ĐẠI HỌC QUỐC GIA THÀNH HỒ CHÍ MINH

TRƯỜNG ĐẠI HỌC KHOA TỰ NHIÊN

--- KHOA CÔNG NGHỆ THÔNG TIN ---

Báo cáo

Lab01: Search

Môn học: Cơ sở trí tuệ nhân tạo

Lớp: CQ2022/22

Sinh viên:

MSSV: 22120032

Họ và tên: Trần Thanh Bình

Giảng viên:

Nguyễn Ngọc Đức

Nguyễn Thị Thu Hằng

Nguyễn Trần Duy Minh

Thành phố Hồ Chí Minh – Năm 2024

MỤC LỤC

1. Thông tin sinh viên. 4
2. Đánh giá mức độ hoàn thành. 4
3. Test case. 5
3.1. Input 01. 5
3.2. Input02. 7
3.3. Input03. 9
3.4. Input04: 11
3.5. Input05. 13
4. Lý thuyết cơ bản của các thuật toán. 15
4.1. Breadth First Search (BFS). 15
4.1.1. Khái niệm. 15
4.1.2. Độ phức tạp thuật toán. 15
4.1.3. Thuộc tính. 16
4.1.4. Triển khai thuật toán. 16
4.2. Depth First Search (DFS). 16
4.2.1. Khái niệm. 16
4.2.2. Độ phức tạp thuật toán. 17
4.2.3. Thuộc tính. 17
4.2.4. Triển khai thuật toán. 18
4.3. Uniform-Cost Search (UCS). 19
4.3.1. Khái niệm. 19
4.3.2. Độ phức tạp thuật toán. 19
4.3.4. Triển khai thuật toán. 19
4.4. Greedy Best First Search (GBFS) 20
4.4.1. Khái niệm. 20
4.4.2. Độ phức tạp thuật toán. 20
4.4.3. Thuộc tính. 20
4.4.4. Triển khai thuật toán. 21

4.5. A* 21
4.5.1. Khái niệm. 21
4.5.2. Độ phức tạp thuật toán. 22
4.5.3. Thuộc tính. 22
4.5.4. Triển khai thuật toán. 22
5. So sánh giữ UCS và A* 24
6. Tài liệu tham khảo 24

1. Thông tin sinh viên

- MSSS: 22120032.
- Họ tên: Trần Thanh Bình.

2. Đánh giá mức độ hoàn thành

- Breadth First Search (BFS): 10/10.
- Depth First Search (DFS): 10/10.
- Uniform-Cost Search (UCS): 10/10.
- Greedy Best First Search (GBFS): 10/10.
- A*: 10/10.

3. Test case

3.1. Input 01

```
1 6
0 0 9 3 0 0 9 0
0 0 8 5 5 7 6 7
6 9 0 4 2 5 6 8
5 1 2 0 2 4 7 9
0 3 6 8 0 6 6 5
0 6 3 9 9 0 4 0
5 1 8 1 5 3 0 5
0 3 9 8 3 0 7 0
```

