**Міністерство освіти і науки України**

**Національний університет «Запорізька Політехніка»**

Кафедра програмних засобів

**ЗВІТ**

з лабораторної роботи №5

з дисципліни «Алгоритми та Структури Даних» на тему:

«Алгоритми обходу графів»

**Виконав:**

Студент групи КНТ-122 О. А. Онищенко

**Прийняли:**

Старший викладач: Л. Ю. Дейнега

2023

**Алгоритми обходу графів**

**Мета роботи**

Вивчити алгоритми обходу графів на основі пошуку в ширину та на основі пошуку в глибину. Навчитися застосовувати алгоритми обходу графів для розв’язання практичних задач.

**Завдання до роботи**

- Розробити програмне забезпечення, в якому реалізується алгоритм обходу графу на основі пошуку в глибину. Передбачити, що граф може бути як орієнтований, так і неорієнтований. В процесі пошуку має бути сформовано ліс пошуку в глибину. Для реалізації має використовуватися стек. Програмне забезпечення має бути побудовано на основі відповідного класу, який повинен дозволяти визначати граф, виконувати пошук в глибину, виводити побудований ліс пошуку в глибину, виводити результат обходу тощо.

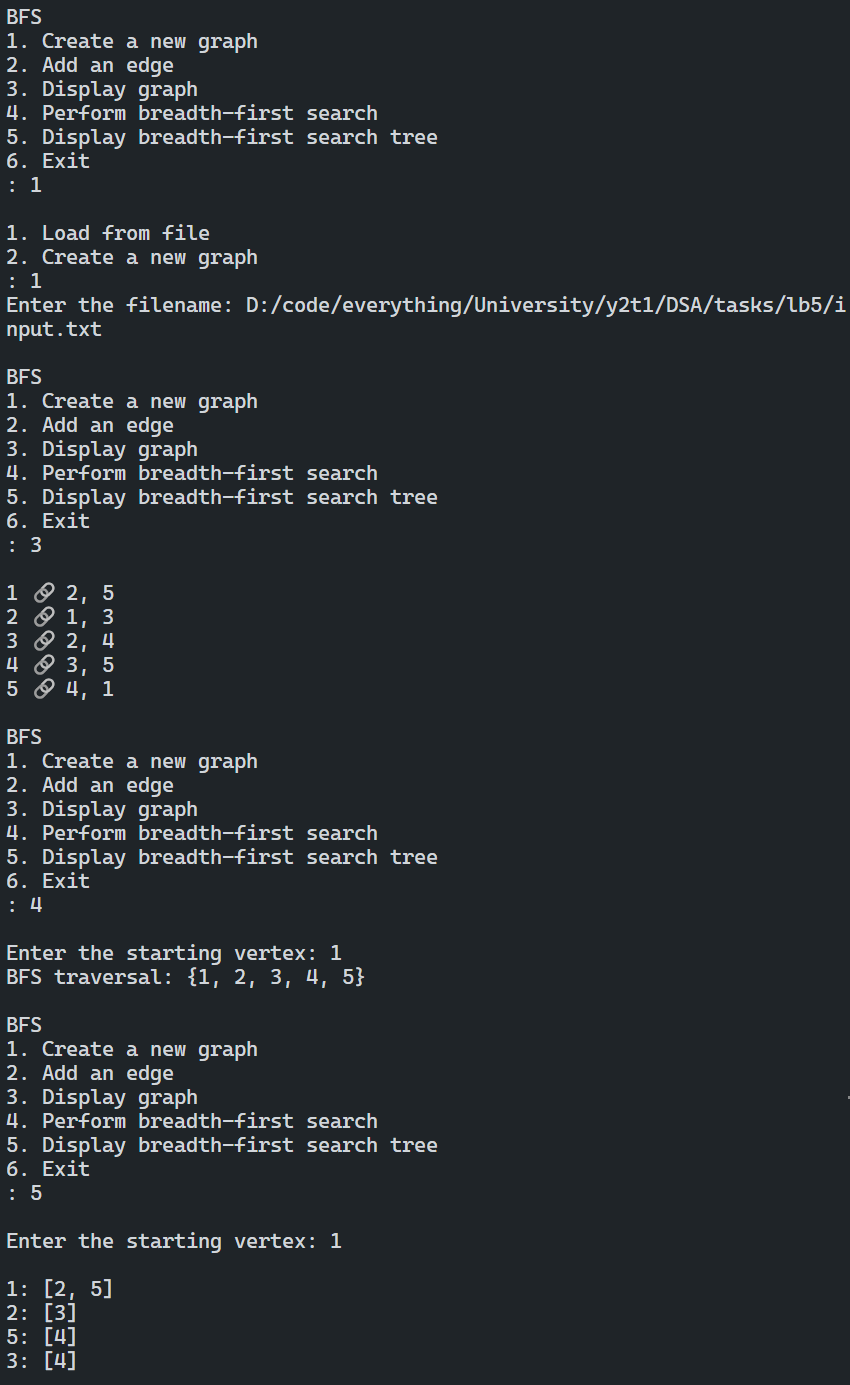
- Розробити програмне забезпечення, в якому реалізується алгоритм обходу графу на основі пошуку в ширину. Передбачити, що граф може бути як орієнтований, так і неорієнтований. В процесі пошуку має бути сформовано дерево пошуку в ширину. Для реалізації має використовуватися черга. Програмне забезпечення має бути побудовано на основі відповідного класу, який повинен дозволяти визначати граф, виконувати пошук в ширину, виводити побудоване дерево пошуку в ширину, виводити результат обходу тощо.

