Wydział Matematyki
i Nauk Informacyjnych

POLITECHNIKA WARSZAWSKA

Teoria algorytmów i obliczeń

Projekt zaliczeniowy

Piotr Jacak
Jakub Kindracki
Wiktor Kobielski
Ernest Mołczan

Koordynator: prof. dr hab. inż. Władysław Homenda

Semestr zimowy 2025/2026

Spis treści

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 $f : V_1 \to V_2$, 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 $S(G) \leq S(G)$.

- **Przechodniość**:

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

  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 $S(G_2) \leq S(G_3)$.

  Załóżmy, 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, ze $|V_2| > |V_3|$, 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:

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

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 przejrzenie 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 $v \in mozliwe\_dopasowania[i]$, jeśli v *not in* $dopasowania$, przypisz $dopasowania[i] = v$ 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, $dopasowania[i] = null$ 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 warunku pierwszego, gdyby warunek nie był spełniony, nie istniałaby co najmniej jedna pętla da 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. **Kosztem 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, π), gdzie:

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

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

Para (C, π) 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, π) będzie osadzeniem k-wierzchołkowym P w G. **Zbiorem brakujących krawędzi** dla osadzenia (C, π) 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 $c[0] < c[1] < \ldots < c[k-1]$, 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 $[0, 1, \ldots, k-1]$
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 $[n-k, n-k+1, \ldots, n-1]$)

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

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

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 ← |items|
2: if n ≤ 1 then
3:     yield items
4:     return
5: end if
6: a ← kopia items                                  ▷ Praca na kopii
7: c ← [0, 0, ..., 0]                  ▷ Liczniki dla algorytmu Heapa, długość n
8: yield kopia a                                    ▷ Pierwsza permutacja
9: i ← 0
10: while i < n do
11:     if c[i] < i then
12:         if i mod 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] ← c[i] + 1
19:         i ← 0
20:     else
21:         c[i] ← 0
22:         i ← 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). □

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 $items^m = items \times items \times \cdots \times items$
- Generowanie w porządku leksykograficznym

Algorithm 3 ProductSequences($items$, m)

Require: $items$ - lista k elementów, m - długość sekwencji
Ensure: Wszystkie m-krotki elementów z $items$

```
k ← |items|
if m = 0 then
    yield [ ]
    return
end if
if k = 0 then
    return                                              ▷ Brak sekwencji
end if
digits ← [0, 0, ..., 0]                    ▷ Tablica m cyfr w systemie bazy k
while true do
    yield [items[digits[0]], items[digits[1]], ..., items[digits[m − 1]]]
    i ← 0                                  ▷ Inkrementacja licznika w systemie bazy k
    while i < m do
        if digits[i] + 1 < k then
            digits[i] ← digits[i] + 1
            break
        else
            digits[i] ← 0
            i ← i + 1
        end if
    end while
    if i = m then
        break          ▷ Przepełnienie - wszystkie sekwencje wygenerowane
    end if
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:

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

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

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

1. Rozpoczyna od $[0, 0, \ldots, 0]$ (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). □

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 $\{C_1, C_2, \ldots, C_N\}$, 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 $\pi_i : V_P \to V_P$ reprezentuje potencjalne uporządkowane mapowanie wierzchołków P na wierzchołki kombinacji C_j.

- *Wejście:* $P = (V_P, E_P)$, $|V_P| = k$
- *Wyjście:* Zbiór wszystkich permutacji $\{\pi_1, \pi_2, \ldots, \pi_M\}$, 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, π_i) obliczamy listę krawędzi, które należy dodać do G, aby podgraf indukowany przez C_j z mapowaniem π_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 Δ 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 $(i_1, i_2, \ldots, i_m)$ 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 $k_1, k_2, \ldots, k_m$, wystarczy dodać $\max(k_1, k_2, \ldots, k_m)$ kopii tej krawędzi, ponieważ krawędzie mogą być współdzielone między kopiami.

Algorytm:

1. Inicjalizujemy mapę częstości: `edgeFrequencyMap = {}`
2. Dla każdej z m kopii ($t = 1, \ldots, m$):
   (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:* $O\left(\binom{n}{k}_m \times (k!)^m \times m \times k^2\right)$

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

