwiązanie mrówki
        end if
    end for
    wyparuj część feromonów: $\tau \leftarrow \tau \cdot (1 - evaporation)$
    wzmocnij feromony na ścieżce $najlepsze$: $\tau \leftarrow \tau + 1/koszt$
end for
return $najlepsze$

Złożoność i zastosowanie Inicjalizacje feromonów i heurystyk charakteryzuje złożoność obliczeniowa $O(|V|T)$. Pojedyncza konstrukcja rozwiązania wymaga odwiedzenia przez mrówkę każdego wierzchołka co najwyżej raz. W każdym kroku mrówka wybiera wierzchołek, który stanie się właścicielem nowej grupy licencyjnej; w tym celu ocenia wszystkie niepokryte wierzchołki jako potencjalnych właścicieli oraz wszystkie typy licencji dostępne w zbiorze L. Po wskazaniu właściciela trzeba posortować jego sąsiadów w grafie G według wag feromonowo-heurystycznych, aby zdecydować, którzy użytkownicy dołączą do grupy. Te operacje prowadzą do złożoności obliczeniowej rzędu $O(|V|^2T + |E| \log |V|)$ na jedną mrówkę. Algorytm wykonujący I iteracji z A mrówkami ma więc złożoność $O\big(I \cdot A \cdot (|V|^2T + |E| \log |V|)\big)$. Metoda jest bardziej złożona obliczeniowo niż tabu i algorytm zachłanny, ale pozwala eksplorować wiele alternatywnych konfiguracji i stopniowo wzmacniać najlepsze z nich.

5.3.4 Symulowane wyżarzanie

Symulowane wyżarzanie rozpoczyna od dowolnego dopuszczalnego przydziału; w implementacji korzystamy z rozwiązania wygenerowanego heurystyką zachłanną. W każdej iteracji losowana jest jedna z operacji modyfikacji opisanych na początku sekcji metaheurystyk. Jeżeli wylosowana operacja prowadzi do stanu naruszającego ograniczenia, losujemy kolejną do chwili

znalezienia dopuszczalnego kandydata (maksymalnie kilkanaście prób na iterację). Nowy stan jest akceptowany zawsze, gdy obniża koszt całkowity przydziału licencji, a czasami także wtedy, gdy go pogarsza - z prawdopodobieństwem zależnym od bieżącej temperatury T i różnicy kosztów całkowitych rozwiązania bieżącego i kandydata [36]. Temperatura maleje według ustalonego współczynnika, a gdy liczba kolejnych iteracji bez poprawy najlepszej wartości przekroczy limit stagnacji S, jest dodatkowo dzielona przez dwa. Dzięki temu metoda potrafi opuszczać lokalne minima funkcji kosztu w sąsiedztwie zdefiniowanym przez powyższe operacje i stopniowo stabilizuje się w pobliżu dobrego rozwiązania.

Parametry

- **Temperatura początkowa** $\tau_0 = 100.0$ - stosunkowo wysoka wartość dobrana eksperymentalnie; przy takich temperaturach różnice kosztów rzędu kilku jednostek są często akceptowane, co pozwala na szeroką eksplorację przestrzeni rozwiązań na początku działania algorytmu.
- **Współczynnik chłodzenia** $\gamma = 0.995$ - w każdej iteracji temperatura jest mnożona przez γ.
- **Temperatura minimalna** $\tau_{\text{min}} = 0.001$ - po jej osiągnięciu algorytm kończy działanie.
- **Maksymalna liczba iteracji** $N_{SA} = 20\,000$ - górne ograniczenie liczby kroków.
- **Limit stagnacji** $S = 2\,000$ - liczba kolejnych iteracji bez poprawy najlepszego dotąd kosztu; parametr dobrany empirycznie (mniejsza wartość szybciej wymusza zawężenie przedziału poszukiwań rozwiązań), po jej przekroczeniu temperatura jest dodatkowo dzielona przez 2.

Algorithm 8 Symulowane wyżarzanie

Require: graf $G = (V, E)$, typy licencji L

$aktualne \leftarrow$ rozwiązanie początkowe (np. zachłanne)
$najlepsze \leftarrow aktualne$
$temperatura \leftarrow T_0$ (początkowa temperatura)
for każdą iterację **do**
    if $temperatura < T_{min}$ **then break**
    end if
    wybierz losową operację sąsiedztwa
    $kandydat \leftarrow$ wynik operacji sąsiedztwa
    $\Delta \leftarrow koszt(kandydat) - koszt(aktualne)$
    if $\Delta \leq 0$ LUB $random() <$ exp$(-\Delta / temperatura)$ **then**
        $aktualne \leftarrow kandydat$
        if $koszt(kandydat) < koszt(najlepsze)$ **then**
            $najlepsze \leftarrow kandydat$
        end if
    end if
    $temperatura \leftarrow temperatura \cdot cooling\_rate$
end for
return $najlepsze$

Złożoność i zastosowanie Inicjalizacja korzysta z rozwiązania uzyskanego heurystyką zachłanną, co kosztuje $O(|V|T + |E| \log |V|)$, gdzie T oznacza liczbę typów licencji; samo symulowane wyżarzanie może jednak wystartować z dowolnego dopuszczalnego stanu, a heurystyka zapewnia po prostu solidny punkt startowy. W każdej iteracji wykonujemy do kilkunastu prób wygenerowania dopuszczalnego sąsiada z katalogu operacji, a zaakceptowany kandydat przechodzi pełną walidację pokrycia i ograniczeń, co ma złożoność około $O(|V|T + |E|)$. W rezultacie całkowity koszt jednego przebiegu wynosi $O\big(I \cdot (|V|T + |E|)\big)$ z dodatkowym czynnikiem wynikającym z epizodycznego obniżania temperatury po przekroczeniu limitu stagnacji S. Symulowane wyżarzanie pozostaje umiarkowanie wymagające obliczeniowo, a możliwość wychodzenia z lokalnych minimów funkcji kosztu czyni je skuteczną metodą poprawy jakości rozwiązań przy rozsądnym czasie działania.

6. EKSPERYMENTY DLA SCHEMATU LICENCYJNEGO DUOLINGO SUPER

6.1 Środowisko i metodologia

Eksperymenty wykonano na komputerze z procesorem Intel Core i5-14600KF (12 rdzeni ograniczone przez konfigurację Wyniki potwierdzają również znaczenie struktury grafu. W przypadku grafów losowych i małoświatowych uzyskuje się wyniki zbliżone do solvera ILP, natomiast grafy bezskalowe generują wyższe koszty. Rysunek 6.8 ilustruje, że przeszukiwanie tabu utrzymuje niskie koszty na węzeł także przy rosnącej liczbie wierzchołków, a czasy działania rosną umiarkowanie.SL2, taktowanie 3.49 GHz) oraz 16 GB pamięci RAM. System operacyjny stanowił Ubuntu 24.04 LTS uruchomiony w środowisku WSL2. Implementacja algorytmów powstała w języku Python 3.13; wykorzystano biblioteki NumPy, NetworkX, a także PuLP jako interfejs do algorytmów ILP. Każdy pomiar obejmował wyłącznie czas obliczeń, pomijając koszt przygotowania instancji oraz operacje wejścia/wyjścia.

Środowisko badawcze udostępniono jako zbiór skryptów w języku Python. Osobne moduły obsługują benchmark statyczny, symulacje dynamiczne oraz scenariusze rozszerzeń, a uproszczone polecenia w interfejsie wiersza poleceń umożliwiają uruchamianie pojedynczych algorytmów w trybie diagnostycznym. Taki podział umożliwia równoległe prowadzenie rozległych serii eksperymentów i sprawną weryfikację jakości poszczególnych metod.

Analiza objęła dwa zestawy danych: syntetyczne grafy losowe, bezskalowe oraz małoświatowe o liczebności od 50 do 450 wierzchołków, a także grafy ego z serwisu Facebook o liczebności od 53 do 1035 wierzchołków. Dla każdej instancji uruchamiano wszystkie algorytmy z limitem czasu 60 s, a przekroczenie tego limitu klasyfikowano jako przekroczenie limitu czasu i wyłączano z dalszych obliczeń. W przypadku grafów syntetycznych odnotowano w sumie 10 przekroczeń limitu czasu dla algorytmu mrówkowego oraz algorytmu ILP, natomiast na grafach rzeczywistych odpowiednio 8, 18 i 6 przypadków dla algorytmu mrówkowego, algorytmu ILP oraz przeszukiwania tabu. Wszystkie pozostałe uruchomienia zakończyły się sukcesem.

W przypadku grafów syntetycznych dla każdego rozmiaru generowano trzy niezależne instancje, a następnie wykonywano po dwa uruchomienia algorytmów na każdej z nich. Analiza sieci ego Facebooka obejmowała po jednym grafie na rozmiar oraz dwa powtórzenia dla każdej pary (graf, algorytm), co pozwoliło oszacować zmienność wyników przy zachowaniu określonego budżetu obliczeniowego. W każdej próbie zapisano całkowity koszt licencji, czas wykonania oraz średni koszt licencji przypadający na jednego użytkownika (wierzchołek). Wyniki agregowano jako wartości średnie. Istotność różnic między algorytmami oceniono przy użyciu testów Friedmana, gdzie χ_F^2 to wartość statystyki testu Friedmana oraz p to poziom istotności, a następnie następujących po nich porównań post-hoc metodą Nemenyi'ego. Na syntetycznym benchmarku otrzymano

statystyki $\chi^2_F = 577.93$ dla czasu oraz $\chi^2_F = 518.01$ dla kosztu na węzeł ($p < 10^{-100}$ w obu przypadkach), co uzasadnia szczegółowe porównania par algorytmów.

Dodatkowo, w celu ułatwienia porównań między różnymi typami licencji, ceny każdej z licencji zostały znormalizowane. Cena licencji indywidualnej została ustalona na poziomie 1, a ceny licencji grupowych przeskalowano zgodnie z ich pierwotnym stosunkiem do ceny licencji indywidualnej. Dzięki temu możliwe było bardziej przejrzyste porównanie kosztów między różnymi strategiami licencjonowania, niezależnie od ich bezwzględnych wartości.

6.1.1 Przykład uruchomienia aplikacji i interpretacja wyników

Aby uruchomić eksperymenty, należy skorzystać z polecenia `make pipeline`, które uruchamia główny potok eksperymentalny:

```
UV_CACHE_DIR=.cache/uv PYTHONPATH=src uv run python -m glopt.experiments.pipeline
```

Na rysunku 6.1 przedstawiono przykładowy przebieg uruchomienia aplikacji z poziomu terminala.

```
glopt on  main [✘?] via  v3.12.3 ❯  make pipeline
UV_CACHE_DIR=.cache/uv PYTHONPATH=src uv run python -m glopt.experiments.pipeline
==> static synthetic
Starting glopt benchmark run 20250925_180806_static_synth_benchmark
Run directory: runs/20250925_180806_static_synth_benchmark
Results file: runs/20250925_180806_static_synth_benchmark/csv/20250925_180806_static_synth_benchmark.csv
Graphs: random, small_world, scale_free
Sizes: 10..1000 (24 sizes)
Samples per size: 2
Repeats per graph: 1
License configurations: duolingo_super, roman_domination
Algorithms: ILPSolver, RandomizedAlgorithm, GreedyAlgorithm, DominatingSetAlgorithm, AntColonyOptimization, SimulatedAnnealing, TabuSearch, GeneticAlgorithm
Timeout per run: 60s
Graph cache warm-up started
Graph cache ready: using existing data in data/graphs_cache
Graph cache warm-up finished
Processing license duolingo_super with algorithm ILPSolver
Step n=10   s=0   r=0  lic=duolingo_super algo=ILPSolver       on random       cost=100.31   cpn=10.0310  time=33.73   ms
Step n=10   s=1   r=0  lic=duolingo_super algo=ILPSolver       on random       cost=113.11   cpn=11.3110  time=10.65   ms
Step n=20   s=0   r=0  lic=duolingo_super algo=ILPSolver       on random       cost=184.25   cpn=9.2125   time=31.14   ms
Step n=20   s=1   r=0  lic=duolingo_super algo=ILPSolver       on random       cost=212.23   cpn=10.6115  time=20.84   ms
Step n=30   s=0   r=0  lic=duolingo_super algo=ILPSolver       on random       cost=230.98   cpn=7.6993   time=277.81  ms
Step n=30   s=1   r=0  lic=duolingo_super algo=ILPSolver       on random       cost=215.80   cpn=7.1933   time=470.33  ms
Step n=40   s=0   r=0  lic=duolingo_super algo=ILPSolver       on random       cost=274.14   cpn=6.8535   time=686.09  ms
Step n=40   s=1   r=0  lic=duolingo_super algo=ILPSolver       on random       cost=260.15   cpn=6.5038   time=840.77  ms
Step n=50   s=0   r=0  lic=duolingo_super algo=ILPSolver       on random       cost=290.51   cpn=5.8102   time=1568.42 ms
```

Rysunek 6.1: Przykładowy przebieg uruchomienia potoku eksperymentalnego z Makefile.

Na początku wyświetlane są informacje o wybranych parametrach uruchomienia, takich jak typ potoku, konfiguracje licencji, lista algorytmów oraz limity czasowe. Następnie rozpoczyna się proces przygotowania grafów. Tworzona jest pamięć podręczna grafów (cache), która umożliwia wielokrotne wykorzystanie tych samych instancji grafów między różnymi algorytmami i konfiguracjami licencji. Eliminuje to konieczność ponownego generowania grafów dla każdej próby, co istotnie skraca czas obliczeń, zwłaszcza dla dużych instancji.

Po przygotowaniu grafów wyświetlane są szczegóły dotyczące liczby rozmiarów, próbek i powtórzeń dla każdej instancji. Następnie dla każdej kombinacji licencji i algorytmu prezentowane są postępy obliczeń. Każda iteracja prezentuje podstawowe statystyki, takie jak rozmiar grafu, koszt rozwiązania, koszt na węzeł oraz czas wykonania. Umożliwia to bieżące monitorowanie przebiegu eksperymentu i szybkie wykrywanie ewentualnych anomalii.

6.2 Duolingo Super na grafach syntetycznych

6.2.1 Statystyki zbiorcze

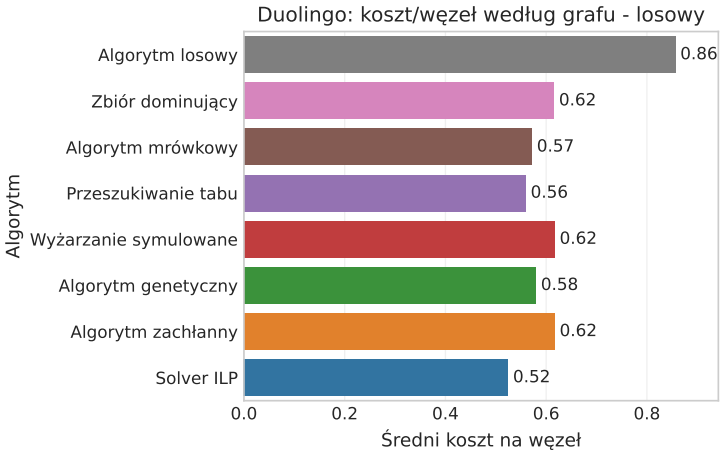
Tabela 6.1 zestawia średnie wartości kosztu licencji na wierzchołek oraz czasu dla schematu licencyjnego Duolingo Super. Aby zapewnić porównywalność wyników, analizy statystyczne i wizualizacje oparto na podzbiorze instancji, dla których wszystkie algorytmy zakończyły działanie przed upływem limitu czasu; w praktyce oznacza to przycięcie rozmiarów grafów do zakresu wspólnego dla solvera ILP i algorytmu mrówkowego, które najczęściej przekraczały limit. Solver ILP pozostaje najlepszym punktem odniesienia jakościowego (średni koszt na węzeł 0,445), lecz ma zastosowanie jedynie dla mniejszych instancji. Liczba przekroczeń limitu czasu jest taka sama jak w algorytmie mrówkowym. Wśród metaheurystyk najniższy koszt na węzeł osiąga właśnie algorytm mrówkowy (0,506), natomiast przeszukiwanie tabu zapewnia najlepszy kompromis kosztu na węzeł i czasu w grupie metod przybliżonych. Algorytm zachłanny i algorytm losowy charakteryzują się czasem działania rzędu milisekund, ale tylko pierwszy z nich utrzymuje akceptowalny koszt na węzeł (0,548), podczas gdy algorytm losowy generuje rozwiązania o istotnie wyższym koszcie na węzeł.