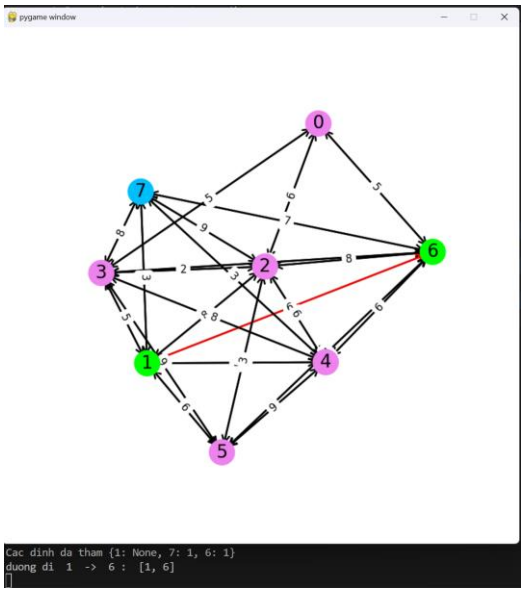


Figure 1: BFS

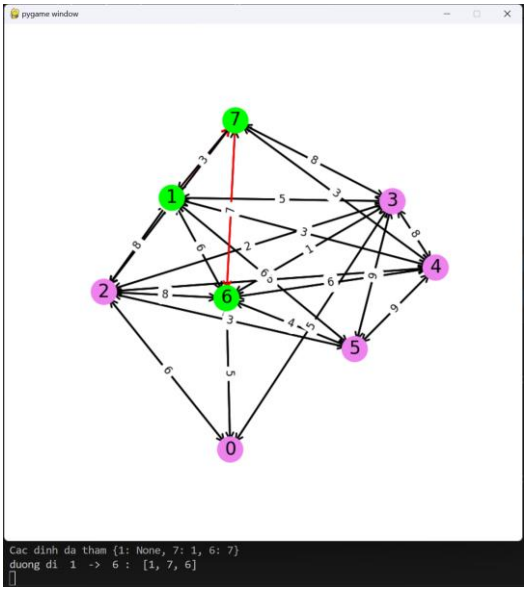


Figure 2: DFS

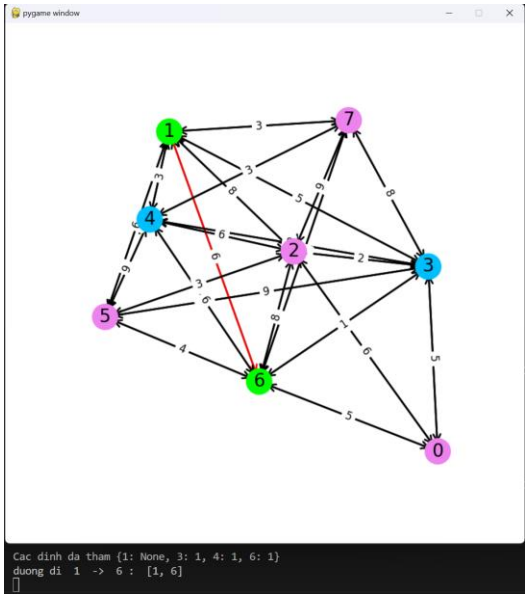


Figure 3: UCS

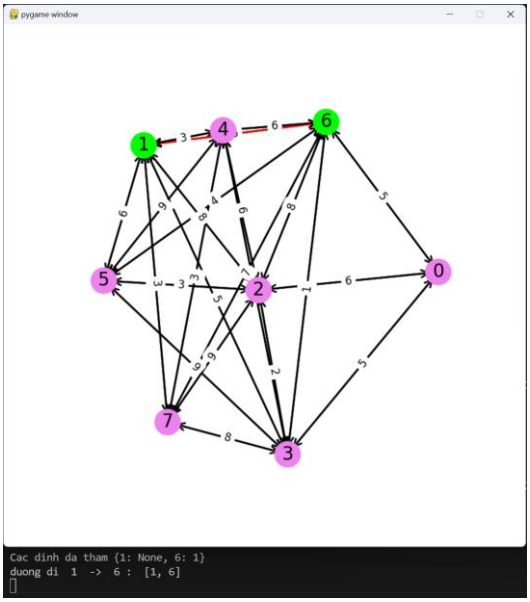


Figure 4: Greedy

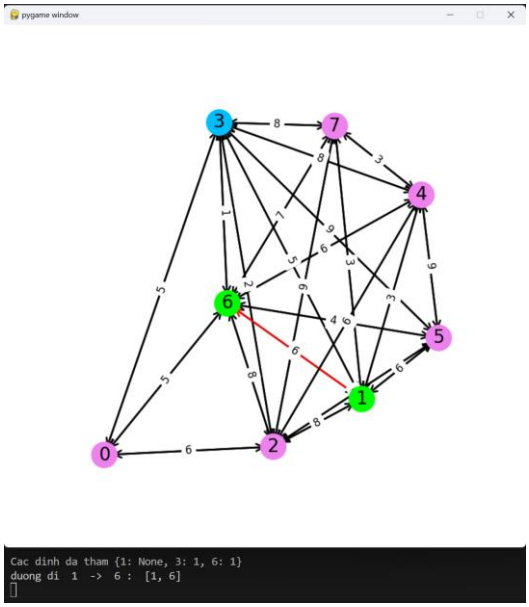
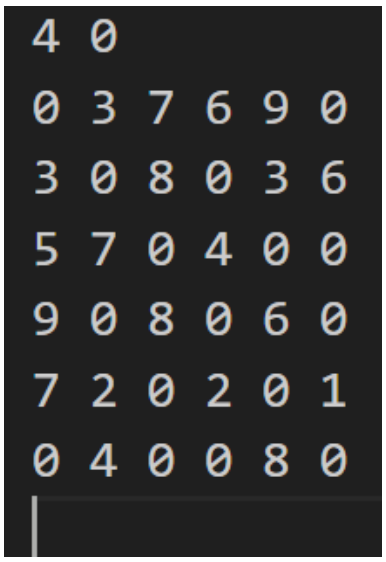


