



**Wydział Matematyki  
i Nauk Informatycznych**

POLITECHNIKA WARSZAWSKA

## **Teoria algorytmów i obliczeń**

Projekt zaliczeniowy

Piotr Jacak  
Jakub Kindracki  
Wiktor Kobielski  
Ernest Mołczan

Koordinator: prof. dr hab. inż. Władysław Homenda

Semestr zimowy 2025/2026

# Spis treści

<b>1</b>	<b>Wstęp</b>	<b>4</b>
<b>2</b>	<b>Definicje pojęć</b>	<b>5</b>
<b>3</b>	<b>Rozmiar multigrafu</b>	<b>6</b>
<b>4</b>	<b>Metryka w zbiorze wszystkich multigrafów</b>	<b>8</b>
<b>5</b>	<b>Minimalne rozszerzenie multigrafu</b>	<b>9</b>
5.1	Algorytm dokładny dla problemu izomorfizmu podgrafu . . . . .	9
5.1.1	Dowód poprawności . . . . .	10
5.1.2	Złożoność obliczeniowa . . . . .	10
<b>6</b>	<b>Minimalne rozszerzenie multigrafu zawierającego m kopii podgrafu P</b>	<b>12</b>
6.1	Motywacja i sformułowanie problemu . . . . .	12
6.2	Definicje formalne . . . . .	13
6.3	Algorytmy pomocnicze . . . . .	15
6.3.1	Algorytm generowania k-kombinacji . . . . .	15
6.3.2	Algorytm generowania permutacji . . . . .	17
6.3.3	Algorytm generowania produktu kartezjańskiego . . . . .	19
6.4	Algorytm dokładny . . . . .	21
6.4.1	Przegląd algorytmu . . . . .	21
6.4.2	Szczegółowy opis algorytmu . . . . .	22
6.4.3	Pseudokod algorytmu . . . . .	24
6.5	Dowód poprawności algorytmu . . . . .	26
6.6	Analiza złożoności obliczeniowej . . . . .	28
6.6.1	Złożoność czasowa . . . . .	28
6.6.2	Złożoność pamięciowa . . . . .	30
6.6.3	Charakterystyka algorytmu . . . . .	31
6.7	Aproksymacyjne minimalne rozszerzenie multigrafu - algorytm pierwszy	31
6.7.1	Opis algorytmu . . . . .	31
6.7.2	Złożoność obliczeniowa . . . . .	32
6.7.3	Uzasadnienie . . . . .	32

<b>7</b>	<b>Aproksymacyjne minimalne rozszerzenie multigrafu - algorytm drugi</b>	<b>33</b>
7.1	Heurystyczna Funkcja Kosztu . . . . .	33
7.2	Opis Algorytmów . . . . .	35
7.2.1	Algorytm Fazy 1: Generowanie Podzbiorów . . . . .	36
7.2.2	Algorytm Fazy 2: Iteracyjne Rozwiązywanie i Rozszerzanie . .	37
7.3	Końcowa Analiza Złożoności Całkowitej . . . . .	41
<b>8</b>	<b>Bibliografia</b>	<b>41</b>

# 1 Wstęp

Niniejsza praca stanowi sprawozdanie z projektu zrealizowanego w ramach przedmiotu **Teoria algorytmów i obliczeń**. Przedmiotem badań są algorytmy operujące na multigrafach, ze szczególnym uwzględnieniem problematyki izomorfizmu podgrafów oraz minimalnych rozszerzeń grafów.

Głównym celem projektu jest opracowanie, analiza teoretyczna oraz implementacja algorytmów rozwiązujących dwa ściśle powiązane problemy. Pierwszym z nich jest weryfikacja, czy dany multigraf  $H$  jest izomorficzny z  $n$  podgrafami multigrafu  $G$ . Drugim, kluczowym zagadnieniem, jest wyznaczenie *minimalnego rozszerzenia* multigrafu  $G$  do postaci  $G'$ , która zawiera co najmniej  $n$  podgrafów izomorficznych z  $H$ .

Realizacja powyższych celów wymagała formalnego zdefiniowania oraz uzasadnienia kilku fundamentalnych pojęć. W pracy zaproponowano autorskie lub bazujące na literaturze definicje:

- *rozmiaru multigrafu*,
- *metryki* w zbiorze multigrafów,
- *minimalnego rozszerzenia* multigrafu.

Pojęcia te stanowią podstawę do dalszej analizy algorytmicznej oraz oceny kosztu operacji.

W ramach pracy przeprowadzono analizę złożoności obliczeniowej opracowanych algorytmów. Zgodnie z założeniami projektu, w przypadku gdy algorytmy dokładne charakteryzują się złożonością wykładniczą, przedstawiono również propozycje algorytmów aproksymacyjnych o złożoności wielomianowej.

## 2 Definicje pojęć

**Definicja 1** (Graf). Grafem nazywamy parę  $G = (V, E)$ , gdzie  $V$  jest zbiorem wierzchołków, a  $E \subseteq V \times V = \{(u, v) : u, v \in V \wedge u \neq v\}$  jest zbiorem krawędzi. Dla każdej pary wierzchołków  $u, v \in V$  istnieje co najwyżej jedna krawędź łącząca wierzchołki  $u$  i  $v$ .

**Definicja 2** (Multigraf). Multigrafem nazywamy graf, w którym pomiędzy dowolnymi dwoma różnymi wierzchołkami  $u, v \in V$  może istnieć więcej niż jedna krawędź.

**Definicja 3** (Graf skierowany). Grafem skierowanym nazywamy parę  $G = (V, E)$ , gdzie  $V$  jest zbiorem wierzchołków, a  $E \subseteq V \times V = \{(u, v) : u, v \in V \wedge u \neq v\}$  jest zbiorem krawędzi. Krawędzie w grafie skierowanym mają określony kierunek, co oznacza, że krawędź  $(u, v)$  jest różna od krawędzi  $(v, u)$ . Definicja jest analogiczna dla multigrafów.

**Definicja 4** (Izomorfizm grafów). Dwa grafy  $G_1 = (V_1, E_1)$  i  $G_2 = (V_2, E_2)$  są izomorficzne, wtedy i tylko wtedy, gdy istnieje bijekcja  $f : V_1 \rightarrow V_2$ , taka że dla każdej krawędzi  $(u, v) \in E_1$  zachodzi  $(f(u), f(v)) \in E_2$ . Definicja ta jest analogiczna dla multigrafów i grafów skierowanych.

**Definicja 5** (Podgraf). Graf  $H = (V_H, E_H)$  nazywamy podgrafem grafu  $G = (V_G, E_G)$ , wtedy i tylko wtedy, gdy  $V_H \subseteq V_G$  oraz  $E_H \subseteq E_G$ . Definicja ta jest analogiczna dla multigrafów i grafów skierowanych.

**Definicja 6** (Macierz sąsiedztwa). Macierzą sąsiedztwa multigrafu  $G = (V, E)$  nazywamy macierz  $A$ , której pole  $A_{uv} = k$ , wtedy i tylko wtedy, gdy istnieje  $k$  krawędzi  $(u, v) \in E$ . W przypadku gdy nie istnieje żadna krawędź pomiędzy wierzchołkami  $u$  i  $v$ , to  $A_{uv} = 0$ .

### 3 Rozmiar multigrafu

**Definicja 7** (Rozmiar multigrafu). Rozmiarem  $S(G)$  multigrafu  $G = (V, E)$  nazywamy parę liczb naturalnych  $(|V|, |E|)$ , gdzie  $|V|$  oznacza liczbę wierzchołków, a  $|E|$  liczbę krawędzi w multigrafie  $G$ .

Zakładamy, że liczby wierzchołków i krawędzi są zapisanymi wcześniej stałymi, więc obliczenie rozmiaru multigrafów jest operacją o złożoności czasowej  $O(1)$ .

**Definicja 8** (Porządek w zbiorze wszystkich multigrafów). Niech  $G_1$  i  $G_2$  będą dwoma multigrafami. Mówimy, że  $G_1$  jest mniejszy, lub równy  $G_2$  wtedy i tylko wtedy, gdy:

$$|V_1| < |V_2| \vee (|V_1| = |V_2| \wedge |E_1| \leq |E_2|)$$

Żeby udowodnić poprawność powyższej definicji porządku wykazujemy, że spełnia ona trzy wymagane własności:

- **Zwrotność:**

$$S(G) \leq S(G)$$

Dla dowolnego multigrafu  $G = (V, E)$ , zachodzi  $|V| = |V| \wedge |E| = |E|$ . Więc w szczególności spełnia on warunek  $|V| = |V| \wedge |E| \leq |E|$  z definicji porządku. Stąd  $S(G) \leq S(G)$ .

- **Przechodność:**

$$S(G_1) \leq S(G_2) \wedge S(G_2) \leq S(G_3) \Rightarrow S(G_1) \leq S(G_3)$$

Weźmy dowolne trzy multigrafy  $G_1 = (V_1, E_1)$ ,  $G_2 = (V_2, E_2)$  oraz  $G_3 = (V_3, E_3)$  takie, że  $S(G_1) \leq S(G_2)$  oraz  $S(G_2) \leq S(G_3)$ .

Założmy, że  $S(G_1) \geq S(G_3)$ . Z definicji to implikuje, że  $|V_1| > |V_3| \vee (|V_1| = |V_3| \wedge |E_1| > |E_3|)$ .

Z założeń wiemy też, że  $|V_2| > |V_1|$ , lub  $|V_2| = |V_1| \wedge |E_2| \geq |E_1|$ .

W pierwszym przypadku z założeń wynika, że  $|V_2| > |V_3|$ , co stoi w sprzeczności z  $S(G_2) \leq S(G_3)$ .

W drugim przypadku, z założeń wynika, że  $|V_2| = |V_3|$  oraz  $|E_2| > |E_3|$ , co również stoi w sprzeczności z  $S(G_2) \leq S(G_3)$ .

W obu przypadkach dochodzimy do sprzeczności, więc nasze początkowe założenie było fałszywe. Stąd  $S(G_1) \leq S(G_3)$ .

- **Antysymetryczność:**

$$S(G_1) \leq S(G_2) \wedge S(G_2) \leq S(G_1) \Rightarrow S(G_1) = S(G_2)$$

Weźmy dowolne dwa multigrafy  $G_1 = (V_1, E_1)$  oraz  $G_2 = (V_2, E_2)$  takie, że  $S(G_1) \leq S(G_2)$  oraz  $S(G_2) \leq S(G_1)$ . Z definicji porządku, z pierwszego założenia wynika, że  $|V_1| < |V_2| \vee (|V_1| = |V_2| \wedge |E_1| \leq |E_2|)$ . Z drugiego założenia wynika, że  $|V_2| < |V_1| \vee (|V_2| = |V_1| \wedge |E_2| \leq |E_1|)$ .

Jeśli  $|V_1| < |V_2|$ , to z drugiego założenia wynika, że  $|V_2| < |V_1|$ , co jest sprzeczne. Analogicznie, jeśli  $|V_2| < |V_1|$ , to z pierwszego założenia wynika, że  $|V_1| < |V_2|$ , co również jest sprzeczne. Zatem musi zachodzić  $|V_1| = |V_2|$ .