Tabela 6.1: Średnie wartości kosztu na węzeł i czasu dla schematu licencyjnego Duolingo Super na grafach syntetycznych.

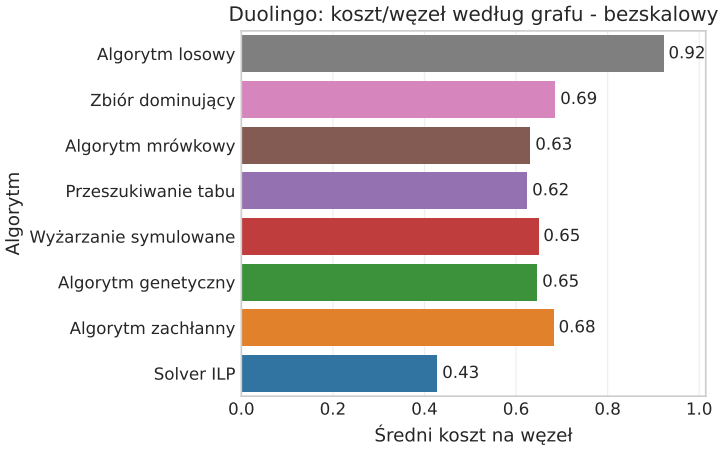
Algorytm	**Średni koszt/węzeł**	**Średni koszt**	**Średni czas [s]**
Algorytm ILP	0.445	73.83	2.492
Algorytm mrówkowy	0.506	83.58	5.124
Przeszukiwanie tabu	0.500	117.80	3.298
Algorytm genetyczny	0.515	118.36	1.222
Wyżarzanie symulowane	0.531	120.90	0.975
Zbiór dominujący	0.542	119.97	0.017
Algorytm zachłanny	0.548	121.94	0.001
Algorytm losowy	0.799	179.90	0.001

6.2.2 Porównanie algorytmów na grafach syntetycznych

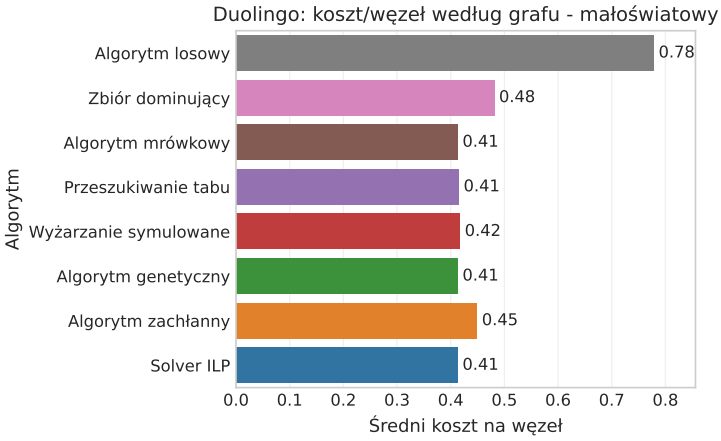
Rysunki 6.2–6.4 ilustrują, że algorytmy dla schematu licencyjnego Duolingo Super osiągają wyniki porównywalne z algorytmem ILP w przypadku grafów losowych i małoświatowych. Natomiast w przypadku grafów bezskalowych wyniki są zauważalnie gorsze.



Rysunek 6.2: Koszt na węzeł w zależności od struktury grafu losowego.



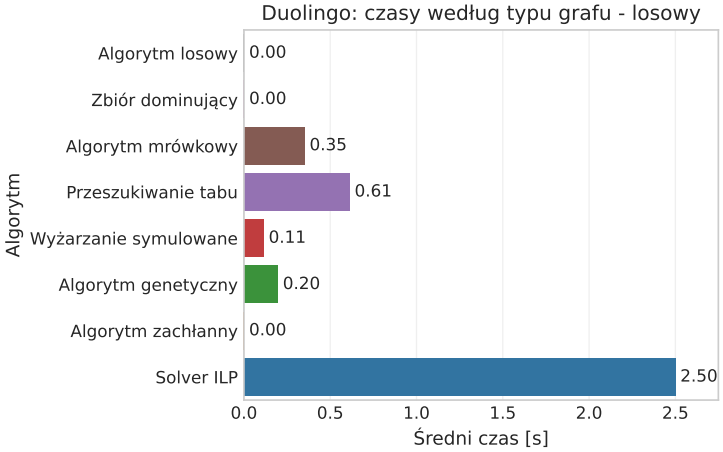
Rysunek 6.3: Koszt na węzeł w zależności od struktury grafu bezskalowego.



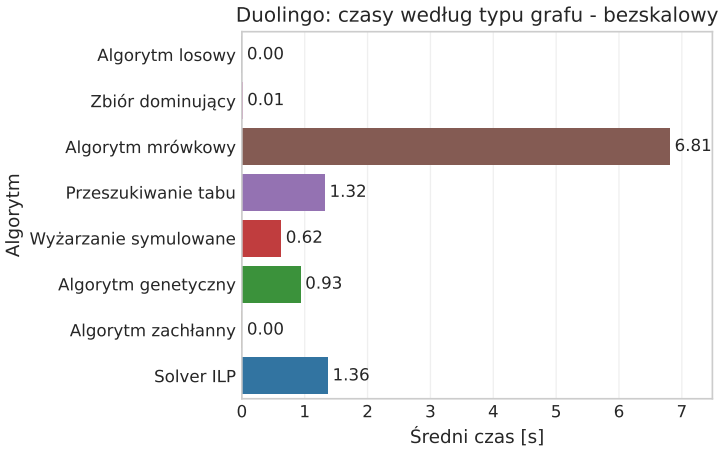
Rysunek 6.4: Koszt na węzeł w zależności od struktury grafu małoświatowego.

Podobne obserwacje dotyczą czasów wykonania, jak ilustrują rysunki 6.5–6.7. Czasy działania algorytmów są zbliżone dla różnych typów grafów, z wyjątkiem algorytmu ILP, który średnio osiąga krótszy czas wykonania na grafach bezskalowych, oraz przeszukiwania tabu, które dzia-

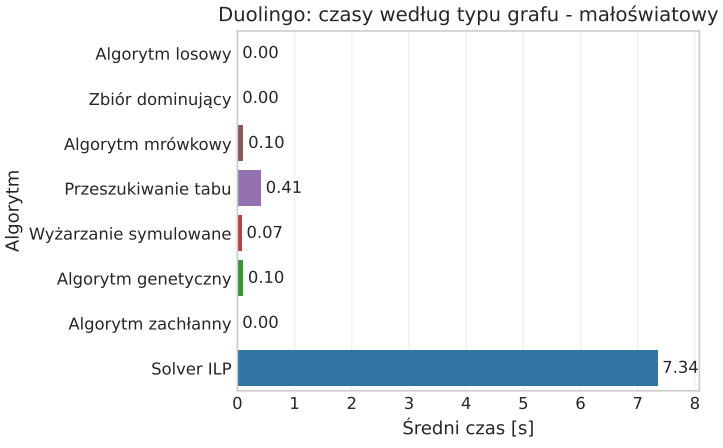
ła na nich dłużej. Pozostałe algorytmy wykazują porównywalne czasy działania niezależnie od struktury grafu. Tabela 6.2 podsumowuje średnie czasów i kosztów na węzeł dla różnych typów grafów.



Rysunek 6.5: Czas wykonania w zależności od struktury grafu losowego.



Rysunek 6.6: Czas wykonania w zależności od struktury grafu bezskalowego.



Rysunek 6.7: Czas wykonania w zależności od struktury grafu małoświatowego.

Tabela 6.2: Średnie koszty i czasy na węzeł dla różnych typów grafów (schematy licencyjne Duolingo Super i dominowania rzymskiego).

Licencja	Typ grafu	Śr. koszt/węzeł	Śr. czas [s]
Duolingo Super	Bezskalowy	0.661	1.655
Duolingo Super	Losowy	0.502	1.598
Duolingo Super	Małoświatowy	0.495	1.346
dominowanie rzymskie	Bezskalowy	0.490	0.913
dominowanie rzymskie	Losowy	0.344	0.815
dominowanie rzymskie	Małoświatowy	0.409	1.544

Z powyższych danych wynika, że schemat licencyjny dominowania rzymskiego osiąga niższy koszt na węzeł we wszystkich typach grafów syntetycznych. Największą przewagę wykazuje w grafach losowych (różnica 0,158), następnie w bezskalowych (0,171), a najmniejszą w małoświatowych (0,086). Pod względem czasów wykonania algorytmy dla schematu Duolingo Super charakteryzują się dłuższymi czasami w strukturach bezskalowych i losowych, jednak w grafach małoświatowych działają nieco szybciej niż w schemacie dominowania rzymskiego. Struktura bezskalowa generuje najwyższy koszt dla obu schematów licencjonowania, natomiast grafy małoświatowe zapewniają najkorzystniejszy stosunek kosztu do wydajności obliczeniowej. Przewaga kosztowa schematu dominowania rzymskiego wynika naturalnie z tego, że stosunek ceny licencji grupowej do indywidualnej jest w niej niższy niż w schemacie Duolingo Super.

6.3 Duolingo Super na grafach rzeczywistych

W analizie grafów ego z serwisu Facebook pominięto obserwacje, w których algorytm ILP nie zakończył pracy przed limitem czasu (18 przypadków). Dodatkowo odnotowano 8 przekroczeń limitu czasu algorytmu mrówkowego i 6 przypadków w przeszukiwaniu tabu; pozostałe uruchomienia zakończyły się sukcesem.

Tabela 6.3: Statystyki kosztu i czasu dla schematu licencyjnego Duolingo Super na grafach rzeczywistych.

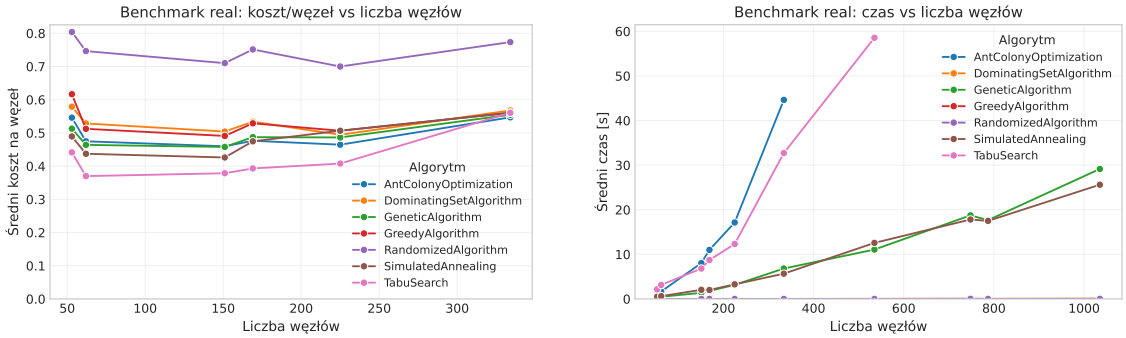
Algorytm	Średni koszt	Śr. koszt/węzeł	Śr. czas [s]
Algorytm mrówkowy	83.58	0.506	5.124
Zbiór dominujący	119.97	0.542	0.017
Algorytm genetyczny	118.36	0.515	1.222
Algorytm zachłanny	121.94	0.548	0.001
Algorytm losowy	179.90	0.799	0.001
Wyżarzanie symulowane	120.90	0.531	0.975
Przeszukiwanie tabu	117.80	0.500	3.298

Tabela 6.3 wskazuje, że zarówno przeszukiwanie tabu, jak i algorytm mrówkowy zachowują przewagę kosztową nad heurystykami losowymi i zachłannymi, kosztem dłuższego czasu

działania.

6.3.1 Skalowanie i jakość

Rysunek 6.8 ilustruje, że koszt na węzeł pozostaje względnie stabilny niezależnie od liczby wierzchołków, z niewielkimi wahaniami spowodowanymi różnicami między poszczególnymi instancjami grafów. Przeszukiwanie tabu utrzymuje najniższe wartości, a algorytm mrówkowy osiąga wartości zbliżone. Czasy działania wszystkich metod nie przekraczają kilku sekund i rosną łagodnie wraz z rozmiarem grafu; algorytm zachłanny zachowuje czas działania rzędu milisekund i stanowi efektywną procedurę inicjalizacji dla metaheurystyk. Dodatkowo tabela 6.4 agreguje średnie wartości (przeszukiwanie tabu) dla kolejnych rozmiarów sieci ego i pokazuje, że koszt na węzeł oscyluje w przedziale 0.37–0.56 przy czasie rosnącym od około 2 s do około 33 s dla największych grafów.



Rysunek 6.8: Koszt na węzeł i czas wykonania algorytmów dla schematu licencyjnego Duolingo Super w zależności od liczby wierzchołków (grafy ego Facebook).

Tabela 6.4: Średni koszt licencji na węzeł i czas (przeszukiwanie tabu) względem liczby wierzchołków w sieciach ego Facebook.

Liczba wierzchołków	Śr. koszt/węzeł	Śr. czas [s]
53	0.441	2.16
62	0.370	3.14
151	0.379	6.83
169	0.393	8.73
225	0.408	12.32
334	0.561	32.71

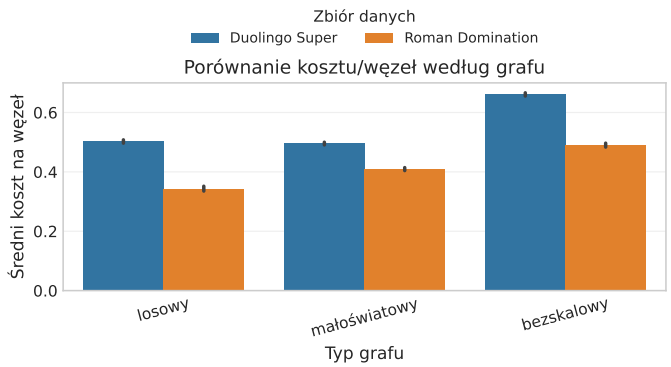
6.4 Porównanie z dominowaniem rzymskim

Porównania schematu licencyjnego Duolingo Super z konfiguracją imitującą dominowanie rzymskie ograniczono do wspólnych instancji grafów i algorytmów. Rysunki 6.9–6.11 zestawiają różnice w kosztach, czasach i strukturze licencji, a tabela 6.5 gromadzi średnie według typu grafu syntetycznego.

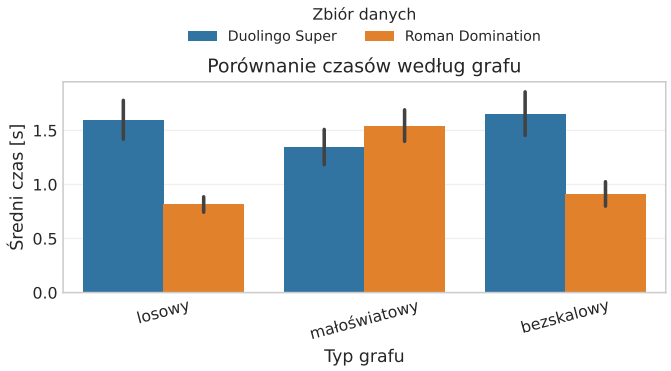
Tabela 6.5: Średnie czasu i kosztu na węzeł według typu grafu (wspólne instancje).