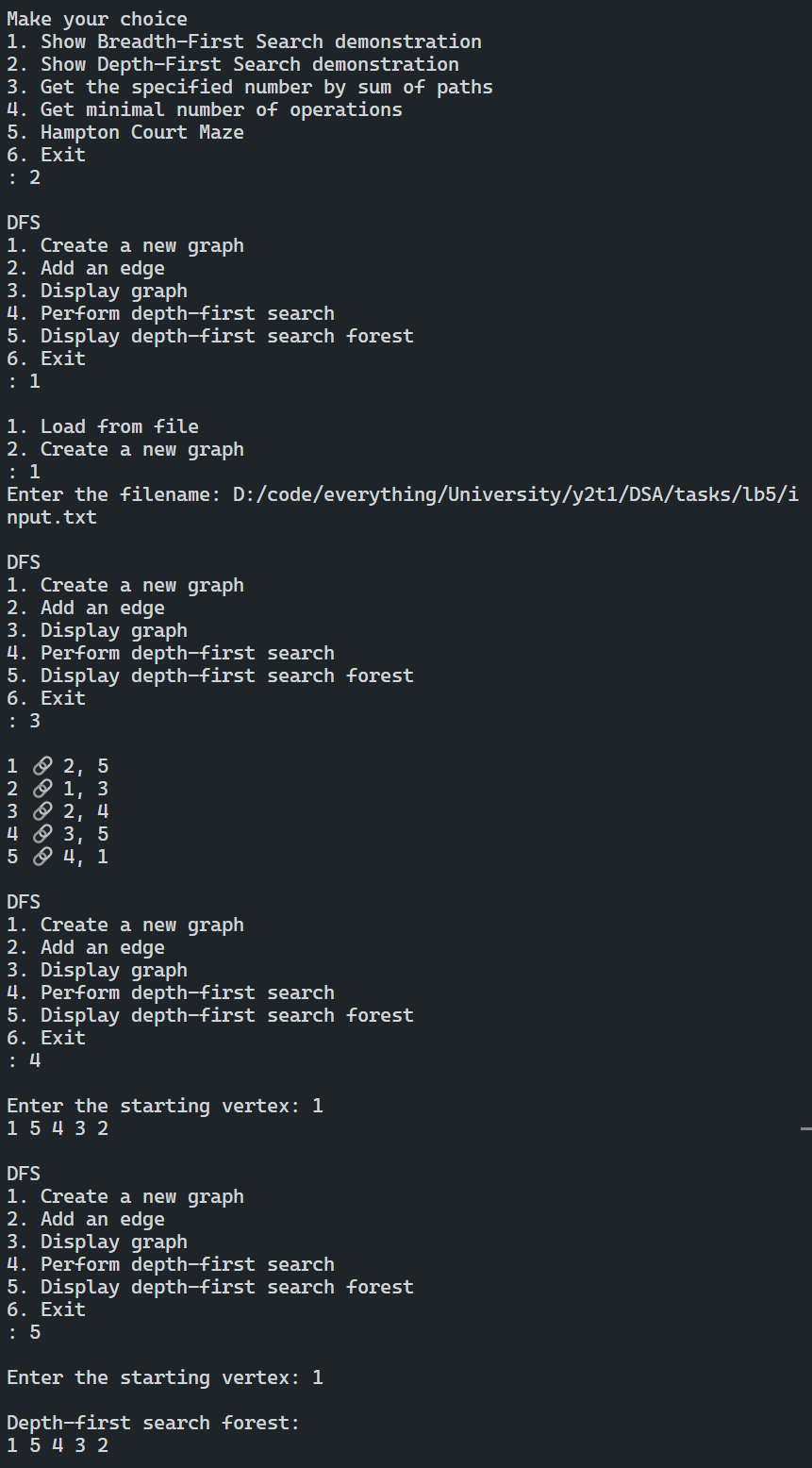
- У заданому користувачем графі поставлено у відповідність кожній вершині деяке ціле число (може бути як від’ємним, так і додатним). Визначити такі шляхи між парами вершин, які в результаті додавання всіх чисел з кожної вершини дозволяють отримати задане користувачем значення.