Wtedy z pierwszego założenia wynika, że  $|E_1| \leq |E_2|$ , a z drugiego, że  $|E_2| \leq |E_1|$ . Stąd  $|E_1| = |E_2|$ .

W rezultacie mamy  $S(G_1) = S(G_2)$ .

## 4 Metryka w zbiorze wszystkich multigrafów

**Definicja 9** (Metryka w zbiorze multigrafów). Niech  $\mathcal{G}$  będzie zbiorem wszystkich multigrafów. **Metryką** w zbiorze  $\mathcal{G}$  nazywamy funkcję:

$$d : \mathcal{G} \times \mathcal{G} \rightarrow \mathbb{N}_0$$

Wartość  $d(G_1, G_2)$  nazywamy **odległością** między multigrafami  $G_1$  i  $G_2$ , a definiujemy ją, jako **minimalną** liczbę operacji dodawania lub usuwania pojedynczej krawędzi lub wierzchołka, za pomocą których można przekształcić graf  $G_1$  w graf izomorficzny z  $G_2$ .

Powyższa definicja spełnia następujące własności metryki:

- **Identyczność nierozróżnialnych:**

Dla dowolnych multigrafów  $G_1$  oraz  $G_2$ ,  $d(G_1, G_2) = 0$  wtedy i tylko wtedy, gdy  $G_1$  jest izomorficzny z  $G_2$ . Wynika to bezpośrednio z definicji naszej metryki.

- **Symetria:**

Dla dowolnych multigrafów  $G_1$  oraz  $G_2$ ,  $d(G_1, G_2) = d(G_2, G_1)$ . Dodawanie i usuwanie krawędzi lub wierzchołków jest operacją odwracalną, więc liczba operacji potrzebnych do przekształcenia  $G_1$  w  $G_2$  jest równa liczbie odwrotnych operacji potrzebnych do przekształcenia  $G_2$  w  $G_1$ .

- **Nierówność trójkąta:**

Dla dowolnych multigrafów  $G_1$ ,  $G_2$  oraz  $G_3$ ,  $d(G_1, G_3) \leq d(G_1, G_2) + d(G_2, G_3)$ . Oznacza to, że najkrótsza droga między dwoma multigrafami nie może być dłuższa niż droga przechodząca przez trzeci multigraf. Jest to prawda, ponieważ każda sekwencja operacji przekształcających  $G_1$  w  $G_2$  oraz  $G_2$  w  $G_3$  może być złożona w jedną sekwencję przekształcającą  $G_1$  w  $G_3$ .



## 5 Minimalne rozszerzenie multigrafu

### 5.1 Algorytm dokładny dla problemu izomorfizmu podgrafu

Mając dane dwa grafy  $G$  i  $H$ , chcemy znaleźć podgrafy  $G$  izomorficzne do  $H$ . Do rozwiązania tego problemu posłuży nam algorytm, który wykorzystuje procedurę Backtrackingu do sprawdzania struktury grafów.

Przed przejściem do algorytmu, zdefiniujmy sobie struktury przydatne nam do implementacji. Niech  $n_G = |V(G)|$  - ilość wierzchołków w grafie  $G$  oraz  $n_H = |V(H)|$  - ilość wierzchołków w Grafie  $H$ .

#### Opis algorytmu:

1. Inicjalizacja macierzy sąsiedztwa grafów  $G$  i  $H$  odpowiednio  $S_G \in \mathbb{N}^{n_G \times n_G}$  i  $S_H \in \mathbb{N}^{n_H \times n_H}$ . Wartość  $S[i, j]$ , to ilość krawędzi pomiędzy  $i$ -tym, a  $j$ -tym wierzchołkiem dla danego grafu.

2. Inicjalizacja kandydatów - Zdefiniujmy sobie listę *mozliwe\_dopasowania*,  $len(mozliwe\_dopasowania) = n_H$ , gdzie pod  $i$ -tym indeksem, będziemy mieli listę możliwych dopasowań dla wierzchołka  $i \in V(H)$ .

Algorytm Ullmana dla grafów prostych zakłada inicjalizację:

$$u \in V(G), u \in mozliwe\_dopasowania[i] \iff deg_G(u) \geq deg_H(i)$$

Jest ona działającą inicjalizacją dla multigrafów, jednak w celach optymalizacji algorytmu, możemy zmienić tę inicjalizację tak, aby zmniejszyć liczbę potencjalnych dopasowań, a co za tym idzie zmniejszyć liczbę gałęzi, które będzie musiał przejść algorytm. Możemy zauważyć, że w macierzach sąsiedztwa na głównej przekątnej pod indeksami  $[i, i]$  znajduje się liczba pętli danego wierzchołka, zatem naszym warunkiem będzie także  $S_G[i, i] \geq S_H[i, i]$ . Biorąc to wszystko razem, otrzymujemy

$$u \in mozliwe\_dopasowania[i] \iff (deg_G(u) \geq deg_H(i)) \wedge (S_G[i, i] \geq S_H[i, i])$$

3. Dla każdej krawędzi, która istnieje między już dopasowanymi wierzchołkami z  $H$ , sprawdź czy istnieje krawędź między ich dopasowaniami z  $G$  i czy ilość krawędzi między dopasowaniami jest większa lub równa niż ilość krawędzi między wierzchołkami. Można to osiągnąć przez przejście wszystkich par już dopasowanych wierzchołków i krawędzi między nimi. Jeśli nie, zwróć False
4. Sprawdź, czy wszystkie wierzchołki nie zostały już dopasowane. Jeśli tak, zwróć True.

5. Dla każdego wierzchołka  $v \in \text{mozliwe\_dopasowania}[i]$ , jeśli  $v \text{ not in } \text{dopasowania}$ , przypisz  $\text{dopasowania}[i] = v$  oraz wywołaj funkcję ponownie dla następnego wierzchołka  $\in V(H)$  z przekazaną kopią. W przypadku wyniku True z tej funkcji, zwróć True, w przypadku False,  $\text{dopasowania}[i] = \text{null}$  i przejdź do następnego kroku tej pętli.
6. W przypadku niedopasowania po wszystkich iteracjach pętli, zwróć False.

### 5.1.1 Dowód poprawności

Najpierw zbadajmy, czy algorytm dobrze inicjalizuje *mozliwe\_dopasowania*. W tym celu rozbijmy wszystkie 3 warunki. Pierwszy warunek mówi o tym, że potencjalne dopasowanie  $v$  dla wierzchołka  $u$ , musi mieć stopień co najmniej równy stopniowi wierzchołka  $u$ . Gdyby tak nie było, w grafie  $G$  nie istniałaby co najmniej jedna krawędź wychodząca z  $v$ , która istniałaby w  $H$  i wychodziłaby z  $u$ , zatem  $v$  nie mogłoby być dopasowaniem dla  $u$ . Drugi warunek mówi o tym, że liczba pętli dla  $v$  musi być co najmniej równa liczbie pętli dla  $u$ . Idea jest taka sama jak warunkowi pierwszego, gdyby warunek nie był spełniony, nie istniałaby co najmniej jedna pętla dla danego wierzchołka, a co za tym idzie, nie mógłby on być dopasowaniem dla  $u$ .

Dalej w algorytmie, przechodzimy po kolei po wierzchołkach z  $H$ . Najpierw sprawdzamy, czy struktura się zgadza dla tych wierzchołków, do których znaleźliśmy już dopasowania. Jeśli choć 1 krawędź istniejąca w  $H$  pomiędzy dwoma wierzchołkami nie będzie istnieć między ich dopasowaniami w  $G$ , algorytm wychodzi z tej ścieżki dopasowań i szuka innych, zatem działa poprawnie.

Następnie sprawdzamy wszystkie z możliwych dopasowań dla danego wierzchołka, zatem sprawdzając tak wszystkie wierzchołki, mamy pewność, że przejdziemy po wszystkich możliwych permutacjach.

### 5.1.2 Złożoność obliczeniowa

Zauważmy, że inicjalizacja *mozliwe\_dopasowania* w taki sposób, że dla każdego wierzchołka  $u \in V(H)$  możliwym dopasowaniem są wszystkie  $v \in V(G)$ , to algorytm przejdzie po wszystkich poddrzewach, zatem w przypadku pesymistycznym do 1 wierzchołka wykona  $n_G$  potencjalnych dopasowań, do drugiego  $n_G - 1$ , ..., a do  $n_H$ -tego,  $(n_G - n_H + 1)$  dopasowań. Zatem mamy

$$\underbrace{(n_G)(n_G - 1) \dots (n_G - n_H + 1)}_{n_H \text{ razy}} \leq n_G^{n_H}$$

W każdej takiej pętli wykonujemy sprawdzenie, czy struktura grafu się zgadza, (krok 4). Zauważmy, że wykonamy tam  $i^2$  porównań, gdzie  $i$  to indeks aktualnie obliczanego wierzchołka. Wiemy że  $i < n_H$ , zatem możemy ograniczyć tę operację:  $i^2 < n_H^2$ . W sumie możemy stwierdzić, że złożoność tego algorytmu wyniesie  $O(n_G^{n_H} n_H^2)$

## 6 Minimalne rozszerzenie multigrafu zawierającego $m$ kopii podgrafu $P$

### 6.1 Motywacja i sformułowanie problemu

W poprzednich sekcjach zajmowaliśmy się problemem weryfikacji istnienia pojedynczego podgrafu izomorficznego z danym wzorcem. W praktycznych zastosowaniach często pojawia się jednak bardziej ogólne zagadnienie: jak minimalnie rozszerzyć graf  $G$ , aby zawierał on  $m$  różnych kopii grafu wzorcowego  $P$ ?

Problem ten ma istotne zastosowania w dziedzinach takich jak:

- **Projektowanie sieci:** Zapewnienie redundancji przez istnienie wielu izomorficznych podsieci
- **Analiza struktur molekularnych:** Identyfikacja powtarzających się motywów strukturalnych
- **Analiza sieci społecznych:** Wykrywanie grup o podobnej strukturze relacji
- **Optymalizacja grafów:** Minimalne modyfikacje zachowujące pożądane właściwości strukturalne

#### Formalne sformułowanie problemu:

*Dane:*

- Multigraf skierowany  $G = (V_G, E_G)$  o  $n$  wierzchołkach (graf "duży")
- Multigraf skierowany  $P = (V_P, E_P)$  o  $k$  wierzchołkach, gdzie  $k \leq n$  (graf "mały", wzorzec)
- Liczba naturalna  $m \geq 1$  - wymagana liczba kopii

*Zadanie:*

- Znaleźć minimalny zbiór krawędzi  $E_{add}$  taki, że graf  $G' = (V_G, E_G \cup E_{add})$  zawiera co najmniej  $m$  podgrafów izomorficznych z  $P$ , przy czym każde dwa podgrafy różnią się przynajmniej jednym wierzchołkiem

## 6.2 Definicje formalne

**Definicja 10** (Rozszerzenie multigrafu). Niech  $G = (V_G, E_G)$  i  $G' = (V_{G'}, E_{G'})$  będą multigrafami. Mówimy, że  $G'$  jest **rozszerzeniem**  $G$ , jeśli:

1.  $V_G \subseteq V_{G'}$  (zbiór wierzchołków  $G$  jest podzbiorem wierzchołków  $G'$ )
2.  $E_G \subseteq E_{G'}$  (zbiór krawędzi  $G$  jest podzbiorem krawędzi  $G'$ )

**Uzasadnienie:** Definicja jest naturalna i oparta na relacji inkluzji zbiorów. Zgodna z intuicją, że rozszerzenie grafu polega na dodaniu nowych wierzchołków i/lub krawędzi przy zachowaniu struktury oryginalnego grafu. Jest spójna z definicją podgrafu ( $G$  jest podgrafem  $G'$ ).