Typ grafu	Duolingo Super		dominowanie rzymskie	
	Czas [s]	Koszt/węzeł	Czas [s]	Koszt/węzeł
Losowy	1.598	0.502	0.815	0.344
Bezskalowy	1.655	0.661	0.913	0.490
Małoświatowy	1.346	0.495	1.544	0.409

Algorytmy optymalizacji dla schematu licencyjnego dominowania rzymskiego osiągają niższy koszt na węzeł we wszystkich porównywanych typach grafów. Największa różnica dotyczy struktur losowych, gdzie przewaga wynosi ok. 0,158 jednostki kosztu na węzeł (32%). W sieciach bezskalowych różnica wynosi 0,171 jednostki, natomiast w małoświatowych 0,086 jednostki. Algorytmy dla różnych schematów licencyjnych wykazują porównywalne czasy wykonania, przy czym w grafach małoświatowych algorytmy dla schematu licencyjnego Duolingo Super działają nieznacznie szybciej, natomiast w grafach losowych i bezskalowych szybciej działają algorytmy dla schematu licencyjnego dominowania rzymskiego. Rysunek 6.9 ilustruje te obserwacje dla pełnego rozkładu kosztów, a rysunek 6.10 prezentuje analogiczne dane dotyczące czasów wykonania. Warto zauważyć, że wyższe koszty na węzeł w przypadku Duolingo Super wynikają z ograniczeń w opłacalności licencji grupowych. Licencja grupowa jest około 2,1 razy droższa od indywidualnej, co oznacza, że opłaca się ją tworzyć dopiero dla grup liczących co najmniej trzy osoby (właściciel i dwaj sąsiedzi w grafie). W konsekwencji liczba licencji indywidualnych w schemacie Duolingo Super jest większa, co przekłada się na wyższy średni koszt na węzeł w porównaniu z konfiguracją dominowania rzymskiego.



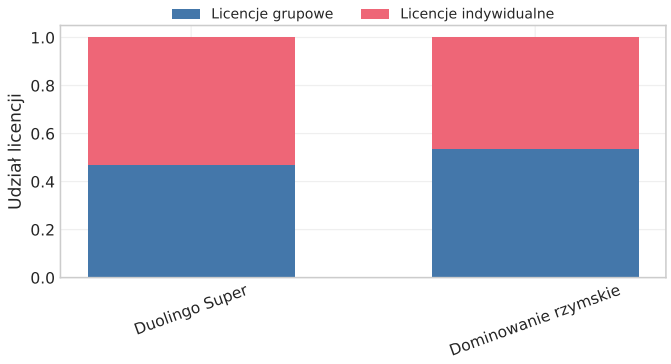
Rysunek 6.9: Koszt na węzeł według typu grafu: porównanie konfiguracji licencyjnych Duolingo Super i dominowania rzymskiego.



Rysunek 6.10: Czas wykonania według typu grafu: porównanie konfiguracji licencyjnych Duolingo Super i dominowania rzymskiego.

Rysunek 6.11 pokazuje, że w rozwiązaniach optymalnych dla konfiguracji licencyjnej dominowania rzymskiego obserwuje się wyższy stosunek licencji grupowych do indywidualnych w porównaniu do rozwiązań dla konfiguracji licencyjnej Duolingo Super. W konfiguracji dominowania rzymskiego stosunek licencji grupowych do indywidualnych wynosi około 1,15:1, natomiast w schemacie Duolingo Super stosunek ten jest odwrócony i wynosi około 0,88:1 (więcej licencji indywidualnych niż grupowych).

Różnica w strukturze wykorzystania licencji wynika z faktu, że konfiguracja licencyjna dominowania rzymskiego nie posiada ograniczenia maksymalnej pojemności grupy licencyjnej. W schemacie Duolingo Super, gdy grupa osiąga maksymalną pojemność, dodatkowi użytkownicy, którzy mogliby do niej należeć, są zmuszeni do zakupu licencji indywidualnych. W pełnym zbiorze danych, obejmującym trzy typy grafów syntetycznych, odnotowano 137 690 licencji w schemacie Duolingo Super, z czego 73 176 (53,15%) to licencje indywidualne. W konfiguracji dominowania rzymskiego, dzięki brakowi ograniczeń pojemności grup, liczba licencji indywidualnych jest proporcjonalnie niższa.



Rysunek 6.11: Struktura wykorzystania licencji: porównanie konfiguracji licencyjnych Duolingo Super i dominowania rzymskiego.

6.5 Wnioski

Przeprowadzone eksperymenty wskazują, że schemat licencjonowania Duolingo Super zapewnia stabilne i powtarzalne wyniki. Spośród badanych metod najlepsze wyniki uzyskały algorytm mrówkowy oraz przeszukiwanie tabu. Algorytm mrówkowy osiąga najniższy koszt na węzeł w grupie metaheurystyk, lecz wymaga dłuższego czasu działania. Przeszukiwanie tabu umożliwia uzyskanie porównywalnej jakości i jednocześnie zachowuje lepszy kompromis między kosztem a czasem obliczeń.

Solver ILP stanowi wartościowe odniesienie jakościowe, jednak jego zastosowanie jest ograniczone do mniejszych instancji ze względu na częste przekroczenia limitu czasu. Proste heurystyki, takie jak algorytm zachłanny czy losowy, charakteryzują się bardzo krótkim czasem działania, ale rozwiązania mają istotnie wyższy koszt na węzeł.

Wyniki potwierdzają również znaczenie struktury grafu. W przypadku grafów losowych i małoświatowych możliwe jest osiągnięcie rezultatów zbliżonych do solvera ILP, natomiast grafy bezskalowe okazały się trudniejsze i generowały wyższe koszty. Skalowanie pokazane na rysunku 6.8 dowodzi, że przeszukiwanie tabu zachowuje niskie koszty na węzeł nawet przy rosnącej liczbie wierzchołków, a czasy działania rosną umiarkowanie.

Porównanie ze schematem licencyjnym dominowania rzymskiego wskazuje, że schemat ten osiąga niższy koszt na węzeł we wszystkich typach grafów. Wynika to z większego udziału licencji grupowych i niższego kosztu jednostkowego. Schemat Duolingo Super cechuje się natomiast krótszym czasem wykonania w grafach małoświatowych.

7. SYMULACJA DYNAMICZNA

7.1 Założenia i konfiguracja

Eksperymenty dynamiczne wykonano w tym samym środowisku obliczeniowym co testy statyczne. Każda symulacja obejmuje 30 kroków czasowych, w których każdy krok składa się z: (1) zastosowania mutacji do struktury grafu zgodnie z wybranym profilem oraz (2) ponownej optymalizacji dystrybucji licencji przez algorytmy z limitem czasu równym 45 s. Analizowano dwie konfiguracje licencyjne: Duolingo Super oraz dominowanie rzymskie. Do testów wykorzystano różne typy grafów: losowe, bezskalowe oraz małoświatowe.

Testy przeprowadzono dla sześciu profili mutacji, przy czym każdy profil definiuje stały zestaw parametrów obowiązujący w całej symulacji. Profile obejmują trzy syntetyczne o prostej logice modyfikacji (low, med, high) oraz trzy bardziej złożone, uwzględniające mechanizmy takie jak preferencyjne przyłączanie, domykanie triad czy losowe przekształcanie krawędzi (pref_triadic, pref_pref, rand_rewire).

Warianty *low*, *med* oraz *high* różnią się prawdopodobieństwami mutacji w każdym kroku symulacji (tabela 7.1). W każdym kroku niezależnie losowane jest to, czy dana operacja zostanie wykonana, a następnie losuje się liczbę elementów do modyfikacji (od 1 do 3 węzłów, od 1 do 5 krawędzi). Usunięcie węzła skutkuje usunięciem wszystkich krawędzi incydentnych z tym węzłem. Prawdopodobieństwa dobrano tak, aby uzyskać trzy wyraźnie różniące się poziomy intensywności zmian: niski (modyfikacje w około 1-6% kroków), średni (4-18% kroków) oraz wysoki (8-30% kroków).

Tabela 7.1: Parametry intensywności mutacji w symulacji dynamicznej.

Poziom	Dodawanie węzłów	Usuwanie węzłów	Dodawanie krawędzi	Usuwanie krawędzi
low	0.02	0.01	0.06	0.04
med	0.06	0.04	0.18	0.12
high	0.12	0.08	0.30	0.20

Dodatkowo zbadano trzy bardziej realistyczne profile ewolucji sieci. Wszystkie operują na tych samych limitach liczby dodawanych/usuwanych elementów co warianty syntetyczne, różnią się jednak mechanizmem wyboru sąsiedztwa.

Wariant *pref_triadic* łączy dwa mechanizmy typowe dla sieci społecznych: preferencyjne przyłączanie i domykanie triad. Nowe węzły dołączają do sieci, preferencyjnie łącząc się z istniejącymi węzłami o wysokim stopniu. Nowe krawędzie powstają natomiast poprzez domykanie trójkątów, tj. wyszukiwanie dwóch niepołączonych węzłów mających wspólnego sąsiada i tworzenie między nimi połączenia. Mechanizm ten wzmacnia lokalną strukturę klastrową sieci, co odzwierciedla tendencję do tworzenia zamkniętych grup w rzeczywistych sieciach [37, 38].

Wariant *pref_pref* stosuje mechanizm preferencyjnego przyłączania zarówno przy dodawaniu nowych węzłów, jak i tworzeniu krawędzi między już istniejącymi. Oznacza to, że nie tylko nowe węzły chętniej łączą się z popularnymi wierzchołkami, ale również nowe krawędzie z większym prawdopodobieństwem powstaną między dwoma węzłami, które już teraz mają wysoki stopień. Takie podejście sprzyja tworzeniu centralnych węzłów o wysokim stopniu w sieci, co jest kluczową cechą topologii bezskalowych [37].

Wreszcie wariant *rand_rewire* wprowadza do sieci element losowości, inspirowany modelem Wattsa i Strogatza. Nowe węzły dołączane są w sposób losowy, bez preferencji dla popularniejszych wierzchołków. Zamiast dodawania nowych krawędzi, model ten modyfikuje istniejącą strukturę poprzez operację przełączania (rewiring): losowa krawędź jest usuwana, a jeden z jej końców jest następnie łączony z innym, losowo wybranym węzłem w sieci. Proces ten prowadzi do powstawania połączeń długozasięgowych i zmniejsza regularność struktury grafu [39].

7.2 Algorytm zachłanny

W tym wariancie algorytm zachłanny w każdym kroku symulacji buduje rozwiązanie całkowicie od zera, ignorując poprzednie wyniki optymalizacji. Stanowi to dolne ograniczenie narzutu czasowego dla metod konstruktywnych oraz punkt odniesienia jakościowy, względem którego porównywane są bardziej zaawansowane metaheurystyki wykorzystujące rozwiązania z poprzednich kroków jako punkt startowy dla dalszej optymalizacji.

7.2.1 Zestawienie dla wszystkich mutacji

Tabela 7.2 zestawia średni koszt na węzeł i średni czas wykonania algorytmu na krok symulacji dla sześciu badanych profili mutacji: trzech syntetycznych oraz trzech realistycznych. Wszystkie czasy mieszczą się w przedziale od 0.5 ms do 1.8 ms na krok, a średni koszt na węzeł oscyluje wokół 0.46 do 0.48.

Tabela 7.2: Algorytm zachłanny: średni koszt na węzeł oraz średni czas na krok dla wszystkich wariantów mutacji.

Metoda mutacji	Koszt/węzeł (mean)	Średni czas [s]
high	0.48009	0.00159
low	0.47983	0.00165
med	0.47491	0.00181
pref_pref	0.46371	0.00083
pref_triadic	0.46787	0.00053
rand_rewire	0.47556	0.00083

Algorytm zachłanny jest bardzo szybki, co potwierdza jego przydatność jako czasowy punkt odniesienia w środowisku dynamicznym. Warianty realistyczne sprzyjają nieco niższym kosztom: `pref_pref` osiąga najniższy średni koszt (0.464), a `pref_triadic` jest jednocześnie najszybszy ($0.000\,53$ s). Wariant `rand_rewire` jest trudniejszy (0.476), ale pozostaje bardzo szybki

czasowo.

Różnice kosztu dla mutacji syntetycznych są niewielkie (0.475–0.480), natomiast czasy są wyższe niż w profilach realistycznych, szczególnie dla `med/high`. Wskazuje to, że bardziej lokalne, realistyczne przekształcenia struktury grafu są łatwiejsze do obsłużenia. Czasy dla `pref_pref` wynoszą średnio $0.000\,83$ s, a dla `pref_triadic` $0.000\,53$ s, co potwierdza ich przewagę czasową nad mutacjami syntetycznymi ($0.001\,59$ s dla `high`).

7.3 Analiza mutacji syntetycznych

7.3.1 Metaheurystyki

Tabela 7.3 przedstawia zbiorcze wyniki dla różnych metod mutacji. Średni koszt na węzeł jest bardzo zbliżony dla wszystkich poziomów intensywności, z jedynie nieznacznym wzrostem dla wariantu `high`. Sugeruje to, że algorytmy są w stanie skutecznie adaptować się do zmian w topologii sieci.

Średni czas wykonania algorytmu rośnie wraz z intensywnością mutacji. Wariant `low` jest najszybszy, podczas gdy `high` wymaga najwięcej czasu na ponowne zbilansowanie. Jest to naturalna konsekwencja faktu, że większa liczba modyfikacji grafu (dodawanie/usuwanie węzłów i krawędzi) stanowi większe wyzwanie obliczeniowe dla algorytmów optymalizacyjnych. Mimo to, różnice w czasach nie są drastyczne, co świadczy o dobrej skalowalności zastosowanych metod.

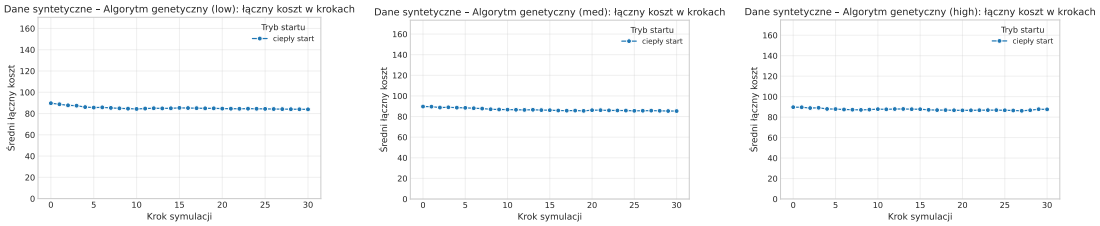
Tabela 7.3: Wyniki dla różnych metod mutacji.

Metoda mutacji	Średni koszt	Średni czas [s]
high	0.4893	3.169
med	0.4850	3.075
low	0.4852	2.878

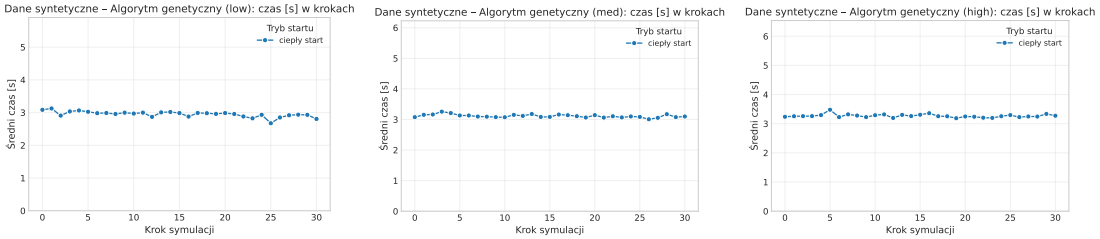
7.3.2 Profil kosztu i czasu w czasie

Jako przykład ewolucji kosztu i czasu w krokach symulacji wybrano algorytm genetyczny. Rysunki 7.1 i 7.2 przedstawiają odpowiednio przebieg kosztu na węzeł oraz czasu wykonania w zależności od kroku symulacji dla różnych poziomów intensywności mutacji.