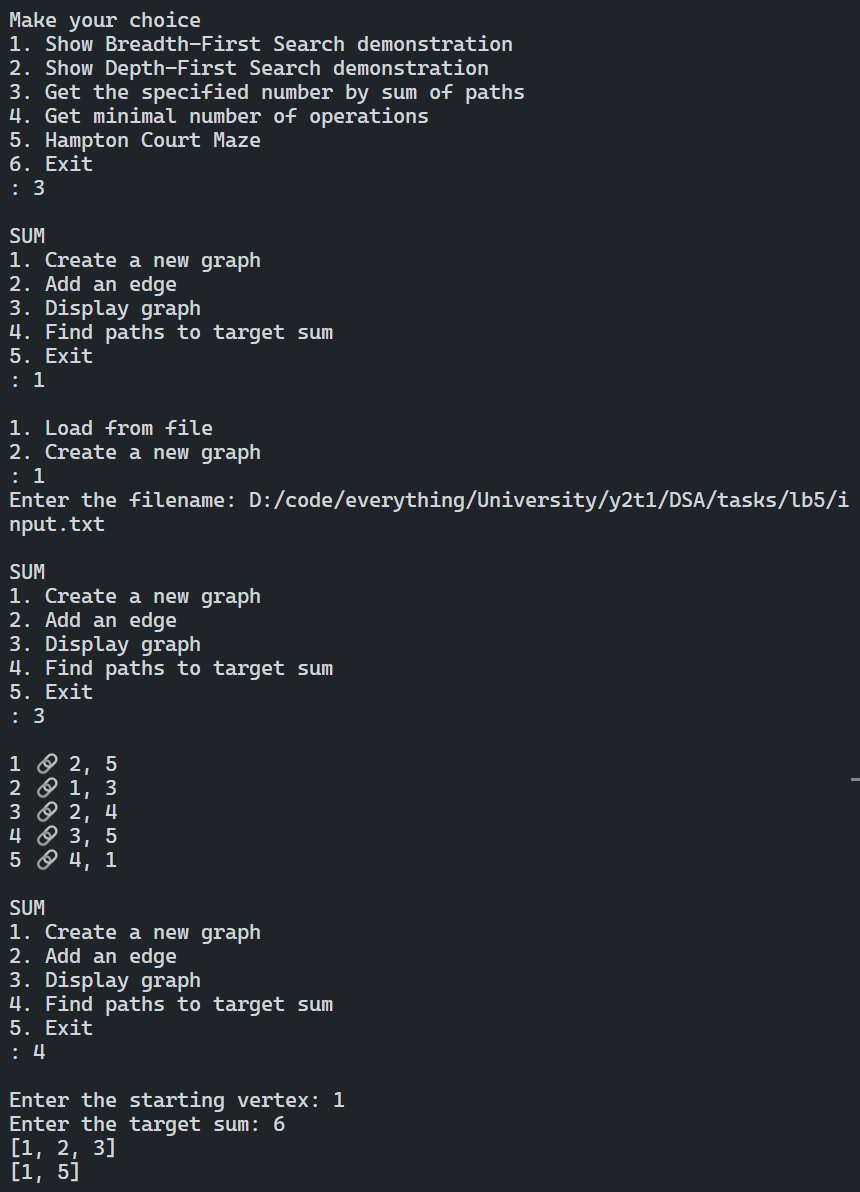
- Задано деякий набір арифметичних операцій (наприклад, додати 3, помножити на 2), які можуть бути виконані над операндом. Визначити мінімальний набір операцій, за допомогою якого можна отримати з одного заданого числа а число b. Якщо таке перетворення за допомогою заданого користувачем набору операцій виконати неможливо, то вивести відповідне повідомлення.