*Figure 5: A**

3.2. Input02



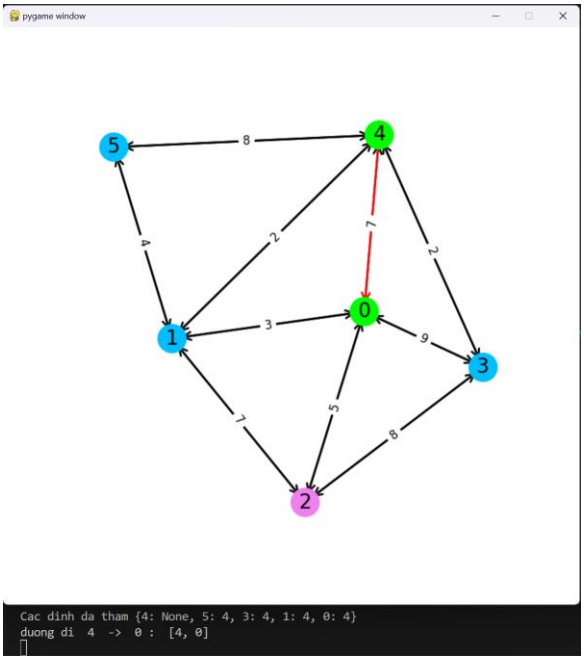


Figure 6: BFS

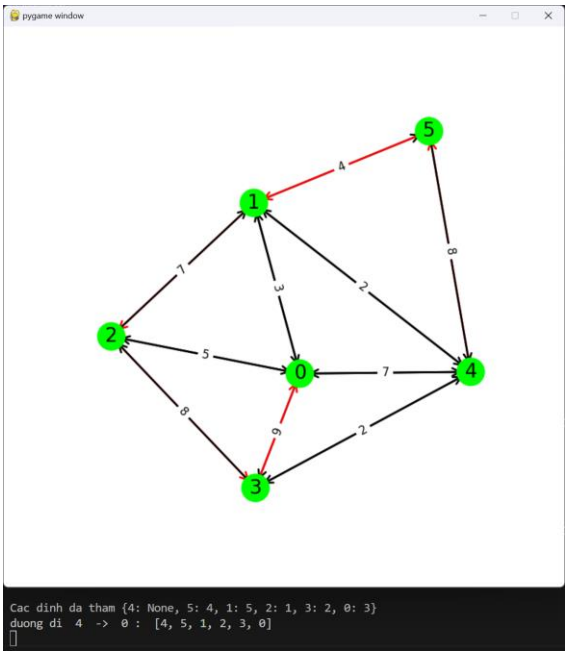


Figure 7: DFS

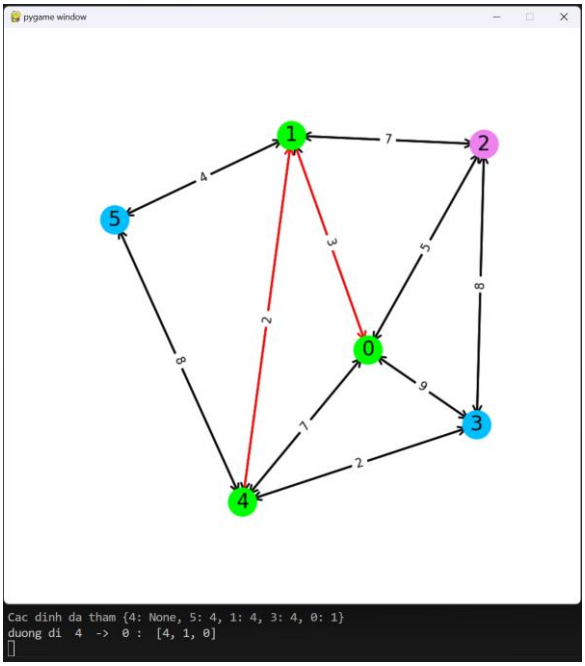


Figure 8: UCS

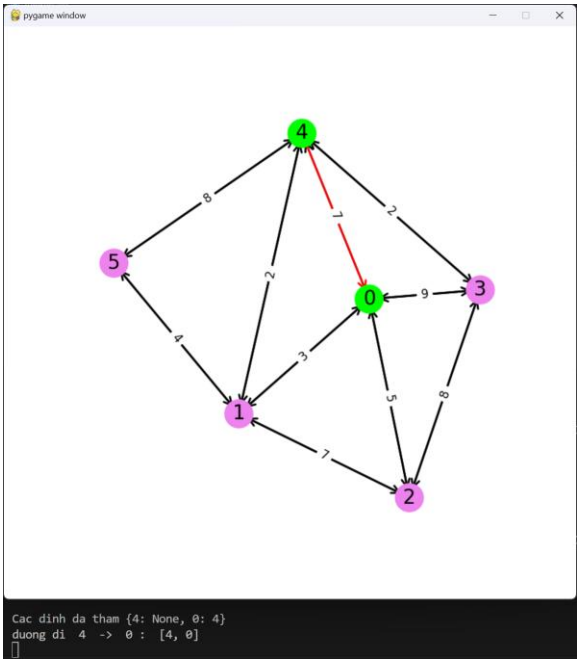


Figure 9: Greedy

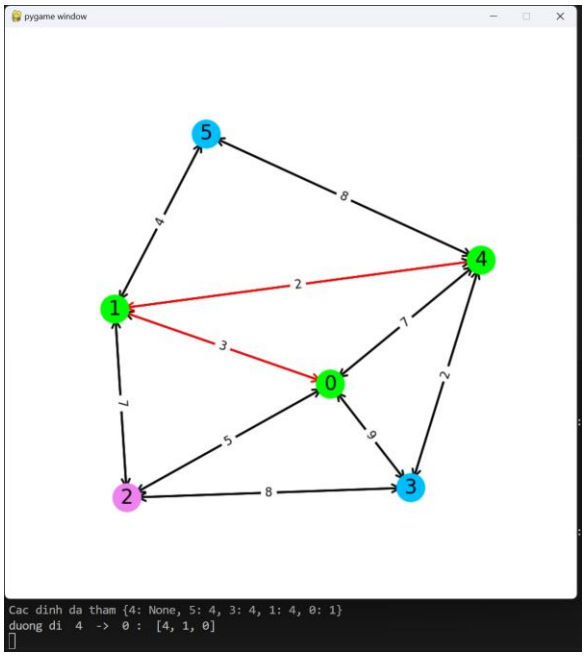


*Figure 10: A**

3.3. Input03

```
2 3
0 7 7 0 3 3 6 4
7 0 1 4 4 0 4 5
6 3 0 0 6 8 8 9
4 5 5 0 7 8 0 1
6 6 4 4 0 0 9 2
0 8 6 3 4 0 5 6
9 7 0 4 7 0 0 0
8 8 7 7 1 8 3 0
```