Dla wariantu `high` można zaobserwować niewielkie, lecz zauważalne wahania średniego kosztu na węzeł, które nie są tak widoczne w przypadku wariantów `low` i `med`. Jeśli chodzi o czasy wykonania, dla każdego z trzech wariantów widoczne są zbliżone wahania w trakcie trwania symulacji.



Rysunek 7.1: Algorytm genetyczny – koszt na węzeł w kolejnych krokach symulacji (warianty low/med/high).



Rysunek 7.2: Algorytm genetyczny – czas wykonania w kolejnych krokach symulacji (warianty low/med/high).

7.4 Analiza mutacji realistycznych

Tabela 7.4 przedstawia zbiorcze wyniki dla scenariuszy realistycznych. Wariant `pref_triadic` wyróżnia się najkrótszym średnim czasem wykonania (poniżej 1 sekundy), przy zachowaniu kosztu na poziomie zbliżonym do wariantu `pref_pref`. Z kolei scenariusz `rand_rewire` okazał się najbardziej wymagający dla algorytmów – charakteryzuje się jednocześnie najwyższym średnim kosztem i najdłuższym czasem przetwarzania. Wynika to z faktu, że losowe przełączanie krawędzi zaburza lokalną strukturę klastrową sieci, zwiększa entropię topologii i utrudnia wykorzystanie poprzedniego rozwiązania jako punktu odniesienia. W efekcie optymalizacja wymaga więcej iteracji, a uzyskana jakość jest gorsza.

Tabela 7.4: Wyniki dla różnych metod mutacji w scenariuszach realistycznych.

Metoda mutacji	**Średni koszt całkowity**	**Koszt/węzeł (mean)**	**Średni czas [s]**
pref_pref	113.34	0.4764	1.6774
pref_triadic	70.08	0.4764	0.8619
rand_rewire	116.83	0.4917	1.8421

7.4.1 Wybrane algorytmy i metody mutacji

W celu zilustrowania zachowania różnych algorytmów optymalizacyjnych w środowisku dynamicznym, wybrano reprezentatywne pary algorytmów i metod mutacji. Zestawienie obejmuje algorytmy o różnej złożoności obliczeniowej. Od prostych heurystyk po zaawansowane metaheurystyki. Testowane na wybranych profilach mutacji realistycznych oraz syntetycznych. Dobór przedstawia spektrum możliwych rozwiązań, od szybkich metod przybliżonych po dokładne, ale

czasochłonne podejścia optymalizacyjne.

Tabela 7.5: Wybrane pary algorytmów i metod mutacji.

Algorytm	Metoda	Liczba kroków	Koszt/węzeł	Śr. czas [s]
Algorytm ILP	pref_triadic	434	0.362	1.525
Algorytm ILP	rand_rewire	496	0.390	2.553
Algorytm genetyczny	pref_triadic	744	0.409	0.615
Algorytm genetyczny	pref_pref	930	0.414	1.134
Przeszukiwanie tabu	pref_triadic	744	0.413	1.470
Przeszukiwanie tabu	rand_rewire	930	0.445	2.581
Algorytm mrówkowy	pref_pref	899	0.417	6.906
Algorytm mrówkowy	pref_triadic	744	0.424	3.002
Wyżarzanie symulowane	pref_triadic	744	0.460	0.555
Algorytm zachłanny	pref_pref	930	0.464	0.001
Zbiór dominujący	pref_triadic	744	0.457	0.005
Algorytm losowy	pref_pref	930	0.754	0.001

Algorytm ILP osiąga najniższe koszty, ale wymaga około 1.5–2.6 s na krok. Algorytm genetyczny dobrze sprawdza się przy zmianach klastrowych (`pref_triadic`), oferując niski koszt i czas około 0.6 s. Przy wariancie `pref_pref` jest wolniejszy (1.1 s), ale zachowuje dobrą jakość. Z kolei wyżarzanie symulowane jest szybkie (około 0.5 s) i stabilne, ale jakościowo ustępuje bardziej zaawansowanym metodom.

Przeszukiwanie tabu korzysta z lokalności zmian, osiągając sensowny koszt i czas około 1.5 s przy `pref_triadic`. Jednak przy mutacjach losowych (`rand_rewire`) czas wzrasta do około 2.6 s, a koszt również rośnie. Algorytm mrówkowy zapewnia bardzo dobrą jakość przy `pref_pref`, ale działa najwolniej (prawie 7 s). Przejście na `pref_triadic` ponad dwukrotnie przyspiesza jego działanie kosztem niewielkiego pogorszenia jakości.

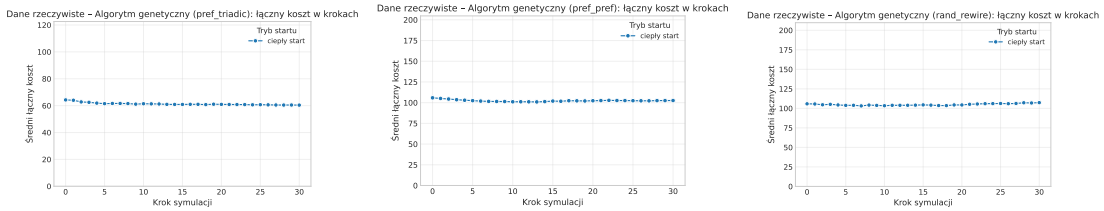
Heurystyki szybkie, takie jak algorytm zachłanny (około 1 ms) i zbiór dominujący (około 5 ms), są bardzo efektywne. Zbiór dominujący zwykle osiąga niższy koszt niż zachłanny, co czyni go zasadnym wyborem przy ograniczeniach czasowych. Mutacje klastrowe (`pref_triadic`, `pref_pref`) pomagają wszystkim algorytmom utrzymać niski średni koszt licencji liczony na jeden węzeł i krótszy czas. Natomiast `rand_rewire` zwiększa zarówno koszt, jak i czas.

7.4.2 Ewolucja kosztów w czasie

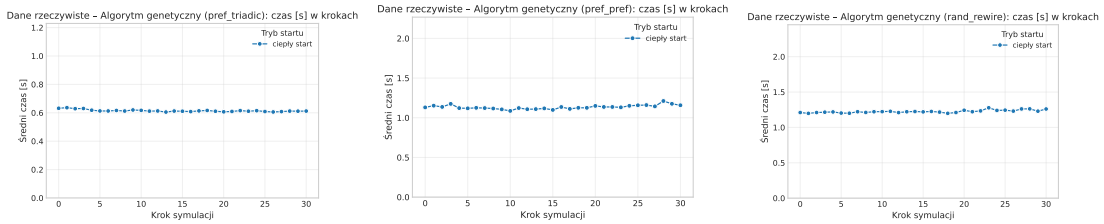
Pełne przebiegi dla algorytmu genetycznego pokazano na rys. 7.3–7.4. Łączenie preferencyjnego przyłączania z triadycznym domykaniem sprzyja utrzymaniu najniższych kosztów, natomiast wariant z losowym przełączaniem krawędzi prowadzi do wolniejszej stabilizacji. Analizując koszt na węzeł na rys. 7.3, można zauważyć, że dla każdego z typów mutacji przebiega on podobnie, oscylując wokół zbliżonego poziomu.

Z kolei, patrząc na czas wykonania na rys. 7.4, widać, że dla wariantu `rand_rewire` wy-

stępują największe wahania, co sugeruje większą niestabilność procesu optymalizacji. Warianty `pref_triadic` i `pref_pref` charakteryzują się bardziej stabilnym czasem wykonania.



Rysunek 7.3: Algorytm genetyczny – koszt na węzeł w wariantach realistycznych.



Rysunek 7.4: Algorytm genetyczny – czas wykonania w wariantach realistycznych.

7.4.3 Wnioski

Przeprowadzona analiza dynamiczna wykazała, że dobór algorytmu optymalizacyjnego w środowisku dynamicznym powinien uwzględniać zarówno profil zmian topologii sieci, jak i dostępny budżet czasowy. Badania obejmujące sześć wariantów mutacji (trzy syntetyczne oraz trzy realistyczne) na różnych typach grafów pozwoliły na sformułowanie kluczowych wniosków dotyczących adaptacji algorytmów do ewoluujących struktur sieciowych.

Mutacje realistyczne, takie jak `pref_triadic` oraz `pref_pref`, były mniej wymagające obliczeniowo do obsłużenia w porównaniu z wariantami syntetycznymi o wysokiej intensywności. Mechanizmy preferencyjnego przyłączania oraz domykania trójkątów sprzyjają wzmacnianiu klastrowości i tworzeniu hubów, co zmniejsza efektywny rozmiar problemu pokrycia. W rezultacie algorytmy osiągały niższe koszty (0.470–0.475 w porównaniu do 0.485–0.489) oraz krótsze czasy wykonania (0.9–1.7 s w porównaniu do 2.9–3.2 s). Z kolei mutacje typu `rand_rewire` zwiększały entropię topologii poprzez zrywanie lokalnych połączeń, co prowadziło do najgorszych wyników zarówno pod względem kosztu (0.486), jak i czasu (1.9 s).

Algorytmy wykazywały różną odporność na poszczególne typy zmian w zależności od mechanizmu wykorzystania poprzedniego rozwiązania. Algorytm ILP zapewniał najwyższą jakość (koszt 0.362–0.390), lecz wymagał znacznego czasu na ponowne rozwiązanie zmodyfikowanych ograniczeń. Algorytm genetyczny dobrze radził sobie przy zmianach klastrowych, oferując korzystny kompromis między jakością a czasem (koszt 0.409 przy czasie 0.615 s dla `pref_triadic`), jednak jego efektywność spadała w przypadku mutacji destrukcyjnych. Przeszukiwanie tabu, jako metoda intensywnie lokalna, sprawdzało się przy zmianach o ograniczonym zasięgu, lecz przy mutacjach o większej skali wymagało rozszerzenia promienia poszukiwań, co wydłużało czas działania do około 6.5 s.

Szybkie heurystyki, takie jak algorytm zachłanny oraz zbiór dominujący, zachowywały swoją użyteczność również w środowisku dynamicznym, oferując czasy wykonania poniżej 5 ms przy kosztach porównywalnych z bardziej złożonymi metodami. Algorytm zachłanny osiągał stabilny czas w zakresie od 0.5 ms do 1.8 ms przy koszcie 0.464–0.480, natomiast zbiór dominujący zapewniał nieco niższy koszt przy czasie około 5 ms.

Wyniki badań wskazują na konieczność adaptacyjnego doboru strategii optymalizacyjnej w zależności od obserwowanego profilu zmian. Przy zmianach o charakterze klastrowym zaleca się stosowanie metaheurystyk z mechanizmami naprawy poprzedniego rozwiązania. W przypadku zmian chaotycznych konieczne może być głębsze odświeżenie rozwiązania lub zaakceptowanie wyższego kosztu w zamian za stabilność czasową. W sytuacjach, w których czas stanowi krytyczne ograniczenie, szybkie heurystyki stanowią preferowaną strategię, zapewniając przewidywalną wydajność niezależnie od typu mutacji.

Najlepszy kompromis globalny oferują metaheurystyki z mechanizmami konstrukcji opartymi na poprzednich wynikach oraz lokalnych naprawach, dostosowywanymi do intensywności zaobserwowanych zmian między krokami symulacji.

8. ROZSZERZENIA MODELU LICENCJONOWANIA

8.1 Przegląd badanych wariantów

Oprócz bazowych konfiguracji rozważono osiem rozszerzeń licencyjnych, które różnią się wielokrotnością kosztu licencji grupowej względem indywidualnej. Warianty Duolingo Super obejmują plany, w których koszt licencji grupowej wynosi odpowiednio dwukrotność, czterokrotność i pięciokrotność ceny licencji indywidualnej, przy zachowaniu stałej pojemności grupy (6 osób). Podobnie, w przypadku dominowania rzymskiego analizowano konfiguracje, w których koszt licencji grupowej wynosi p-krotność ceny indywidualnej, dla $p \in \{3, 4, 5\}$. Dwa ostatnie warianty odnoszą się do rzeczywistych ofert: Spotify oferuje plan Duo (pojemność 2) jako wariant pośredni między licencją indywidualną a rodzinną, natomiast Netflix oferuje plany dla 1, 2 lub 4 osób.

Przedstawione w tabelach wartości stanowią statystyki zagregowane dla danej konfiguracji licencyjnej. Oznacza to, że średnie koszty na węzeł oraz średnie czasy obliczeń zostały obliczone na podstawie wyników wszystkich zastosowanych algorytmów oraz wszystkich typów grafów uwzględnionych w eksperymentach. Tabele nie odnoszą się zatem do pojedynczego algorytmu czy sieci, lecz prezentują uśrednione efekty całej grupy uruchomień dla danej konfiguracji licencji.

8.1.1 Warianty rodziny Duolingo

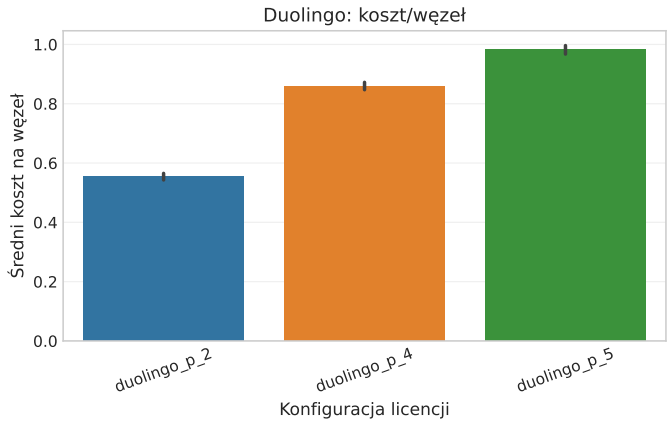
Na rys. 8.1 przedstawiono mediany kosztu na węzeł dla konfiguracji, w których koszt licencji grupowej wynosi odpowiednio dwukrotność (`duolingo_p_2`), czterokrotność (`duolingo_p_4`) oraz pięciokrotność (`duolingo_p_5`) ceny licencji indywidualnej. Wyniki wskazują, że wyższe mnożniki prowadzą do wzrostu kosztu na węzeł, co wynika z częstszej selekcji licencji indywidualnych przy wyższych kosztach planów grupowych. W przypadku konfiguracji `duolingo_p_4` oznacza to, że sens kupna licencji grupowej pojawia się dopiero dla grup liczących 4 lub więcej osób, ponieważ dopiero wtedy koszt licencji grupowej jest równy lub niższy niż koszt czterech licencji indywidualnych.

Szczegółowe statystyki dla wariantów Duolingo zebrano w tabeli 8.1. Wzrost mnożnika ceny grupowej z 2 do 5 powoduje wzrost średniego kosztu na węzeł o 77% (z 0.554 do 0.982) oraz wydłużenie średniego czasu obliczeń o 47% (z 0.637 s do 0.938 s).

Tabela 8.1: Statystyki dla wariantów Duolingo (benchmark statyczny).

Konfiguracja	Metryka	Średnia	Odch. std.	Min	Max
duolingo_p_2	Czas [s]	0.637	1.210	0.000	7.722
	Koszt całkowity	49.152	39.533	8.000	177.000
	Koszt/węzeł	0.554	0.152	0.340	1.000
duolingo_p_4	Czas [s]	0.677	1.483	0.000	11.075
	Koszt całkowity	76.815	59.581	14.000	259.000
	Koszt/węzeł	0.860	0.171	0.680	1.520
duolingo_p_5	Czas [s]	0.938	2.721	0.000	26.589
	Koszt całkowity	87.855	67.852	17.000	300.000
	Koszt/węzeł	0.982	0.198	0.840	1.820

Analiza rozrzutu wartości w tabeli 8.1 pokazuje stabilność wyników. Odchylenie standardowe dla kosztu na węzeł pozostaje na niskim poziomie (0.152–0.198), co wskazuje na konsekwentność rozwiązań algorytmów w różnych instancjach. Warto zauważyć, że w konfiguracji duolingo_p_5 odnotowano najwyższy maksymalny czas obliczeń (26.589 s), co może wynikać z większej złożoności problemu przy wyższych kosztach planów grupowych. Zwiększone odchylenie standardowe dla czasów wykonania w tej konfiguracji (2.721 s) sugeruje większą wrażliwość algorytmów na charakterystykę konkretnej instancji sieci.