**Definicja 11** (Koszt rozszerzenia). Niech  $G = (V_G, E_G)$  i  $G' = (V_{G'}, E_{G'})$  będą multigrafami takimi, że  $G'$  jest rozszerzeniem  $G$ . **Koszt** rozszerzenia  $\gamma(G, G')$  nazywamy parę liczb naturalnych:

$$\gamma(G, G') = (|V_{G'} \setminus V_G|, |E_{G'} \setminus E_G|)$$

gdzie:

- $|V_{G'} \setminus V_G|$  to liczba dodanych wierzchołków
- $|E_{G'} \setminus E_G|$  to liczba dodanych krawędzi

**Uzasadnienie:** Koszt uwzględnia dwie podstawowe operacje rozszerzania grafu. W kontekście naszego algorytmu skupiamy się głównie na dodawaniu krawędzi, zakładając stałą liczbę wierzchołków ( $V_G = V_{G'}$ ).

**Definicja 12** (Porządek leksykograficzny na kosztach). Dla dwóch kosztów  $(v_1, e_1)$  i  $(v_2, e_2)$  definiujemy porządek leksykograficzny:

$$(v_1, e_1) < (v_2, e_2) \iff v_1 < v_2 \vee (v_1 = v_2 \wedge e_1 < e_2)$$

**Uzasadnienie:** Porządek leksykograficzny priorytetyzuje minimalizację liczby dodanych wierzchołków, a następnie krawędzi.

**Definicja 13** (Osadzenie k-wierzchołkowe). **Osadzenie k-wierzchołkowe** grafu  $P$  w grafie  $G$  definiujemy przez parę  $(C, \pi)$ , gdzie:

1.  $C \subseteq V_G$  jest **k-kombinacją** - podzbiorem wierzchołków takim, że  $|C| = k = |V_P|$

2.  $\pi : V_P \rightarrow C$  jest **k-permutacją** - bijekcją mapującą wierzchołki  $P$  na wierzchołki  $C$

Para  $(C, \pi)$  definiuje potencjalne osadzenie  $P$  w  $G$  poprzez podgraf indukowany przez  $C$  z odpowiednim mapowaniem wierzchołków.

**Uzasadnienie:** Formalizuje procedurę przeszukiwania przestrzeni możliwych osadzeń. Bezpośrednio odpowiada implementacji (kombinacje i permutacje w kodzie). Liczba możliwych osadzeń wynosi  $\binom{n}{k} \times k!$ .

**Definicja 14** (Brakujące krawędzie dla osadzenia). Niech  $G = (V_G, E_G)$  i  $P = (V_P, E_P)$  będą multigrafami, gdzie  $|V_P| = k \leq |V_G| = n$ . Niech  $(C, \pi)$  będzie osadzeniem k-wierzchołkowym  $P$  w  $G$ . **Zbiorem brakujących krawędzi** dla osadzenia  $(C, \pi)$  nazywamy multizbiór:

$$\Delta((C, \pi), G, P) = \{(\pi(u), \pi(v)) : u, v \in V_P\}$$

z krotnościami:

$$\text{mult}_\Delta(\pi(u), \pi(v)) = \max(0, A_P[u][v] - A_G[\pi(u)][\pi(v)])$$

gdzie  $A_P, A_G$  są macierzami sąsiedztwa odpowiednio grafów  $P$  i  $G$ .

**Uzasadnienie:** Umożliwia kwantyfikację odległości między potencjalnym osadzeniem a rzeczywistym izomorfizmem. Stanowi podstawę algorytmu konstrukcji minimalnego rozszerzenia. Uwzględnia krotności krawędzi (multigrafowość).

**Definicja 15** (Minimalne rozszerzenie zawierające  $m$  kopii podgrafu  $P$ ). Niech  $G = (V_G, E_G)$  i  $P = (V_P, E_P)$  będą multigrafami, gdzie  $|V_P| \leq |V_G|$ , oraz niech  $m \geq 1$  będzie liczbą naturalną. **Minimalnym rozszerzeniem**  $G$  zawierającym  $m$  kopii  $P$  nazywamy multigraf  $G' = (V_{G'}, E_{G'})$  spełniający następujące warunki:

1.  $G'$  jest rozszerzeniem  $G$  (tj.  $V_G \subseteq V_{G'}, E_G \subseteq E_{G'}$ )
2.  $G'$  zawiera co najmniej  $m$  podgrafów izomorficznych z  $P$ , przy czym każde dwa podgrafy różnią się przynajmniej jednym wierzchołkiem (tzn. dla dowolnych dwóch podgrafów  $H_i, H_j$  zachodzi  $V_{H_i} \neq V_{H_j}$ )
3. Koszt rozszerzenia  $\gamma(G, G')$  jest minimalny w sensie porządku leksykograficznego wśród wszystkich rozszerzeń spełniających warunki 1 i 2

**Uzasadnienie:** Warunek  $V_{H_i} \neq V_{H_j}$  zapewnia, że kopie są rzeczywiście różne (nie są tym samym podgrafem), ale dopuszcza częściowe pokrywanie się zbiorów wierzchołków. Jest to słabsze wymaganie niż pełna rozłączność wierzchołkowa, ale wystarczające do sensownego policzenia  $m$  różnych kopii grafu  $P$ . W implementacji zakładamy  $V_G = V_{G'}$  (nie dodajemy wierzchołków), więc minimalizujemy tylko liczbę dodanych krawędzi. Definicja jest operacyjna i pozwala na konstrukcję algorytmów.

## 6.3 Algorytmy pomocnicze

Algorytm główny wykorzystuje trzy fundamentalne algorytmy kombinatoryczne: generowanie  $k$ -kombinacji, generowanie permutacji oraz generowanie produktu kartezjańskiego ( $m$ -krotek). W tej sekcji przedstawiamy szczegółowe opisy tych algorytmów wraz z dowodami poprawności i analizą złożoności.

### 6.3.1 Algorytm generowania $k$ -kombinacji

**Problem:** Dla danego zbioru  $n$  elementów wygenerować wszystkie jego  $k$ -elementowe podzbiory (kombinacje).

**Właściwości:**

- Liczba  $k$ -kombinacji ze zbioru  $n$ -elementowego:  $\binom{n}{k} = \frac{n!}{k!(n-k)!}$
- Kombinacje są nieuporządkowane (zbiory, nie ciągi)
- Kolejność generowania: leksykograficzna według indeksów

---

**Algorithm 1** GenerateCombinations(*items*, *k*)

---

**Require:** *items* - lista *n* elementów, *k* - rozmiar kombinacji

**Ensure:** Wszystkie *k*-kombinacje elementów z *items*

```
1: if k = 0 then
2:   yield []                                ▷ Pusta kombinacja
3:   return
4: end if
5: if k > n then
6:   return                                ▷ Brak kombinacji
7: end if
8: c ← [0, 1, 2, ..., k - 1]              ▷ Początkowa kombinacja indeksów
9: while true do
10:  yield [items[c[0]], items[c[1]], ..., items[c[k - 1]]]
11:  i ← k - 1
12:  while i ≥ 0 and c[i] = n - k + i do
13:    i ← i - 1
14:  end while
15:  if i < 0 then
16:    break                                ▷ Wszystkie kombinacje wygenerowane
17:  end if
18:  c[i] ← c[i] + 1
19:  for j ← i + 1 to k - 1 do
20:    c[j] ← c[j - 1] + 1
21:  end for
22: end while
```

---

**Dowód poprawności:**

*Dowód.* Algorytm reprezentuje kombinacje jako rosnące ciągi indeksów  $c[0] < c[1] < \dots < c[k-1]$ , gdzie  $0 \leq c[i] \leq n-1$ .

**Niezmiennik:** W każdej iteracji głównej pętli, tablica *c* reprezentuje poprawną *k*-kombinację (ściśle rosnący ciąg indeksów).

**Kompletność:** Algorytm generuje wszystkie kombinacje, ponieważ:

1. Rozpoczyna od najmniejszej kombinacji  $[0, 1, \dots, k-1]$
2. W każdej iteracji znajduje najbardziej prawy indeks *i*, który można zwiększyć (linie 10-12)
3. Zwiększa *c*[*i*] i ustawia następne indeksy jako kolejne liczby (linie 16-18)



4. To jest standardowy algorytm generowania kombinacji w porządku leksykograficznym
5. Kończy gdy nie można zwiększyć żadnego indeksu (największa kombinacja  $[n - k, n - k + 1, \dots, n - 1]$ )

**Brak duplikatów:** Każda kombinacja jest unikalna, ponieważ algorytm ściśle następuje porządek leksykograficzny i nigdy nie cofa się do wcześniej wygenerowanych kombinacji.

**Poprawność struktury:** Niezmiennik  $c[0] < c[1] < \dots < c[k-1]$  jest zachowany w każdej iteracji przez konstrukcję algorytmu (linie 16-18).  $\square$

**Złożoność obliczeniowa:**

- **Liczba iteracji:**  $\binom{n}{k}$  (liczba kombinacji do wygenerowania)
- **Koszt jednej iteracji:**  $O(k)$  - wyprodukowanie kombinacji i znalezienie indeksu do zwiększenia
- **Złożoność czasowa całkowita:**  $O\left(\binom{n}{k} \cdot k\right)$
- **Złożoność pamięciowa:**  $O(k)$  - przechowywanie tablicy indeksów

### 6.3.2 Algorytm generowania permutacji

**Problem:** Dla danego zbioru  $n$  elementów wygenerować wszystkie jego permutacje (uporządkowania).

**Właściwości:**

- Liczba permutacji zbioru  $n$ -elementowego:  $n!$
- Wykorzystujemy algorytm Heapa - minimalizuje liczbę zamian elementów
- Generowanie w miejscu (in-place)

---

**Algorithm 2** GeneratePermutations(*items*)

---

**Require:** *items* - lista  $n$  elementów

**Ensure:** Wszystkie permutacje elementów z *items*

```
1:  $n \leftarrow |items|$ 
2: if  $n \leq 1$  then
3:   yield items
4:   return
5: end if
6:  $a \leftarrow$  kopia items ▷ Praca na kopii
7:  $c \leftarrow [0, 0, \dots, 0]$  ▷ Liczniki dla algorytmu Heapa, długość  $n$ 
8: yield kopia a ▷ Pierwsza permutacja
9:  $i \leftarrow 0$ 
10: while  $i < n$  do
11:   if  $c[i] < i$  then
12:     if  $i \bmod 2 = 0$  then
13:       swap( $a[0]$ ,  $a[i]$ )
14:     else
15:       swap( $a[c[i]]$ ,  $a[i]$ )
16:     end if
17:     yield kopia a
18:      $c[i] \leftarrow c[i] + 1$ 
19:      $i \leftarrow 0$ 
20:   else
21:      $c[i] \leftarrow 0$ 
22:      $i \leftarrow i + 1$ 
23:   end if
24: end while
```

---

**Dowód poprawności:**

*Dowód.* Algorytm Heapa wykorzystuje nierekurencyjną implementację z licznikami  $c[i]$  reprezentującymi stan rekurencji.