```
// Generuj k-kombinacje wierzchołków G
combinations ← GenerateCombinations(V_G, k)
indexToSubset ← IndexMap(combinations)
// Generuj permutacje wierzchołków P
permutations ← GeneratePermutations(V_P)
indexToPermutation ← IndexMap(permutations)
missingEdgesMatrix ← Array[|indexToPermutation|][|indexToSubset|]
for i ← 0 to |indexToPermutation| − 1 do
    for j ← 0 to |indexToSubset| − 1 do
        π ← indexToPermutation[i]; C ← indexToSubset[j]
        missingEdgesMatrix[i][j] ← [ ]
        for u ← 0 to k − 1 do
            for v ← 0 to k − 1 do
                u′ ← C[π[u]]; v′ ← C[π[v]]
                Δ ← max(0, A_P[π[u]][π[v]] − A_G[u′][v′])
                for t ← 0 to Δ − 1 do
                    missingEdgesMatrix[i][j].append((u′, v′))
                end for
            end for
        end for
    end for
end for
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 ← GenerateCombinations(indexToSubset.keys, m)
2: minimalEdges ← null; minimalSize ← ∞
3: for each {j_1, ..., j_m} in mCombinations do
4:     for each (i_1, ..., i_m) in ProductSeq(indexToPermutation.keys, m) do
5:         edgeFreqMap ← {}
6:         for t ← 0 to m − 1 do
7:             missingEdges ← missingEdgesMatrix[i_t][j_t]
8:             localFreq ← {}
9:             for each e in missingEdges do
10:                localFreq[e] ← localFreq[e] + 1
11:            end for
12:            for each (e, freq) in localFreq do
13:                edgeFreqMap[e] ← max(edgeFreqMap[e], freq)
14:            end for
15:        end for
16:        addedEdges ← [ ]
17:        for each (e, freq) in edgeFreqMap do
18:            for t ← 0 to freq − 1 do
19:                addedEdges.append(e)
20:            end for
21:        end for
22:        if |addedEdges| < minimalSize then
23:            minimalSize ← |addedEdges|; minimalEdges ← 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)
Załóżmy, 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 $C_t \subseteq V_G$ $(t = 1, \ldots, m)$
   - Permutacja $\pi_t : V_P \to V_P$
   - 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 $\{C_1, \ldots, C_m\}$ i krotkę permutacji $(\pi_1, \ldots, \pi_m)$ 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 $k_1, k_2, \ldots$, dodajemy $\max(k_1, k_2, \ldots)$ 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). □

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)

6.6.4 Heurystyki i algorytmy aproksymacyjne

1. Algorytm zachłanny (Greedy)
Idea: Zamiast rozważać wszystkie możliwe m-kombinacje osadzeń, wybieraj zachłannie następne najlepsze osadzenie.
Algorytm:

- Dla pierwszej kopii P: znajdź osadzenie wymagające najmniej krawędzi
- Dodaj te krawędzie do G
- Dla kolejnych kopii: znajdź osadzenie wymagające najmniej nowych krawędzi (biorąc pod uwagę już dodane)
- Powtórz m razy

Złożoność: $O\left(m \times \binom{n}{k} \times k! \times k^2\right)$
Jakość: Nie gwarantuje optymalności, ale może dać dobre przybliżenie w krótszym czasie.
2. Algorytm genetyczny
Idea: Ewolucyjne poszukiwanie dobrego rozwiązania.
Komponenty:

- **Populacja:** Zbiór kandydatów rozwiązań (m-krotki osadzeń)

- **Funkcja przystosowania:** Liczba krawędzi do dodania (minimalizowana)
- **Operatory:** Krzyżowanie (wymiana osadzeń między rozwiązaniami), mutacja (losowa zmiana osadzenia)
- **Selekcja:** Wybór najlepszych rozwiązań do następnego pokolenia

Zalety: Możliwość znalezienia dobrych rozwiązań dla dużych instancji.
Wady: Brak gwarancji optymalności, wymaga tuningu parametrów.