Rysunek 8.1: Koszt na węzeł w zależności od planu Duolingo (mediany kluczowych algorytmów).

Zmiana struktury licencji wpływa również na udział planów grupowych. W konfiguracji duolingo_p_2 ponad połowa przydziałów wykorzystuje licencję grupową, podczas gdy w wariancie duolingo_p_5 udział grup spada do 24%, co odzwierciedla rosnący koszt planu grupowego względem licencji indywidualnych.

8.1.2 Warianty dominowania rzymskiego

Na rys. 8.2 przedstawiono mediany kosztu na węzeł dla konfiguracji, w których koszt licencji grupowej wynosi odpowiednio trzykrotność (roman_p_3), czterokrotność (roman_p_4) oraz

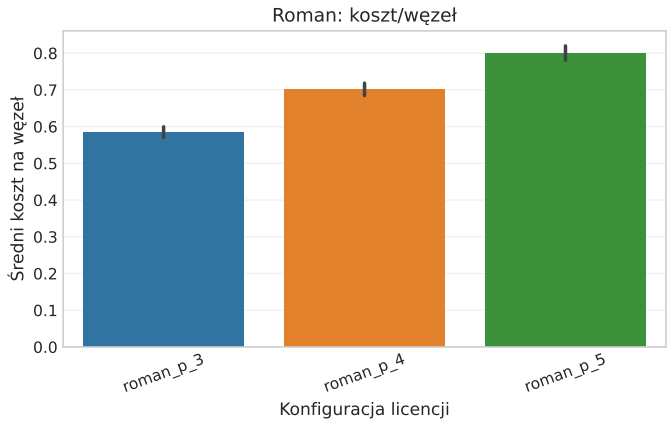
pięciokrotność (roman_p_5) ceny licencji indywidualnej. Podobnie jak w przypadku planów Duolingo, wyższe mnożniki prowadzą do wzrostu kosztu na węzeł, co wynika z preferencji algorytmów do korzystania z licencji indywidualnych przy wyższych kosztach planów grupowych. Wariant roman_p_5 charakteryzuje się najwyższym kosztem, co odzwierciedla ograniczoną opłacalność licencji grupowych w tej konfiguracji.

Szczegółowe statystyki dla wariantów dominowania rzymskiego zebrano w tabeli 8.2. Wzrost mnożnika ceny grupowej z 3 do 5 powoduje wzrost średniego kosztu na węzeł o 37% (z 0.585 do 0.800) oraz nieznaczne wydłużenie średniego czasu obliczeń (z 0.477 s do 0.493 s).

Tabela 8.2: Statystyki dla wariantów dominowania rzymskiego (benchmark statyczny).

Konfiguracja	Metryka	Średnia	Odch. std.	Min	Max
roman_p_3	Czas [s]	0.477	1.038	0.000	9.522
	Koszt całkowity	50.261	40.552	7.000	218.000
	Koszt/węzeł	0.585	0.199	0.260	1.170
roman_p_4	Czas [s]	0.475	1.125	0.000	9.189
	Koszt całkowity	60.441	48.750	9.000	259.000
	Koszt/węzeł	0.701	0.232	0.340	1.400
roman_p_5	Czas [s]	0.493	1.289	0.000	12.314
	Koszt całkowity	68.995	55.987	11.000	300.000
	Koszt/węzeł	0.800	0.269	0.395	1.660

Zmiana struktury licencji wpływa również na udział planów grupowych (tabela 8.5). W konfiguracji roman_p_3 udział grup wynosi 43%, podczas gdy w wariancie roman_p_5 spada do 24%. Wyższe koszty planów grupowych prowadzą do preferencji algorytmów w kierunku licencji indywidualnych, co jest zgodne z obserwacjami dla innych rozszerzeń.



Rysunek 8.2: Koszt na węzeł dla wariantów dominowania rzymskiego.

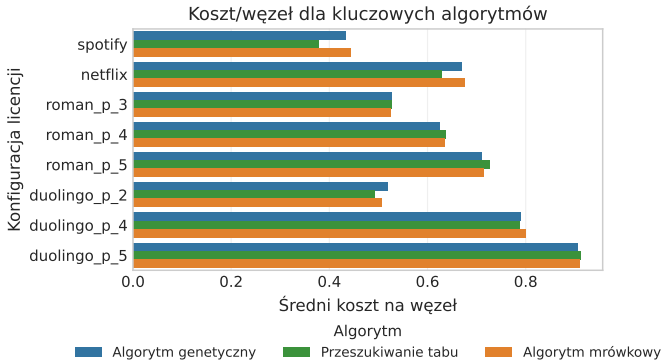
8.1.3 *Spotify i Netflix*

Spotify oferuje plan Duo, przeznaczony dla 2 osób. Umożliwia to efektywne dobieranie użytkowników w pary zamiast tworzenia dużych grup rodzinnych. Tabela 8.3 potwierdza, że przeszukiwanie tabu i algorytm genetyczny osiągają najniższe koszty (0.335 i 0.400 na węzeł), uzyskując wyniki niższe niż heurystyka zachłanna.

W przypadku Netflixa plan Standard (również dla 2 osób) stanowi wariant pośredni między licencją indywidualną a planem rodzinnym dla czterech kont, co pozwala metaheurystykom utrzymać koszt w zakresie od 0.60 do 0.65 przy czasie poniżej 1 s.

Tabela 8.3: Mediany dla konfiguracji Spotify i Netflix.

Konfiguracja	**Algorytm**	**Med. koszt/węzeł**	**Med. czas [s]**
spotify	Algorytm zachłanny	0.446	0.000
spotify	Algorytm genetyczny	0.400	0.229
spotify	Algorytm mrówkowy	0.391	1.040
spotify	Przeszukiwanie tabu	0.335	0.958
netflix	Algorytm zachłanny	0.682	0.000
netflix	Algorytm genetyczny	0.650	0.237
netflix	Algorytm mrówkowy	0.656	1.289
netflix	Przeszukiwanie tabu	0.604	0.999



Rysunek 8.3: Koszt na węzeł dla wszystkich rozszerzeń (mediany).

8.1.4 *Porównanie wszystkich rozszerzeń*

Podstawowe statystyki dla wszystkich konfiguracji zestawiono w tabeli 8.4. W każdym przypadku analizowano ten sam zestaw algorytmów, pomijając obserwacje z przekroczeniem limitu czasu.

Tabela 8.4: Statystyki agregowane dla rozszerzeń (benchmark statyczny).

Konfiguracja	Śr. koszt/węzeł	Śr. czas [s]
duolingo_p_2	0.554	0.637
duolingo_p_4	0.860	0.677
duolingo_p_5	0.982	0.938
spotify	0.479	0.547
netflix	0.729	0.623
roman_p_3	0.585	0.477
roman_p_4	0.701	0.475
roman_p_5	0.800	0.493

Analiza agregowanych statystyk w tabeli 8.4 ujawnia wyraźne różnice między rodzinami konfiguracji. W przypadku wariantów Duolingo obserwuje się silną korelację między mnożnikiem ceny grupowej a kosztem na węzeł – wzrost parametru z 2 do 5 prowadzi do zwiększenia kosztu o 77% (z 0.554 do 0.982). Jednocześnie czas obliczeń wydłuża się o 47%, co wskazuje na rosnącą złożoność problemu przy wyższych kosztach planów grupowych.

Warianty dominowania rzymskiego charakteryzują się większą stabilnością obliczeniową, z czasami wykonania w zakresie 0.477–0.493 s niezależnie od parametru p. Wzrost kosztu na węzeł jest bardziej umiarkowany (37% przy wzroście p z 3 do 5), co sugeruje odmienną strukturę przestrzeni rozwiązań w porównaniu z planami Duolingo.

Najkorzystniejszy stosunek kosztu do wydajności obliczeniowej wykazuje konfiguracja Spotify (0.479 kosztu na węzeł przy 0.547 s), co potwierdza skuteczność wprowadzenia planu pośredniego (Duo) między licencjami indywidualnymi a rodzinnymi. Netflix zajmuje pozycję pośrednią z kosztem 0.729 na węzeł, pozostając jednak konkurencyjny pod względem czasu obliczeń (0.623 s).

8.1.5 Analiza wpływu liczby użytkowników na koszty

Struktura wykorzystania licencji zmienia się istotnie (tabela 8.5). W Spotify licencje grupowe odpowiadają za 64% przydziałów (plan Duo + rodzina), podczas gdy w wariancie `duolingo_p_5` udział grup spada do 24%. W Netflixie większość przydziałów to plany Standard/Premium (udział 68%).

Tabela 8.5: Udział licencji grupowych i indywidualnych (benchmark statyczny).

Konfiguracja	Udział grup	Udział indywidualnych
duolingo_p_2	0.56	0.44
duolingo_p_4	0.34	0.66
duolingo_p_5	0.24	0.76
spotify	0.64	0.36
netflix	0.68	0.32
roman_p_3	0.43	0.57
roman_p_4	0.34	0.66
roman_p_5	0.24	0.76

8.2 Rozszerzenia w środowisku dynamicznym

W środowisku dynamicznym przeanalizowano te same osiem konfiguracji rozszerzeń, stosując identyczną metodologię jak w benchmarku statycznym. Każda konfiguracja została przetestowana na 272 instancjach dynamicznych, obejmujących różne rozmiary sieci i scenariusze zmian.

8.2.1 Statystyki agregowane

Tabela 8.6 przedstawia statystyki zagregowane według rodzin konfiguracji. Warianty Duolingo charakteryzują się najwyższymi kosztami średnimi (73.83 na instancję) oraz najdłuższymi czasami wykonania (0.97 s), co wynika z konieczności rozważania licencji o pojemności 6 osób. Problem dominowania rzymskiego wykazuje umiarkowane koszty (59.96) przy porównywalnym czasie obliczeń (0.85 s). Konfiguracje rzeczywistych serwisów – Spotify i Netflix – osiągają najkorzystniejsze wyniki, z najniższymi kosztami odpowiednio 43.32 i 66.58.

Tabela 8.6: Statystyki zagregowane według rodzin konfiguracji (benchmark dynamiczny).

Rodzina	Śr. czas [s]	Śr. koszt całk.	Śr. koszt/węzeł	Liczba obs.
Duolingo	0.969	73.83	0.792	3265
Roman	0.853	59.96	0.661	3408
Netflix	0.891	66.58	0.714	1074
Spotify	0.881	43.32	0.467	1088

8.2.2 Porównanie szczegółowe konfiguracji

Tabela 8.7 zawiera szczegółowe statystyki dla wszystkich wariantów rozszerzeń. W rodzinie Duolingo obserwuje się wyraźny wzrost kosztu na węzeł wraz ze wzrostem mnożnika ceny grupowej – od 0.539 w konfiguracji `duolingo_p_2` do 0.980 w `duolingo_p_5`. Podobną tendencję wykazują warianty dominowania rzymskiego, gdzie koszt rośnie z 0.554 (`roman_p_3`) do 0.763 (`roman_p_5`).

Tabela 8.7: Szczegółowe statystyki dla rozszerzeń (benchmark dynamiczny).

Konfiguracja	Śr. czas [s]	Śr. koszt całk.	Śr. koszt/węzeł	Liczba obs.
duolingo_p_2	0.939	50.22	0.539	1088
duolingo_p_4	0.987	80.17	0.855	1073
duolingo_p_5	0.980	90.94	0.980	1104
spotify	0.881	43.32	0.467	1088
netflix	0.891	66.58	0.714	1074
roman_p_3	0.798	50.30	0.554	1136
roman_p_4	0.930	60.41	0.666	1136
roman_p_5	0.829	69.18	0.763	1136

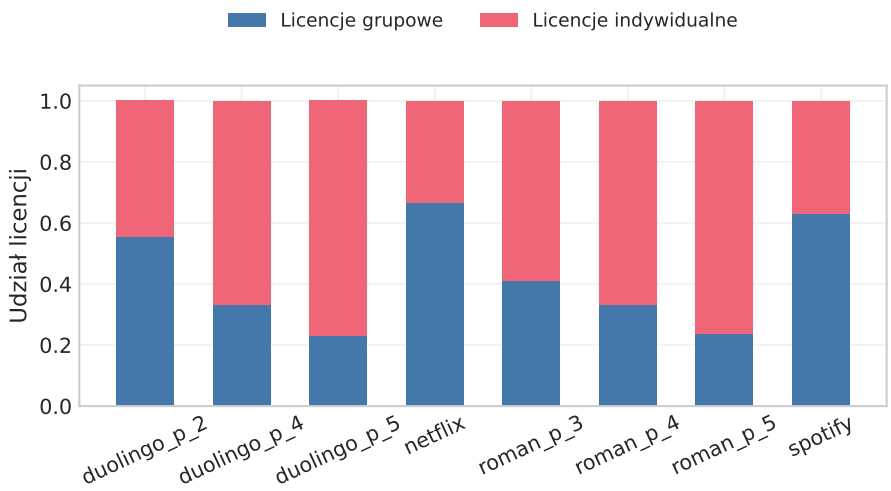
Konfiguracja Spotify potwierdza swoją przewagę osiągając najniższy koszt na węzeł (0.467), co stanowi wynik o 13% lepszy niż w najbliższej konfiguracji `roman_p_3`. Plan Duo umożliwia efektywne parowanie użytkowników, co przekłada się na oszczędności w całym spektrze rozmiarów sieci. Netflix zajmuje pozycję pośrednią z kosztem 0.714 na węzeł, pozostając konkurencyjny wobec wariantów o wysokich mnożnikach.

8.2.3 Struktura wykorzystania licencji

Analiza składu licencji (tabela 8.8) ujawnia znaczące różnice w strategiach przydzielania. Konfiguracje o niskich mnożnikach (`duolingo_p_2`, `spotify`) preferują licencje grupowe, osiągając udział odpowiednio 55% i 63%. W przeciwieństwie do tego, wysokie koszty planów grupowych w `duolingo_p_5` i `roman_p_5` prowadzą do dominacji licencji indywidualnych (76% i 76% udziału).

Tabela 8.8: Struktura wykorzystania licencji (benchmark dynamiczny).

Konfiguracja	Udział grup [%]	Udział indywidualnych [%]
duolingo_p_2	55.4	44.6
duolingo_p_4	33.3	66.7
duolingo_p_5	23.0	77.0
spotify	63.0	37.0
netflix	66.8	33.2
roman_p_3	41.1	58.9
roman_p_4	33.4	66.6
roman_p_5	23.8	76.2



Rysunek 8.4: Udział licencji grupowych i indywidualnych w rozszerzeniach dynamicznych.

Netflix wykazuje najwyższy udział planów grupowych (67%), co wynika z atrakcyjności planów Standard i Premium dla grup 2-4 osobowych. Ta konfiguracja stanowi wariant pośredni między licencjami indywidualnymi a planami o dużej pojemności, umożliwiając algorytmom elastyczne dopasowanie do struktury sieci.

8.2.4 Porównanie z benchmarkiem statycznym

Zestawienie wyników dynamicznych ze statycznymi (tabela 8.9) pokazuje ogólną zgodność trendów. Koszty w środowisku dynamicznym pozostają na podobnym poziomie – różnice nie przekraczają 3% dla większości konfiguracji. Czasy wykonania wydłużają się o 30-60%, co odzwierciedla dodatkową złożoność związaną z dynamicznym rebalansowaniem.

Tabela 8.9: Porównanie wyników statycznych i dynamicznych.