**Kompletność:** Algorytm generuje dokładnie  $n!$  permutacji, ponieważ:

1. Dla każdego  $i$  wartość  $c[i]$  przyjmuje wartości od 0 do  $i - 1$ , co daje  $i$  możliwości
2. Całkowita liczba stanów:  $1 \cdot 2 \cdot 3 \cdots n = n!$
3. Każdemu stanowi odpowiada unikalna permutacja

**Brak duplikatów:** Każda kombinacja wartości liczników  $c$  występuje dokładnie raz, co gwarantuje unikalność permutacji.

**Minimalna liczba zamian:** Algorytm Heapa minimalizuje liczbę zamian - średnio około 1 zamiany na permutację (znacznie lepiej niż  $O(n)$  w algorytmach naiwnych).  $\square$

**Złożoność obliczeniowa:**

- **Liczba iteracji:**  $n!$  (liczba permutacji)
- **Koszt jednej iteracji:**  $O(1)$  - zamiany elementów i kopiowanie
- **Koszt kopiowania permutacji:**  $O(n)$  na permutację
- **Złożoność czasowa całkowita:**  $O(n! \cdot n)$
- **Złożoność pamięciowa:**  $O(n)$  - tablice  $a$  i  $c$

### 6.3.3 Algorytm generowania produktu kartezjańskiego

**Problem:** Dla danego zbioru  $items$  i liczby  $m$  wygenerować wszystkie  $m$ -krotki (sekwencje długości  $m$ ) złożone z elementów  $items$  (z powtórzeniami).

**Właściwości:**

- Liczba  $m$ -krotek ze zbioru  $k$ -elementowego:  $k^m$
- Odpowiada produktowi kartezjańskiemu  $items^m = items \times items \times \dots \times items$
- Generowanie w porządku leksykograficznym

---

**Algorithm 3** ProductSequences(*items*, *m*)

---

**Require:** *items* - lista *k* elementów, *m* - długość sekwencji

**Ensure:** Wszystkie *m*-krotki elementów z *items*

```
1:  $k \leftarrow |items|$ 
2: if  $m = 0$  then
3:   yield []
4:   return
5: end if
6: if  $k = 0$  then
7:   return ▷ Brak sekwencji
8: end if
9:  $digits \leftarrow [0, 0, \dots, 0]$  ▷ Tablica m cyfr w systemie bazy k
10: while true do
11:   yield [ $items[digits[0]], items[digits[1]], \dots, items[digits[m-1]]$ ]
12:    $i \leftarrow 0$  ▷ Inkrementacja licznika w systemie bazy k
13:   while  $i < m$  do
14:     if  $digits[i] + 1 < k$  then
15:        $digits[i] \leftarrow digits[i] + 1$ 
16:       break
17:     else
18:        $digits[i] \leftarrow 0$ 
19:        $i \leftarrow i + 1$ 
20:     end if
21:   end while
22:   if  $i = m$  then
23:     break ▷ Przepełnienie - wszystkie sekwencje wygenerowane
24:   end if
25: end while
```

---

**Dowód poprawności:**

*Dowód.* Algorytm traktuje sekwencje jako liczby w systemie pozycyjnym o podstawie *k*.

**Reprezentacja:** Każda *m*-krotka odpowiada liczbie w systemie bazy *k*:

$$\text{number} = digits[0] + digits[1] \cdot k + digits[2] \cdot k^2 + \dots + digits[m-1] \cdot k^{m-1}$$

gdzie  $0 \leq digits[i] < k$ .

**Kompletność:** Algorytm generuje wszystkie liczby od 0 do  $k^m - 1$ :

1. Rozpoczyna od  $[0, 0, \dots, 0]$  (liczba 0)
2. W każdej iteracji inkrementuje liczbę o 1 (linie 13-20)
3. Kończy przy przepełnieniu (liczba  $k^m$ , co odpowiada stanowi po ostatniej sekwencji)
4. Każdej liczbie odpowiada unikalna  $m$ -krotka

**Brak duplikatów:** Ponieważ algorytm ściśle liczy od 0 do  $k^m - 1$ , każda sekwencja jest wygenerowana dokładnie raz.

**Poprawność inkrementacji:** Linie 13-20 implementują standardową inkrementację w systemie pozycyjnym z propagacją przeniesienia (carry).  $\square$

#### **Złożoność obliczeniowa:**

- **Liczba iteracji:**  $k^m$  (liczba  $m$ -krotek)
- **Koszt jednej iteracji:**  $O(m)$  w najgorszym przypadku (propagacja przeniesienia przez wszystkie pozycje)
- **Średni koszt iteracji:**  $O(1)$  (przeniesienie rzadko propaguje daleko)
- **Złożoność czasowa całkowita:**  $O(k^m \cdot m)$  (pesymistyczna),  $O(k^m)$  (średnia)
- **Złożoność pamięciowa:**  $O(m)$  - tablica *digits*

#### **Porównanie z innymi metodami:**

- **Rekurencja:** Intuicyjna, ale wymaga  $O(m)$  stosu dla każdej sekwencji
- **Podejście iteracyjne:** Używane tutaj - wydajniejsze pamięciowo,  $O(m)$  bez nadmiarowych wywołań

## **6.4 Algorytm dokładny**

### **6.4.1 Przegląd algorytmu**

Algorytm oparty jest na pełnym przeszukiwaniu przestrzeni rozwiązań. Strategia polega na:

1. Wygenerowaniu wszystkich możliwych osadzeń grafu  $P$  w grafie  $G$
2. Obliczeniu brakujących krawędzi dla każdego osadzenia

3. Rozważeniu wszystkich możliwych  $m$ -krotnych osadzeń dla  $m$  kopii
4. Wybraniu osadzenia minimalizującego liczbę dodanych krawędzi

Algorytm składa się z dwóch głównych faz:

- **Faza 1:** Generowanie osadzeń i macierzy brakujących krawędzi
- **Faza 2:** Znajdowanie minimalnego rozszerzenia dla  $m$  kopii

#### 6.4.2 Szczegółowy opis algorytmu

**Faza 1: Generowanie osadzeń i macierzy brakujących krawędzi**

##### Krok 1: Generowanie $k$ -kombinacji wierzchołków $G$

Dla grafu  $G$  o  $n$  wierzchołkach generujemy wszystkie możliwe  $k$ -kombinacje wierzchołków, gdzie  $k = |V_P|$ . Każda kombinacja  $C_j \subseteq V_G$  reprezentuje potencjalny zbiór wierzchołków, na które można zmapować graf  $P$ .

- *Wejście:*  $G = (V_G, E_G)$ ,  $|V_G| = n$ ,  $k = |V_P|$
- *Wyjście:* Zbiór wszystkich  $k$ -kombinacji  $\{C_1, C_2, \dots, C_N\}$ , gdzie  $N = \binom{n}{k}$
- *Implementacja:* Algorytm 1 (GenerateCombinations) - szczegóły w sekcji 6.3.1
- *Złożoność:*  $O\left(\binom{n}{k} \times k\right)$

##### Krok 2: Generowanie permutacji wierzchołków $P$

Dla grafu  $P$  o  $k$  wierzchołkach generujemy wszystkie możliwe permutacje. Każda permutacja  $\pi_i : V_P \rightarrow V_P$  reprezentuje potencjalne uporządkowane mapowanie wierzchołków  $P$  na wierzchołki kombinacji  $C_j$ .

- *Wejście:*  $P = (V_P, E_P)$ ,  $|V_P| = k$
- *Wyjście:* Zbiór wszystkich permutacji  $\{\pi_1, \pi_2, \dots, \pi_M\}$ , gdzie  $M = k!$
- *Implementacja:* Algorytm 2 (GeneratePermutations, algorytm Heapa) - szczegóły w sekcji 6.3.2
- *Złożoność:*  $O(k! \times k)$

##### Krok 3: Obliczanie brakujących krawędzi

Dla każdej pary  $(C_j, \pi_i)$  obliczamy listę krawędzi, które należy dodać do  $G$ , aby podgraf indukowany przez  $C_j$  z mapowaniem  $\pi_i$  był izomorficzny z  $P$ .

Dla każdej pary wierzchołków  $(u, v) \in V_P \times V_P$ :

- Obliczamy obrazy:  $u' = C_j[\pi_i(u)]$ ,  $v' = C_j[\pi_i(v)]$
- Liczba krawędzi w  $P$ :  $e_P = A_P[\pi_i(u)][\pi_i(v)]$
- Liczba krawędzi w  $G$ :  $e_G = A_G[u'][v']$
- Brakujące krawędzie:  $\Delta = \max(0, e_P - e_G)$
- Dodajemy  $\Delta$  kopii krawędzi  $(u', v')$  do listy brakujących krawędzi

Wynik zapisujemy w macierzy `missingEdgesMatrix[i][j]` jako listę brakujących krawędzi dla permutacji  $i$  i kombinacji  $j$ .

- *Złożoność pojedynczego obliczenia*:  $O(k^2)$
- *Całkowita złożoność*:  $O\left(\binom{n}{k} \times k! \times k^2\right)$
- *Pamięć*:  $O\left(\binom{n}{k} \times k! \times k^2\right)$  w najgorszym przypadku

## **Faza 2: Znajdowanie minimalnego rozszerzenia dla $m$ kopii**

### **Krok 4: Generowanie $m$ -kombinacji osadzeń**

### **Krok 4: Generowanie $m$ -kombinacji osadzeń**

Generujemy wszystkie możliwe sposoby wyboru  $m$  różnych kombinacji wierzchołków spośród  $N = \binom{n}{k}$  dostępnych kombinacji. Każda taka  $m$ -kombinacja reprezentuje wybór  $m$  różnych podzbiorów dla  $m$  kopii grafu  $P$ . Ponieważ są to kombinacje (a nie permutacje z powtórzeniami), każde dwa podzbiory w wybranej  $m$ -krotce są różne, co zapewnia, że odpowiadające im kopie różnią się przynajmniej jednym wierzchołkiem.