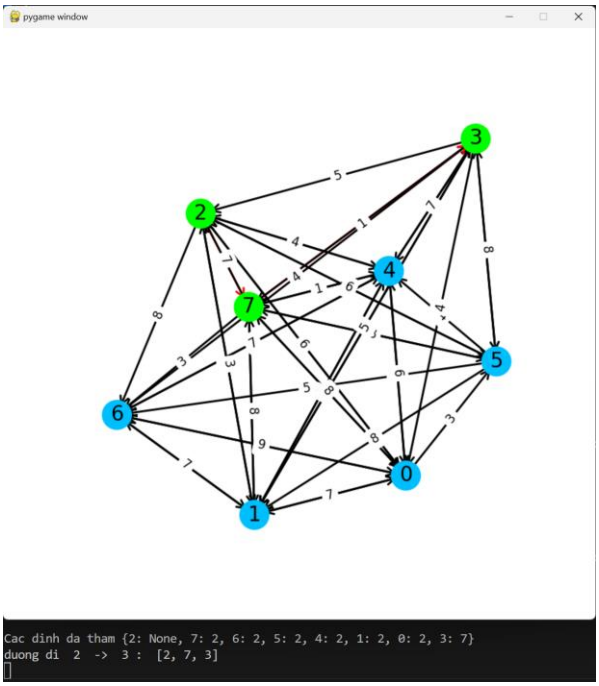


Figure 11: BFS

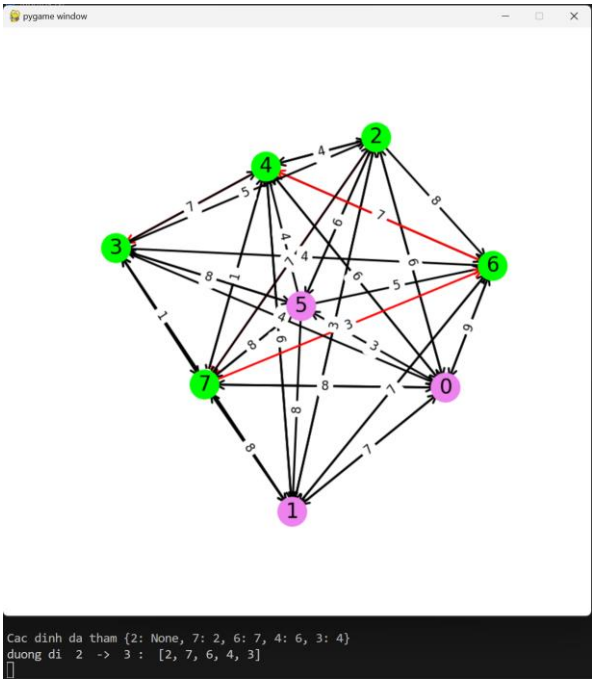


Figure 12: DFS

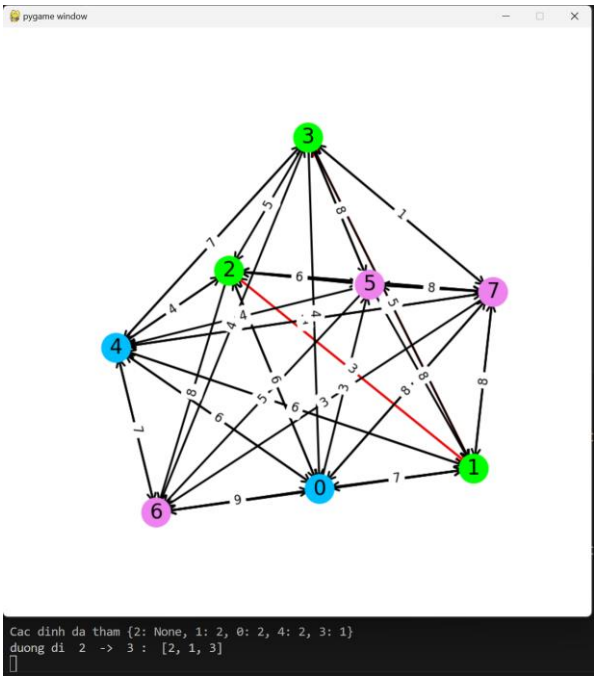


Figure 13: UCS

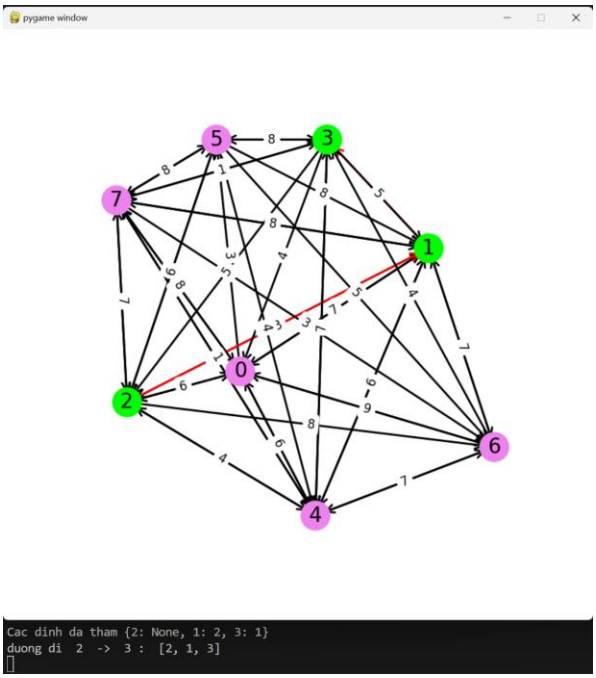


Figure 14: Greedy

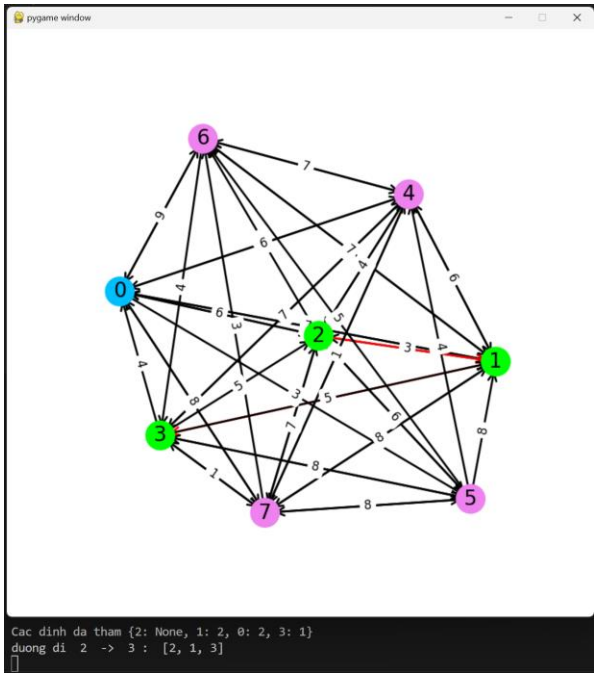


Figure 15: Astar

3.4. Input04

```
4 2
0 3 7 6 9 0
3 0 8 0 3 6
5 7 0 4 0 5
9 0 8 0 6 0
7 2 0 2 0 1
3 4 1 0 8 0
```