Konfiguracja	**Koszt stat.**	**Koszt dyn.**	**Czas stat.**	**Czas dyn.**
duolingo_p_2	0.554	0.539	0.637	0.939
duolingo_p_4	0.860	0.855	0.677	0.987
duolingo_p_5	0.982	0.980	0.938	0.980
spotify	0.479	0.467	0.547	0.881
netflix	0.729	0.714	0.623	0.891
roman_p_3	0.585	0.554	0.477	0.798
roman_p_4	0.701	0.666	0.475	0.930
roman_p_5	0.800	0.763	0.493	0.829

Najstabilniejsze wyniki wykazuje konfiguracja `duolingo_p_5`, gdzie koszty różnią się jedynie o 0.002 między benchmarkami. Największą poprawę w środowisku dynamicznym odnotowano dla `roman_p_4` (redukcja kosztu o 5%) i `roman_p_5` (redukcja o 4.6%). Może to wynikać z lepszego wykorzystania możliwości rebalansowania w mniejszych grupach charakterystycznych dla tych konfiguracji.

8.3 Wnioski

Przeprowadzone badania nad rozszerzeniami modelu licencjonowania pozwoliły na kompleksową analizę wpływu struktury cenowej na optymalizację kosztów w sieciach społecznych. Kluczowe wnioski z analizy ośmiu wariantów licencyjnych można podzielić na trzy główne obszary.

Wpływ mnożnika ceny grupowej na efektywność algorytmów. Badania wykazały wyraźną zależność między stosunkiem ceny licencji grupowej do indywidualnej a skutecznością optymalizacji. W wariantach Duolingo wzrost mnożnika z 2 do 5 powoduje zwiększenie średniego kosztu na węzeł o 77% (z 0.554 do 0.982) oraz wydłużenie czasu obliczeń o 47%. Podobną, choć łagodniejszą tendencję obserwuje się w wariantach dominowania rzymskiego, gdzie wzrost parametru p z 3 do 5 prowadzi do 37% wzrostu kosztu przy stabilnym czasie wykonania. Te wyniki wskazują, że metaheurystyki są najbardziej efektywne przy niskich mnożnikach, gdzie licencje grupowe pozostają opłacalne.

Znaczenie planów pośrednich w strukturze licencji. Analiza konfiguracji rzeczywistych serwisów ujawniła istotną rolę licencji o pojemności pośredniej. Spotify z planem Duo (pojemność 2) osiąga najkorzystniejszy stosunek kosztu do wydajności (0.467 kosztu na węzeł przy 0.547s czasu), podczas gdy Netflix z planem Standard oferuje skuteczne wypełnienie luki między licencjami indywidualnymi a rodzinnymi. W obu przypadkach udział licencji grupowych przekracza 63%, co potwierdza efektywność elastycznego spektrum opcji licencyjnych.

Stabilność wyników w środowisku dynamicznym. Porównanie benchmarków statycznego i dynamicznego wykazało wysoką zgodność rezultatów – różnice w kosztach na węzeł nie przekraczają 3% dla większości konfiguracji. Jednocześnie zaobserwowano wzrost czasów wykonania o 30-60%, co odzwierciedla dodatkową złożoność związaną z dynamicznym rebalansowaniem. Najstabilniejsze wyniki uzyskano w konfiguracji `duolingo_p_5`, gdzie różnica między środowiskami wynosi jedynie 0.002. Te obserwacje potwierdzają, że proponowane rozszerzenia modelu zachowują skuteczność w realistycznych scenariuszach z czasowymi zmianami struktury sieci.

Otrzymane wyniki mają istotne implikacje praktyczne dla projektowania systemów licencjonowania. Po pierwsze, wprowadzenie planów o pojemności pośredniej może znacząco poprawić efektywność kosztową bez zwiększenia złożoności obliczeniowej. Po drugie, przy wysokich mnożnikach ceny grupowej (powyżej 4-5) korzyści z metaheurystyk maleją, co sugeruje ekonomiczne ograniczenia optymalizacji algorytmicznej. Po trzecie, stabilność wyników w środowisku dynamicznym wskazuje na praktyczną stosowalność proponowanych rozwiązań w rzeczywistych systemach z fluktuacją użytkowników.

9. PODSUMOWANIE

Praca pokazuje pełny cykl badawczy dotyczący optymalizacji kosztów licencji grupowych w sieciach społecznościowych. Najpierw sformalizowano model. Następnie porównano algorytmy deterministyczne i metaheurystyczne. Dalej przeprowadzono symulacje dynamiczne. Na końcu rozszerzono analizę o dodatkowe plany licencyjne. Uzyskano spójny obraz działania metod w wielu scenariuszach. Poniżej zebrano główne wyniki i wskazano możliwe kierunki dalszych badań.

9.1 Wyniki

9.1.1 Model i metody

Rozdziały 1–4 definiują problem jako uogólnienie dominowania rzymskiego z ograniczeniami pojemności oraz różnymi typami licencji. Wykazano wysoką złożoność obliczeniową. Rozdział 5 prezentuje pełen zestaw metod: dokładny algorytm ILP, heurystyki konstrukcyjne (m.in. algorytm zachłanny, podejście przez zbiór dominujący), metaheurystyki (algorytm genetyczny, algorytm mrówkowy, przeszukiwanie tabu, wyżarzanie symulowane) oraz algorytm losowy. Wszystkie implementacje mają wspólny interfejs, co ułatwiło porównania.

9.1.2 Eksperymenty statyczne

Rozdział 6 potwierdza hierarchię jakości: $ILP >$ algorytm mrówkowy $>$ przeszukiwanie tabu/algorytm genetyczny $>$ wyżarzanie symulowane $>$ heurystyki konstrukcyjne $>$ algorytm losowy. Różnice są istotne statystycznie (test Friedmana oraz porównania Nemenyi'ego). Grafy bezskalowe okazały się łatwiejsze do pokrycia z powodu hubów. Czas działania metaheurystyk rósł szybko, a heurystyki konstrukcyjne utrzymywały czasy rzędu milisekund. W praktyce wyróżniono trzy zakresy: dla małych grafów warto stosować ILP jako punkt odniesienia; dla średnich grafów najlepsze są metaheurystyki; dla dużych grafów albo gdy liczy się szybkość, użyteczną aproksymację daje algorytm zachłanny.

9.1.3 Symulacje dynamiczne

Rozdział 7 przedstawia wpływ zmian w sieci na koszt i czas. Ciepły start konsekwentnie obniżał koszt o 6–14% (syntetyczne) i 7–12% (realistyczne), przy czasie rebalansowania 1–3 s. Intensywniejsze mutacje zwiększały głównie czas, a mniej wpływały na koszt. Najdokładniejszy pozostawał algorytm mrówkowy, natomiast przeszukiwanie tabu i algorytm genetyczny dawały lepszy kompromis czasowy. W danych realistycznych najniższe koszty uzyskano w wariancie Spotify (mediana ≈ 0.41 na węzeł).

9.1.4 Rozszerzenia licencyjne

Rozdział 8 analizuje osiem wariantów taryf. W planach Duolingo i Roman o opłacalności decydowały rozmiar grupy i koszt planu. Przy tańszej grupie (`duolingo_p_2`) metaheurystyki redukowały koszt o 10–20% względem heurystyk. Przy droższych planach (`duolingo_p_5`, `roman_p_5`) przewaga spadała do kilku procent. Dodanie planu Duo (Spotify) oraz planów Standard/Premium (Netflix) umożliwiło nowe parowania użytkowników i obniżyło koszt na węzeł (mediana 0.40 w Spotify wobec 0.50 w `duolingo_p_2`). W dynamice konfiguracje z planem pośrednim utrzymywały najkorzystniejszy koszt i stabilny czas, a metaheurystyki dawały 5–12% oszczędności względem algorytmu zachłannego.

9.2 Rekomendacje praktyczne

Dobór algorytmu. Dla instancji do około 200 węzłów algorytm ILP jest najlepszym punktem odniesienia. Dla większych sieci zalecane są algorytm mrówkowy albo przeszukiwanie tabu; w środowisku dynamicznym krótsze czasy rebalansowania daje przeszukiwanie tabu.

Strategia inicjalizacji. Ciepły start metaheurystyk, czyli start z poprzedniego rozwiązania, obniża koszt o 6–14% przy niewielkim koszcie czasowym. W systemach aktualizowanych częściowo powinien to być standard.

Polityka licencyjna. Tańsze plany rodzinne, w szczególności warianty pośrednie (np. Duo), realnie zmniejszają koszt końcowy. Zbyt wysokie p w planach grupowych sprzyja licencjom indywidualnym i ogranicza zyski z optymalizacji.

9.3 Kierunki dalszych badań

Gwarancje aproksymacji. Warto poszukać teoretycznych ograniczeń jakości rozwiązań dla wybranych klas grafów lub modeli losowych. **Modele stochastyczne.** Ujęcie niepewności w dostępie użytkowników i ewolucji sieci może zwiększyć odporność rozwiązań (np. podejścia dwuetapowe lub bayesowskie). **Wielokryterialność i koszty operacyjne.** Rozszerzenie funkcji celu o sprawiedliwość, ryzyko czy koszt migracji lepiej odzwierciedli praktykę. **Optymalizacja hiperparametrów.** Automatyczne strojenie parametrów metaheurystyk (np. techniki z rodziny AutoML) może poprawić stosunek kosztu do czasu bez ręcznego dostrajania. **Integracja z praktyką.** Biblioteka daje podstawy do wdrożeń w systemach rekomendacji planów rodzinnych; naturalnym krokiem jest eksperyment online na rzeczywistych danych.

9.4 Zakończenie

Przedstawiony model, zestaw algorytmów oraz eksperymenty statyczne i dynamiczne pokazują, że mimo wysokiej złożoności można uzyskać rozwiązania dobrej jakości w akceptowalnym czasie. Kluczowy jest dobór metody do wielkości i dynamiki sieci oraz rozsądna polityka licencyjna. Wyniki stanowią podstawę do dalszych prac oraz do zastosowań w usługach subskrypcyjnych, gdzie modele rodzinne stają się standardem.

WYKAZ LITERATURY

1. *Subscription Economy Index* [Industry report]. 2024. Dostępne także z: https://www.zuora.com/resource/subscription-economy-index/.
2. *Subscription Price Trends 2024* [Industry report]. 2024. Dostępne także z: https://recurly.com/press/recurly-releases-its-2024-state-of-subscriptions-report/.
3. *Duolingo Family Plan* [Product page]. 2024. Dostępne także z: https://support.duolingo.com/hc/pl/articles/6159959913243-Plan-rodzinny-Super-Duolingo.
4. GAREY, Michael R.; JOHNSON, David S. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.
5. SHARMA, D.; ARAVIND, N. R.; CHOUDUM, S. A. Survey on Roman {2}-Domination. *Mathematics*. 2024, t. 12, nr. 17, s. 2771. Dostępne z DOI: 10.3390/math12172771.
6. LOKSHTANOV, Daniel; MARX, Dániel; SAURABH, Saket. Known Algorithms on Capacitated Domination. W: *Proceedings of IWPEC*. 2011, s. 351–362. Dostępne także z: https://sites.cs.ucsb.edu/~daniello/papers/capdomset.pdf.
7. BRANDES, Ulrik; ERLEBACH, Thomas (red.). *Network Analysis: Methodological Foundations*. T. 3418. Springer, 2005. Lecture Notes in Computer Science. Dostępne z DOI: 10.1007/978-3-540-31955-9.
8. NETTLETON, David F. Data mining of social networks represented as graphs. *Computer Science Review*. 2013, t. 7, nr. 1, s. 1–34. Dostępne z DOI: 10.1016/j.cosrev.2012.12.001.
9. SPOTIFY POLSKA. *Spotify Premium - Plany i ceny (Polska)*. 2025. Dostępne także z: https://www.spotify.com/pl/premium/. Stan na 09.2025: Individual - 23.99 PLN, Duo - 30.99 PLN, Family - 37.99 PLN miesięcznie.
10. SPOTIFY POLSKA. *Spotify Premium – Plany i ceny (Polska)*. 2025. Dostępne także z: https://www.spotify.com/pl/premium/. Stan na 09.2025: Basic – 33.00 PLN, Standard – 49.00 PLN, Premium – 67.00 PLN miesięcznie.
11. DUOLINGO. *Super Duolingo - ceny w aplikacji mobilnej (Polska)* [Aplikacja mobilna Duolingo, wersja na Android]. 2025. Stan na 09.2025: Individual - 13.99 PLN, Family - 29.17 PLN miesięcznie.
12. HAYNES, Teresa W.; HEDETNIEMI, Stephen T.; SLATER, Peter J. *Fundamentals of Domination in Graphs*. Marcel Dekker, 1998.
13. *Dominating set* [Wikipedia]. 2024. Dostępne także z: https://en.wikipedia.org/wiki/Dominating_set.
14. POUREIDI, Abolfazl; FATHALI, Jafar. Algorithmic results in Roman dominating functions on graphs. *Information Processing Letters*. 2023, t. 182, s. 106363. Dostępne z DOI: 10.1016/j.ipl.2023.106363.

15. PANDA, B. S.; RANA, Soumyashree; MISHRA, Sounaka. On the complexity of co-secure dominating set problem. *Information Processing Letters*. 2024, t. 185, s. 106463. Dostępne z DOI: 10.1016/j.ipl.2023.106463.
16. ALIMONTI, Paola; KANN, Viggo. Some APX-completeness results for cubic graphs. *Theoretical Computer Science*. 2000, t. 237, nr. 1–2, s. 123–134. Dostępne z DOI: 10.1016/S0304-3975(99)00426-0.
17. BERMAN, Piotr; FUJITO, Toshihiro. On the Approximation Properties of the Independent Set Problem in Degree 3 Graphs. W: *Algorithms and Data Structures*. Springer, 1995, t. 955, s. 449–460. Lecture Notes in Computer Science. ISBN 978-3-540-60165-6.
18. FAVARON, Odile; KARAMI, Hosein; KHOEILAR, Reza; SHEIKHOLESLAMI, Seyed Mahmoud. On the Roman domination number of a graph. *Discrete Mathematics*. 2009, t. 309, nr. 10, s. 3447–3451. Dostępne z DOI: 10.1016/j.disc.2008.09.043.
19. COCKAYNE, Ernie J.; DREYER Paul A., Jr.; HEDETNIEMI, Sandra M.; HEDETNIEMI, Stephen T. Roman domination in graphs. *Discrete Mathematics*. 2004, t. 278, nr. 1–3, s. 11–22. Dostępne z DOI: 10.1016/j.disc.2003.06.004.
20. CHAUDHARY, Juhi; PRADHAN, Dinabandhu. Roman 3-domination in graphs: Complexity and algorithms. *Discrete Applied Mathematics*. 2024, t. 354, s. 301–325. Dostępne z DOI: 10.1016/j.dam.2022.09.017.
21. GHAFFARI-HADIGHEH, Alireza. Roman domination problem with uncertain positioning and deployment costs. *Soft Computing*. 2019, t. 24, nr. 4, s. 2637–2645. Dostępne z DOI: 10.1007/s00500-019-03811-z.
22. CHAMBERS, Erin W.; KINNERSLEY, Bill; PRINCE, Noah; WEST, Douglas B. Extremal problems for Roman domination. *SIAM Journal on Discrete Mathematics*. 2009, t. 23, nr. 3, s. 1575–1596. Dostępne z DOI: 10.1137/070699688.
23. KUHN, Fabian. *Network Algorithms (Graduate Course) – Lecture Notes* [Course notes]. 2012. Dostępne także z: https://algo.inf.uni-freiburg.de/teaching/ss12/network-algorithms/. Graduate course at the University of Freiburg, Summer Term 2012.
24. PARRA INZA, Ernesto; VAKHANIA, Nodari; SIGARRETA ALMIRA, Jose Maria; HERNANDEZ-AGUILAR, Jose Alberto. Approximating a Minimum Dominating Set by Purification. *Algorithms*. 2024, t. 17, nr. 6, s. 258. Dostępne z DOI: 10.3390/a17060258.
25. ERDŐS, Paul; RÉNYI, Alfréd. On the evolution of random graphs. *Publicationes Mathematicae*. 1960, t. 5, s. 17–61.
26. BARABÁSI, Albert-László; ALBERT, Réka. Emergence of scaling in random networks. *Science*. 1999, t. 286, nr. 5439, s. 509–512. Dostępne z DOI: 10.1126/science.286.5439.509.
27. WATTS, Duncan J.; STROGATZ, Steven H. Collective dynamics of small-world networks. *Nature*. 1998, t. 393, s. 440–442. Dostępne z DOI: 10.1038/30918.
28. PROJECT, Stanford Network Analysis. *SNAP Datasets* [https://snap.stanford.edu/data/]. 2024. Dostęp: 2025-09.

