تمرین ۱ – کاربرد سیستم های هوشمند در پزشکی

مازندرانیان – ۸۳۰۴۰۲۰۶۶

Q1

```python
from collections import deque
from scipy.io import loadmat
from scipy.sparse import csc_matrix
import numpy as np

file_path = "ConnectionMatrix.mat"
mat_data = loadmat(file_path)

node_connections = mat_data['NodeConnections']
node_connections_dense = node_connections.toarray()

def bfs(graph, start, goal):
    start -= 1
    goal -= 1
    queue = deque([(start, [start])])
    visited = set()

    while queue:
        node, path = queue.popleft()
        if node in visited:
            continue

        visited.add(node)
        if node == goal:
            return [p + 1 for p in path]

        for neighbor, connected in enumerate(graph[node]):
            if connected and neighbor not in visited:
                queue.append((neighbor, path + [neighbor]))

    return None

start_node = 1
goal_node = 20

graph = node_connections_dense
path_bfs = bfs(graph, start_node, goal_node)

print(path_bfs)
```

این کد از چند بخش اصلی تشکیل شده است. ابتدا، با استفاده از کتابخانه‌های deque از ماژول collections، loadmat از scipy.io، و csc_matrix از scipy.sparse داده‌ها بارگذاری می‌شوند. فایل **ConnectionMatrix.mat** شامل ماتریسی از اتصالات گره‌ها است که پس از بارگذاری با استفاده از متد toarray() به یک آرایه متراکم تبدیل می‌شود. سپس، تابع bfs که

الگوریتم جستجوی اول سطح را پیاده‌سازی می‌کند، تعریف شده است. این تابع از یک **صف** برای مدیریت گره‌ها و مسیرها استفاده کرده و با استفاده از مجموعه‌ای به نام visited، گره‌های بازدید شده را پیگیری می‌کند تا از حلقه‌های بی‌پایان جلوگیری کند. در نهایت، در کد اصلی، گراف (ماتریس اتصالات) به عنوان ورودی به تابع bfsداده می‌شود و مسیر یافت شده بین گره شروع و هدف محاسبه و چاپ می‌شود.

Q2

```python
from collections import deque
from scipy.io import loadmat
from scipy.sparse import csc_matrix
import numpy as np

file_path = "ConnectionMatrix.mat"
mat_data = loadmat(file_path)

node_connections = mat_data['NodeConnections']
node_connections_dense = node_connections.toarray()

def dfs(graph, start, goal):
    start -= 1
    goal -= 1
    stack = [(start, [start])]
    visited = set()

    while stack:
        node, path = stack.pop()
        if node in visited:
            continue

        visited.add(node)
        if node == goal:
            return [p + 1 for p in path]

        for neighbor, connected in reversed(list(enumerate(graph[node]))):
            if connected and neighbor not in visited:
                stack.append((neighbor, path + [neighbor]))

    return None

start_node = 1
goal_node = 20

graph = node_connections_dense
path_dfs = dfs(graph, start_node, goal_node)

print(path_dfs)
```

الگوریتم جستجوی عمقی (DFS) را برای گرافی که در قالب یک ماتریس ارتباطی از یک فایل بارگذاری شده، پیاده‌سازی می‌کند. ابتدا فایل شامل اطلاعات گراف بارگذاری می‌شود و سپس با استفاده از یک تبدیل، آن را به یک آرایه چگال تبدیل می‌کند. در الگوریتمDFS ، گره‌ها به‌صورت عمیق بررسی می‌شوند تا زمانی که به گره هدف برسند یا همه مسیرها بررسی شوند. در نهایت، اگر

مسیری بین گره شروع و هدف پیدا شود، مسیر را برمی‌گرداند. در این کد همسایه‌ها در ترتیب معکوس پیمایش می‌شوند که ممکن است ترتیب گره‌های بازدید شده را تغییر دهد. به‌طور کلی، این کد به دنبال پیدا کردن مسیری بین دو گره خاص در گراف است .

Q3

```python
from collections import deque
from scipy.io import loadmat
from scipy.sparse import csc_matrix
import numpy as np

file_path = "ConnectionMatrix.mat"
mat_data = loadmat(file_path)

node_connections = mat_data['NodeConnections']
node_connections_dense = node_connections.toarray()

def best_first_search(graph, start, goal, heuristic):
    start -= 1
    goal -= 1
    priority_queue = [(heuristic[start], start, [start])]
    visited = set()

    while priority_queue:
        priority_queue.sort()
        _, node, path = priority_queue.pop(0)

        if node in visited:
            continue

        visited.add(node)
        if node == goal:
            return [p + 1 for p in path]

        for neighbor, connected in enumerate(graph[node]):
            if connected and neighbor not in visited:
                priority_queue.append((heuristic[neighbor], neighbor, path + [neighbor]))

    return None

start_node = 1
goal_node = 20

heuristic = np.random.rand(len(node_connections_dense))

graph = node_connections_dense
path_best_first = best_first_search(graph, start_node, goal_node, heuristic)

print(path_best_first)
```