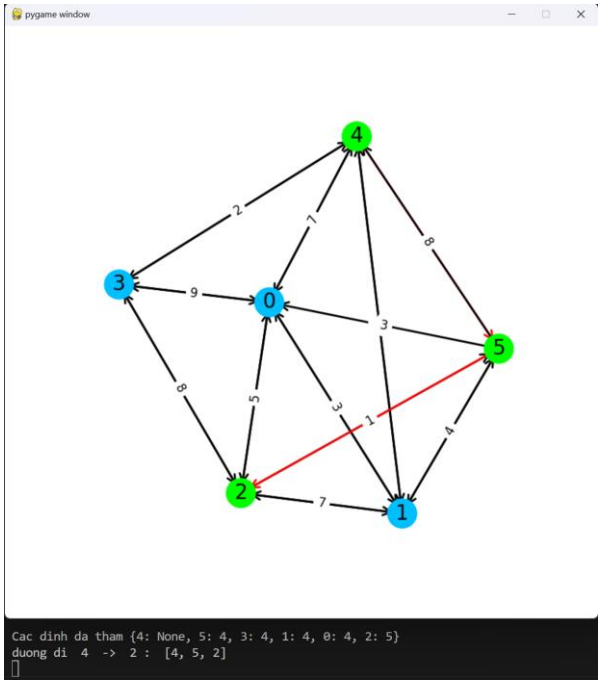


Figure 16: BFS

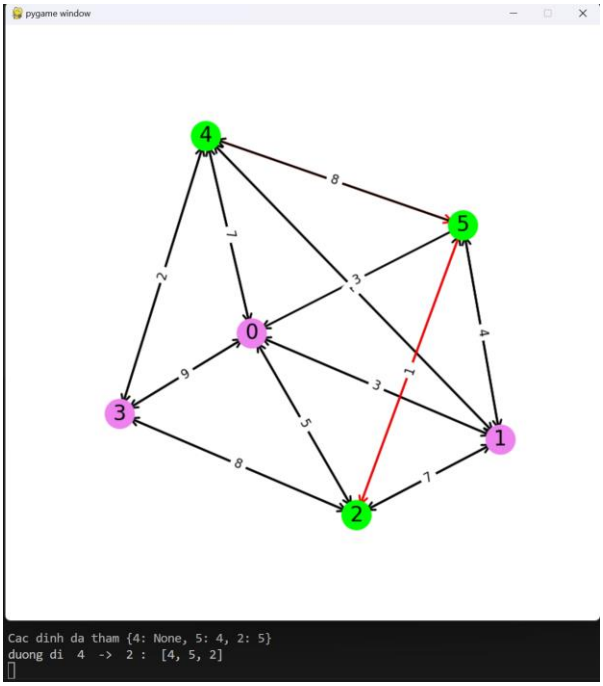


Figure 17: DFS

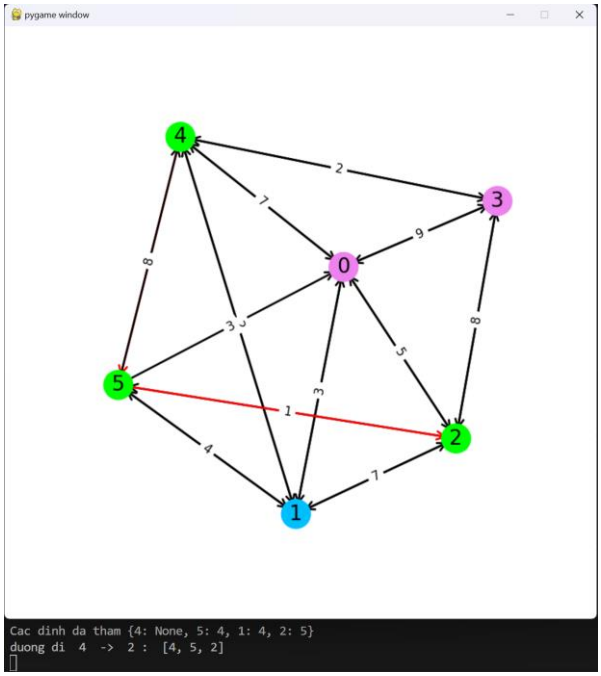


Figure 18: UCS

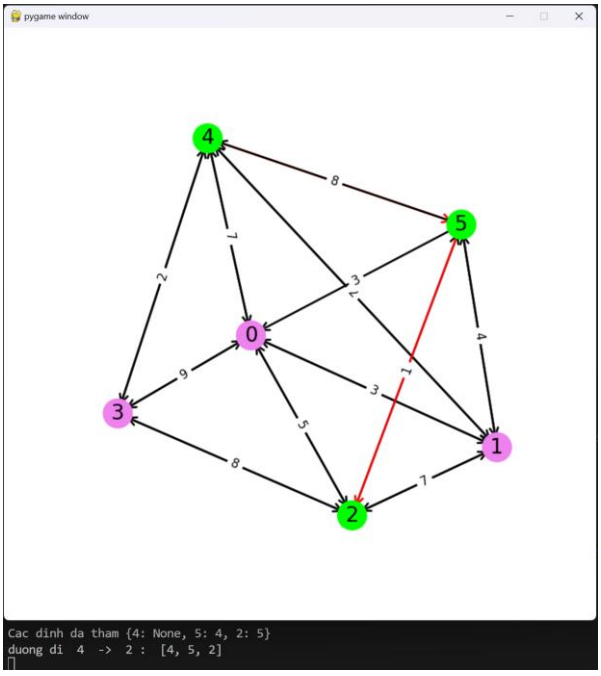


Figure 19: Greedy

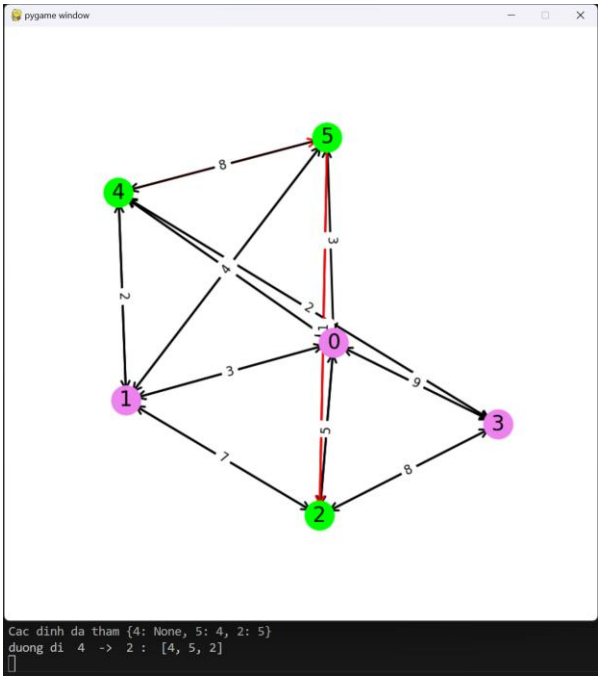


*Figure 20: A**

3.5. Input05

```
2 1
0 0 9 3 0 0 9
0 0 8 5 5 7 6
6 9 0 4 2 5 6
5 1 2 0 2 4 7
0 3 6 8 0 6 6
0 6 3 9 9 0 4
5 1 8 1 5 3 0
```