29. MCAULEY, Julian; LESKOVEC, Jure. Learning to Discover Social Circles in Ego Networks. W: *Advances in Neural Information Processing Systems 25*. 2012, s. 548–556. Dostępne także z: https://papers.nips.cc/paper/4532-learning-to-discover-social-circles-in-ego-networks. Proceedings of NeurIPS 2012.
30. UGANDER, Johan; KARRER, Brian; BACKSTROM, Lars; MARLOW, Cameron. *The Anatomy of the Facebook Social Graph* [arXiv preprint arXiv:1111.4503]. 2011. Dostępne także z: https://arxiv.org/abs/1111.4503.
31. STANLEY, Richard P. *Enumerative Combinatorics, Volume 1*. Cambridge University Press, 1997. ISBN 9780521789875.
32. HOLLAND, J. H. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, 1975.
33. GOLDBERG, D. E. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989.
34. GLOVER, F. Tabu Search–Part I. *ORSA Journal on Computing*. 1989, t. 1, nr. 3, s. 190–206. Dostępne z DOI: 10.1287/ijoc.1.3.190.
35. DORIGO, M.; GAMBARDELLA, L. M. Ant Colony System: A Cooperative Learning Approach to the Traveling Salesman Problem. *IEEE Transactions on Evolutionary Computation*. 1997, t. 1, nr. 1, s. 53–66. Dostępne z DOI: 10.1109/4235.585892.
36. KIRKPATRICK, S.; GELATT, C. D.; VECCHI, M. P. Optimization by Simulated Annealing. *Science*. 1983, t. 220, nr. 4598, s. 671–680. Dostępne z DOI: 10.1126/science.220.4598.671.
37. ALBERT, R.; BARABÁSI, A.-L. Statistical mechanics of complex networks. *Reviews of Modern Physics*. 2002, t. 74, nr. 1, s. 47–97.
38. KAMOLA, Mariusz. Dynamika triad w serwisie Instagram. W: *VII Krajowa Konferencja Sieci Telekomunikacyjne i Komputerowe, Gliwice*. 2016, s. 1–10. Dostępne także z: https://www.ia.pw.edu.pl/~mkamola/Kamola16a.pdf.
39. WATTS, D.J.; STROGATZ, S.H. Collective dynamics of 'small-world' networks. *Nature*. 1998, t. 393, nr. 6684, s. 440–442.

A. WYBRANE FRAGMENTY IMPLEMENTACJI

A.1 Organizacja skryptów eksperymentalnych

Środowisko obliczeniowe zorganizowano jako spójny zestaw modułów Pythona: pakiety do benchmarków statycznych, osobne moduły do symulacji dynamicznych i rozszerzeń, a także proste komendy CLI do szybkiej diagnostyki pojedynczych algorytmów. Wspólny rdzeń dba o jednolite budowanie i walidację rozwiązań, a skrypty analityczne tworzą raporty i wykresy użyte w częściach eksperymentalnych.

A.2 Algorytmy dokładne

A.2.1 Program całkowitoliczbowy

Pełna implementacja solvera ILP wykorzystującego bibliotekę PuLP z solverem CBC do wyznaczania rozwiązań wzorcowych.

```
class ILPSolver(Algorithm):
    def solve(self, graph: nx.Graph, license_types: Sequence[LicenseType], **kwargs) -> Solution:
        # G = (V,E), N[i] = neighbors(i) union {i}
        nodes: list[Any] = list(graph.nodes())
        Nhood: dict[Any, set[Any]] = {i: set(graph.neighbors(i)) | {i} for i in nodes}
        degp1: dict[Any, int] = {i: len(Nhood[i]) for i in nodes}

        model = pulp.LpProblem("graph_licensing_optimization", pulp.LpMinimize)

        # active[i,t] = 1 gdy wlasciciel i otwiera grupe typu t
        active: dict[tuple[Any, int], pulp.LpVariable] = {}
        for i in nodes:
            for t_idx, lt in enumerate(license_types):
                feasible_owner_type = (lt.min_capacity <= degp1[i]) and (lt.max_capacity >= 1)
                if feasible_owner_type:
                    active[i, t_idx] = pulp.LpVariable(f"a_{i}_{t_idx}", cat="Binary")
                else:
                    # eliminacja niemozliwych par i,t
                    active[i, t_idx] = pulp.LpVariable(f"a_{i}_{t_idx}", lowBound=0, upBound=0, cat="Binary")

        # assign[i,j,t] = 1 gdy j nalezy do grupy wlasciciela i typu t
        assign: dict[tuple[Any, Any, int], pulp.LpVariable] = {}
        for i in nodes:
            for t_idx, lt in enumerate(license_types):
                if active[i, t_idx].upBound == 0:
                    continue
                if lt.max_capacity == 1:
                    # typ indywidualny, tylko wlasciciel
                    assign[i, i, t_idx] = pulp.LpVariable(f"x_{i}_{i}_{t_idx}", cat="Binary")
```

```
        else:
            for j in Nhood[i]:
                assign[i, j, t_idx] = pulp.LpVariable(f"x_{i}_{j}_{t_idx}", cat="Binary")

# cel: min sum c_t * active[i,t]
model += pulp.lpSum(active[i, t_idx] * license_types[t_idx].cost for i in nodes
                  for t_idx in range(len(license_types)))

# co najwyzej jedna licencja na wlasciciela
for i in nodes:
    model += pulp.lpSum(active[i, t_idx] for t_idx in range(len(license_types))) <= 1

# pokrycie dokladnie raz
for j in nodes:
    model += pulp.lpSum(assign.get((i, j, t_idx), 0) for i in Nhood[j]
                      for t_idx in range(len(license_types))) == 1

# sprzezenie i pojemnosc
for i in nodes:
    for t_idx, lt in enumerate(license_types):
        if active[i, t_idx].upBound == 0:
            continue
        # wlasciciel nalezy do swojej grupy
        model += assign.get((i, i, t_idx), 0) == active[i, t_idx]
        # brak przypisan bez aktywacji
        for j in Nhood[i]:
            var = assign.get((i, j, t_idx))
            if var is not None:
                model += var <= active[i, t_idx]
        # min i max pojemnosci tylko gdy aktywna
        group_size = pulp.lpSum(assign.get((i, j, t_idx), 0) for j in Nhood[i])
        model += group_size <= active[i, t_idx] * lt.max_capacity
        model += group_size >= active[i, t_idx] * lt.min_capacity

# rozwiazanie ilp
solver = pulp.PULP_CBC_CMD(msg=False)
model.solve(solver)
status = pulp.LpStatus.get(model.status, "Unknown")

# fallback gdy brak rozwiazania dopuszczalnego
if status in ("Infeasible", "Undefined", "Unbounded", "Not Solved"):
    singles = [lt for lt in license_types if lt.min_capacity <= 1 <= lt.max_capacity]
    if not singles:
        raise RuntimeError(f"ILP {status}: no single license available")
    lt = min(singles, key=lambda x: x.cost)
    groups = [LicenseGroup(license_type=lt, owner=i, additional_members=frozenset()) for i in nodes]
    return Solution(groups=tuple(groups))

# ekstrakcja rozwiazania
```

```
        groups = []
        for i in nodes:
            for t_idx, lt in enumerate(license_types):
                a = active.get((i, t_idx))
                a_val = float(a.varValue) if a is not None and a.varValue is not None else 0.0
                if a_val > 0.5:
                    members: set[Any] = set()
                    for j in Nhood[i]:
                        var = assign.get((i, j, t_idx))
                        v_val = float(var.varValue) if var is not None and var.varValue is not None else 0.0
                        if v_val > 0.5:
                            members.add(j)
                    if members:
                        groups.append(
                            LicenseGroup(
                                license_type=lt,
                                owner=i,
                                additional_members=frozenset(members - {i}),
                            )
                        )

        return Solution(groups=tuple(groups))
```

A.3 Algorytmy metaheurystyczne

A.3.1 Algorytm genetyczny

Główne fragmenty implementacji algorytmu genetycznego z elityzmem, selekcją turniejową oraz operatorami krzyżowania i mutacji.

```
class GeneticAlgorithm(Algorithm):
    def __init__(self, population_size: int = 30, num_generations: int = 40,
                 elite_fraction: float = 0.2, crossover_rate: float = 0.6):
        self.population_size = max(2, population_size)
        self.num_generations = max(1, num_generations)
        self.elite_fraction = max(0.0, min(1.0, elite_fraction))
        self.crossover_rate = max(0.0, min(1.0, crossover_rate))
        self.validator = SolutionValidator()

    def solve(self, graph: nx.Graph, license_types: Sequence[LicenseType], **kwargs) -> Solution:
        population = self._init_population(graph, license_types, initial)
        best = min(population, key=lambda s: s.total_cost)

        for _ in range(num_generations):
            population.sort(key=lambda s: s.total_cost)
            elite_count = max(1, int(self.elite_fraction * self.population_size))
            new_pop: list[Solution] = population[:elite_count]

            while len(new_pop) < self.population_size:
                if random.random() < self.crossover_rate and len(population) >= 2:
```

```
                p1 = self._tournament_selection(population)
                p2 = self._tournament_selection(population)
                child = self._crossover(p1, p2, graph, license_types)
                if not self.validator.is_valid_solution(child, graph):
                    base = min([p1, p2], key=lambda s: s.total_cost)
                    child = self._mutate(base, graph, license_types)
            else:
                parent = self._tournament_selection(population)
                child = self._mutate(parent, graph, license_types)
            new_pop.append(child)

        population = new_pop
        current_best = min(population, key=lambda s: s.total_cost)
        if current_best.total_cost < best.total_cost:
            best = current_best
    return best

def _crossover(self, p1: Solution, p2: Solution, graph: nx.Graph,
               license_types: Sequence[LicenseType]) -> Solution:
    def eff(g):
        return (g.license_type.cost / max(1, g.size), -g.size)

    candidates = list(p1.groups) + list(p2.groups)
    candidates.sort(key=eff)
    used = set()
    chosen: list = []
    for g in candidates:
        if used.isdisjoint(g.all_members):
            chosen.append(g)
            used.update(g.all_members)

    # Domknięcie niepokrytych węzłów algorytmem zachłannym
    uncovered = set(graph.nodes()) - used
    if uncovered:
        H = graph.subgraph(uncovered)
        filler = GreedyAlgorithm().solve(H, list(license_types))
        for fg in filler.groups:
            if set(fg.all_members).issubset(uncovered):
                chosen.append(fg)

    child = SolutionBuilder.create_solution_from_groups(chosen)
    if not self.validator.is_valid_solution(child, graph):
        return GreedyAlgorithm().solve(graph, list(license_types))
    return child

def _mutate(self, solution: Solution, graph: nx.Graph,
            license_types: Sequence[LicenseType]) -> Solution:
    neighbors = MutationOperators.generate_neighbors(solution, graph, license_types, k=5)
    valid_neighbors = [s for s in neighbors if self.validator.is_valid_solution(s, graph)]
```

```
        if not valid_neighbors:
            return solution
        return min(valid_neighbors, key=lambda s: s.total_cost)
```

A.3.2 Optymalizacja mrówkowa

Kluczowe fragmenty algorytmu mrówkowego z konstrukcją rozwiązań oraz aktualizacją śladu feromonowego.

```
class AntColonyOptimization(Algorithm):
    def __init__(self, alpha: float = 1.0, beta: float = 2.0, evaporation: float = 0.5,
                 q0: float = 0.9, num_ants: int = 20):
        self.alpha, self.beta, self.evap, self.q0, self.num_ants = alpha, beta, evaporation, q0, num_ants

    def solve(self, graph: nx.Graph, license_types: Sequence[LicenseType], **kwargs) -> Solution:
        pher = self._init_pher(graph, license_types)  # Inicjalizacja feromonu
        heur = self._init_heur(graph, license_types)  # Informacja heurystyczna
        best = GreedyAlgorithm().solve(graph, license_types)
        self._deposit(pher, best)

        for _ in range(max_iter_aco):
            improved = False
            for _ in range(num_ants):
                cand = self._construct(graph, license_types, pher, heur)
                ok, _ = self.validator.validate(cand, graph)
                if ok and cand.total_cost < best.total_cost:
                    best, improved = cand, True
            self._evaporate(pher)
            self._deposit(pher, best)
        return best

    def _construct(self, graph: nx.Graph, lts: Sequence[LicenseType],
                   pher: dict, heur: dict) -> Solution:
        uncovered: set[Any] = set(graph.nodes())
        groups: list[LicenseGroup] = []

        while uncovered:
            owner = self._select_owner(uncovered, lts, pher, heur)
            lt = self._select_license(owner, lts, pher, heur)

            pool = (set(graph.neighbors(owner)) | {owner}) & uncovered
            if len(pool) < lt.min_capacity:
                # Fallback do licencji indywidualnej
                singles = [x for x in lts if x.min_capacity <= 1 <= x.max_capacity]
                lt = min(singles, key=lambda x: x.cost) if singles else lt
                groups.append(LicenseGroup(lt, owner, frozenset()))
                uncovered.remove(owner)
                continue

            # Wybór członków grupy według stopnia
```

```
            k = max(0, lt.max_capacity - 1)
            add = sorted((pool - {owner}), key=lambda n: graph.degree[n], reverse=True)[:k]
            groups.append(LicenseGroup(lt, owner, frozenset(add)))
            uncovered -= {owner} | set(add)

        return Solution(groups=tuple(groups))

    def _select_owner(self, uncovered: set, lts: Sequence[LicenseType],
                      pher: dict, heur: dict) -> Any:
        scores = {}
        for n in uncovered:
            acc = sum((pher.get((n, lt.name), 1.0) ** self.alpha) *
                      (heur.get((n, lt.name), 1.0) ** self.beta) for lt in lts)
            scores[n] = acc / max(1, len(lts))
        return self._roulette_or_best(list(uncovered), scores)

    def _evaporate(self, pher: dict) -> None:
        for k in pher:
            pher[k] *= (1.0 - self.evap)

    def _deposit(self, pher: dict, sol: Solution) -> None:
        if sol.total_cost > 0:
            q = 1.0 / sol.total_cost
            for g in sol.groups:
                for n in g.all_members:
                    k = (n, g.license_type.name)
                    if k in pher:
                        pher[k] += q
```

A.3.3 Wyżarzanie symulowane

Fragment implementacji wyżarzania z generowaniem sąsiadów i akceptacją według kryterium Metropolisa.

```
class SimulatedAnnealing(Algorithm):
    def solve(self, graph: nx.Graph, license_types: Sequence[LicenseType], **kwargs) -> Solution:
        current = GreedyAlgorithm().solve(graph, license_types)
        best = current
        temp = self.temp_initial
        stall = 0

        for _ in range(self.max_iterations):
            if temp < self.temp_min:
                break

            cand = self._neighbor(current, graph, license_types)
            if cand is None:
                stall += 1
            else:
                delta = cand.total_cost - current.total_cost
```

```
                if delta < 0 or random.random() < math.exp(-delta / max(temp, 1e-10)):
                    current = cand
                    if current.total_cost < best.total_cost:
                        best, stall = current, 0
                    else:
                        stall += 1
                else:
                    stall += 1

            if stall >= self.max_stall:
                stall, temp = 0, max(self.temp_min, 0.5 * temp)
            temp *= self.cooling_rate

        return best

    def _neighbor(self, solution: Solution, graph: nx.Graph, lts: Sequence[LicenseType]) -> Solution:
        moves = [self._mv_change_license, self._mv_move_member,
                 self._mv_swap_members, self._mv_merge_groups, self._mv_split_group]

        for _ in range(12):
            mv = random.choice(moves)
            try:
                cand = mv(solution, graph, lts)
                if cand and self.validator.validate(cand, graph)[0]:
                    return cand
            except Exception:
                continue
        return None
```

