PRZESZUKIWANIE W GŁĄB (DEPTH-FIRST SEARCH, DFS)

Przeszukiwanie w głąb (DFS) to jedna z podstawowych metod przeszukiwania grafów i drzew, szeroko stosowana w sztucznej inteligencji i informatyce. Główna idea polega na eksplorowaniu jednej ścieżki aż do jej końca (czyli aż do napotkania liścia w drzewie lub węzła bez sąsiadów w grafie), zanim algorytm zacznie badać inne ścieżki. DFS działa na zasadzie "zagłębiania się" jak najgłębiej w strukturę problemu, a dopiero potem wraca, jeśli nie znalazł rozwiązania na danej ścieżce.

Jak działa DFS?

Kroki algorytmu są następujące:

1. **Zacznij od węzła początkowego (startowego)**.
2. Oznacz węzeł jako odwiedzony.
3. **Wybierz pierwszy nieodwiedzony sąsiedni węzeł** i przejdź do niego.
4. Powtórz kroki od 2 do 3 dla każdego kolejnego węzła, aż:
   - Zostanie znalezione rozwiązanie, albo
   - Nie będzie już nieodwiedzonych sąsiadów.
5. Jeśli dotarłeś do węzła, który nie ma nieodwiedzonych sąsiadów, wróć do poprzedniego węzła (backtracking) i spróbuj inną ścieżką.
6. Kontynuuj, aż odwiedzisz wszystkie węzły lub znajdziesz rozwiązanie.

Przykład działania DFS w drzewie

Wyobraź sobie drzewo, w którym węzły są oznaczone literami:

1. Zaczynamy od węzła **A**.
2. Przechodzimy do **B** (pierwszy sąsiad węzła A).
3. Przechodzimy do **E** (pierwszy sąsiad B).
4. Ponieważ **E** nie ma sąsiadów, cofamy się do **B**.
5. Przechodzimy do **F** (drugi sąsiad B).
6. Przechodzimy do **H** (jedyny sąsiad F).
7. **H** nie ma sąsiadów, więc cofamy się do **F**, a potem do **B**, a następnie do **A**.
8. Przechodzimy do **C** (drugi sąsiad A).
9. **C** nie ma sąsiadów, więc cofamy się do **A**.
10. Przechodzimy do **D** (trzeci sąsiad A).
11. Przechodzimy do **G** (jedyny sąsiad D).
12. **G** nie ma sąsiadów, więc cofamy się do **D** i kończymy przeszukiwanie.

Przykład 1. Znalezienie ścieżki w labiryncie

Mamy labirynt przedstawiony jako siatkę (np. 2D listę), gdzie:

- 0 oznacza wolne pole (przejezdne),
- 1 oznacza przeszkodę (ściana, przez którą nie można przejść),
- Znajdź ścieżkę od punktu startowego (S) do punktu końcowego (E), poruszając się tylko na pola przylegające pionowo lub poziomo.

S	0	1	0	0
1	0	1	1	0
1	0	0	1	0
1	1	0	0	0
1	0	0	1	E

Drzewo decyzyjne zostanie zbudowane na podstawie poniższych rozważań:

1. Startujemy od punktu (0, 0) i mamy kilka możliwych ruchów, jednak możemy pójść w **prawo** do (0, 1) (bo w lewo i w górę jest poza granicami, a w dół mamy ścianę).
2. Z punktu (0, 1) możemy: pójść w **dół** do (1, 1) (bo reszta opcji to albo ściany, albo powrót do (0, 0)).
3. Z punktu (1, 1) możemy: pójść w **dół** do (2, 1).
4. Z punktu (2, 1) mamy dwie opcje: pójść w **prawo** do (2, 2), bo idąc w **dół** do (3, 1) natrafilibyśmy na ścianę, więc nie możemy tam pójść.
5. Z punktu (2, 2) możemy: pójść w **dół** do (3, 2).
6. Z punktu (3, 2) możemy: pójść w **prawo** do (3, 3).
7. Z punktu (3, 3) możemy: pójść w **prawo** do (3, 4).
8. Z punktu (3, 4) możemy: pójść w **dół** do punktu końcowego (4, 4) -> cel osiągnięty.