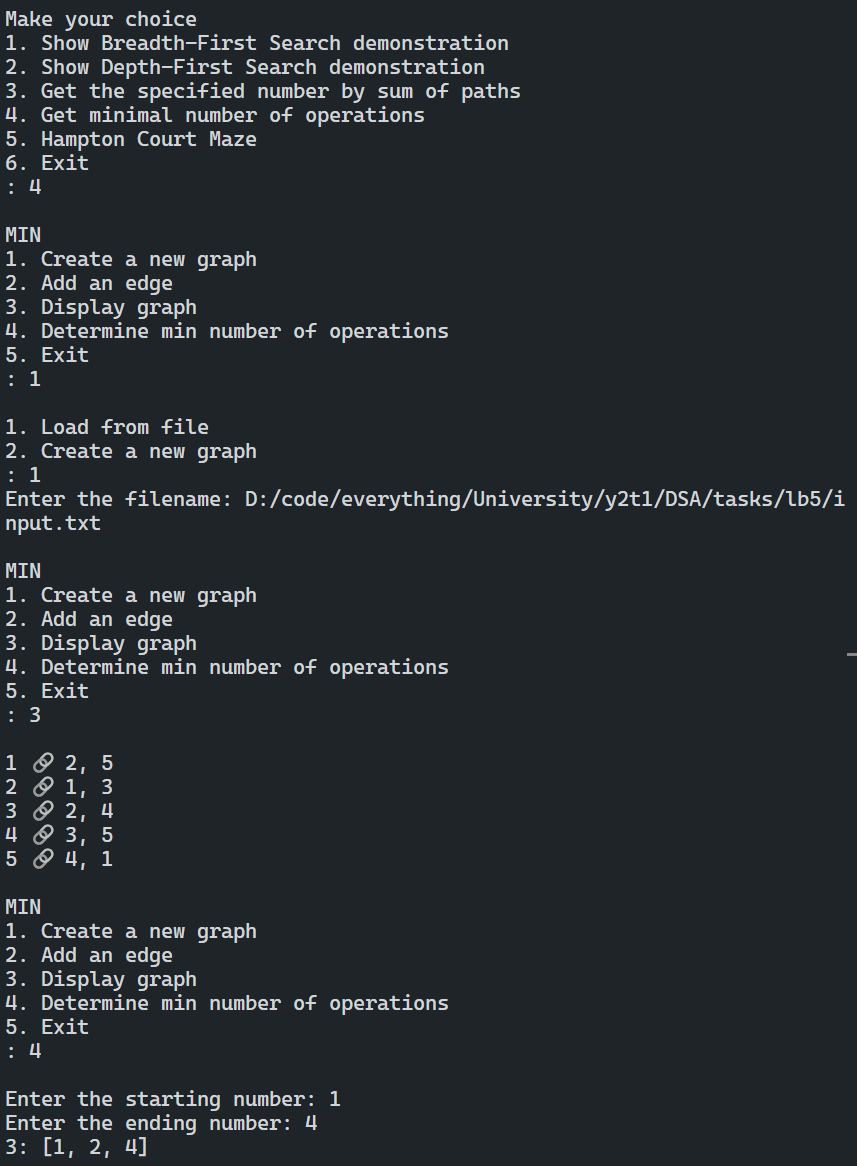
- Гемптон-Кортський лабіринт площею у 60 акрів привертає увагу багатьох туристів. Ваш побратим перед тим, як потрапити до одного з таких лабіринтів і продемонструвати свої здібності, вирішив вивчити план лабіринту та звернувся до Вас по допомогу, яким чином знайти шлях у лабіринті. Змоделюйте лабіринт за допомогою вершин, що відповідають входу в лабіринт, виходу, глухим кутам, всім точкам лабіринту, в яких є можливість вибору шляху, та з’єднань даних вершин ребрами, що відповідають шляхам у лабіринті