- *Liczba  $m$ -kombinacji*:  $\binom{\binom{n}{k}}{m}$
- *Złożoność*:  $O\left(\binom{\binom{n}{k}}{m} \times m\right)$

### **Krok 5: Generowanie $m$ -krotek permutacji**

Dla każdej  $m$ -kombinacji podzbiorów, generujemy wszystkie możliwe  $m$ -krotki permutacji. Każda  $m$ -krotka  $(i_1, i_2, \dots, i_m)$  określa, jaką permutację stosujemy dla każdej z  $m$  kopii.

- *Liczba  $m$ -krotek*:  $(k!)^m$
- *Implementacja*: Algorytm 3 (ProductSequences) - szczegóły w sekcji 6.3.3
- *Złożoność*:  $O((k!)^m \times m)$

### Krok 6: Obliczanie unii zbiorów krawędzi

Dla każdej konfiguracji ( $m$ -kombinacja podzbiorów,  $m$ -krotka permutacji) obliczamy minimalny zbiór krawędzi potrzebny do stworzenia  $m$  kopii grafu  $P$ .

Kluczowa obserwacja: jeśli wielokrotne kopie wymagają tej samej krawędzi  $(u, v)$  z krotnościami  $k_1, k_2, \dots, k_m$ , wystarczy dodać  $\max(k_1, k_2, \dots, k_m)$  kopii tej krawędzi, ponieważ krawędzie mogą być współdzielone między kopiami.

Algorytm:

1. Inicjalizujemy mapę częstości: `edgeFrequencyMap = {}`
2. Dla każdej z  $m$  kopii ( $t = 1, \dots, m$ ):
  - (a) Pobieramy brakujące krawędzie dla kopii  $t$
  - (b) Tworzymy lokalną mapę częstości krawędzi dla tej kopii
  - (c) Aktualizujemy globalną mapę: dla każdej krawędzi  $e$ , ustawiamy  
`edgeFrequencyMap[e] = max(edgeFrequencyMap[e], localFrequency[e])`
3. Konwertujemy mapę częstości na listę krawędzi
4. Jeśli rozmiar listy jest mniejszy niż dotychczasowe minimum, aktualizujemy rozwiązanie
  - *Złożoność:*  $O\left(\binom{n}{m} \times (k!)^m \times m \times k^2\right)$

#### 6.4.3 Pseudokod algorytmu

Algorytm został podzielony na dwie części dla lepszej czytelności: Fazę 1 (generowanie osadzeń i obliczanie brakujących krawędzi) oraz Fazę 2 (znajdowanie minimalnego rozszerzenia dla  $m$  kopii).



---

**Algorithm 4** MinimalGraphExtension - Faza 1: Generowanie osadzeń

---

**Require:**  $G = (V_G, E_G)$  - multigraf "duży",  $|V_G| = n$

**Require:**  $P = (V_P, E_P)$  - multigraf "mały",  $|V_P| = k$

**Ensure:** Macierz *missingEdgesMatrix* z brakującymi krawędziami

```
1: // Generuj k-kombinacje wierzchołków G
2: combinations ← GenerateCombinations( $V_G, k$ )
3: indexToSubset ← IndexMap(combinations)
4: // Generuj permutacje wierzchołków P
5: permutations ← GeneratePermutations( $V_P$ )
6: indexToPermutation ← IndexMap(permutations)
7: missingEdgesMatrix ← Array[|indexToPermutation|][|indexToSubset|]
8: for i ← 0 to |indexToPermutation| - 1 do
9:   for j ← 0 to |indexToSubset| - 1 do
10:     $\pi \leftarrow \text{indexToPermutation}[i]$ ;  $C \leftarrow \text{indexToSubset}[j]$ 
11:    missingEdgesMatrix[i][j] ← []
12:    for u ← 0 to k - 1 do
13:      for v ← 0 to k - 1 do
14:         $u' \leftarrow C[\pi[u]]$ ;  $v' \leftarrow C[\pi[v]]$ 
15:         $\Delta \leftarrow \max(0, A_P[\pi[u]][\pi[v]] - A_G[u'][v'])$ 
16:        for t ← 0 to  $\Delta - 1$  do
17:          missingEdgesMatrix[i][j].append(( $u', v'$ ))
18:        end for
19:      end for
20:    end for
21:  end for
22: end for
23: return missingEdgesMatrix, indexToSubset, indexToPermutation
```

---

---

**Algorithm 5** MinimalGraphExtension - Faza 2: Znajdowanie minimalnego rozszerzenia

---

**Require:** *missingEdgesMatrix*, *indexToSubset*, *indexToPermutation*, *m*

**Ensure:** Lista krawędzi do dodania do *G*

```

1: mCombinations  $\leftarrow$  GenerateCombinations(indexToSubset.keys, m)
2: minimalEdges  $\leftarrow$  null; minimalSize  $\leftarrow$   $\infty$ 
3: for each  $\{j_1, \dots, j_m\}$  in mCombinations do
4:   for each  $(i_1, \dots, i_m)$  in ProductSeq(indexToPermutation.keys, m) do
5:     edgeFreqMap  $\leftarrow$  {}
6:     for t  $\leftarrow$  0 to m - 1 do
7:       missingEdges  $\leftarrow$  missingEdgesMatrix[i_t][j_t]
8:       localFreq  $\leftarrow$  {}
9:       for each e in missingEdges do
10:        localFreq[e]  $\leftarrow$  localFreq[e] + 1
11:      end for
12:      for each (e, freq) in localFreq do
13:        edgeFreqMap[e]  $\leftarrow$  max(edgeFreqMap[e], freq)
14:      end for
15:    end for
16:    addedEdges  $\leftarrow$  []
17:    for each (e, freq) in edgeFreqMap do
18:      for t  $\leftarrow$  0 to freq - 1 do
19:        addedEdges.append(e)
20:      end for
21:    end for
22:    if |addedEdges| < minimalSize then
23:      minimalSize  $\leftarrow$  |addedEdges|; minimalEdges  $\leftarrow$  addedEdges
24:    end if
25:  end for
26: end for
27: return minimalEdges

```

---

## 6.5 Dowód poprawności algorytmu

**Twierdzenie 1** (Poprawność algorytmu MinimalGraphExtension). *Algorytm MinimalGraphExtension zwraca minimalną listę krawędzi do dodania do  $G$ , aby zawierał  $m$  różnych kopii  $P$  (tzn. kopii różniących się przynajmniej jednym wierzchołkiem).*

*Dowód.* Dowód podzielimy na trzy części: kompletność, poprawność i minimalność.

### Część 1: Kompletność (algorytm znajduje rozwiązanie, jeśli istnieje)

Założmy, że istnieje rozszerzenie  $G'$  grafu  $G$  zawierające  $m$  różnych kopii  $P$  (tzn. kopii różniących się przynajmniej jednym wierzchołkiem), osiągnięte przez dodanie zbioru krawędzi  $E_{add}$ .

1. Dla każdej z  $m$  kopii  $P$  w  $G'$  istnieje:
  - $k$ -kombinacja wierzchołków  $C_t \subseteq V_G$  ( $t = 1, \dots, m$ )
  - Permutacja  $\pi_t : V_P \rightarrow V_P$
  - Takie że podgraf  $G'$  indukowany przez  $C_t$  z odpowiednim mapowaniem jest izomorficzny z  $P$
2. Kombinacje  $C_t$  są różne ( $C_i \neq C_j$  dla  $i \neq j$ ), co zapewnia, że kopie różnią się przynajmniej jednym wierzchołkiem
3. Algorytm generuje wszystkie możliwe  $m$ -kombinacje  $k$ -podzbiorów (linia 3 Fazy 2), co gwarantuje, że wybranych  $m$  podzbiorów jest parami różnych
4. Dla każdej  $m$ -kombinacji, algorytm generuje wszystkie możliwe  $m$ -krotki permutacji (linia 4 Fazy 2)
5. Zatem algorytm rozważy kombinację  $\{C_1, \dots, C_m\}$  i krotkę permutacji  $(\pi_1, \dots, \pi_m)$  odpowiadającą rzeczywistemu rozwiązaniu
6. Dla tej kombinacji i krotki permutacji, algorytm obliczy dokładnie te same krawędzie, które są w  $E_{add}$  (linie 5-16 Fazy 2)

### Część 2: Poprawność (dodane krawędzie są wystarczające)

Dla dowolnej  $m$ -kombinacji podzbiorów i  $m$ -krotki permutacji rozważanej przez algorytm:

1. Dla każdej z  $m$  kopii (linie 33-42):
  - Algorytm oblicza brakujące krawędzie zapisane wcześniej w *missingEdgesMatrix*
  - Te krawędzie są dokładnie tymi, których brakuje do stworzenia izomorfizmu (z Fazy 1)
2. Operacja maksimum na krotnościach (linia 41) zapewnia:
  - Jeśli wielokrotne kopie potrzebują tej samej krawędzi  $(u, v)$  z krotnościami  $k_1, k_2, \dots$ , dodajemy  $\max(k_1, k_2, \dots)$  kopii

- To jest *wystarczające*, bo krawędzie mogą być współdzielone między kopiami
- To jest *konieczne*, bo każda kopia wymaga odpowiedniej krotności

3. Po dodaniu krawędzi z *addedEdges* do *G*:

- Każda z *m* kopii ma wystarczającą liczbę krawędzi między każdą parą wierzchołków
- Każda kopia jest izomorficzna z *P*

### Część 3: Minimalność (algorytm znajduje minimum)

1. Algorytm przeszukuje wszystkie możliwe sposoby osadzenia *m* kopii *P* w *G* (linie 30-31)
2. Dla każdego sposobu oblicza minimalną liczbę krawędzi potrzebnych do realizacji tego osadzenia (linie 33-48)
3. Wybiera osadzenie wymagające najmniejszej liczby krawędzi (linie 49-52)
4. Nie istnieje sposób osadzenia *m* kopii *P* w *G* wymagający mniej krawędzi, bo wszystkie sposoby zostały rozważone

Zatem algorytm jest poprawny - zwraca minimalną liczbę krawędzi wystarczających do stworzenia *m* różnych kopii *P* w *G* (różniących się przynajmniej jednym wierzchołkiem).  $\square$

## 6.6 Analiza złożoności obliczeniowej

### 6.6.1 Złożoność czasowa

Oznaczmy:

- $n = |V_G|$  - liczba wierzchołków dużego grafu *G*
- $k = |V_P|$  - liczba wierzchołków małego grafu *P*
- $m$  - liczba wymaganych kopii *P* w *G*

**Rozbicie na fazy:**

1. **Faza 1a - Generowanie kombinacji:**  $O\left(\binom{n}{k} \times k\right)$

- Liczba kombinacji:  $\binom{n}{k}$
- Koszt generowania jednej kombinacji:  $O(k)$

2. **Faza 1b - Generowanie permutacji:**  $O(k! \times k)$

- Liczba permutacji:  $k!$
- Koszt generowania jednej permutacji:  $O(k)$

3. **Faza 1c - Obliczanie missingEdgesMatrix:**  $O\left(\binom{n}{k} \times k! \times k^2\right)$

- Dla każdej z  $\binom{n}{k}$  kombinacji
- Dla każdej z  $k!$  permutacji
- Porównanie  $k^2$  par krawędzi