Kod w Python:

```
def dfs(maze, start, end): # przyjmuje labirynt, punkt startowy
oraz punkt końcowy
    rows, cols = len(maze), len(maze[0])
    stack = [(start, [start])]  # stack = stos zawiera krotki:
(bieżąca pozycja, dotychczasowa ścieżka) do śledzenia aktualnej
pozycji
```

```
    visited = set()  # zestaw do śledzenia odwiedzonych pozycji

    while stack:
        (current_pos, path) = stack.pop()
        x, y = current_pos

        # Jeśli osiągniemy koniec, zwracamy znalezioną ścieżkę
        if current_pos == end:
            return path

        # Jeśli bieżąca pozycja była już odwiedzona, pomijamy ją
        if current_pos in visited:
            continue

        visited.add(current_pos) # dodajemy bieżącą pozycję do
odwiedzonych

        # Przeglądamy sąsiadów: góra, dół, lewo, prawo
        for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
            new_x, new_y = x + dx, y + dy
            # Sprawdzamy, czy nowa pozycja jest w granicach
labiryntu i czy jest wolna (0)
            if 0 <= new_x < rows and 0 <= new_y < cols and
maze[new_x][new_y] == 0:
                stack.append(((new_x, new_y), path + [(new_x,
new_y)]))

    return None  # Jeśli nie znaleziono ścieżki, zwracamy None

# Definiujemy labirynt jako siatkę 2D (0 - przejezdne, 1 - ściana)
maze = [
    [0, 0, 1, 0, 0],
    [1, 0, 1, 1, 0],
    [1, 0, 0, 1, 0],
    [1, 1, 0, 0, 0],
    [1, 0, 0, 1, 0]
]

start = (0, 0)  # # Punkt startowy (S)
end = (4, 4)    # Punkt końcowy  (E)

# Szukamy ścieżki używając DFS
```

```
path = dfs(maze, start, end)

if path:
    print("Path found:", path)
else:
    print("No path found.")
```

W obecnej sytuacji powinniśmy otrzymać znalezioną ścieżkę (tą samą co zaznaczona jest powyżej):

```
Path found: [(0, 0), (0, 1), (1, 1), (2, 1), (2, 2), (3, 2), (3, 3), (3, 4), (4, 4)]
```

Zadanie 1. Znalezienie ścieżki w labiryncie

Mamy labirynt przedstawiony jako siatkę (np. 2D listę), gdzie:

- 0 oznacza wolne pole (przejezdne),
- 1 oznacza przeszkodę (ściana, przez którą nie można przejść),
- Znajdź ścieżkę od punktu startowego (S) do punktu końcowego (E), poruszając się tylko na pola przylegające pionowo lub poziomo.

S	**0**	**1**	**0**	**0**	**0**	**0**
1	**0**	**1**	**1**	**1**	**1**	**0**
1	**0**	**0**	**0**	**0**	**1**	**0**
1	**0**	**1**	**1**	**0**	**1**	**0**
0	**0**	**0**	**1**	**0**	**0**	**0**
0	**1**	**0**	**1**	**1**	**1**	**0**
0	**1**	**1**	**0**	**0**	**0**	**E**

Spróbuj rozpisać drzewo decyzyjne dla powyższego przypadku.

Przykład 2. Liczenie wysp

Mamy daną tablicę 2D, w której każda komórka zawiera "1" (ląd) lub "0" (woda). Wyspa jest grupą sąsiadujących ze sobą komórek "1" połączonych w pionie lub poziomie (diagonalne połączenia nie są brane pod uwagę). Zadaniem jest policzenie liczby wysp dla poniższego przykładu:

```
tablica = [
  [1, 1, 0, 0, 0],
  [1, 1, 0, 0, 0],
  [0, 0, 1, 0, 0],
  [0, 0, 0, 1, 1]
]
```

Algorytm DFS będzie przeszukiwał każdą wyspę zaczynając od znalezienia "1" i eksplorując wszystkie sąsiadujące komórki lądu, zmieniając ich wartość na "0", aby oznaczyć je jako odwiedzone. Każde rozpoczęcie DFS liczymy jako jedną wyspę.

