Kamil Połacik

Zadanie 9. Cykle Hamiltona *(***)*

Zaprojektuj algorytm, który dla danego grafu skierowanego *G*, wypisze wszystkie cykle hamiltonowskie w *G*. Algorytm powinien działać w czasie *O*(*n*!), gdzie *n* jest liczbę wierzchołków w grafie *G*.

Rysunek 1. Treść zadania 9. Cykle Hamiltona (***)

Wstęp teoretyczny

Graf [1] to struktura matematyczna składająca się z dwóch zbiorów: zbioru wierzchołków V oraz zbioru krawędzi E, które są parami wierzchołków. W grafie skierowanym (ang. *directed graph, digraph*) krawędzie posiadają kierunek - para (u,v) oznacza krawędź prowadzącą od wierzchołka u do v.

Na potrzeby zadania graf reprezentowany jest za pomocą macierzy sąsiedztwa - kwadratowej macierzy $n \times n$, gdzie wartość 1 w komórce (i,j) oznacza istnienie krawędzi skierowanej od i do j.

Cykl Hamiltona [2] (ang. *Hamiltonian cycle*) to cykl prosty w grafie, który przechodzi przez każdy wierzchołek dokładnie jeden raz, a następnie wraca do wierzchołka początkowego. Formalne warunki dla cyklu Hamiltona:

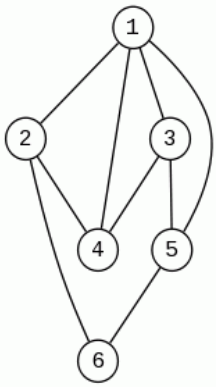
- Cykl zawiera dokładnie n różnych wierzchołków (gdzie $\mathrm{n}=|\mathrm{V}|$),
- Każdy z tych wierzchołków występuje w cyklu dokładnie raz,
- Cykl kończy się w wierzchołku, z którego się rozpoczął.

Zadanie polega na odnalezieniu wszystkich cykli Hamiltona w zadanym grafie skierowanym. Problem ten ma znaczenie teoretyczne i praktyczne — występuje m.in. w optymalizacji tras (np. w problemie komiwojażera).

Problem znajdowania cyklu Hamiltona należy do klasy problemów NP-trudnych. Oznacza to, że:

- Nie jest znany algorytm działający w czasie wielomianowym dla dowolnego grafu,
- W najgorszym przypadku liczba możliwych permutacji wierzchołków wynosi $(\mathrm{n}-1)!$, ponieważ można dowolnie wybrać wierzchołek startowy, a pozostałe $\mathrm{n}-1$ permutować.

W zaprojektowanym algorytmie używamy metody backtrackingu, która w najgorszym przypadku przeszukuje przestrzeń rozmiaru $\mathrm{O}(\mathrm{n}!)$, więc algorytm działa w czasie wykładniczym względem liczby wierzchołków.



Rysunek 2. Graf nieskierowany

Przykładowy cykl: 1,2,6,5,3,4,1

Opis algorytmu

Algorytm przeszukuje wszystkie możliwe ścieżki w grafie, które mogą tworzyć cykl Hamiltona. Działa rekurencyjnie metodą *backtrackingu*, próbując zbudować ścieżkę, która odwiedza każdy wierzchołek dokładnie raz i wraca do punktu początkowego.

1. Dla każdego wierzchołka v w grafie:
    - Ustaw v jako wierzchołek startowy.
    - Rozpocznij rekurencyjne przeszukiwanie, budując ścieżkę zawierającą odwiedzone wierzchołki.
    - Na każdym etapie próbuj przejść do każdego nieodwiedzonego sąsiada, dla którego istnieje krawędź z bieżącego wierzchołka.
    - Jeśli ścieżka zawiera już wszystkie n wierzchołki, sprawdź, czy istnieje krawędź z ostatniego wierzchołka do pierwszego. Jeśli tak — cykl Hamiltona został znaleziony.
2. Jeśli cykl spełnia warunki — zapisz go do listy rozwiązań.