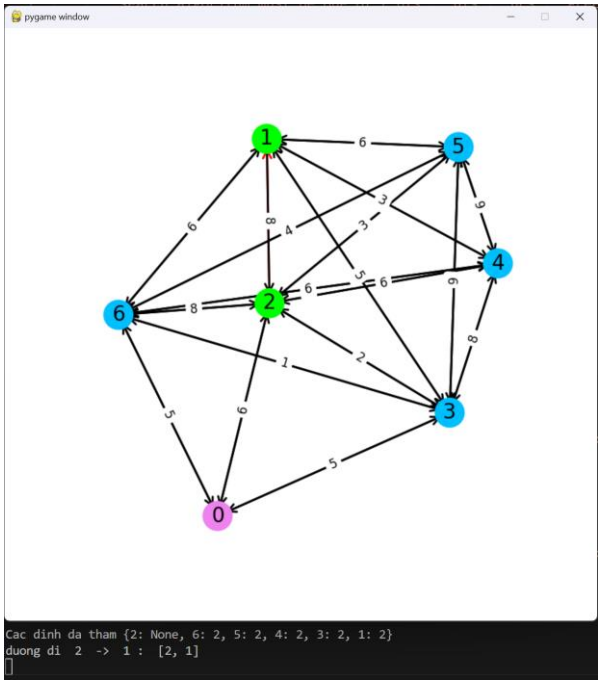


Figure 21: BFS

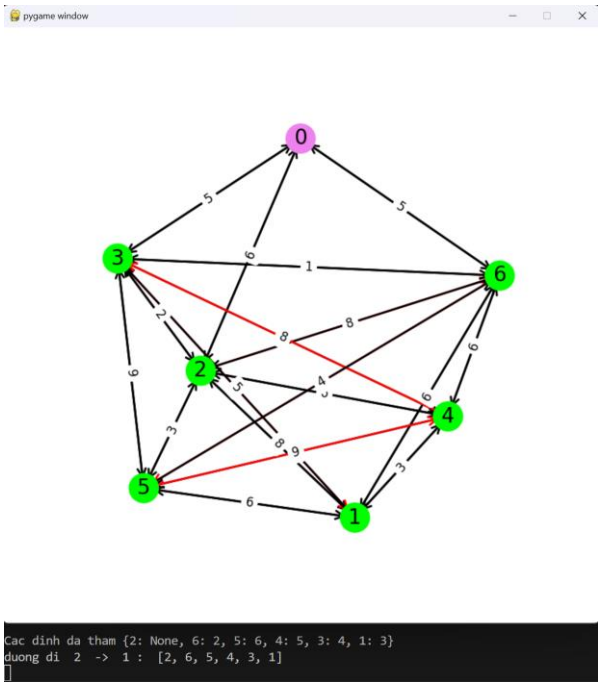


Figure 22: DFS

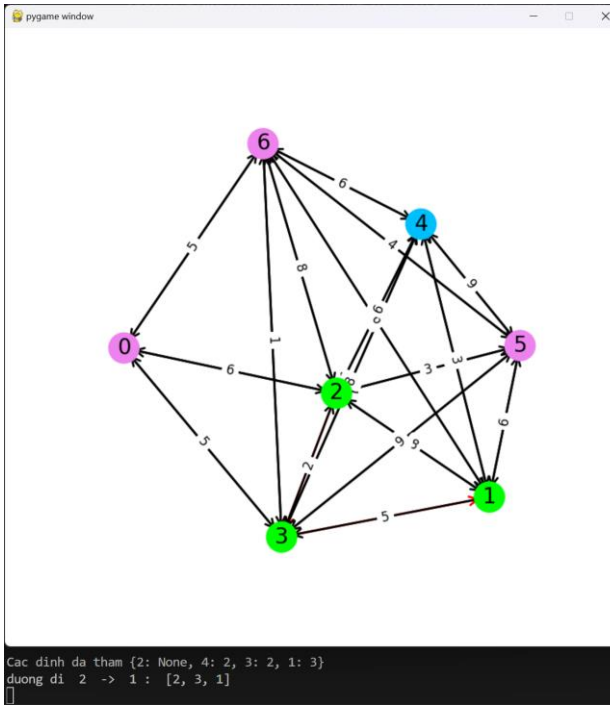


Figure 23: UCS

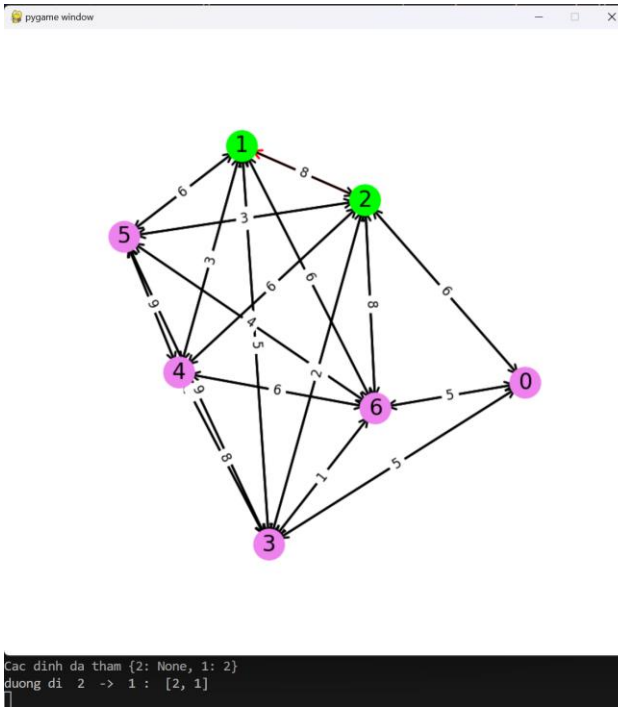


Figure 24: Greedy

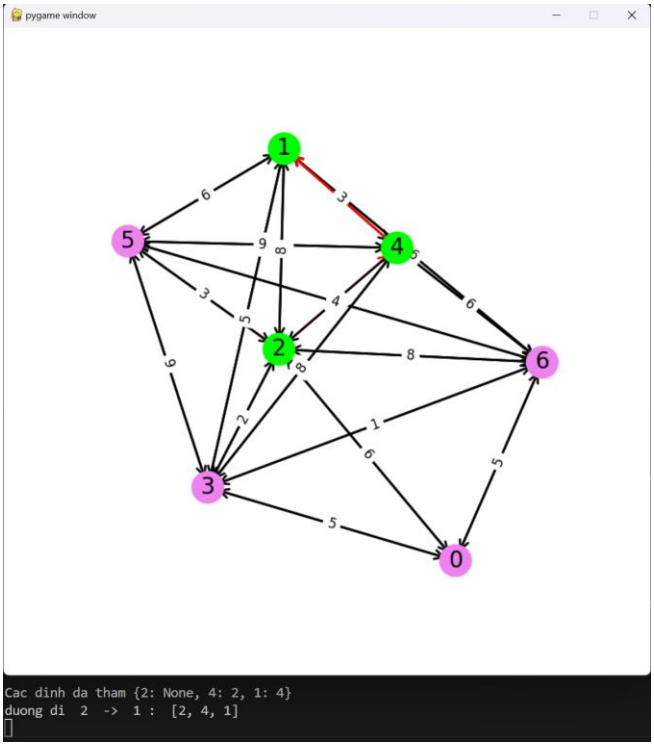


*Figure 25: A**

4. Lý thuyết cơ bản của các thuật toán

4.1. Breadth First Search (BFS)

4.1.1. Khái niệm

Breadth First Search (BFS) là là một thuật toán duyệt đồ thị cơ bản. Nó bắt đầu từ một nút gốc, sau đó duyệt qua tất cả các nút kề với nó. Sau khi tất cả các nút kề được duyệt, thuật toán sẽ tiếp tục duyệt đến các nút kề của những nút đó.

4.1.2. Độ phức tạp thuật toán

- **Thời gian:** Trong quá trình duyệt, BFS khám phá toàn bộ các đỉnh và các cạnh trong đồ thị. Vì vậy, độ phức tạp thời gian của BFS là O(V+E) (V là số lượng đỉnh, E là số lượng cạnh.
- **Không gian:** BFS sử dụng một hàng đợi để lưu trữ các đỉnh cần duyệt. Vậy nên, độ phức tạp về không gian của BFS là O(V) (V là số lượng đỉnh).

4.1.3. Thuộc tính

- **Tính đầy đủ:** BFS là một thuật toán đầy đủ, nghĩa là nó luôn luôn tìm ra một đường đi nếu nó có tồn tại.
- **Tính tối ưu:** BFS là thuật toán tối ưu nếu trọng số các cạnh đều bằng nhau hoặc đồ thị không có trong số thì BFS sẽ tìm ra đường đi ngắn nhất tới đích..

4.1.4. Triển khai thuật toán

```python
def BFS(matrix, start, end):

    # TODO:

    path=[] # lưu kết quả đường đi từ điểm bắt đầu đến đích
    visited={} #lưu các đỉnh đã thăm
    q = queue.Queue() #tạo danh sách hàng đợi lưu các đỉnh sẽ thăm
    q.put((start,None)) #thêm điểm bắt đầu vào danh sách hàng đợi và đỉnh đã thăm nó
    visited={start: None} # thêm điểm bắt đầu vào đỉnh đã thăm
    visit={start: None}# đánh dấu đã được duyệt trong danh sách hàng đợi

    while q:# vòng lặp kết thúc khi danh sách hàng đợi rộng
        current_node,front_node = q.get() #lấy đỉnh nằm đầu tiên của hàng đợi và đỉnh đã thăm nó
        visited[current_node] = front_node# lưu đỉnh đã thăm

        if current_node == end:# nếu như đỉnh đang duyệt giống với đích thì dừng vòng lặp
            break

        for neighbor in range(len(matrix[current_node]) - 1, -1, -1):# duyệt các đỉnh hàng xóm của đỉnh hiện tại
            if matrix[current_node][neighbor] != 0 and neighbor not in visit:# điều kiện để thêm vào danh sách hàng đợi
                q.put((neighbor,current_node))#thêm vào cách đỉnh cạnh nó vào danh sách hàng đợi
                visit[neighbor] = current_node  # Đánh các đỉnh chuẩn bị thăm
    # nếu tìm thấy đỉnh thì lưu đường đi vào path
    if end in visited:
        current_node = end
        while current_node is not None:
            path.append(current_node)
            current_node = visited[current_node]
        path.reverse()
    print("")
    print("Cac dinh da tham" ,visited)
    print("duong di ",start, " -> ", end,": ",path)
    return visited, path
```

4.2. Depth First Search (DFS)

4.2.1. Khái niệm

Depth First Search (DFS) là một thuật toán duyệt được sử dụng trong các cấu trúc dữ liệu dạng cây và đồ thị. Thuật toán này thường bắt đầu bằng cách khám phá nút sâu nhất trong biên (frontier). Bắt đầu từ nút gốc, thuật toán tiến hành tìm kiếm đến mức sâu nhất của cây tìm kiếm cho đến khi gặp các nút không có nút kế tiếp nào. Nếu gặp

nút có các nút kế tiếp chưa được mở rộng, thì thuật toán sẽ quay lui (backtrack) đến nút sâu nhất tiếp theo để khám phá các con đường thay thế.