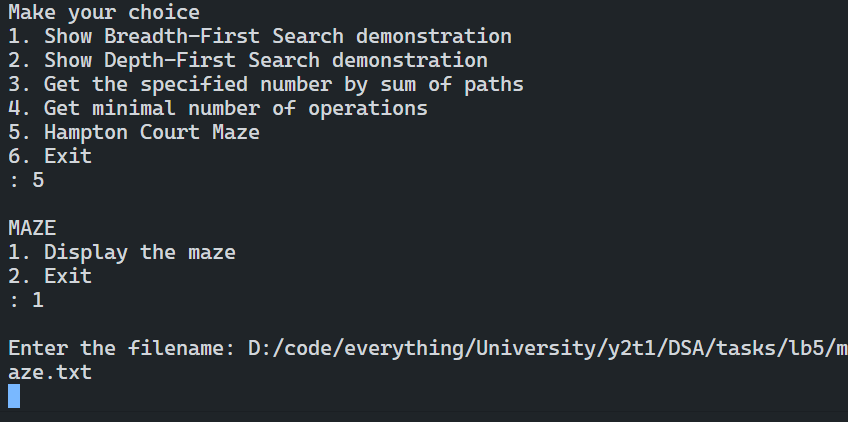
**Результати виконання роботи**

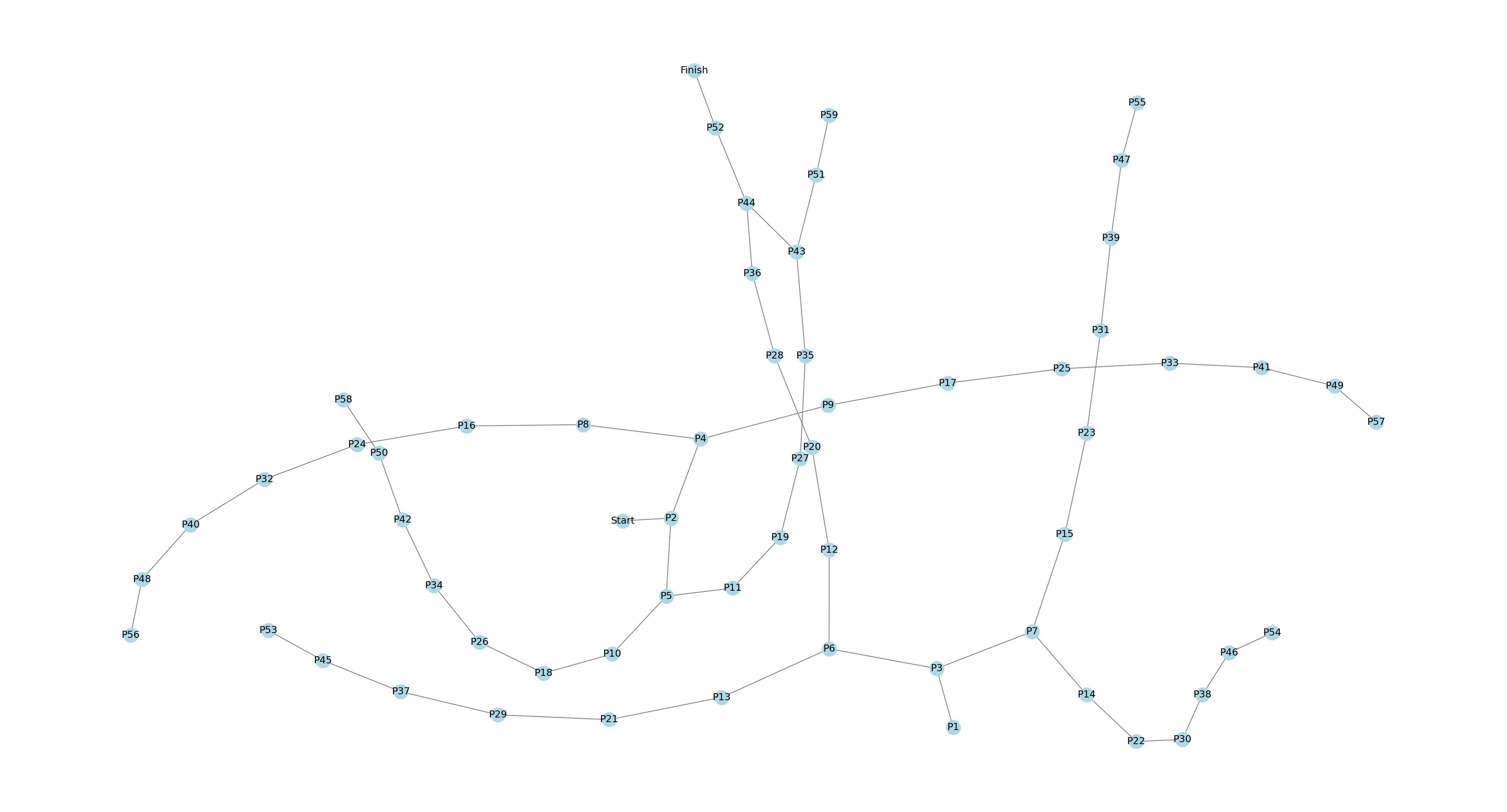












**Код**

def breadthFirstSearch():

    from collections import defaultdict, deque

    class Graph:

        def \_\_init\_\_(self, directed=False):

            self.graph = defaultdict(list)

            self.directed = directed

        def addEdge(self, u, v):

            self.graph[u].append(v)

            if not self.directed:

                self.graph[v].append(u)

        def BFS(self, start):

            visited = set()

            queue = deque([start])

            while queue:

                vertex = queue.popleft()

                if vertex not in visited:

                    visited.add(vertex)

                    queue.extend(set(self.graph[vertex]) - visited)

            return visited

        def displayTree(self, start):

            visited = set()

            queue = deque([start])

            tree = defaultdict(list)

            while queue:

                vertex = queue.popleft()

                if vertex not in visited:

                    visited.add(vertex)

                    for neighbor in self.graph[vertex]:

                        if neighbor not in visited:

                            tree[vertex].append(neighbor)

                            queue.append(neighbor)

            return tree

        def displayResults(self, start):

            visited = self.BFS(start)

            print(f"BFS traversal: {visited}")

        def displayGraph(self):

            for key, value in self.graph.items():

                print(f"{key} 🔗 {', '.join(map(str, value))}")

        def loadFromFile(self, filename):

            with open(filename, "r") as file:

                for line in file:

                    u, v = line.strip().split()

                    self.addEdge(int(u), int(v))

    def main():

        g = None

        while True:

            print("\nBFS")

            print("1. Create a new graph")

            print("2. Add an edge")

            print("3. Display graph")

            print("4. Perform breadth-first search")

            print("5. Display breadth-first search tree")

            print("6. Exit")

            choice = int(input(": "))

            print()

            if choice == 1:

                print("1. Load from file")

                print("2. Create a new graph")

                choice = int(input(": "))

                choice = 1

                if choice == 1:

                    filename = input("Enter the filename: ")

                    g = Graph()

                    g.loadFromFile(filename)

                elif choice == 2:

                    directed = input("Is the graph directed? (y/n): ") == "y"

                    g = Graph(directed)

            elif choice == 2:

                u = int(input("Enter the first vertex: "))

                v = int(input("Enter the second vertex: "))

                g.addEdge(u, v)

            elif choice == 3:

                g.displayGraph()

            elif choice == 4:

                v = int(input("Enter the starting vertex: "))

                g.displayResults(v)

            elif choice == 5:

                v = int(input("Enter the starting vertex: "))

                print()

                tree = dict(g.displayTree(v))

                for key, value in tree.items():

                    print(f"{key}: {value}")

            else:

                break

    main()

def depthFirstSearch():

    from collections import defaultdict

    class Graph:

        def \_\_init\_\_(self, directed=False):

            self.graph = defaultdict(list)

            self.directed = directed

        def addEdge(self, u, v):

            self.graph[u].append(v)

            if not self.directed:

                self.graph[v].append(u)

        def DFS(self, v, visited=None):

            if visited is None:

                visited = set()

            stack = [v]

            while stack:

                vertex = stack.pop()

                if vertex not in visited:

                    visited.add(vertex)

                    print(vertex, end=" ")

                    stack.extend(set(self.graph[vertex]) - visited)

            return visited

        def displayForest(self, v):

            print("Depth-first search forest:")

            self.DFS(v)

            print()

        def displayGraph(self):

            for key, value in self.graph.items():

                print(f"{key} 🔗 {', '.join(map(str, value))}")

        def loadFromFile(self, filename):

            with open(filename, "r") as file:

                for line in file:

                    u, v = line.strip().split()

                    self.addEdge(int(u), int(v))

    def main():

        g = None

        while True:

            print("\nDFS")

            print("1. Create a new graph")

            print("2. Add an edge")

            print("3. Display graph")

            print("4. Perform depth-first search")

            print("5. Display depth-first search forest")

            print("6. Exit")

            choice = int(input(": "))

            print()

            if choice == 1:

                print("1. Load from file")

                print("2. Create a new graph")

                choice = int(input(": "))

                choice = 1

                if choice == 1:

                    filename = input("Enter the filename: ")

                    g = Graph()

                    g.loadFromFile(filename)

                elif choice == 2:

                    directed = input("Is the graph directed? (y/n): ") == "y"

                    g = Graph(directed)

            elif choice == 2:

                u = int(input("Enter the first vertex: "))

                v = int(input("Enter the second vertex: "))

                g.addEdge(u, v)

            elif choice == 3:

                g.displayGraph()

            elif choice == 4:

                v = int(input("Enter the starting vertex: "))

                g.DFS(v)

                print()

            elif choice == 5:

                v = int(input("Enter the starting vertex: "))

                print()

                g.displayForest(v)

            else:

                break

    main()

def getNumberBySumOfPaths():

    from collections import defaultdict

    class Graph:

        def \_\_init\_\_(self, directed=False):

            self.graph = defaultdict(list)

            self.directed = directed

        def addEdge(self, u, v):

            self.graph[u].append(v)

            if not self.directed:

                self.graph[v].append(u)

        def findPaths(self, start, targetSum):

            visited = set()

            path = []