3. Symulowane wyżarzanie (Simulated Annealing)

Idea: Iteracyjne ulepszanie rozwiązania z możliwością akceptacji gorszych rozwiązań.

Algorytm:

- Start: losowa m-krotka osadzeń
- Iteracyjnie: modyfikuj losowo jedno osadzenie
- Akceptuj jeśli poprawia rozwiązanie lub z prawdopodobieństwem zależnym od "temperatury"
- "Temperatura"maleje z czasem, redukując akceptację gorszych rozwiązań

Zalety: Pozwala na wyjście z lokalnych minimów.

4. Programowanie całkowitoliczbowe (ILP)

Idea: Sformułuj problem jako program całkowitoliczbowy.
Zmienne:

- $x_{ij} \in \{0, 1\}$ - czy osadzenie i jest wybrane dla kopii j
- $y_e \in \mathbb{N}_0$ - liczba kopii krawędzi e do dodania

Ograniczenia:

- Każda kopia musi mieć dokładnie jedno osadzenie: $\sum_i x_{ij} = 1$ dla każdego j
- Osadzenia muszą być różne (odpowiadać różnym kombinacjom wierzchołków)
- Dla każdej krawędzi e: $y_e \geq \max_j\{\Delta_{ij}(e) \cdot x_{ij}\}$ gdzie $\Delta_{ij}(e)$ to krotność e w brakujących krawędziach osadzenia i dla kopii j

Cel: Minimalizuj $\sum_e y_e$
Zalety: Optymalne rozwiązanie (jeśli solver zakończy się w rozsądnym czasie).
Wady: Może być wolne dla dużych instancji.

6.7 Podsumowanie

W niniejszej sekcji przedstawiono kompleksowe podejście do problemu minimalnego rozszerzenia multigrafu zawierającego m kopii podgrafu wzorcowego:

- **Sformułowano problem** i uzasadniono jego praktyczne znaczenie
- **Zdefiniowano formalnie** kluczowe pojęcia: rozszerzenie, koszt rozszerzenia, osadzenie k-wierzchołkowe, brakujące krawędzie, minimalne rozszerzenie zawierające m kopii
- **Opracowano algorytm dokładny** oparty na pełnym przeszukiwaniu przestrzeni rozwiązań
- **Udowodniono poprawność** algorytmu (kompletność, poprawność, minimalność)
- **Przeprowadzono szczegółową analizę złożoności**:
  - Złożoność czasowa: $O\left(\binom{n}{k} \times k! \times k^2 + \binom{\binom{n}{k}}{m} \times (k!)^m \times m \times k^2\right)$
  - Złożoność pamięciowa: $O\left(\binom{n}{k} \times k! \times k^2\right)$
  - Problem jest NP-trudny
- **Zaprezentowano przykład działania** algorytmu na konkretnych danych
- **Zaproponowano optymalizacje** implementacyjne i algorytmy aproksymacyjne (zachłanny, genetyczny, simulated annealing, ILP)

Algorytm dokładny jest praktyczny dla małych wartości k ($k \leq 6 - 7$) i $n \leq 20$. Dla większych instancji zaleca się stosowanie algorytmów aproksymacyjnych lub heurystyk.

6.8 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.

6.8.1 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 - $K[i, j] = max_{k=1,...,m}(C_k[i, j])$, gdzie C_k dla $k = 1, ..., m$ to m najlepszych macierzy kosztu.

7. Zwróć macierz K reprezentującą minimalne rozszerzenie.

6.8.2 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. Czynnik $|V_1|\cdot|E_1|\cdot|E_2|$ to koszt zachłannego przeszukiwania grafu w celu minimalizacji kosztu. m to liczba wybieranych macierzy kosztu (liczba kopii). Opisany algorytm ma zatem złożoność wielomianową.

6.8.3 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.