Warunki:

- Każdy wierzchołek może wystąpić tylko raz w danej ścieżce.
- Krawędzie muszą istnieć (czyli graf G musi mieć skierowaną krawędź).
- Na końcu musi istnieć krawędź z ostatniego wierzchołka do pierwszego, by zakończyć cykl.

Krok po kroku

1. Wybieramy wierzchołek początkowy – zaczynamy od jednego z wierzchołków w grafie.

2. Tworzymy pustą ścieżkę (listę odwiedzonych wierzchołków) i dodajemy do niej wierzchołek startowy.

3. Z aktualnego wierzchołka próbujemy przejść do każdego innego wierzchołka, który:
    - nie został jeszcze odwiedzony,
    - oraz do którego istnieje skierowana krawędź.

4. Jeśli takie przejście jest możliwe, dodajemy nowy wierzchołek do ścieżki i rekurencyjnie kontynuujemy budowanie cyklu.

5. Gdy ścieżka zawiera wszystkie wierzchołki:

    - Sprawdzamy, czy istnieje powrót do wierzchołka startowego.
    - Jeśli tak — mamy pełny cykl Hamiltona i zapisujemy go jako rozwiązanie.

6. Cofamy się (*backtracking*) — usuwamy ostatni wierzchołek ze ścieżki i próbujemy inne możliwości. Dzięki temu algorytm przeszukuje wszystkie potencjalne drogi prowadzące do cyklu Hamiltona.

Pseudokod

```
1.  FUNKCJA znajdz_cykle_hamiltona(graf):
2.      n ← liczba wierzchołków w grafie
3.      cykle ← pusta lista                    // zapisuje cykle Hamiltona
4.        FUNKCJA backtrack(ścieżka, odwiedzone):
5.                obecny ← ostatni wierzchołek w ścieżce
6.                JEŚLI długość(ścieżka) = n:
7.                      JEŚLI istnieje krawędź z obecny do ścieżka[0]:  // sprawdzamy, czy cykl się domyka
8.                         dodaj (ścieżka + [ścieżka[0]]) do cykle
9.                      ZAKOŃCZ (return)
10.               DLA sąsiad od 0 DO n - 1:
11.                 JEŚLI istnieje krawędź z obecny do sąsiad ORAZ sąsiad NIE został odwiedzony:
12.                      oznacz sąsiad jako odwiedzony
13.                      wywołaj backtrack(ścieżka + [sąsiad], odwiedzone)
14.                      oznacz sąsiad jako NIE odwiedzony (cofnięcie – backtracking)
// Uruchamiamy backtracking dla każdego wierzchołka jako punktu startowego
15.     DLA startowy_wierzchołek od 0 DO n - 1:
16.               odwiedzone ← tablica n elementów o wartości FALSE
17.               oznacz startowy_wierzchołek jako odwiedzony
18.               wywołaj backtrack([startowy_wierzchołek], odwiedzone)
19.     ZWRÓĆ cykle
```

Opis algorytmu

Algorytm rozpoczyna swoje działanie od przyjęcia grafu w postaci macierzy sąsiedztwa. Na tej podstawie ustala, ile wierzchołków zawiera graf, ponieważ liczba ta będzie kluczowa do sprawdzania, czy dana ścieżka zawiera już wszystkie wierzchołki i potencjalnie może być cyklem Hamiltona. Następnie tworzona jest pusta lista, do której będą dodawane wszystkie znalezione cykle.

Wewnątrz głównej funkcji znajduje się funkcja pomocnicza o nazwie backtrack, która odpowiada za budowanie ścieżek w grafie metodą rekurencyjną. Funkcja ta przyjmuje jako argument bieżącą ścieżkę (czyli listę wierzchołków, które już zostały odwiedzone) oraz tablicę informującą, które wierzchołki już zostały odwiedzone. Na początku każdej rekurencji sprawdzany jest ostatni wierzchołek ścieżki, ponieważ z niego będziemy próbować kontynuować dalszą drogę.