4.2.2. Độ phức tạp thuật toán

- **Thời gian:** DFS sẽ thăm tất cả các đỉnh và cạnh của đồ thị. Mỗi đỉnh được thăm một lần và mỗi cạnh cũng được kiểm tra một lần nên độ phức tạp là O(V + E) (V là số lượng đỉnh, E là số cạnh).
- **Không gian:** DFS sử dụng cấu trung Stack để theo dõi các đỉnh truy cập nên độ phức tạp không gian là O(V) (V là số lượng đỉnh).

4.2.3. Thuộc tính

- **Tính hoàn chỉnh:** DFS không là thuật toán hoàn chỉnh nếu như đồ thì vô hạng hoặc có chu trình.
- **Tính tối ưu:** DFS không là thuật toán tới ưu dù cho DFS tìm thấy đường đi nhưng đường đi đó đảm bảo rằng tốt nhất hoặc ngắn nhất.

4.2.4. Triển khai thuật toán

```python
def DFS(matrix, start, end):

    path = []
    visited = {start: None}  # Lưu các đỉnh đã thăm
    stack = [(start, None)]  # Tạo ngăn xếp để lưu đỉnh

    while stack:
        current_node, previous = stack.pop()  # Lấy đỉnh cuối cùng trong ngăn xếp
        visited[current_node] = previous  # Lưu đỉnh cha

        if current_node == end:  # Kiểm tra nếu đã đến đích
            break

        # Duyệt các đỉnh hàng xóm của đỉnh hiện tại
        for neighbor in range(len(matrix[current_node])):
            if matrix[current_node][neighbor] != 0 and neighbor not in visited:
                stack.append((neighbor, current_node))  # Thêm đỉnh vào ngăn xếp

    # Nếu tìm thấy đỉnh đích, xây dựng đường đi
    if end in visited:
        current_node = end
        while current_node is not None:
            path.append(current_node)
            current_node = visited[current_node]
        path.reverse()  # Đảo ngược để có đường đi từ start đến end

    print("")
    print("Các đỉnh đã thăm:", visited)
    print("Đường đi từ", start, "->", end, ":", path)
    return visited, path
```

4.3. Uniform-Cost Search (UCS)

4.3.1. Khái niệm

Uniform-Cost Search là là một thuật toán tìm kiếm phổ biến được sử dụng trong trí tuệ nhân tạo (AI) để tìm đường đi có chi phí thấp nhất trong đồ thị. Đây là một biến thể của thuật toán Dijkstra và đặc biệt hữu ích khi tất cả các cạnh của đồ thị có trọng số khác nhau và mục tiêu là tìm đường đi có tổng chi phí nhỏ nhất từ nút bắt đầu đến nút đích.

4.3.2. Độ phức tạp thuật toán

- **Thời gian**: cấp số mũ (b^d).
- **Không gian**: cấp số mũ (b^d).

  + Trong đó: b là số đỉnh con mà mỗi đỉnh có thể phát sinh, d là chiều sau của nút mục tiêu.

4.3.3. Thuộc tính

- **Tính hoàn chỉnh:** UCS là thuật toán hoàn chỉnh vì UCS đảm bảo rằng nó sẽ duyệt tất cả các nút có chi phí từ thấp đến lớn.
- **Tính tối ưu:** UCS là thuật toán tối ưu đảm bảo rằng sẽ tìm thấy đường đi có chi phí thấp nhất tới đích nếu có.

4.3.4. Triển khai thuật toán

```python
def UCS(matrix, start, end):
    # TODO:
    path = []
    visited = {start: None}
    priority_queue = queue.PriorityQueue()
    priority_queue.put((0, (start,None)))  # Hàng đợi lưu (chi phí, (đỉnh, đỉnh cha))

    while not priority_queue.empty():

        cost, (current_node,front_node) = priority_queue.get()# Lấy đỉnh có chi phí nhỏ nhất
        visited[current_node] = front_node  # Lưu đỉnh cha của current_node


        if current_node == end: # Kiểm tra nếu đã đến đỉnh đích
            break

         # Duyệt qua các đỉnh kề của current_node
        for neighbor in range(len(matrix[current_node]) - 1, -1, -1):
            if matrix[current_node][neighbor] != 0 and neighbor not in visited:# kiểm tra đỉnh đã thăm và có đường đi tới đỉnh không.
                new_cost = cost + matrix[current_node][neighbor]# tính phí đi từ điểm bắt đầu đến đỉnh kề.
                priority_queue.put((new_cost, (neighbor,current_node)))# thêm các định kề vào danh sách hàng đợi
    # Nếu tìm thấy đỉnh đích, xây dựng đường đi
    if end in visited:
        current_node = end
        while current_node is not None:
            path.append(current_node)
            current_node = visited[current_node]
        path.reverse() # Đảo ngược để có đường đi từ start đến end

    print("")
    print("Cac dinh da tham" ,visited)
    print("duong di ",start, " -> ", end,": ",path)
    return visited, path
```

4.4. Greedy Best First Search (GBFS)

4.4.1. Khái niệm.

Greedy Best-First Search (GBFS) là một thuật toán tìm kiếm tham lam, nó cố gắng tìm đường đi có vẻ triển vọng nhất từ điểm bắt đầu đến đích. Nó ưu tiên các đường đi có vẻ triển vọng hơn, nhưng không đảm bảo đó là đường đi ngắn nhất. Thuật toán này hoạt động bằng cách đánh giá chi phí của mỗi đường đi có thể và mở rộng đường đi với chi phí thấp nhất. Quá trình này lặp lại cho đến khi tìm thấy đích.

4.4.2. Độ phức tạp thuật toán.

- **Thời gian:** cấp số mũ (b^m). Thực thi nhanh nếu heurictic hợp lý.
- **Không gian:** cấp số mũ (b^m).

4.4.3. Thuộc tính.

- **Tính tối ưu:** GBFS là thuật toán không tối ưu vì không xét tổng chi phí đường đi, nên nó có thể không tìm được đường ngắn nhất đến đích.

- **Tính hoàn chỉnh:** GBFS là thuật toán không hoàn chỉnh trong một số trường hợp: Với các đồ thị có vô hạn đường đi hoặc khi heuristic có thể gây sai lệch, GBFS có thể không tìm được lời giải.