6.9 Aproksymacyjne minimalne rozszerzenie multigrafu - algorytm drugi

Algorytm implementuje metodę Murty'ego do znajdowania j najlepszych (o najniższym koszcie) rozwiązań problemu przypisania, z kluczową modyfikacją: problem dotyczy przypisania k wierszy "wzorca"do N kolumn "grafu"($k \leq N$), modelowanego jako problem $N \times N$, gdzie $N - k$ wierszy to "atrapy".

6.9.1 Notacja i Definicje

Definicja 16 (Problem Przypisania)**.** Dany jest graf dwudzielny z wierzchołkami $U = \{u_1, \ldots, u_N\}$ (wiersze) i $V = \{v_1, \ldots, v_N\}$ (kolumny) oraz macierz kosztów M o wymiarach $N \times N$. Celem jest znalezienie permutacji σ zbioru $\{1, \ldots, N\}$ minimalizującej całkowity koszt:

$$C(\sigma) = \sum_{i=1}^{N} M[i, \sigma(i)]$$

Definicja 17 (Problem k-Mapowania)**.** W naszym przypadku interesuje nas tylko przypisanie pierwszych k wierszy (reprezentujących P). Mapowanie f jest iniekcją $f : \{1, \ldots, k\} \to \{1, \ldots, N\}$. Algorytm `linear_sum_assignment` znajduje pełną permutację σ, z której my ekstrahujemy f jako:

$$f(i) = \sigma(i) \quad \text{dla } i \in \{1, \ldots, k\}$$

Zbiór wierszy "atrap"to $D = \{k + 1, \ldots, N\}$. Ich przypisania są ignorowane.

Definicja 18 (Problem z Ograniczeniami)**.** Definiujemy $H(M')$ jako funkcję rozwiązującą problem przypisania (np. algorytm węgierski) na macierzy M'. Rozwiązanie podproblemu z ograniczeniami jest realizowane przez modyfikację macierzy kosztów:

- $I \subset \{1..k\} \times \{1..N\}$: zbiór "wymuszonych"przypisań (includes).
- $E \subset \{1..k\} \times \{1..N\}$: zbiór "zabronionych"przypisań (excludes).

Funkcja `solve_with_constraints(I, E)` rozwiązuje problem $H(M')$ na zmodyfikowanej macierzy M', gdzie:

$$M'[i,j] = \begin{cases} -\infty & \text{jeśli } (i,j) \in I \\ +\infty & \text{jeśli } (i,j) \notin I \text{ oraz } \exists j' : (i,j') \in I \\ +\infty & \text{jeśli } (i,j) \in E \\ M[i,j] & \text{w przeciwnym razie} \end{cases}$$

Funkcja zwraca $(\text{Koszt}(f), f)$ używając oryginalnej macierzy M do obliczenia kosztu, lub $(\text{None}, \text{None})$, jeśli ograniczenia I nie zostały spełnione.

6.9.2 Heurystyczna Funkcja Kosztu

Algorytm Murty'ego (opisany poniżej) operuje na macierzy kosztów M, która stanowi heurystyczną, "liniową"aproksymację rzeczywistego, "kwadratowego"kosztu rozszerzenia. Wybór tej heurystyki ma kluczowy wpływ na jakość znajdowanych rozwiązań – lepsza heurystyka (bliższa rzeczywistemu kosztowi) sprawi, że j najlepszych rozwiązań heurystycznych będzie z większym prawdopodobieństwem odpowiadać j najlepszym rozwiązaniom rzeczywistym.

Definicja 19 (Heurystyka Różnicy Stopni)**.** Podstawową i szybką heurystyką jest dopasowanie wierzchołków P i G na podstawie ich łącznej konektywności. Definiujemy koszt $M[i,j]$ (dla $i \leq k$) jako absolutną różnicę stopni całkowitych (suma stopni wejściowych i wychodzących):

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

gdzie $p_i \in V_P$ oraz $g_j \in V_G$.