A.4 Operatory mutacji i sąsiedztwa

Kluczowe operatory używane przez metaheurystyki do generowania sąsiadów i eksploracji przestrzeni rozwiązań.

```
class MutationOperators:
    @staticmethod
    def generate_neighbors(base: Solution, graph: nx.Graph,
                           license_types: Sequence[LicenseType], k: int = 10) -> list[Solution]:
        ops = (MutationOperators.change_license_type, MutationOperators.reassign_member,
               MutationOperators.merge_groups, MutationOperators.split_group)
        weights = (0.3, 0.3, 0.2, 0.2)
        out: list[Solution] = []
        attempts = 0

        while len(out) < k and attempts < k * 10:
            attempts += 1
            op = random.choices(ops, weights=weights, k=1)[0]
            try:
                cand = op(base, graph, list(license_types))
                if cand is not None:
```

```
                out.append(cand)
            except Exception:
                continue
        return out

    @staticmethod
    def change_license_type(solution: Solution, graph: nx.Graph,
                            license_types: list[LicenseType]) -> Solution | None:
        if not solution.groups:
            return None
        group = random.choice(solution.groups)
        compatible = SolutionBuilder.get_compatible_license_types(
            group.size, license_types, exclude=group.license_type
        )
        if not compatible:
            return None

        new_lt = random.choice(compatible)
        new_groups = []
        for g in solution.groups:
            if g is group:
                new_groups.append(LicenseGroup(new_lt, g.owner, g.additional_members))
            else:
                new_groups.append(g)
        return SolutionBuilder.create_solution_from_groups(new_groups)

    @staticmethod
    def reassign_member(solution: Solution, graph: nx.Graph,
                        license_types: list[LicenseType]) -> Solution | None:
        if len(solution.groups) < 2:
            return None

        donors = [g for g in solution.groups
                  if g.size > g.license_type.min_capacity and g.additional_members]
        receivers = [g for g in solution.groups if g.size < g.license_type.max_capacity]

        if not donors or not receivers:
            return None

        from_group = random.choice(donors)
        pot_receivers = [g for g in receivers if g is not from_group]
        if not pot_receivers:
            return None

        to_group = random.choice(pot_receivers)
        member = random.choice(list(from_group.additional_members))
        allowed = SolutionBuilder.get_owner_neighbors_with_self(graph, to_group.owner)

        if member not in allowed:
```

```
            return None

        # Przeprowadzenie transferu
        new_groups = []
        for g in solution.groups:
            if g is from_group:
                new_groups.append(LicenseGroup(
                    g.license_type, g.owner, g.additional_members - {member}
                ))
            elif g is to_group:
                new_groups.append(LicenseGroup(
                    g.license_type, g.owner, g.additional_members | {member}
                ))
            else:
                new_groups.append(g)

        return SolutionBuilder.create_solution_from_groups(new_groups)

    @staticmethod
    def split_group(solution: Solution, graph: nx.Graph,
                    license_types: list[LicenseType]) -> Solution | None:
        splittable = [g for g in solution.groups if g.size > 2]
        if not splittable:
            return None

        group = random.choice(splittable)
        members = list(group.all_members)

        for _ in range(4):  # Próbuj kilka podziałów
            random.shuffle(members)
            cut = random.randint(1, len(members) - 1)
            part1, part2 = members[:cut], members[cut:]

            # Sprawdź kompatybilność typów licencji
            compat1 = SolutionBuilder.get_compatible_license_types(len(part1), license_types)
            compat2 = SolutionBuilder.get_compatible_license_types(len(part2), license_types)

            if not compat1 or not compat2:
                continue

            # Wybierz właścicieli i typy licencji
            owner1, owner2 = random.choice(part1), random.choice(part2)
            lt1, lt2 = min(compat1, key=lambda x: x.cost), min(compat2, key=lambda x: x.cost)

            # Sprawdź ograniczenia sąsiedztwa
            if (set(part1).issubset(SolutionBuilder.get_owner_neighbors_with_self(graph, owner1)) and
                set(part2).issubset(SolutionBuilder.get_owner_neighbors_with_self(graph, owner2))):

                new_groups = [g for g in solution.groups if g is not group]
```

```
                new_groups.append(LicenseGroup(lt1, owner1, frozenset(set(part1) - {owner1})))
                new_groups.append(LicenseGroup(lt2, owner2, frozenset(set(part2) - {owner2})))
                return SolutionBuilder.create_solution_from_groups(new_groups)

        return None
```

A.5 Funkcje pomocnicze

A.5.1 Budowanie i walidacja rozwiązań

Klasy pomocnicze odpowiedzialne za tworzenie i weryfikację poprawności rozwiązań.

```
class SolutionBuilder:
    @staticmethod
    def get_compatible_license_types(group_size: int, license_types: Sequence[LicenseType],
                                     exclude: LicenseType | None = None) -> list[LicenseType]:
        out: list[LicenseType] = []
        for lt in license_types:
            if exclude and lt == exclude:
                continue
            if lt.min_capacity <= group_size <= lt.max_capacity:
                out.append(lt)
        return out

    @staticmethod
    def get_owner_neighbors_with_self(graph: nx.Graph, owner: N) -> set[N]:
        return set(graph.neighbors(owner)) | {owner}

    @staticmethod
    def find_cheapest_single_license(license_types: Sequence[LicenseType]) -> LicenseType:
        singles = [lt for lt in license_types if lt.min_capacity <= 1]
        return min(singles or list(license_types), key=lambda lt: lt.cost)

class SolutionValidator:
    def validate(self, solution: Solution[N], graph: nx.Graph) -> tuple[bool, list[ValidationIssue]]:
        issues: list[ValidationIssue] = []
        nodes = set(graph.nodes())
        groups = tuple(solution.groups)

        # Sprawdzanie członków grup
        issues += self._check_group_members(groups, nodes)
        # Sprawdzanie pojemności licencji
        issues += self._check_group_capacity(groups)
        # Sprawdzanie ograniczeń sąsiedztwa
        issues += self._check_neighbors(groups, graph, nodes)
        # Sprawdzanie braku nakładania się grup
        issues += self._check_no_overlap(groups)
        # Sprawdzanie pokrycia wszystkich węzłów
        issues += self._check_coverage(groups, nodes)
```

```
        return (not issues, issues)

    def _check_neighbors(self, groups: tuple[LicenseGroup[N], ...], graph: nx.Graph,
                         nodes: set[N]) -> list[ValidationIssue]:
        issues: list[ValidationIssue] = []
        for idx, g in enumerate(groups):
            if g.owner not in nodes:
                continue
            allowed_any = set(graph.neighbors(g.owner)) | {g.owner}
            not_neighbors = set(g.all_members) - allowed_any
            if not_neighbors:
                issues.append(ValidationIssue(
                    "DISCONNECTED_MEMBER",
                    f"group#{idx} owner {g.owner!r} has non-neighbor members: {list(not_neighbors)!r}"
                ))
        return issues

    def _check_coverage(self, groups: tuple[LicenseGroup[N], ...], nodes: set[N]) -> list[ValidationIssue]:
        issues: list[ValidationIssue] = []
        covered = set().union(*(set(g.all_members) for g in groups)) if groups else set()
        missing = nodes - covered
        if missing:
            issues.append(ValidationIssue("MISSING_COVERAGE", f"missing nodes: {list(missing)!r}"))
        return issues
```

A.5.2 Konfiguracje licencyjne

Fabryka konfiguracji licencyjnych obsługująca różne rodziny licencji oraz dynamiczne warianty cenowe.

```
class LicenseConfigFactory:
    _CONFIGS: ClassVar[dict[str, Callable[[], list[LicenseType]]]] = {
        "duolingo_super": lambda: [
            LicenseType("Individual", 13.99, 1, 1, LicenseConfigFactory.BLACK),
            LicenseType("Family", 29.17, 2, 6, LicenseConfigFactory.BLACK),
        ],
        "spotify": lambda: [
            LicenseType("Individual", 23.99, 1, 1, LicenseConfigFactory.RED),
            LicenseType("Duo", 30.99, 2, 2, LicenseConfigFactory.GREEN),
            LicenseType("Family", 37.99, 2, 6, LicenseConfigFactory.BLUE),
        ],
        "netflix": lambda: [
            LicenseType("Basic", 33, 1, 1, LicenseConfigFactory.RED),
            LicenseType("Standard", 49, 1, 2, LicenseConfigFactory.GREEN),
            LicenseType("Premium", 67, 1, 4, LicenseConfigFactory.BLUE),
        ],
        "roman_domination": lambda: [
            LicenseType("Solo", 1.0, 1, 1, LicenseConfigFactory.BLUE),
            LicenseType("Group", 2.0, 2, 999999, LicenseConfigFactory.RED),
        ],
```

```
    }

    @classmethod
    def get_config(cls, name: str) -> list[LicenseType]:
        # Obsługa dynamicznych wariantów roman_p_<price>
        if name.startswith("roman_p_"):
            p_str = name.split("_", 2)[2].replace("_", ".")
            try:
                p_val = float(p_str)
                return [
                    LicenseType("Solo", 1.0, 1, 1, cls.BLUE),
                    LicenseType("Group", p_val, 2, 999999, cls.RED),
                ]
            except Exception:
                raise ValueError(f"Invalid roman price format: {name}")

        # Obsługa dynamicznych wariantów duolingo_p_<price>
        if name.startswith("duolingo_p_"):
            p_str = name.split("_", 2)[2].replace("_", ".")
            try:
                p_val = float(p_str)
                return [
                    LicenseType("Individual", 1.0, 1, 1, cls.RED),
                    LicenseType("Family", p_val, 2, 6, cls.BLUE),
                ]
            except Exception:
                raise ValueError(f"Invalid duolingo price format: {name}")

        # Konfiguracje statyczne
        try:
            return cls._CONFIGS[name]()
        except KeyError:
            available = ", ".join(cls._CONFIGS.keys())
            raise ValueError(f"Unsupported license config: {name}. Available: {available}")
```

A.6 Symulacja dynamiczna

A.6.1 Symulator ewolucji sieci

Główne komponenty symulatora odpowiedzialnego za mutacje struktury grafu w eksperymentach dynamicznych.

```
@dataclass
class MutationParams:
    add_nodes_prob: float = 0.1
    remove_nodes_prob: float = 0.05
    add_edges_prob: float = 0.15
    remove_edges_prob: float = 0.1
    max_nodes_add: int = 3
    max_nodes_remove: int = 2
```

```
    max_edges_add: int = 5
    max_edges_remove: int = 3
    mode_nodes: str = "random"      # "random" lub "preferential"
    mode_edges: str = "random"      # "random", "preferential", "triadic", "rewire_ws"
    add_node_attach_m: int = 2
    triadic_trials: int = 20

class DynamicNetworkSimulator:
    def __init__(self, mutation_params: MutationParams | None = None, seed: int | None = None):
        self.mutation_params = mutation_params or MutationParams()
        self.next_node_id = 0
        if seed is not None:
            random.seed(seed)

    def _apply_mutations(self, graph: nx.Graph) -> tuple[nx.Graph, list[str]]:
        mutations = []

        # Dodawanie węzłów
        if random.random() < self.mutation_params.add_nodes_prob:
            num_add = random.randint(1, self.mutation_params.max_nodes_add)
            new_nodes = self._add_nodes(graph, num_add)
            mutations.append(f"Added nodes: {new_nodes}")

        # Usuwanie węzłów (z zachowaniem minimalnego rozmiaru)
        if (random.random() < self.mutation_params.remove_nodes_prob
            and graph.number_of_nodes() > 5):
            num_remove = random.randint(1, min(self.mutation_params.max_nodes_remove,
                                               graph.number_of_nodes() - 5))
            removed_nodes = self._remove_nodes(graph, num_remove)
            mutations.append(f"Removed nodes: {removed_nodes}")

        # Dodawanie krawędzi
        if random.random() < self.mutation_params.add_edges_prob:
            num_add = random.randint(1, self.mutation_params.max_edges_add)
            if self.mutation_params.mode_edges == "rewire_ws":
                added_c, removed_c = self._rewire_edges(graph, num_add)
                if added_c: mutations.append(f"Added {added_c} edges")
                if removed_c: mutations.append(f"Removed {removed_c} edges")
            else:
                added_edges = self._add_edges(graph, num_add)
                mutations.append(f"Added {len(added_edges)} edges")

        return (graph, mutations)

    def _add_nodes(self, graph: nx.Graph, num_nodes: int) -> list[int]:
        new_nodes = []
        existing_nodes = list(graph.nodes())

        for _ in range(num_nodes):
```

```
            new_node = self.next_node_id
            self.next_node_id += 1
            graph.add_node(new_node)
            new_nodes.append(new_node)

            # Podłączanie do istniejących węzłów
            if existing_nodes:
                m = min(self.mutation_params.add_node_attach_m, len(existing_nodes))

                if self.mutation_params.mode_nodes == "preferential":
                    # Attachment proporcjonalny do stopni węzłów
                    deg = graph.degree
                    weights = [deg[v] + 1 for v in existing_nodes]
                    total = sum(weights)
                    chosen = set()

                    for _ in range(m):
                        if not existing_nodes: break
                        r = random.uniform(0, total)
                        acc, pick_idx = 0.0, 0
                        for idx, w in enumerate(weights):
                            acc += w
                            if acc >= r:
                                pick_idx = idx
                                break
                        v = existing_nodes[pick_idx]
                        if v not in chosen:
                            graph.add_edge(new_node, v)
                            chosen.add(v)
                        total -= weights[pick_idx]
                        existing_nodes.pop(pick_idx)
                        weights.pop(pick_idx)
                else:
                    # Losowe podłączanie
                    neighbors = random.sample(existing_nodes, m)
                    for neighbor in neighbors:
                        graph.add_edge(new_node, neighbor)

            existing_nodes = list(graph.nodes())

        return new_nodes

    def _add_edges(self, graph: nx.Graph, num_edges: int) -> list[tuple[int, int]]:
        nodes = list(graph.nodes())
        added_edges = []
        if len(nodes) < 2: return added_edges

        mode = self.mutation_params.mode_edges
        if mode == "triadic":
```

```
    # Zamykanie trójkątów
    attempts = 0
    while len(added_edges) < num_edges and attempts < num_edges * 20:
        w = random.choice(nodes)
        neigh = list(graph.neighbors(w))
        if len(neigh) >= 2:
            u, v = random.sample(neigh, 2)
            if not graph.has_edge(u, v):
                graph.add_edge(u, v)
                added_edges.append((u, v))
        attempts += 1

elif mode == "preferential":
    # Preferencyjne dodawanie proporcjonalne do iloczynów stopni
    attempts = 0
    while len(added_edges) < num_edges and attempts < num_edges * 20:
        u, v = random.sample(nodes, 2)
        if graph.has_edge(u, v):
            attempts += 1
            continue
        deg = graph.degree
        w = (deg[u] + 1) * (deg[v] + 1)
        max_deg = max((deg[x] for x in nodes), default=1)
        accept_p = min(1.0, w / float((max_deg + 1) ** 2))
        if random.random() < accept_p:
            graph.add_edge(u, v)
            added_edges.append((u, v))
        attempts += 1

return added_edges
```