Q1_Without_Visited_List

```python
from collections import deque
from scipy.io import loadmat
import numpy as np

file_path = "ConnectionMatrix.mat"
mat_data = loadmat(file_path)

node_connections = mat_data['NodeConnections']
graph = node_connections.toarray()

def bfs(graph, start, goal):
    start -= 1
    goal -= 1
    queue = deque([(start, [start])])

    while queue:
        node, path = queue.popleft()

        if node == goal:
            return [p + 1 for p in path]

        for neighbor, connected in enumerate(graph[node]):
            if connected:
                queue.append((neighbor, path + [neighbor]))

    return None

start_node = 1
goal_node = 20

path_bfs = bfs(graph, start_node, goal_node)

print(path_bfs)
```

این کد الگوریتم جستجوی بهترین اول (Best First Search) را پیاده‌سازی می‌کند تا مسیری از گره شروع به گره هدف در یک گراف پیدا کند. ابتدا ماتریس ارتباطات گراف از یک فایل بارگذاری می‌شود و به فرمت چگال تبدیل می‌گردد. سپس یک هیوستیک تصادفی برای هر گره تولید می‌شود که به‌عنوان معیاری برای اولویت‌بندی گره‌ها در صف استفاده می‌شود. در هر مرحله، گره‌ای که کمترین مقدار هیوستیک را دارد از صف اولویت خارج شده و بررسی می‌شود. همسایه‌های آن گره به صف اضافه می‌شوند و این فرایند ادامه می‌یابد تا زمانی که گره هدف پیدا شود. در نهایت، مسیر از گره شروع به گره هدف برگردانده می‌شود.

Q2_Without_Visited_List

```python
from scipy.io import loadmat
import numpy as np
from scipy.sparse import csr_matrix

file_path = "ConnectionMatrix.mat"
mat_data = loadmat(file_path)

node_connections = mat_data['NodeConnections']
graph = csr_matrix(node_connections)

def dfs(graph, start, goal):
    start -= 1
    goal -= 1
    stack = [start]
    parent = {start: None}

    while stack:
        node = stack.pop()

        if node == goal:
            path = []
            while node is not None:
                path.append(node + 1)
                node = parent[node]
            return path[::-1]

        for neighbor in reversed(graph[node].indices):
            if neighbor not in parent:
                stack.append(neighbor)
                parent[neighbor] = node

    return None

start_node = 1
goal_node = 20

path_dfs = dfs(graph, start_node, goal_node)

print(path_dfs)
```

الگوریتم جستجوی عمقی پیشرفته (Progressive Deepening Search) را پیاده‌سازی می‌کند که ترکیبی از جستجوی عمقی با عمق محدود است. الگوریتم به‌طور تدریجی عمق جستجو را از صفر شروع کرده و تا عمق مشخصی (که در اینجا ۵۰ است)

افزایش می‌دهد. در هر مرحله، جستجوی عمقی محدود (DLS) اجرا می‌شود و گره‌ها بررسی می‌شوند تا زمانی که به گره هدف برسند یا عمق جستجو تمام شود. اگر مسیر پیدا نشود، عمق جستجو بیشتر می‌شود تا زمانی که مسیر به گره هدف پیدا شود یا عمق حداکثر جستجو به پایان برسد.

Q3_Without_Visited_List

```python
from scipy.io import loadmat
import numpy as np

file_path = "ConnectionMatrix.mat"
mat_data = loadmat(file_path)

node_connections = mat_data['NodeConnections']
node_connections_dense = node_connections.toarray()

def best_first_search(graph, start, goal, heuristic):
    start -= 1
    goal -= 1
    priority_queue = [(heuristic[start], start, [start])]

    while priority_queue:
        priority_queue.sort()
        _, node, path = priority_queue.pop(0)

        if node == goal:
            return [p + 1 for p in path]

        for neighbor, connected in enumerate(graph[node]):
            if connected:
                priority_queue.append((heuristic[neighbor], neighbor, path + [neighbor]))

    return None

start_node = 1
goal_node = 20

heuristic = np.random.rand(len(node_connections_dense))

graph = node_connections_dense
path_best_first = best_first_search(graph, start_node, goal_node, heuristic)

print(path_best_first)
```

Q4_Without_Visited_List

```
from scipy.io import loadmat
import numpy as np

file_path = "ConnectionMatrix.mat"
mat_data = loadmat(file_path)

node_connections = mat_data['NodeConnections']
node_connections_dense = node_connections.toarray()

def progressive_deepening_search(graph, start, goal, max_depth):
    start -= 1
    goal -= 1

    def dls(node, path, depth):
        if depth < 0:
            return None
        if node == goal:
            return [p + 1 for p in path]

        for neighbor, connected in enumerate(graph[node]):
            if connected:
                result = dls(neighbor, path + [neighbor], depth - 1)
                if result:
                    return result
        return None

    for depth in range(max_depth + 1):
        result = dls(start, [start], depth)
        if result:
            return result

    return None

start_node = 1
goal_node = 20
max_search_depth = 50

graph = node_connections_dense
path_progressive_deepening = progressive_deepening_search(graph, start_node, goal_node,
max_search_depth)

print(path_progressive_deepening)
```

Algorithm	Time Complexity	Space Complexity
Depth-First Search (DFS)	$O(V + E)$	$O(H)$ (max depth)
Breadth-First Search (BFS)	$O(V + E)$	$O(V)$
Best-First Search	$O(E \log V)$	$O(V)$
Iterative Deepening Search (IDS)	$O(b^d)$	$O(H)$ (max depth)