Złożoność Obliczeniowa Heurystyki Koszt zbudowania macierzy M przy użyciu tej heurystyki składa się z dwóch części:

1. Obliczenie wszystkich stopni dla P: Wymaga to zsumowania macierzy A_P, co ma koszt $\mathcal{O}(k^2)$.

2. Obliczenie wszystkich stopni dla G: Wymaga zsumowania macierzy A_G, co ma koszt $\mathcal{O}(N^2)$.

3. Wypełnienie k pierwszych wierszy macierzy M: Wymaga $k \cdot N$ operacji (pobranie i odjęcie obliczonych wcześniej stopni).

Całkowity koszt przygotowania macierzy heurystycznej M jest zdominowany przez obliczenie stopni w grafie G, a zatem wynosi $\mathcal{O}(N^2)$. Koszt ten jest ponoszony jednokrotnie przed uruchomieniem procedury solvera.

Walidacja Heurystyka ta jest prosta i szybka. Bardziej zaawansowane (i potencjalnie dokładniejsze) funkcje heurystyczne, np. uwzględniające stopnie sąsiadów lub ważące osobno stopnie wejściowe i wyjściowe, mogą zostać opracowane i zweryfikowane w fazie testów empirycznych algorytmu.

6.9.3 Algorytm Główny (`compute_k_best`)

Algorytm wykorzystuje kolejkę priorytetową (min-heap) Q do przechowywania kandydatów na rozwiązania. Każdy element w Q to krotka: (C, I, E, f), gdzie C to koszt mapowania f, a I i E to zbiory ograniczeń, które wygenerowały to rozwiązanie.

1. **Inicjalizacja:**

   - Stwórz pustą listę rozwiązań L_{sol}.
   - Stwórz pusty zbiór "widzianych"mapowań U_{seen}.
   - Stwórz pustą kolejkę priorytetową Q.
   - Rozwiąż problem bez ograniczeń: $(C_1, f_1) \leftarrow$ `solve_with_constraints`$(\emptyset, \emptyset)$.
   - **if** $C_1 \neq$ None **then** Q.push$((C_1, \emptyset, \emptyset, f_1))$.

2. **Pętla główna:**

3. **while** Q nie jest pusta **and** $|L_{sol}| <$ max_solutions:

   (a) $(C, I, E, f) \leftarrow Q$.pop().

(b) **if** $f \in U_{seen}$ **then continue** (pomiń duplikat).

(c) U_{seen}.add(f).

(d) L_{sol}.append((C, f)).

(e) **Partycjonowanie (Rozgałęzienie):**

(f) $I_{accum} \leftarrow I$ (akumulator wymuszonych krawędzi dla tej gałęzi).

(g) **for** $i = 1$ **to** k:

- $(r, c) \leftarrow (i, f(i))$ (krawędź do partycjonowania).
- $E_{new} \leftarrow E \cup \{(r, c)\}$ (nowy zbiór zabronionych).
- *// Generowanie podproblemu P1: wymuś (1..i-1), zabroń (i)*
- $(C_{new}, f_{new}) \leftarrow$ `solve_with_constraints`(I_{accum}, E_{new}).
- **if** $C_{new} \neq$ None **then** Q.push($(C_{new}, I_{accum}, E_{new}, f_{new})$).
- *// Przygotowanie do następnej iteracji: wymuś (i)*
- $I_{accum} \leftarrow I_{accum} \cup \{(r, c)\}$.

4. **Zakończenie:** Zwróć L_{sol}.

6.9.4 Dowód Poprawności

Twierdzenie 2 (Zakończenie Algorytmu)**.** *Algorytm* `compute_k_best` *zawsze kończy działanie w skończonym czasie.*

Dowód. Liczba możliwych mapowań f (iniekcji z $\{1..k\}$ do $\{1..N\}$) jest skończona i wynosi $P(N, k) = N!/(N - k)!$. Zbiór `seen_mappings` (U_{seen}) gwarantuje, że każde unikalne mapowanie f jest dodawane do L_{sol} i partycjonowane co najwyżej raz. Ponieważ pętla `while` jest również ograniczona przez `max_solutions`, algorytm musi się zakończyć. □