Kod w Python:

```
def dfs(tablica, x, y):
    # Sprawdzamy czy współrzędne są w granicach tablicy i czy
komórka to ląd (1)
    if x < 0 or x >= len(tablica) or y < 0 or y >= len(tablica[0])
or tablica[x][y] == 0:
        return

    # Oznaczamy bieżącą komórkę jako odwiedzoną (zamieniamy ją na
0)
    tablica[x][y] = 0

    # Sprawdzamy sąsiadów w czterech kierunkach: góra, dół, lewo,
prawo
    dfs(tablica, x - 1, y)  # góra
    dfs(tablica, x + 1, y)  # dół
    dfs(tablica, x, y - 1)  # lewo
    dfs(tablica, x, y + 1)  # prawo


def liczba_wysp(tablica):
    if not tablica:
        return 0
```

```
    licznik_wysp = 0

    # Przechodzimy przez każdą komórkę w tablicy
    for i in range(len(tablica)):
        for j in range(len(tablica[0])):
            # Jeśli znajdziemy ląd (1), uruchamiamy DFS i
zwiększamy licznik wysp
            if tablica[i][j] == 1:
                dfs(tablica, i, j)
                licznik_wysp += 1

    return licznik_wysp

# Przykładowa tablica
tablica = [
    [1, 1, 0, 0, 0],
    [1, 1, 0, 0, 0],
    [0, 0, 1, 0, 0],
    [0, 0, 0, 1, 1]
]

# Liczymy liczbę wysp
print("Liczba wysp:", liczba_wysp(tablica))
```

W obecnej sytuacji powinniśmy otrzymać ilość wysp:

```
Liczba wysp: 3
```

Zadanie 2. Odwracanie słowa (do zastanowienia się w domu)

Użyj DFS do budowy nowego słowa, przeszukując je od końca do początku.

Spróbuj wykorzystać funkcję ***`reverse_dfs`****, która przyjmuje dwa argumenty: oryginalne słowo oraz indeks aktualnie przetwarzanej litery.*
Jeśli indeks jest mniejszy od 0, zwraca pusty ciąg (koniec rekurencji). W przeciwnym razie zwraca bieżącą literę (znajdującą się pod indeksem) oraz rekurencyjne wywołanie dla poprzedniego indeksu.

```
Dla słowa hello powinniśmy otrzymać odwrócone słowo: olleh  😊
```

PRZESZUKIWANIE WSZERZ (BREADTH-FIRST SEARCH, BFS)

Przeszukiwanie wszerz (BFS) to algorytm służący do przeszukiwania grafów lub drzew, który działa w sposób poziomy, eksplorując wszystkie sąsiadujące węzły przed przejściem do następnego poziomu. Tak więc różnicą miedzy DFS a BFS jest kolejność przetwarzania. BFS przetwarza węzły poziomo (wszystkie sąsiednie węzły przed przejściem do następnego poziomu), podczas gdy DFS przetwarza węzły głęboko (zachodzi w dół, aż do końca gałęzi, zanim wróci). Kolejną różnicą jest to, że BFS używa kolejki, podczas gdy DFS używa stosu (lub rekurencji).

Przykład 3. Znalezienie ścieżki w labiryncie (najkrótszej ścieżki)

Mamy labirynt przedstawiony jako siatkę (np. 2D listę), gdzie:

- 0 oznacza wolne pole (przejezdne),
- 1 oznacza przeszkodę (ściana, przez którą nie można przejść),
- Znajdź ścieżkę od punktu startowego (S) do punktu końcowego (E), poruszając się tylko na pola przylegające pionowo lub poziomo.