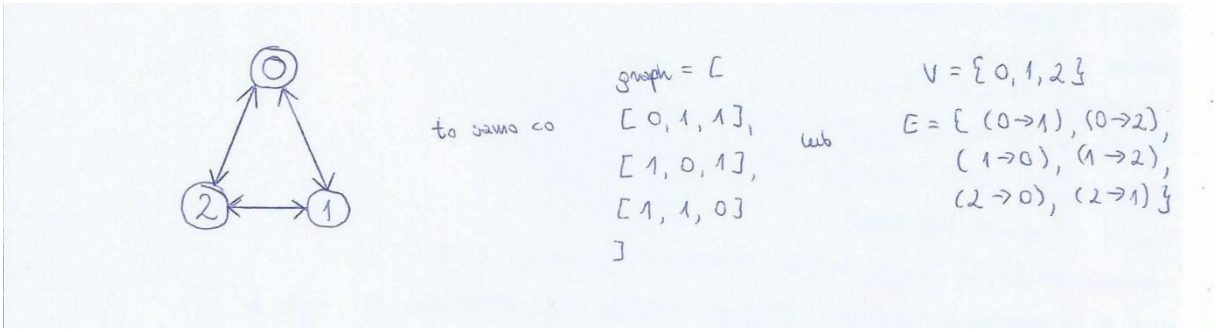
Jeśli okaże się, że długość ścieżki osiągnęła wartość równą liczbie wierzchołków w grafie oznacza to, że odwiedziliśmy każdy wierzchołek dokładnie jeden raz. Wtedy należy sprawdzić, czy istnieje krawędź z ostatniego odwiedzonego wierzchołka do wierzchołka początkowego - ponieważ tylko w takim przypadku ścieżka może zostać zamknięta w cykl. Jeśli taka krawędź istnieje, dodajemy ścieżkę (z dołączonym powrotem do początku) do listy cykli Hamiltona. Jeśli nie - przerywamy bieżącą rekurencję i nie kontynuujemy tej gałęzi przeszukiwania.

Jeżeli długość bieżącej ścieżki nie osiągnęła jeszcze liczby wszystkich wierzchołków, funkcja backtrack kontynuuje eksplorację grafu, próbując przejść do każdego z możliwych sąsiadów aktualnego wierzchołka. Każdy sąsiad jest rozważany tylko wtedy, gdy istnieje bezpośrednia krawędź prowadząca do niego oraz nie został jeszcze odwiedzony w bieżącej ścieżce. Jeśli te warunki są spełnione, oznaczamy wierzchołek jako odwiedzony i rekurencyjnie wywołujemy backtrack, przekazując nową, rozszerzoną ścieżkę oraz zaktualizowaną listę odwiedzonych wierzchołków. Po zakończeniu tej rekurencji wykonujemy tzw. „cofnięcie” (ang. *backtracking*) — czyli oznaczamy bieżący wierzchołek jako nieodwiedzony i usuwamy go z aktualnej ścieżki. Dzięki temu algorytm może eksplorować inne możliwe ścieżki, które również przechodzą przez ten wierzchołek, ale w innej kolejności.

Na zakończenie główna funkcja uruchamia backtrack osobno dla każdego wierzchołka jako punktu startowego. Oznacza to, że próbujemy znaleźć wszystkie możliwe cykle Hamiltona zaczynające się w różnych miejscach, co zapewnia kompletność przeszukiwania. Dla każdego z tych punktów tworzona jest nowa tablica odwiedzin, gdzie tylko punkt startowy jest zaznaczony jako odwiedzony, a pozostałe nie. Następnie algorytm uruchamia funkcję backtrack, przekazując jako początkową ścieżkę listę zawierającą tylko ten jeden wierzchołek startowy.

Gdy cały proces dobiegnie końca, funkcja główna zwraca listę cykli Hamiltona, które udało się odnaleźć. Każdy z tych cykli jest reprezentowany jako lista wierzchołków, w której pierwszy i ostatni element są takie same — co oznacza, że cykl został poprawnie zamknięty.