4. **Faza 2a - Generowanie m-kombinacji osadzeń:**  $O\left(\binom{\binom{n}{k}}{m} \times m\right)$

- Liczba  $m$ -kombinacji:  $\binom{\binom{n}{k}}{m}$

5. **Faza 2b - Główna pętla przeszukiwania:**

$$O\left(\binom{\binom{n}{k}}{m} \times (k!)^m \times m \times k^2\right)$$

- Dla każdej  $m$ -kombinacji podzbiorów:  $\binom{\binom{n}{k}}{m}$
- Dla każdej  $m$ -krotki permutacji:  $(k!)^m$
- Dla każdej z  $m$  kopii:  $m$
- Obliczanie częstości krawędzi:  $O(k^2)$  dla jednej kopii

**Całkowita złożoność czasowa:**

$$T(n, k, m) = O\left(\binom{n}{k} \times k! \times k^2\right) + O\left(\binom{\binom{n}{k}}{m} \times (k!)^m \times m \times k^2\right)$$

**Dominujący składnik** dla małych  $m$  (gdy  $m \ll \binom{n}{k}$ ):

$$T(n, k, m) = O\left(\binom{n}{k} \times k! \times k^2\right)$$

**Dominujący składnik** dla większych  $m$ :

$$T(n, k, m) = O\left(\binom{\binom{n}{k}}{m} \times (k!)^m \times m \times k^2\right)$$

**Oszacowania asymptotyczne:**

- $\binom{n}{k} = O\left(\frac{n^k}{k!}\right)$  - wielomianowe względem  $n$  dla stałego  $k$
- $k!$  - silniowe względem  $k$
- $\binom{\binom{n}{k}}{m} \approx O((n^k)^m)$  dla dużych  $n$
- $(k!)^m$  - wykładnicze względem  $m$

### 6.6.2 Złożoność pamięciowa

**Główne struktury danych:**

1. **missingEdgesMatrix:**  $O\left(\binom{n}{k} \times k! \times k^2\right)$ 
  - Tablica 2D o wymiarach  $k! \times \binom{n}{k}$
  - Każda komórka zawiera listę brakujących krawędzi (w najgorszym  $O(k^2)$  krawędzi)
2. **indexToSubset:**  $O\left(\binom{n}{k} \times k\right)$ 
  - Przechowuje  $\binom{n}{k}$  kombinacji, każda długości  $k$
3. **indexToPermutation:**  $O(k! \times k)$ 
  - Przechowuje  $k!$  permutacji, każda długości  $k$
4. **Zmienne tymczasowe w głównej pętli:**  $O(k^2)$

**Całkowita złożoność pamięciowa:**

$$S(n, k, m) = O\left(\binom{n}{k} \times k! \times k^2\right)$$

**Uwaga:** Złożoność pamięciowa jest niezależna od  $m$  (nie przechowujemy wszystkich  $m$ -krotek, generujemy je leniwie).

### 6.6.3 Charakterystyka algorytmu

#### Klasa złożoności:

- Problem jest **NP-trudny** (redukcja z problemu izomorfizmu podgrafów)
- Algorytm dokładny ma złożoność **wykładniczą** względem  $k$  (ze względu na  $k!$ )
- Algorytm ma złożoność **wielomianowo-wykładniczą** względem  $m$

#### Praktyczne ograniczenia:

- Algorytm jest wykonalny dla małych wartości  $k$  ( $k \leq 6 - 7$ ) i  $n \leq 20$
- Dla większych wartości  $k$  lub  $n$  algorytm staje się niepraktyczny
- Wartość  $m$  ma mniejszy wpływ na czas wykonania niż  $k$  (dla małych  $m$ )

## 6.7 Aproksymacyjne minimalne rozszerzenie multigrafu - algorytm pierwszy

Zdefiniowane są dwa skierowane multigrafy: mniejszy  $G_1 = (V_1, E_1)$  oraz większy  $G_2 = (V_2, E_2)$ . Multigrafy  $G_1$  i  $G_2$  są reprezentowane przez macierze sąsiedztwa  $A_{G_1}$  i  $A_{G_2}$ . Wartość komórki  $A_{G_1}[i, j]$  oznacza liczbę krawędzi skierowanych od wierzchołka  $i$  do wierzchołka  $j$  w grafie  $G_1$ . Podana jest także liczba szukanych kopii  $m$ . Idea algorytmu polega na iteracyjnym znajdowaniu *najtańszej* kopii  $G_1$  w  $G_2$ .

### 6.7.1 Opis algorytmu

1. Iteruj po wszystkich możliwych nasionach, czyli parach  $(u_1, u_2)$ , gdzie  $u_1 \in V_1$  i  $u_2 \in V_2$ .
2. Ustal zerową macierz kosztu  $C_{u_1, u_2}$  o rozmiarze  $|V_2|$  na  $|V_2|$ , która reprezentuje jakie i ile krawędzi skierowanych należy dodać do  $G_2$  dla danych nasion  $(u_1, u_2)$ .
3. Do mapowania dodaj takich sąsiadów nasion  $u'_1 \in V_1$  i  $u'_2 \in V_2$ , które minimalizują koszt zdefiniowany wzorem:

$$Koszt = \max(0, (A_{G_1}[u_1, u'_1] - A_{G_2}[u_2, u'_2])) + \max(0, A_{G_1}[u'_1, u_1] - A_{G_2}[u'_2, u_2])) \quad (1)$$

Zapisz do odpowiednich komórek macierzy  $C_{u_1, u_2}$ :

$$C_{u_1, u_2}[u_2, u'_2] = C_{u_1, u_2}[u_2, u'_2] + \max(0, (A_{G_1}[u_1, u'_1] - A_{G_2}[u_2, u'_2])) \quad (2)$$

$$C_{u_1, u_2}[u'_2, u_2] = C_{u_1, u_2}[u'_2, u_2] + \max(0, (A_{G_1}[u'_1, u_1] - A_{G_2}[u'_2, u_2])) \quad (3)$$

4. Następnie próbuj zachłannie rozszerzyć mapowanie na resztę wierzchołków  $G_1$ . Do mapowania dodawaj tylko wierzchołki, które nie zostały jeszcze zmapowane.
5. Ze wszystkich macierzy kosztu ( $|V_1| \cdot |V_2|$  macierzy) wybierz  $m$  najlepszych. Przez najlepszą macierz rozumiemy taką, dla której suma wartości we wszystkich komórkach jest najmniejsza. Każda macierz kosztu odpowiada jednemu mapowaniu - jeśli dowolna para z  $m$  macierzy mapuje te same wierzchołki w  $G_2$ , wybierz kolejną  $m + 1$  najlepszą macierz kosztu i ponownie sprawdź warunek. Docelowo, żadna para z wybranych  $m$  macierzy kosztu nie może mapować tych samych wierzchołków w  $G_2$ .
6. Macierz  $K$  o rozmiarze  $|V_2|$  na  $|V_2|$  skonstruuj w następujący sposób -  $K[i, j] = \max_{k=1, \dots, m}(C_k[i, j])$ , gdzie  $C_k$  dla  $k = 1, \dots, m$  to  $m$  najlepszych macierzy kosztu.
7. Zwróć macierz  $K$  reprezentującą minimalne rozszerzenie.

### 6.7.2 Złożoność obliczeniowa

Pesymistyczna złożoność opisanego algorytmu aproksymacyjnego równa jest:

$$O(|V_1| \cdot |V_2| \cdot (|V_1| \cdot |E_1| \cdot |E_2|) \cdot m) = O(|V_1|^2 \cdot |V_2| \cdot |E_1| \cdot |E_2| \cdot m) \quad (4)$$

Czynnik  $|V_1| \cdot |V_2|$  odpowiada za iteracyjne wybieranie nasion do mapowania. Czynniki  $|V_1| \cdot |E_1| \cdot |E_2|$  to koszt zachłannego przeszukiwania grafu w celu minimalizacji kosztu.  $m$  to liczba wybieranych macierzy kosztu (liczba kopii). Opisany algorytm ma zatem złożoność wielomianową.

### 6.7.3 Uzasadnienie

Opisany algorytm jest heurystyką zachłanną naszego autorstwa. Algorytm gwarantuje, że macierz  $K$  rzeczywiście uczyni  $G_2$  rozszerzeniem zawierającym kopie  $G_1$ .

Z definicji każdej macierzy  $C_{u_1, u_2}$  wpisy odpowiadają dokładnie brakującym krawędziom. Jeśli na końcu algorytm utworzy macierz  $K$  zgodnie z opisem, to po dodaniu



tych krawędzi w  $G_2$  wszystkie odwzorowania skonstruowane przez algorytm staną się izomorficznymi (liczbowo zgodnymi) kopiami  $G_1$ . Zatem algorytm zwraca dopuszczalne rozwiązanie.

Niestety nie ma dowodu, że algorytm daje rozwiązanie optymalne, ani że ma stałą gwarancję aproksymacji. To heurystyka zachłanna — lokalnie wybiera najtańsze mapowanie — ale problem minimalnego rozszerzenia (znalezienie najmniejszego zbioru dodatkowych krawędzi, by powstały  $m$  kopii) jest kombinatorycznie trudny i algorytm zachłanny może prowadzić do lokalnie optymalnych, ale globalnie złych decyzji.

## 7 Aproksymacyjne minimalne rozszerzenie multigrafu - algorytm drugi

Algorytm polega na losowym wybraniu  $n$   $k$ -elementowych podzbiorów wierzchołków z grafu  $G$ , a następnie użyciu pewnej heurystyki do stworzenia macierzy kosztów przypisania pomiędzy konkretnymi wierzchołkami z grafu  $P$ , oraz grafu  $G$ . Następnie na tej macierzy wywoływany jest Algorytm Węgierski, który znajduje najlepsze mapowanie wierzchołków, w sensie przyjętej heurystyki. Ta operacja jest powtarzana na każdym z  $n$  podzbiorów, tak żeby uzyskać wymagane  $n$  podgrafów grafu  $G$ , różniących się przynajmniej jednym wierzchołkiem.

### 7.1 Heurystyczna Funkcja Kosztu

Algorytm Węgierski operuje na macierzy kosztów  $M$ , która stanowi heurystyczną aproksymację rzeczywistego kosztu rozszerzenia. Wybór tej heurystyki ma kluczowy wpływ na jakość znajdowanych rozwiązań — lepsza heurystyka (bliższa rzeczywistemu kosztowi) sprawi, że najlepsze rozwiązania heurystyczne będą z większym prawdopodobieństwem odpowiadać najlepszym rozwiązaniom rzeczywistym.

Poniżej przedstawiono sześć propozycji heurystyk do obliczania macierzy kosztów  $M$ , które chcielibyśmy poddać testom w celu wybrania funkcji dającej najlepsze rezultaty.

1. **Heurystyka Różnicy Stopni.** Jest to najprostsza i najszybsza heurystyka. Zakłada, że wierzchołki o podobnej łącznej liczbie połączeń (niezależnie od kierunku) powinny być ze sobą mapowane.