S	1	0	0	0
0	1	0	1	0
0	0	0	1	0
0	1	0	0	0
0	0	0	1	E

Kod w Python:

```
from collections import deque

def bfs(maze, start, end):
    rows, cols = len(maze), len(maze[0])
    queue = deque([start])  # Inicjalizacja kolejki z punktem
startowym
    visited = set()  # Zbiór odwiedzonych węzłów
    visited.add(start)

    # Kierunki ruchu (góra, dół, lewo, prawo)
    directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]
```

```
    # Słownik do przechowywania ścieżki
    parent = {start: None}

    while queue:
        current = queue.popleft()  # Usuwa węzeł z początku kolejki

        # Sprawdź, czy osiągnięto cel
        if current == end:
            path = []
            while current is not None:
                path.append(current)
                current = parent[current]
            return path[::-1]  # Odwróć ścieżkę

        # Sprawdź sąsiadów
        for direction in directions:
            neighbor = (current[0] + direction[0], current[1] +
direction[1])
            if (0 <= neighbor[0] < rows and
                0 <= neighbor[1] < cols and
                maze[neighbor[0]][neighbor[1]] == 0 and
                neighbor not in visited):
                queue.append(neighbor)
                visited.add(neighbor)
                parent[neighbor] = current  # Ustaw rodzica

    return None  # Brak ścieżki

# Przykładowe użycie
maze = [
    [0, 1, 0, 0, 0],
    [0, 1, 0, 1, 0],
    [0, 0, 0, 1, 0],
    [0, 1, 0, 0, 0],
    [0, 0, 0, 1, 0]
]

start = (0, 0)  # Punkt startowy
end = (4, 4)    # Punkt docelowy
path = bfs(maze, start, end)

if path:
```

```
    print("Najkrótsza ścieżka w labiryncie:")
    print(path)
else:
    print("Nie znaleziono ścieżki.")
```

Kilka informacji:

`from collections import deque` *to polecenie w Pythonie, które importuje klasę* `deque` *z modułu* `collections`
gdzie `deque` *(skrót od "double-ended queue") to struktura danych, która pozwala na efektywne dodawanie i usuwanie elementów z obu końców (przodu i tyłu).*

Ja używać `deque`*:*

```
from collections import deque

# Tworzenie deque
queue = deque()

# Dodawanie elementów
queue.append(1)  # Dodaje 1 na końcu
queue.append(2)  # Dodaje 2 na końcu
queue.appendleft(0)  # Dodaje 0 na początku

print(queue)  # Wypisze: deque([0, 1, 2])

# Usuwanie elementów
print(queue.popleft())  # Usuwa i wypisuje: 0
print(queue.pop())  # Usuwa i wypisuje: 2

print(queue)  # Wypisze: deque([1])
```

W kontekście BFS, `deque` jest używane do przechowywania węzłów, które mają być przetworzone. Umożliwia to szybkie dodawanie nowych węzłów na końcu oraz usuwanie węzłów z przodu, co jest kluczowe dla działania algorytmu przeszukiwania wszerz.

Przykład 4. Odwracanie Słowa z Użyciem BFS

Odwracanie słowa przy użyciu algorytmu przeszukiwania wszerz (BFS) jest interesującym zastosowaniem, mimo że BFS nie jest najbardziej typowym wyborem do tego zadania. Możemy jednak użyć BFS do zbudowania odwróconego słowa, traktując litery jako węzły w grafie (Każda litera w słowie będzie węzłem, a kolejność liter zdefiniuje krawędzie między nimi). Użyjemy kolejki do przetwarzania liter, które będą dodawane w odwrotnej kolejności.

Kod w Python:

```
from collections import deque

def reverse_bfs(word):    # Przyjmuje słowo jako argument
    queue = deque(word)   # Inicjalizuje kolejkę z literami słowa
    reversed_word = ""

    while queue:        # W pętli przetwarzane są kolejne litery
        letter = queue.pop()   # Usuwa literę z końca kolejki
        reversed_word += letter   # Dodaje literę do odwróconego
słowa

    return reversed_word

# Przykładowe użycie
word = "hello"
reversed_word = reverse_bfs(word)

# Wynik
print("Odwrócone słowo:", reversed_word)
```

Zadanie 3. Liczenie wysp (dla chętnych)

Liczenie wysp to popularne zadanie, które można rozwiązać za pomocą algorytmu przeszukiwania wszerz (BFS) lub przeszukiwania w głąb (DFS) (co już zrobiliśmy).

Użyjemy macierzy dwuwymiarowej, gdzie 1 reprezentuje ląd, a 0 reprezentuje wodę.

```
maze = [
    [1, 1, 0, 0, 0],
    [0, 1, 0, 1, 1],
    [0, 0, 0, 0, 0],
    [1, 0, 1, 0, 1],
    [1, 1, 0, 0, 0]
]
```