Działanie algorytmu krok po kroku



Rysunek 3. Wykorzystywany graf do przykładu oraz jego możliwe formy

Przebieg dla startowy_wierzchołek = 0
Krok 1:
Ścieżka: [0]
Odwiedzone: [True, False, False]
Szukamy sąsiadów 0 → ma połączenia do 1 i 2

Krok 2: Idziemy do 1
Ścieżka: [0, 1]
Odwiedzone: [True, True, False]
Sąsiedzi 1 → może iść do 0 (odwiedzony), 2

Krok 3: Idziemy do 2
Ścieżka: [0, 1, 2]
Odwiedzone: [True, True, True]
Długość = 3 → sprawdzamy, czy 2 → 0
Istnieje → cykl: [0, 1, 2, 0]

Cofanie:
Cofamy się do [0, 1] wszyscy sąsiedzi odwiedzeni więc kolejny powrót do [0]

Krok 4: Z [0] teraz do 2
Ścieżka: [0, 2]
Odwiedzone: [True, False, True]
Sąsiedzi 2 → może iść do 0 (odwiedzony), 1

Krok 5: Do 1
Ścieżka: [0, 2, 1]
Odwiedzone: [True, True, True]
1 → 0 istnieje → cykl: [0, 2, 1, 0]

Z start = 0 mamy:
[0, 1, 2, 0]
[0, 2, 1, 0]

Dla wierzchołka startowego [0] zostały sprawdzone wszystkie możliwe ścieżki, zmiana wierzchołka startowego na [1] i ponowne uruchomienie algorytmu.

Przebieg dla startowy_wierzchołek = 1
[1] → 0 → 2 → powrót do 1
[1, 0, 2, 1]
[1] → 2 → 0 → powrót do 1
[1, 2, 0, 1]

Przebieg dla startowy_wierzchołek = 2
[2] → 0 → 1 → powrót do 2
[2, 0, 1, 2]
[2] → 1 → 0 → powrót do 2
[2, 1, 0, 2]

[0, 1, 2, 0]
[0, 2, 1, 0]
[1, 0, 2, 1]
[1, 2, 0, 1]
[2, 0, 1, 2]
[2, 1, 0, 2]

Razem: 6 cykli Hamiltona, co odpowiada $3!$ permutacjom — każda z nich tworzy cykl.

Algorytm opiera się na rekurencyjnym przeszukiwaniu wszystkich możliwych ścieżek, które:

- startują z każdego wierzchołka (zostanie uruchomiony na każdym wierzchołku niezależnie od tego jaki był startowy)
- odwiedzają wszystkie inne
- sprawdza, czy ostatni wierzchołek łączy się ze startowym.

Dla n wierzchołków liczba możliwych permutacji długości n wynosi $n! = n * (n-1) * (n-2) * ... * 1$.
W związku z tym liczba operacji rośnie bardzo szybko. Dla podanego przykładu 3 wierzchołków, wszystkie możliwe przejścia to $3! = 3 * 2 * 1 = 6$ (w przykładzie wszystkie są cyklami Hamiltona – najgorszy przypadek, algorytm musi sprawdzić wszystkie możliwe permutacje, z których każda tworzy cykl Hamiltona). Dla przykładu szybkości wzrostu ilość obliczeń dla $5!$ jest to 120, a dla $10!$ już 3628800.

Złożoność obliczeniowa

Aby utworzyć cykl Hamiltona, musimy przejść przez każdy z n wierzchołków dokładnie raz i wrócić do punktu startowego. Oznacza to, że szukamy permutacji wierzchołków, w których każdy wierzchołek występuje dokładnie raz przy czym, aby ścieżka utworzyła cykl Hamiltona, ostatni wierzchołek musi posiadać krawędź prowadzącą do pierwszego. Liczba możliwych permutacji wierzchołków to $n!$, ponieważ:

- Dla pierwszego miejsca mamy do wyboru n wierzchołków,
- dla drugiego $n - 1$,
- dla trzeciego $n - 2$ itd.,
- aż do ostatniego (1 możliwość),
- czyli: $n \times (n-1) \times (n-2) \times ... \times 1 = n!$

Jeśli graf jest pełny (każdy wierzchołek ma krawędź do każdego innego), to każda permutacja wierzchołków jest potencjalnym cyklem Hamiltona, np. dla $n=4$ mamy $4! = 24$ możliwe ścieżki.