**Wzór:**

$$M[i, j] = |\deg_{\text{total}}(p_i) - \deg_{\text{total}}(g_j)|$$

gdzie  $\deg_{\text{total}}(v) = \deg_{\text{in}}(v) + \deg_{\text{out}}(v)$ .

**Złożoność budowy:**  $\mathcal{O}(N^2 + k^2 + kN)$ . Zdominowana przez wstępne obliczenie stopni:  $\mathcal{O}(N^2)$ .

2. **Heurystyka Dopasowania Stopni Skierowanych.** Nieco bardziej zaawansowana wersja dla grafów skierowanych. Koszt  $M[i, j]$  jest sumą absolutnych różnic stopni wejściowych i wyjściowych.

**Wzór:**

$$M[i, j] = |\deg_{\text{in}}(p_i) - \deg_{\text{in}}(g_j)| + |\deg_{\text{out}}(p_i) - \deg_{\text{out}}(g_j)|$$

**Złożoność budowy:**  $\mathcal{O}(N^2 + k^2 + kN)$ . Również zdominowana przez  $\mathcal{O}(N^2)$ .

3. **Heurystyka Dopasowania Stopni Skierowanych z Ignorowaniem Nadmiaru.** Wersja modyfikowana poprzedniej heurystyki, która ignoruje nadmiar stopni w większym grafie. Pomaga to uniknąć karania wierzchołków w  $G$  o większych stopniach niż ich odpowiedniki w  $P$ .

**Wzór:**

$$M[i, j] = \max(0, \deg_{\text{in}}(p_i) - \deg_{\text{in}}(g_j)) + \max(0, \deg_{\text{out}}(p_i) - \deg_{\text{out}}(g_j))$$

**Złożoność budowy:**  $\mathcal{O}(N^2 + k^2 + kN)$ . Również zdominowana przez  $\mathcal{O}(N^2)$ .

4. **Heurystyka Histogramu Stopni Sąsiadów.** Ta heurystyka analizuje najbliższe otoczenie. Dla każdego wierzchołka  $p_i$  i  $g_j$  tworzony jest histogram stopni ich bezpośrednich sąsiadów. Koszt  $M[i, j]$  to odległość między tymi dwoma rozkładami.

**Wzór:** Niech  $H_P(p_i)$  będzie histogramem stopni sąsiadów  $p_i$ , a  $H_G(g_j)$  analogicznym histogramem dla  $g_j$ . Niech  $S$  będzie większą z dwóch wartości stopni wierzchołków. Wtedy:

$$M[i, j] = \sum_{s=0}^S |H_P(p_i)[s] - H_G(g_j)[s]|$$

Zakładamy, że histogramy są normalizowane do tej samej długości  $S$ .

**Złożoność budowy:** Wymaga obliczenia stopni ( $\mathcal{O}(N^2)$ ), zbudowania  $N + k$  histogramów (w  $\mathcal{O}(N^2)$ ) oraz obliczenia  $kN$  odległości (w  $\mathcal{O}(kN^2)$ ). Całkowity koszt to  $\mathcal{O}(N^2 + kN^2)$ .

5. **Heurystyka Dopasowania Struktury (Liczba Trójkątów).** Ta heurystyka próbuje uchwycić naturę problemu przez mierzenie lokalnej struktury. Wierzchołek  $p_i$  będący częścią wielu trójkątów powinien być mapowany na  $g_j$ , który również uczestniczy w wielu trójkątach.

**Wzór:**

$$M[i, j] = \alpha \cdot |\deg(p_i) - \deg(g_j)| + \beta \cdot \max(0, \text{triangles}(p_i) - \text{triangles}(g_j))$$

gdzie  $\text{triangles}(v)$  to liczba trójkątów, w których uczestniczy wierzchołek  $v$ . Karany jest tylko niedobór trójkątów w  $g_j$  względem  $p_i$ .

**Złożoność budowy:** Zdominowana przez koszt zliczania trójkątów w grafie  $G$ , który wynosi  $\mathcal{O}(N^3)$  (przy użyciu mnożenia macierzy).

6. **Heurystyka Zachłannego Dopasowywania Sąsiadów.** Najbardziej złożona obliczeniowo, ale potencjalnie najdokładniejsza heurystyka. Próbuje bezpośrednio oszacować koszt w najbliższym otoczeniu. Koszt  $M[i, j]$  jest sumą "najlepszych" możliwych dopasowań dla sąsiadów  $p_i$  wśród sąsiadów  $g_j$ .

**Wzór (konceptyjny):**

$$M[i, j] = \sum_{p_k \in \mathcal{N}(p_i)} \left( \min_{g_l \in \mathcal{N}(g_j)} C(p_k, g_l) \right)$$

gdzie  $\mathcal{N}(v)$  to sąsiedztwo  $v$ , a  $C$  to prosta heurystyka kosztu (np. z Opcji 1).

**Złożoność budowy:** Dla prostej heurystyki wypełnienie tymczasowej macierzy kosztów, to koszt  $\mathcal{O}(N^2)$ . Wypełnienie jednej komórki  $M[i, j]$  kosztuje  $\mathcal{O}(\deg(p_i) \cdot \deg(g_j))$ . W najgorszym przypadku dla grafów gęstych (gdy  $\deg \approx k$  lub  $N$ ), całkowity koszt wypełnienia  $kN$  komórek wynosi  $\mathcal{O}(k^2 N^2)$ . Czyli koszt całej heurystyki to  $\mathcal{O}(N^2 + k^2 N^2)$ , czyli  $\mathcal{O}(k^2 N^2)$ .

## 7.2 Opis Algorytmów

Ta część opisuje dwie części algorytmu aproksymacyjnego: fazę generowania k-elementowych podzbiorów wierzchołków większego grafu oraz fazę znajdowania najlepszych mapowań dla tych podzbiorów. Lepsze rozwiązanie można uzyskać poprzez iterowanie po podzbiórach i znajdowanie najlepszych mapowań dla każdego z nich, np. korzystając z Algorytmu Murty'ego, jednak z powodu ograniczenia wielomianowej złożoności czasowej, w tej wersji algorytmu generujemy tylko  $n$  podzbiorów.

1. **Faza 1:** Najpierw generowana jest lista  $\mathcal{L}$   $n$   $k$ -elementowych podzbiorów wierzchołków z  $G$  ( $S \subset V_G$ ).
2. **Faza 2:** Następnie, algorytm iteruje  $n$  razy. W każdej iteracji  $i$ , bierze  $i$ -ty podzbiór  $S_i$  z listy  $\mathcal{L}$  i rozwiązuje dla niego *mały* ( $k \times k$ ) problem przypisania (Algorytmem Węgierskim), aby znaleźć najlepsze mapowanie  $f_i : V_P \rightarrow S_i$ . Znalezione rozszerzenie jest dodawane do grafu  $G$ .

### 7.2.1 Algorytm Fazy 1: Generowanie Podzbiorów

Celem tej fazy jest szybkie wygenerowanie listy  $\mathcal{L}$   $n$  unikalnych  $k$ -elementowych podzbiorów  $V_G$ .

---

**Algorithm 6** Faza 1: Generowanie  $n$  Pierwszych Podzbiorów Leksykograficznych

---

**Require:** Graf  $G$  (o  $N$  wierzchołkach,  $V_G = \{v_1, \dots, v_N\}$ ), Liczba podzbiorów  $n$ ,  
Rozmiar podzbioru  $k$

**Ensure:** Lista  $\mathcal{L}$  (do  $n$ ) unikalnych  $k$ -elementowych podzbiorów  $V_G$

```

1:  $\mathcal{L} \leftarrow []$ 
2: if  $n = 0$  or  $k > N$  then
3:   return  $\mathcal{L}$ 
4: end if
5:  $\mathcal{C} \leftarrow \text{COMBINATIONS\_GENERATOR}(V_G, k)$   $\triangleright$  Inicjalizuj generator kombinacji  $\binom{N}{k}$ 
6: for  $i \leftarrow 1$  to  $n$  do
7:   if  $\mathcal{C}.\text{hasNext}()$  then
8:      $S \leftarrow \mathcal{C}.\text{next}()$   $\triangleright$  Pobierz następny podzbiór, np.  $\{v_1, \dots, v_k\}$ 
9:      $\mathcal{L}.\text{append}(S)$ 
10:  else
11:    break  $\triangleright$  Zakończ, jeśli jest mniej niż  $n$  możliwych kombinacji
12:  end if
13: end for
14: return  $\mathcal{L}$ 

```

---

### Dowód Poprawności (Faza 1)

Poprawność procedury oznacza gwarancję zakończenia i zwrócenia listy  $\mathcal{L}$  zawierającej  $n$  (lub mniej, jeśli niemożliwe) unikalnych  $k$ -elementowych podzbiorów.

- **Zakończenie:** Pętla **for** (linia 6) jest ściśle ograniczona przez  $n$ . W każdej iteracji wywoływana jest operacja `next()` generatora kombinacji. Generator ten

zakończy działanie (zwracając **false** w linii 7) po wygenerowaniu wszystkich  $\binom{N}{k}$  kombinacji. Ponieważ  $n$  jest skończone, algorytm zawsze się zakończy.

- **Unikalność:** Standardowy generator kombinacji (leksykograficzny) z definicji produkuje każdą kombinację (podzbiór) dokładnie jeden raz. Ponieważ algorytm tylko pobiera elementy z tego generatora, lista  $\mathcal{L}$  z definicji zawiera unikalne podzbiory.
- **Ilość Wyjścia:** Algorytm próbuje pobrać  $n$  podzbiorów. Jeśli  $n > \binom{N}{k}$ , generator wyczerpie się wcześniej, a pętla zostanie przerwana (linia 10). Algorytm poprawnie zwróci  $\binom{N}{k}$  podzbiorów, czyli wszystkie, które istnieją. W przeciwnym razie zwróci dokładnie  $n$  podzbiorów.

### Analiza Złożoności (Faza 1)

Analiza ta dotyczy kosztu wygenerowania  $n$  podzbiorów.

- **Inicjalizacja (linie 1-5):** Generator jest inicjowany na pierwszym elemencie, np.  $\{v_1, \dots, v_k\}$  w czasie  $\mathcal{O}(k)$ .
- **Pętla Główna (linie 6-12):** Wykonuje się  $n$  razy.
- **Operacje w pętli:**
  - Wygenerowanie `next()` (linia 8): Koszt wygenerowania następnej kombinacji leksykograficznej z poprzedniej jest w zamortyzowanym czasie stałym  $\mathcal{O}(1)$ , a w najgorszym przypadku wynosi  $\mathcal{O}(k)$ .
  - Dodanie do listy  $\mathcal{L}$  (linia 9): Wymaga skopiowania podzbioru o rozmiarze  $k$ , co ma koszt  $\mathcal{O}(k)$ .
  - Koszt pętli wynosi  $n \cdot \mathcal{O}(k)$ .

**Całkowita złożoność obliczeniowa (Faza 1):**  $\mathcal{O}(n \cdot k)$ .

### 7.2.2 Algorytm Fazy 2: Iteracyjne Rozwiązywanie i Rozszerzanie

Faza 2 pobiera listę  $\mathcal{L}$   $n$  podzbiorów-kandydatów i iteracyjnie znajduje najlepsze mapowanie  $V_P \rightarrow S_i$  dla każdego  $S_i \in \mathcal{L}$ , stosując rozszerzenie do grafu  $G$ .