            self.\_findPathsHelper(start, targetSum, visited, path)

        def \_findPathsHelper(self, vertex, targetSum, visited, path):

            visited.add(vertex)

            path.append(vertex)

            if sum(path) == targetSum:

                print(path)

            for neighbor in self.graph[vertex]:

                if neighbor not in visited:

                    self.\_findPathsHelper(neighbor, targetSum, visited, path)

            path.pop()

            visited.remove(vertex)

        def displayGraph(self):

            for key, value in self.graph.items():

                print(f"{key} 🔗 {', '.join(map(str, value))}")

        def loadFromFile(self, filename):

            with open(filename, "r") as file:

                for line in file:

                    u, v = line.strip().split()

                    self.addEdge(int(u), int(v))

    def main():

        g = None

        while True:

            print("\nSUM")

            print("1. Create a new graph")

            print("2. Add an edge")

            print("3. Display graph")

            print("4. Find paths to target sum")

            print("5. Exit")

            choice = int(input(": "))

            print()

            if choice == 1:

                print("1. Load from file")

                print("2. Create a new graph")

                choice = int(input(": "))

                choice = 1

                if choice == 1:

                    filename = input("Enter the filename: ")

                    g = Graph()

                    g.loadFromFile(filename)

                elif choice == 2:

                    directed = input("Is the graph directed? (y/n): ") == "y"

                    g = Graph(directed)

            elif choice == 2:

                u = int(input("Enter the first vertex: "))

                v = int(input("Enter the second vertex: "))

                g.addEdge(u, v)

            elif choice == 3:

                g.displayGraph()

            elif choice == 4:

                v = int(input("Enter the starting vertex: "))

                targetSum = int(input("Enter the target sum: "))

                g.findPaths(v, targetSum)

            else:

                break

    main()

def getMinimalNumberOfOperations():

    from collections import defaultdict, deque

    class Graph:

        def \_\_init\_\_(self, directed=False):

            self.graph = defaultdict(list)

            self.directed = directed

        def addEdge(self, u, v):

            self.graph[u].append(v)

            if not self.directed:

                self.graph[v].append(u)

        def minOperations(self, a, b, maxNumber=None):

            queue = deque([a])

            length = {a: 0}

            previous = {a: None}

            while queue:

                currentValue = queue.popleft()

                def tryPerforming(nextValue):

                    if maxNumber is not None and nextValue > maxNumber:

                        return

                    if nextValue in length:

                        return

                    queue.append(nextValue)

                    length[nextValue] = length[currentValue] + 1

                    previous[nextValue] = currentValue

                tryPerforming(currentValue + 1)

                tryPerforming(currentValue - 1)

                tryPerforming(currentValue \* 2)

                if currentValue % 2 == 0:

                    tryPerforming(currentValue // 2)

                tryPerforming(currentValue \* 3)

                if currentValue % 3 == 0:

                    tryPerforming(currentValue // 3)

                if b in length:

                    break

            path = [b]

            while path[-1] != a:

                path.append(previous[path[-1]])

            path.reverse()

            return path

        def displayGraph(self):

            for key, value in self.graph.items():

                print(f"{key} 🔗 {', '.join(map(str, value))}")

        def loadFromFile(self, filename):

            with open(filename, "r") as file:

                for line in file:

                    u, v = line.strip().split()

                    self.addEdge(int(u), int(v))

    def main():

        g = None

        while True:

            print("\nMIN")

            print("1. Create a new graph")

            print("2. Add an edge")

            print("3. Display graph")

            print("4. Determine min number of operations")

            print("5. Exit")

            choice = int(input(": "))

            print()

            if choice == 1:

                print("1. Load from file")

                print("2. Create a new graph")

                choice = int(input(": "))

                choice = 1

                if choice == 1:

                    filename = input("Enter the filename: ")

                    g = Graph()

                    g.loadFromFile(filename)

                elif choice == 2:

                    directed = input("Is the graph directed? (y/n): ") == "y"

                    g = Graph(directed)

            elif choice == 2:

                u = int(input("Enter the first vertex: "))

                v = int(input("Enter the second vertex: "))

                g.addEdge(u, v)

            elif choice == 3:

                g.displayGraph()

            elif choice == 4:

                if g is not None:

                    a = int(input("Enter the starting number: "))

                    b = int(input("Enter the ending number: "))

                    path = g.minOperations(a, b)

                    print(f"{len(path)}: {path}")

                else:

                    print("No graph loaded.")