4.4.4. Triển khai thuật toán.

```python
def GBFS(matrix, start, end):
    # TODO:
    path = []
    visited = {start: None}
    pq = queue.PriorityQueue()
    pq.put((0, (start,None)))# Hàng đợi lưu (trọng số của cạnh, (đỉnh, đỉnh cha))
    while not pq.empty():
        _, (current_node,front_node) = pq.get()# Lấy đỉnh có trọng số nhỏ nhất
        visited[current_node] = front_node #lưu đỉnh cha của node
        if current_node == end: #kiểm tra đã tới đích chưa
            break
        # duyệt các đỉnh kề của node
        for neighbor in range(len(matrix[current_node]) - 1, -1, -1):
            weight = matrix[current_node][neighbor] #lấy cân nặng từ node đến đỉnh con
            if weight and neighbor not in visited:  #kiểm tra đỉnh con đã thăm và có đường đi tới đỉnh con không
                if(neighbor == end):# kiểm tra đỉnh kề có phải đỉnh đích không
                    weight = 0
                pq.put((weight, (neighbor,current_node)))# thêm đỉnh kề vào danh sách hàng đợi
    # Nếu tìm thấy đỉnh đích, xây dựng đường đi
    if end in visited:
        current_node = end
        while current_node is not None:
            path.append(current_node)
            current_node = visited[current_node]
        path.reverse()# Đảo ngược để có đường đi từ start đến end
    print("")
    print("Cac dinh da tham" ,visited)
    print("duong di ",start, " -> ", end,": ",path)
    return visited, path
```

4.5. A*

4.5.1. Khái niệm.

A* là giải thuật tìm kiếm trong đồ thị, tìm đường đi từ một đỉnh hiện tại đến đỉnh đích có sử dụng hàm để ước lượng khoảng cách hay còn gọi là hàm Heuristic. Từ trạng thái hiện tại A* xây dựng tất cả các đường đi có thể đi dùng hàm ước lược khoảng cách (hàm Heuristic) để đánh giá đường đi tốt nhất có thể đi. Tùy theo mỗi dạng bài khác nhau mà hàm Heuristic sẽ được đánh giá khác nhau. A* luôn tìm được đường đi ngắn nhất nếu tồn tại đường đi như thế. A* lưu giữ một tập các đường đi qua đồ thị, từ đỉnh bắt đầu đến đỉnh kết thúc, tập các đỉnh có thể đi tiếp được lưu trong danh sách hàng đợi

ưu tiên. Thứ tự ưu tiên cho một đường đi đươc quyết định bởi hàm Heuristic được đánh giá f(x) = g(x) + h(x) (g(x) là chi chi phí của đường đi từ điểm xuất phát cho đến thời điểm hiện tại, h(x) là hàm ước lượng chi phí từ đỉnh hiện tại đến đỉnh đích).

4.5.2. Độ phức tạp thuật toán.

- **Thời gian:** Độ phức tạp về thời gian của thuật toán tìm kiếm A * phụ thuộc vào hàm heuristic và số lượng nút được mở rộng theo cấp số nhân với độ sâu của nghiệm d. Vì vậy, độ phức tạp thời gian là O (b^d), trong đó b là hệ số phân nhánh.
- **Không gian:** Độ phức tạp không gian của thuật toán tìm kiếm A * là O (b^d).

4.5.3. Thuộc tính.

- **Tính hoàn chỉnh:** A* là thuật toán hoàn chỉnh đảm bảo tìm thấy giải pháp nếu có với điều kiện là hệ số phân nhánh là hữu hạn và chí phí cho mọi di chuyển là cố định.
- **Tính tối ưu:** A* là thuật toán tối ưu đảm bảo tình thấy đường đi có chi phí thấp nhất nếu như thỏa 2 điều kiện sau:
    - Có thể chấp nhận: điều kiện đầu tiên yêu cầu để tối ưu là h (n) phải là một heuristic có thể chấp nhận được cho tìm kiếm cây A *. Một heuristic có thể chấp nhận được là bản chất lạc quan.
    - Tính nhất quán: Điều kiện bắt buộc thứ hai là tính nhất quán chỉ dành cho tìm kiếm đồ thị A *.

4.5.4. Triển khai thuật toán.

```python
def heuristic(star,end,pos): # tính heuristic
    x1,y1=pos[star]
    x2,y2=pos[end]
    return ((x1-x2)**2+ (y1-y2)**2)**0.5

def Astar(matrix, start, end, pos):
    # TODO:
    path=[]
    visited={start: None}
    pq=queue.PriorityQueue()
    pq.put((heuristic(start,end,pos),(start,None)))# Hàng đợi lưu (f(x)= h(x) + g(x), (đỉnh, đỉnh cha))
                                                   # h(x) là hàm ước lượng khoảng cách từ điểm tới đích
                                                   # g(x) là chi phi đi từ đỉnh bắt đầu đến đỉnh
    while not pq.empty():
        cost,(current_node,front_node) = pq.get()# lấy đỉnh có f(x) nhỏ nhất có trong danh sách
        visited[current_node] =front_node # lưu nút cha của đỉnh.

        if current_node == end:#kiểm tra đỉnh có phải là đích không
            break
        # duyệt có đỉnh kề của đỉnh
        for neighbor in range(len(matrix[current_node]) - 1, -1, -1):
            # kiểm tra các đỉnh kề đã được thăm hay chưa có đường đi tới đỉnh kề không
            if matrix[current_node][neighbor]!=0 and neighbor not in visited:
                #tính f(x) của đỉnh kề
                new_cost = cost-heuristic(current_node,end,pos)+matrix[current_node][neighbor]+heuristic(neighbor,end,pos)
                pq.put((new_cost,(neighbor,current_node))) # thêm đỉnh kề vào danh sách hàng đợi ưu tiên

     # Nếu tìm thấy đỉnh đích, xây dựng đường đi
    if end in visited:
        current_node = end
        while current_node is not None:
            path.append(current_node)
            current_node = visited[current_node]
        path.reverse()# Đảo ngược để có đường đi từ start đến end
    print("")
    print("Cac dinh da tham" ,visited)
    print("duong di ",start, " -> ", end,": ",path)
    return visited, path
```

5. So sánh giữ UCS và A*

	UCS	A*
Giống nhau	+ Hai thuật toán đều tìm ra đường đi có chi phí tốt nhất. + Độ phức tạp về không gian và thời gian đều là cấp số mũ.	
Khác nhau	+ Dựa vào chi phí thực tế từ điểm xuất phát đến các trạng thái khác.	+ Dựa vào tổng chi phí thứ tế từ điểm xuất phát đến các trạng thái khác và chi phí dự đoán từ các trạng thái khác đến đích theo công thức f(x) = g(x) + h(x).
	+ Dựa vào chi phí thấp nhất để định hướng đường đi.	+ Dựa vào heuristic để định hướng tìm kiếm những điểm gần đích.
	+ Thích hợp cho các bài toán không có thông tin gì về đỉnh đích.	+ Nếu bảng heuristic hợp lý thì sẽ tìm thấy điểm đích nhanh hơn.

Kết luận: UCS chỉ dựa vào chi phí thực từ điểm bắt đầu, trong khi A* dùng thêm hàm heuristic để hướng dẫn tìm kiếm, giúp tìm đích hiệu quả hơn khi có thông tin về vị trí mục tiêu. Nếu hàm heuristic tốt thì A* có xu hướng nhanh hơn UCS trong các không gian tìm kiếm lớn.

6. Tài liệu tham khảo

Các thuật toán Informed Search Algorithms - w3seo

Best First Search (Informed Search) - GeeksforGeeks

artificial intelligence - What is the difference between uniform-cost search and best-first search methods? - Stack Overflow

Greedy Best first search algorithm - GeeksforGeeks

Uniform-Cost Search (Dijkstra for large Graphs) - GeeksforGeeks

ChatGPT

HẾT