---

**Algorithm 7** Faza 2: Iteracyjne Rozwiązywanie na Podzbiorach

---

**Require:** Lista podzbiorów  $\mathcal{L}$ , Oryginalny graf  $A_G$ , Wzorzec  $A_P$

**Ensure:** Końcowa macierz  $A_{final}$ , Całkowity koszt  $C_{total}$

```
1:  $A_{curr} \leftarrow \text{COPY}(A_G)$ 
2:  $C_{total} \leftarrow 0$ 
3: for each  $S_i$  in  $\mathcal{L}$  do ▷ Iteruj  $n$  razy
4:
5:    $D_{curr} \leftarrow \text{GETDEGREES}(A_{curr})$  ▷ Tablica stopni wierzchołków w  $A_{curr}$ 
6:    $D_P \leftarrow \text{GETDEGREES}(A_P)$  ▷ Tablica stopni wierzchołków w  $P$ 
7:
8:    $M_{sub} \leftarrow \text{HEURYSTYKA}(S_i, A_{curr}, A_P, D_{curr}, D_P)$  ▷ Wypełnij  $M_{sub}$  wagami
9:
10:   $f_i \leftarrow \text{HUNGARIANALGORITHM}(M_{sub})$  ▷ Wywołaj Algorytm Węgiński
11:
12:   $cost_i \leftarrow 0$  ▷ Zastosuj rozszerzenie dla tego mapowania
13:  for  $u \leftarrow 1$  to  $k$  do ▷ Iteruj  $k \times k$  w macierzy wzorca  $P$ 
14:    for  $v \leftarrow 1$  to  $k$  do
15:       $g_u \leftarrow f_i(u)$ 
16:       $g_v \leftarrow f_i(v)$ 
17:       $w_{needed} \leftarrow A_P[u, v]$ 
18:       $w_{current} \leftarrow A_{curr}[g_u, g_v]$ 
19:      if  $w_{needed} > w_{current}$  then
20:         $w_{missing} \leftarrow w_{needed} - w_{current}$ 
21:         $A_{curr}[g_u, g_v] \leftarrow w_{needed}$ 
22:         $cost_i \leftarrow cost_i + w_{missing}$ 
23:      end if
24:    end for
25:  end for
26:   $C_{total} \leftarrow C_{total} + cost_i$ 
27: end for
28:  $A_{final} \leftarrow A_{curr}$ 
29: return  $A_{final}, C_{total}$ 
```

---

**Dowód Poprawności (Faza 2)** Poprawność procedury (Algorytm 7) opiera się na dwóch głównych filarach: gwarancji zakończenia oraz poprawności iteracyjnego rozszerzania grafu.

- **Zakończenie:** Algorytm składa się z jednej głównej pętli **for** (linia 2), która

iteruje po liście  $\mathcal{L}$ . Ponieważ lista  $\mathcal{L}$  jest skończona (zawiera  $n$  podzbiorów), pętla `for` wykona się  $n$  razy. Musimy przeanalizować operacje wewnątrz pętli:

1. **GetDegrees** (linie 5-6): Obliczenie stopni dla macierzy  $N \times N$  i  $k \times k$  jest operacją o skończonej, wielomianowej złożoności (odpowiednio  $\mathcal{O}(N^2)$  i  $\mathcal{O}(k^2)$ ).
2. **Heurystyka** (linia 8): Wypełnienie macierzy  $k \times k$  jest operacją o złożoności zależnej od wybranej heurystyki. Dla prostszych heurystyk, jak liczenie różnicy stopni, przy podanej macierzy stopni i przy wielkości macierzy kosztów  $k \times k$ , jej złożoność to  $\mathcal{O}(k^2)$ .
3. **HungarianAlgorithm** (linia 10): Wywołanie Algorytmu Węgierskiego na macierzy  $k \times k$  ma złożoność wielomianową  $\mathcal{O}(k^3)$ .
4. Pętla rozszerzenia (linie 13-25): Jest to zagnieżdżona pętla wykonująca  $k \times k = k^2$  operacji  $\mathcal{O}(1)$ , stąd jej koszt to  $\mathcal{O}(k^2)$ .

Ponieważ  $n, k, N$  są skończone, a wszystkie operacje wewnątrz pętli głównej mają skończoną, wielomianową złożoność, cały algorytm gwarantuje zakończenie działania.

- **Poprawność Rozszerzenia:** Musimy udowodnić, że końcowy graf  $A_{final}$  zawiera  $n$  podgrafów izomorficznych z  $P$ , zgodnie ze znalezionymi mapowaniami  $f_1, \dots, f_n$ . Niech  $A_{curr}^{(i)}$  będzie stanem macierzy  $A_{curr}$  na początku  $i$ -tej iteracji pętli `for` (gdzie  $A_{curr}^{(1)} = A_G$ ). Niezmiennik:  $A_{curr}^{(i)}$  zawiera podgrafy  $f_1, \dots, f_{i-1}$ .
  - **Baza** ( $i = 1$ ):  $A_{curr}^{(1)} = A_G$ . Niezmiennik jest trywialnie prawdziwy (zawiera 0 podgrafów).
  - **Krok Indukcyjny:** Zakładamy, że  $A_{curr}^{(i)}$  zawiera  $f_1, \dots, f_{i-1}$ . W  $i$ -tej iteracji algorytm znajduje nowe mapowanie  $f_i$  (linie 5-10). Następnie pętla rozszerzenia (linie 13-25) iteruje przez wszystkie  $k^2$  par w  $P$ . W linii 19, algorytm sprawdza, czy wymagana krawędź  $(f_i(u), f_i(v))$  o krotności  $w_{needed}$  istnieje. Jeśli nie ( $w_{needed} > w_{current}$ ), algorytm *dodaje* brakujące krawędzie (lub zwiększa krotność) w linii 21 ( $A_{curr}[g_u, g_v] \leftarrow w_{needed}$ ).
  - Operacja ta jest monotoniczna: krotności krawędzi w  $A_{curr}$  nigdy nie maleją ( $A_{curr}^{(i+1)} \geq A_{curr}^{(i)}$ ).
  - Zatem, po zakończeniu pętli (linii 13-25), graf  $A_{curr}^{(i+1)}$  gwarantuje zawarcie podgrafu  $f_i$ . Ponieważ krawędzie są tylko dodawane,  $A_{curr}^{(i+1)}$  nadal zawiera wszystkie poprzednie podgrafy  $f_1, \dots, f_{i-1}$ .

- **Zakończenie:** Po  $n$  iteracjach,  $A_{final} = A_{curr}^{(n+1)}$  zawiera wszystkie  $n$  podgrafów  $f_1, \dots, f_n$ .

Algorytm poprawnie również sumuje koszt  $C_{total}$  jako sumę kosztów  $cost_i$  poniesionych w każdej iteracji, odzwierciedlając fakt, że krawędzie dodane we wcześniejszych iteracjach zmniejszają koszt rozszerzeń w późniejszych iteracjach.

**Analiza Złożoności (Faza 2)** Analizujemy koszt obliczeniowy (czasowy) Algorytmu 7 w zależności od  $N$  (rozmiar  $G$ ),  $k$  (rozmiar  $P$ ) oraz  $n$  (liczba podzbiorów w  $\mathcal{L}$ ).

- **Operacje Wstępne (linie 1-2):** Kopiowanie macierzy  $A_G$  ma koszt  $\mathcal{O}(N^2)$ .
- **Pętla Główna (linia 4):** Pętla `for` wykonuje się  $n$  razy. Musimy przeanalizować koszt *jednej* iteracji.
- **Koszt jednej iteracji (linie 6-27):**
  - Linia 6, `GetDegrees(A_curr)`: Wymaga przejścia przez macierz  $N \times N$ , aby obliczyć stopnie dla  $A_{curr}$ . Koszt:  $\mathcal{O}(N^2)$ .
  - Linia 7, `GetDegrees(A_P)`: Wymaga przejścia przez macierz  $k \times k$ . Koszt:  $\mathcal{O}(k^2)$ .
  - Linia 9, `Heurystyka`: Wypełnia macierz  $M_{sub}$  o rozmiarze  $k \times k$ . Zakładając, że stopnie są już obliczone, każda z  $k^2$  komórek jest obliczana w czasie  $\mathcal{O}(1)$ . Koszt:  $\mathcal{O}(k^2)$ .
  - Linia 11, `HungarianAlgorithm(M_sub)`: Wywołanie Algorytmu Węgierskiego na macierzy  $k \times k$ . Koszt:  $\mathcal{O}(k^3)$ .
  - Linie 14-26, Pętla rozszerzenia: Zagnieżdżone pętle `for` wykonują  $k \times k$  iteracji. Wszystkie operacje wewnątrz (mapowanie, odczyt, zapis) mają koszt  $\mathcal{O}(1)$ . Koszt:  $\mathcal{O}(k^2)$ .

- **Koszt całkowity jednej iteracji:** Sumując powyższe kroki:

$$C_{iteracja} = \mathcal{O}(N^2 + k^2 + k^2 + k^3 + k^2) = \mathcal{O}(N^2 + k^3)$$

- **Koszt Całkowity Fazy 2:** Całkowity koszt to koszt inicjalizacji plus  $n$  razy koszt jednej iteracji.

$$C_{Faza\ 2} = \mathcal{O}(N^2) + \mathcal{O}(n \cdot (N^2 + k^3)) = \mathcal{O}(N^2 + n \cdot N^2 + n \cdot k^3)$$

**Całkowita złożoność obliczeniowa (Faza 2):**  $\mathcal{O}(n \cdot N^2 + n \cdot k^3)$ .



### 7.3 Końcowa Analiza Złożoności Całkowitej

Całkowity koszt algorytmu aproksymacyjnego v.4 to suma kosztów obu faz:

$$C_{\text{total}} = C_{\text{Faza 1}} + C_{\text{Faza 2}}$$

$$C_{\text{total}} = \mathcal{O}(nk) + \mathcal{O}(n \cdot N^2 + n \cdot k^3)$$

**Całkowita złożoność obliczeniowa algorytmu:**  $\mathcal{O}(nk) + \mathcal{O}(n \cdot N^2 + n \cdot k^3)$ .

#### Uwaga

Złożoność ta jest drastycznie niższa niż  $\mathcal{O}(J \cdot k \cdot N^3)$  poprzedniego algorytmu. Ceną za tę szybkość jest jakość heurystyki. Poprzedni algorytm (oparty na Murty'm) gwarantował znalezienie  $n$  najlepszych rozwiązań (w sensie heurystyki  $M$ ) spośród *wszystkich*  $\binom{N}{k}$  podzbiorów. Ten algorytm przeszukuje tylko  $n$  preselekcjonowanych podzbiorów; jeśli optymalne rozwiązanie leży poza nimi, algorytm ten go nie znajdzie.

## 8 Bibliografia