            else:

                break

    main()

def hamptonMaze():

    from collections import defaultdict, deque

    import networkx as nx

    import matplotlib.pyplot as plt

    class Graph:

        def \_\_init\_\_(self, directed=False):

            self.graph = defaultdict(list)

            self.directed = directed

        def addEdge(self, u, v):

            self.graph[u].append(v)

            if not self.directed:

                self.graph[v].append(u)

        def loadFromFile(self, filename):

            with open(filename, "r") as file:

                for line in file:

                    u, v = line.strip().split()

                    self.addEdge(u, v)

        def drawGraph(self):

            G = nx.Graph()

            for node in self.graph:

                G.add\_node(node)

            for node, edges in self.graph.items():

                for edge in edges:

                    G.add\_edge(node, edge)

            nx.draw(G, with\_labels=True, node\_color="lightblue", edge\_color="gray")

            plt.show()

    def main():

        g = None

        while True:

            print("\nMAZE")

            print("1. Display the maze")

            print("2. Exit")

            choice = int(input(": "))

            print()

            if choice == 1:

                g = Graph()

                filename = input("Enter the filename: ")

                g.loadFromFile(filename)

                g.drawGraph()

            else:

                break

    main()

def menu():

    while True:

        print("\nMake your choice")

        print("1. Show Breadth-First Search demonstration")

        print("2. Show Depth-First Search demonstration")

        print("3. Get the specified number by sum of paths")

        print("4. Get minimal number of operations")

        print("5. Hampton Court Maze")

        print("6. Exit")

        choice = int(input(": "))

        if choice == 1:

            breadthFirstSearch()

        elif choice == 2:

            depthFirstSearch()

        elif choice == 3:

            getNumberBySumOfPaths()

        elif choice == 4:

            getMinimalNumberOfOperations()

        elif choice == 5:

            hamptonMaze()

        else:

            break

menu()

// input.txt

1 2

2 3

3 4

4 5

5 1

// maze.txt

Start P2

P1 P3

P2 P4

P2 P5

P3 P6

P3 P7

P4 P8

P4 P9

P5 P10

P5 P11

P6 P12

P6 P13

P7 P14

P7 P15

P8 P16

P9 P17

P10 P18

P11 P19

P12 P20

P13 P21

P14 P22

P15 P23

P16 P24

P17 P25

P18 P26

P19 P27

P20 P28

P21 P29

P22 P30

P23 P31

P24 P32

P25 P33

P26 P34

P27 P35

P28 P36

P29 P37

P30 P38

P31 P39

P32 P40

P33 P41

P34 P42

P35 P43

P36 P44

P37 P45

P38 P46

P39 P47

P40 P48

P41 P49

P42 P50

P43 P51

P43 P44

P44 P52

P45 P53

P46 P54

P47 P55

P48 P56

P49 P57

P50 P58

P51 P59

P52 Finish

**Висновки**

Таким чином, ми вивчили алгоритми обходу графів на основі пошуку в ширину та на основі пошуку в глибину. Також ми навчилися застосовувати алгоритми обходу графів для розв’язання практичних задач.

**Контрольні питання**

**Які існують способи зображення графів та яким чином вони пов’язані між собою?**

Графи можуть бути представлені кількома способами, включаючи матриці суміжності, списки суміжності та списки ребер.

- Матриця суміжності - це двовимірний масив, де комірка в i-му рядку та j-му стовпчику дорівнює 1, якщо існує ребро з вершини i до вершини j. Вона корисна для щільних графів, де кількість ребер близька до кількості вершин у квадраті. Однак він не є ефективним для розріджених графів, оскільки більшість клітинок у матриці є нульовими.

- Список суміжності - це набір невпорядкованих списків, які використовуються для представлення скінченного графа. Кожен список описує множину сусідів вершини графа. Він ефективний для розріджених графів, оскільки зберігає лише ті ребра, які існують.

- Список ребер - це список ребер, що з’єднують вершини графу. Кожне ребро - це пара вершин. Він корисний для представлення графа, коли ребра є більш важливими, ніж вершини.

**У чому полягає алгоритм обходу графа на основі пошуку в ширину?**

Пошук в ширину (BFS) - це алгоритм обходу графа, який досліджує всі вершини графа в першу чергу в ширину, тобто досліджує всі вершини на поточній глибині, перш ніж перейти до вершин на наступному рівні глибини. BFS використовує структуру даних у вигляді черги, щоб відстежувати наступну вершину для відвідування.

**У чому полягає алгоритм обходу графа на основі пошуку в глибину?**

Пошук в глибину (DFS) - це алгоритм обходу графа, який досліджує якомога далі вздовж кожної гілки, перш ніж повернутися назад. DFS використовує структуру даних стек для відстеження наступного вузла для відвідування.