Cykl Hamiltona nie może odwiedzać żadnego wierzchołka więcej niż raz. Aby ścieżka mogła zostać uznana za cykl Hamiltona, ostatni wierzchołek musi posiadać krawędź prowadzącą do pierwszego, zamykając tym samym cykl. Algorytm backtrackingu przeszukuje wszystkie możliwe ścieżki rekurencyjnie, rozgałęziając się w każdym kroku do wszystkich sąsiadów, którzy nie zostali jeszcze odwiedzeni. W najgorszym przypadku — gdy graf jest pełny (każdy wierzchołek jest połączony z każdym innym) — liczba potencjalnych ścieżek do sprawdzenia odpowiada wszystkim permutacjom wierzchołków, czyli $n!$. Dodatkowo, przy każdym rozszerzeniu ścieżki sprawdzamy, czy przejście do kolejnego wierzchołka jest możliwe (czy istnieje odpowiednia krawędź), co wykonuje się w czasie stałym $O(1)$. Mimo to, głównym czynnikiem wpływającym na złożoność algorytmu pozostaje ogromna liczba możliwych permutacji — czyli rozmiar przestrzeni przeszukiwań.

Algorytm działa w czasie $O(n!)$, ponieważ próbuje przejść przez wszystkie możliwe permutacje wierzchołków. Optymalizacje (takie jak kończenie ścieżki wcześniej, jeśli nie da się jej domknąć) pomagają skrócić czas działania, ale w najgorszym przypadku nie da się uniknąć eksploracji przestrzeni o rozmiarze $n!$. Problem cyklu Hamiltona jest NP-trudny, dlatego żadna znana metoda nie potrafi go rozwiązać w czasie wielomianowym (dla ogólnego grafu), co czyni $O(n!)$ najlepszym oszacowaniem dla takiego pełnego przeszukiwania.

Podsumowanie

Złożoność dla każdego przykładu zależy od liczby połączeń w grafie. Liczba realnie przeszukanych ścieżek w grafie może być mniejsza, ponieważ niektóre permutacje są wcześnie odrzucane z powodu braku krawędzi lub tylko jedna pełna ścieżka prowadzi do cyklu Hamiltona. W najgorszym możliwym przypadku liczba ścieżek do sprawdzenia to $n!$, ponieważ wszystkie ścieżki muszą zostać sprawdzone. W praktyce, im mniej połączeń w grafie, tym szybciej algorytm może zakończyć przeszukiwanie, ponieważ wiele ścieżek zostaje odrzuconych już na wczesnym etapie. Niemniej jednak, z punktu widzenia analizy złożoności obliczeniowej, kluczowy jest rozmiar przestrzeni możliwych permutacji, a ta wciąż rośnie wykładniczo wraz z liczbą wierzchołków. Dlatego problem cyklu Hamiltona pozostaje obliczeniowo kosztowny i trudny do rozwiązania dla dużych grafów.