Twierdzenie 3 (Kompletność i Optymalność)**.** *Algorytm generuje rozwiązania w kolejności niemalejącego kosztu. Jeśli istnieje j unikalnych rozwiązań, algorytm znajdzie je w j pierwszych (unikalnych) iteracjach pętli.*

Dowód. Dowód opiera się na poprawności schematu partycjonowania Murty'ego. Niech S będzie zbiorem wszystkich dozwolonych mapowań (rozwiązań). Niech f_1 będzie optymalnym rozwiązaniem (1-szym najlepszym). W kroku 2.c.i, algorytm generuje k podproblemów $P_1, \ldots, P_k$ na podstawie f_1. □

Lemat 4 (Partycjonowanie Murty'ego)**.** *Zbiory rozwiązań dla podproblemów $P_1, \ldots, P_k$ są parami rozłączne, a ich suma (unia) jest równa $S \setminus \{f_1\}$.*

Dowód (Szkic). Niech $f_1 = \{(1, c_1), \ldots, (k, c_k)\}$.

- P_1 zawiera rozwiązania, które *nie* mają $(1, c_1)$.
- P_2 zawiera rozwiązania, które *mają* $(1, c_1)$, ale *nie* mają $(2, c_2)$.
- P_i zawiera rozwiązania, które *mają* $\{(1, c_1), \ldots, (i-1, c_{i-1})\}$, ale *nie* mają (i, c_i).

Są one z definicji rozłączne. Weźmy dowolne rozwiązanie $f' \in S \setminus \{f_1\}$. Musi ono różnić się od f_1 na co najmniej jednej pozycji. Niech i będzie pierwszym indeksem (wierszem), gdzie $f'(i) \neq c_i$. Oznacza to, że f' pasuje do f_1 na pozycjach $1, \ldots, i-1$, ale nie na i. Zatem f' należy do zbioru P_i. W ten sposób $\bigcup_{i=1}^{k} P_i = S \setminus \{f_1\}$. □

Ponieważ (1) partycjonowanie jest kompletne i rozłączne, (2) optymalne rozwiązanie każdego podproblemu P_i (czyli $H(P_i)$) jest najlepszym kandydatem z tego podzbioru, oraz (3) kolejka priorytetowa Q zawsze przechowuje najlepszych kandydatów ze wszystkich dotychczas wygenerowanych podzbiorów, to gdy `Q.pop()` zwraca f_j, musi to być rozwiązanie o globalnie j-tym najniższym koszcie spośród wszystkich możliwych rozwiązań.

7 Analiza Złożoności

Analizujemy koszt znalezienia j najlepszych rozwiązań.

7.0.1 Złożoność Pamięciowa

- Macierz kosztów `cost_matrix` (oryginalna i kopie): $\mathcal{O}(N^2)$.
- Lista rozwiązań `solutions`: Przechowuje j mapowań, każde o rozmiarze k. Koszt: $\mathcal{O}(jk)$.
- Zbiór `seen_mappings`: Przechowuje j krotek o rozmiarze k. Koszt: $\mathcal{O}(jk)$.
- Kolejka priorytetowa `queue`: W najgorszym razie, każde z j znalezionych rozwiązań generuje k nowych kandydatów. Rozmiar kolejki jest rzędu $\mathcal{O}(jk)$. Każdy element przechowuje ograniczenia I i E, które mogą rosnąć do $\mathcal{O}(k)$. Koszt kolejki: $\mathcal{O}(jk \cdot k) = \mathcal{O}(jk^2)$.

Całkowita złożoność pamięciowa: $\mathcal{O}(N^2 + jk^2)$.

7.0.2 Złożoność Obliczeniowa (Czasowa)

- Główna pętla `while` wykonuje się $\mathcal{O}(j)$ razy (ignorując duplikaty, których w praktyce jest niewiele w porównaniu do j).
- Wewnątrz pętli, pętla `for` (partycjonowanie) wykonuje się k razy.
- Wewnątrz pętli `for` wywoływana jest funkcja `solve_with_constraints`.
- Koszt `solve_with_constraints` jest zdominowany przez `linear_sum_assignment`, którego złożoność dla macierzy $N \times N$ wynosi $\mathcal{O}(N^3)$.
- Operacje na kolejce priorytetowej (`push`/`pop`) mają koszt $\mathcal{O}(\log |Q|) = \mathcal{O}(\log(jk))$.

Całkowity koszt to suma kosztów $\mathcal{O}(j)$ kroków "wyjęcia"rozwiązania i $\mathcal{O}(j \cdot k)$ kroków "rozwiązania podproblemu".

$$C_{\text{total}} = \underbrace{\mathcal{O}(j \cdot \log(jk))}_{\text{Zarządzanie kolejką}} + \underbrace{\mathcal{O}(j \cdot k \cdot N^3)}_{\text{Rozwiązywanie podproblemów}}$$

Ponieważ $\mathcal{O}(N^3)$ jest znacznie większe niż $\mathcal{O}(\log(jk))$, ten drugi człon jest pomijalny. **Całkowita złożoność obliczeniowa: $\mathcal{O}(j \cdot k \cdot N^3)$.**

7.0.3 Krok 2: Zastosowanie Rozszerzeń i Końcowa Analiza Złożoności

Powyższa analiza dotyczy kosztu procedury-solvera, która znajduje n najlepszych mapowań heurystycznych (ustawiając $j = n$). Pełen algorytm aproksymacyjny składa się z dwóch głównych faz:

1. **Faza 1 (Znalezienie mapowań):** Wywołanie procedury `compute_k_best` z $j = n$. Koszt tej operacji, jak wykazano powyżej, wynosi:

   $$C_{\text{solver}} = \mathcal{O}(n \cdot k \cdot N^3)$$

   Wynikiem jest lista $L_{sol} = \{f_1, \ldots, f_n\}$ zawierająca n unikalnych mapowań.

2. **Faza 2 (Zastosowanie rozszerzeń):** Po uzyskaniu listy L_{sol}, algorytm musi zastosować je do grafu A_G, aby utworzyć końcowe rozszerzenie A_{final}. Procedura ta iteruje n razy (raz dla każdego mapowania $f_i \in L_{sol}$). Wewnątrz każdej iteracji, sprawdza k^2 potencjalnych krawędzi (z A_P) i aktualizuje macierz A_{final} (inicjalizowaną jako A_G), zliczając przy tym dodane krawędzie.

Definicja 20 (Rzeczywisty Koszt Rozszerzenia C_{ext})**.** Koszt ten, w przeciwieństwie do heurystyki liniowej, jest kosztem "kwadratowym"i reprezentuje faktyczną liczbę krawędzi do dodania.

$$C_{ext}(f, A) = \sum_{u,v \in V_P} \max(0, A_P[u, v] - A[f(u), f(v)])$$

Koszt Fazy 2 (zastosowania rozszerzeń) jest następujący:

$$C_{\text{rozszerzenie}} = \sum_{i=1}^{n} (\text{koszt obliczenia } C_{ext}(f_i, A_{curr}) \text{ i aktualizacji } A_{curr})$$

Koszt obliczenia C_{ext} i aktualizacji macierzy dla jednego mapowania f_i wymaga iteracji przez $k \times k$ par wierzchołków P, a więc wynosi $\mathcal{O}(k^2)$. Całkowity koszt Fazy 2 to:

$$C_{\text{rozszerzenie}} = \sum_{i=1}^{n} \mathcal{O}(k^2) = \mathcal{O}(n \cdot k^2)$$

Całkowity koszt algorytmu to suma kosztów obu faz:

$$C_{\text{total}} = C_{\text{solver}} + C_{\text{rozszerzenie}}$$

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

Ponieważ $N \geq k$, złożoność $\mathcal{O}(N^3)$ jest zawsze asymptotycznie większa lub równa $\mathcal{O}(k^2)$. W związku z tym, dominującym członem jest koszt Fazy 1 (znalezienia mapowań).

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