Implementacja algorytmu w Python

```
# Funkcja znajduje wszystkie cykle Hamiltona w grafie reprezentowanym jako macierz sąsiedztwa
def find_hamiltonian_cycles(graph):
    n = len(graph)          # liczba wierzchołków w grafie
    cycles = []             # lista do przechowywania znalezionych cykli Hamiltona

    # Funkcja pomocnicza do rekursywnego przeszukiwania ścieżek (backtracking)
    def backtrack(path, visited):
        current = path[-1]    # ostatni odwiedzony wierzchołek

        # Sprawdzenie czy ścieżka zawiera wszystkie wierzchołki
        if len(path) == n:
            # Jeśli istnieje krawędź powrotna do wierzchołka startowego, to mamy cykl Hamiltona
            if graph[current][path[0]] == 1:
                cycles.append(path + [path[0]])  # dodajemy domknięty cykl do listy
            return

        # Iteracja po wszystkich możliwych sąsiadach bieżącego wierzchołka
        for neighbor in range(n):
            # Jeśli istnieje krawędź i wierzchołek nie był jeszcze odwiedzony
            if graph[current][neighbor] == 1 and not visited[neighbor]:
                visited[neighbor] = True              # oznacz wierzchołek jako odwiedzony
                backtrack(path + [neighbor], visited)   # rekurencyjne wywołanie z nową ścieżką
                visited[neighbor] = False             # cofnięcie (backtracking)
```

```
    # Przeszukiwanie rozpoczynamy od każdego wierzchołka jako punktu startowego
    for start in range(n):
        visited = [False] * n      # tablica odwiedzonych wierzchołków
        visited[start] = True      # oznacz wierzchołek startowy jako odwiedzony
        backtrack([start], visited)  # rozpocznij budowanie ścieżki od wierzchołka startowego

    return cycles  # zwróć wszystkie znalezione cykle Hamiltona

# Funkcja testująca — uruchamia algorytm na podanym grafie i wypisuje wyniki
def test_graph(graph, name):
    print(f"\n{'='*50}")
    print(f"{name}")
    print(f"{'='*50}")
    cycles = find_hamiltonian_cycles(graph)
    if cycles:
        print(f"\nPodsumowanie: Liczba cykli Hamiltona: {len(cycles)}")
        for cycle in cycles:
            print(" -> ".join(str(node) for node in cycle))
    else:
        print("\nPodsumowanie: Brak cykli Hamiltona w grafie.")

# Przykładowe grafy do testów (reprezentacja jako macierz sąsiedztwa)
graph1 = [
    [0, 1, 0],
    [0, 0, 1],
    [1, 0, 0]
]
graph2 = [
```

```
    [0, 1, 1],
    [1, 0, 1],
    [1, 1, 0]
]
graph3 = [
    [0, 1, 0],
    [0, 0, 1],
    [0, 0, 0]
]
graph4 = [
    [0, 1, 1, 0],
    [1, 0, 1, 1],
    [1, 1, 0, 1],
    [0, 1, 1, 0]
]
# Uruchomienie testów tylko jeśli skrypt jest uruchamiany bezpośrednio
if __name__ == "__main__":
    test_graph(graph1, "Przykład 1: Jeden cykl Hamiltona")
    test_graph(graph2, "Przykład 2: Wszystkie cykle to cykle Hamiltona")
    test_graph(graph3, "Przykład 3: Brak cykli Hamiltona")
    test_graph(graph4, "Przykład 4: Macierz 4x4")
```

Wyniki implementacji

Przykład 1: Jeden cykl Hamiltona

Podsumowanie: Liczba cykli Hamiltona: 3
0 -> 1 -> 2 -> 0
1 -> 2 -> 0 -> 1
2 -> 0 -> 1 -> 2

Przykład 2: Wszystkie cykle to cykle Hamiltona

Podsumowanie: Liczba cykli Hamiltona: 6
0 -> 1 -> 2 -> 0
0 -> 2 -> 1 -> 0
1 -> 0 -> 2 -> 1
1 -> 2 -> 0 -> 1
2 -> 0 -> 1 -> 2
2 -> 1 -> 0 -> 2

Przykład 3: Brak cykli Hamiltona

Podsumowanie: Brak cykli Hamiltona w grafie.

Przykład 4: Macierz 4x4

Podsumowanie: Liczba cykli Hamiltona: 8
0 -> 1 -> 3 -> 2 -> 0
0 -> 2 -> 3 -> 1 -> 0
1 -> 0 -> 2 -> 3 -> 1
1 -> 3 -> 2 -> 0 -> 1
2 -> 0 -> 1 -> 3 -> 2
2 -> 3 -> 1 -> 0 -> 2
3 -> 1 -> 0 -> 2 -> 3
3 -> 2 -> 0 -> 1 -> 3

Bibliografia

[1] - Cormen et al., Introduction to Algorithms, MIT Press
[2] - Michael R. Garey, David S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, W.H. Freeman, 1979