BÁO CÁO ĐỒ ÁN CUỐI KÌ
CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT - 22CTT5

SEARCH ALGORITHMS
DFS - BFS - ASTAR - DIJKTRA

Giảng viên hướng dẫn

Ths. NGUYỄN BẢO LONG

Thành viên nhóm

Võ Quốc Quang - 22120299

Hồ Quang Sang - 22120309

Mục lục

1 Thuật toán DFS **3**
1.1 Ý tưởng chung . . . 3
1.2 Mã giả thuật toán . . . 3
1.3 Phân tích thuật toán . . . 4
1.3.1 Tính Đầy Đủ (Completeness) . . . 4
1.3.2 2. Tính Tối Ưu (Optimality): . . . 4
1.3.3 Độ Phức Tạp (Complexity): . . . 5
1.4 Ví dụ . . . 5

2 Thuật toán BFS **6**
2.1 Ý tưởng chung . . . 6
2.2 Mã giả thuật toán . . . 6
2.3 Phân tích thuật toán . . . 7
2.3.1 Tính Đầy Đủ (Completeness) . . . 7
2.3.2 2. Tính Tối Ưu (Optimality): . . . 7
2.3.3 Độ Phức Tạp (Complexity): . . . 7
2.4 Ví dụ . . . 8

3 Thuật toán A-Star **9**
3.1 Ý tưởng chung . . . 9
3.2 Mã giả thuật toán . . . 9
3.3 Phân tích thuật toán . . . 11
3.3.1 Tính Đầy Đủ (Completeness) . . . 11
3.3.2 2. Tính Tối Ưu (Optimality): . . . 11
3.3.3 Độ Phức Tạp (Complexity): . . . 11
3.4 Ví dụ . . . 12

4 Thuật toán Dijkstra **12**

4.1 Ý tưởng chung 12
4.2 Mã giả thuật toán 13
4.3 Phân tích thuật toán 14
4.3.1 Tính Đầy Đủ (Completeness) 14
4.3.2 2. Tính Tối Ưu (Optimality): 14
4.3.3 Độ Phức Tạp (Complexity): 15
4.4 Ví dụ 15

5 Tài liệu tham khảo **16**

1 Thuật toán DFS

1.1 Ý tưởng chung

Thuật toán DFS (Depth First Search) được triển khai dựa trên các bước:

- Bắt Đầu từ Một Nút: Thuật toán bắt đầu từ một nút gốc và đi dọc theo mỗi nút kề cho đến khi không thể đi được(bị chặn) nữa rồi mới quay ngược lại nút kề gần nhất với điểm bị chặn.
- Sử Dụng Ngăn Xếp (Stack): DFS sử dụng một ngăn xếp để quản lý các nút cần xử lý. Nút hiện tại sẽ được đẩy vào ngăn xếp và nó sẽ tiếp tục đi sâu vào nút kề của nút hiện tại, mỗi nút kề sẽ được đẩy vào ngăn xếp để sử dụng khi di chuyển hoặc quay ngược lại nếu không còn đường đi.
- Backtracking: Khi đến một nút không có nút kề nào khác hoặc tất cả các nút kề đã được thăm, nó sẽ quay ngược lại (sử dụng các nút kề trong có ngăn xếp) để tìm đường đi khác.
- Đánh Dấu Các Nút Đã Thăm: Để tránh việc lặp lại những nút đã đi qua, mỗi nút được thăm sẽ được đánh dấu(sử dụng CTDL set). Khi một nút được lấy ra khỏi ngăn xếp để xử lý, nếu nó đã được đánh dấu là đã đi qua, thuật toán sẽ bỏ qua nó.
- Sử dụng vòng lặp: Thuật toán sẽ lặp lại các bước trên cho đến khi đi đến điểm đích hoặc ngăn xếp rỗng(không tìm được đường nào đến điểm đích).

1.2 Mã giả thuật toán

```
proceduce DFS(g: SearchSpace)
    // Khởi tạo stack cho các nút chưa xét và set cho các nút đã xét
    open_set = stack with start node
    closed_set = empty set

    // Tạo một dictionary để lưu trữ nút cha của mỗi nút
    father = dictionary with -1 for each node in grid_cells

    // Lặp cho đến khi stack trống
    while open_set is not empty
        // Lấy nút cuối cùng từ stack (DFS)
        current_id = pop from open_set

        // Bỏ qua nếu nút đã xét
        if current_id in closed_set THEN
            CONTINUE

        // Đánh dấu nút là đã xét
        ADD current_id to closed_set

        // Lấy nút hiện tại từ grid_cells
```

```
        current_node = g.grid_cells[current_id]

        // Kiểm tra nếu đạt đích
        if g.is_goal(current_node) THEN
            // Vẽ đường đi và kết thúc
            draw_path(g, father)
            return

        // Xét các nút kề
        for each neighbor_node in g.get_neighbors(current_node)
            // Bỏ qua nếu nút kề đã xét
            IF neighbor_node.id in closed_set THEN
                CONTINUE

            // Thêm nút kề vào stack
            ADD neighbor_node.id to open_set

            // Cập nhật nút cha cho nút kề
            father[neighbor_node.id] = current_id

    // Nếu không tìm thấy đường đi
    RAISE error 'not implemented'
end proceduce
```

1.3 Phân tích thuật toán

1.3.1 Tính Đầy Đủ (Completeness)

Thuật toán DFS được cài đặt sở hữu cơ chế không đi lại các đỉnh đã đi qua nên thuật toán này là đầy đủ, nghĩa là nó sẽ tìm ra một đường đi nếu đường đi này thực sự tồn tại. Nếu không có cơ chế này, thuật toán DFS chỉ đầy đủ khi đồ thị là hữu hạn và không có chu trình và có thể không đầy đủ nếu đồ thị có chu trình.

1.3.2 2. Tính Tối Ưu (Optimality):

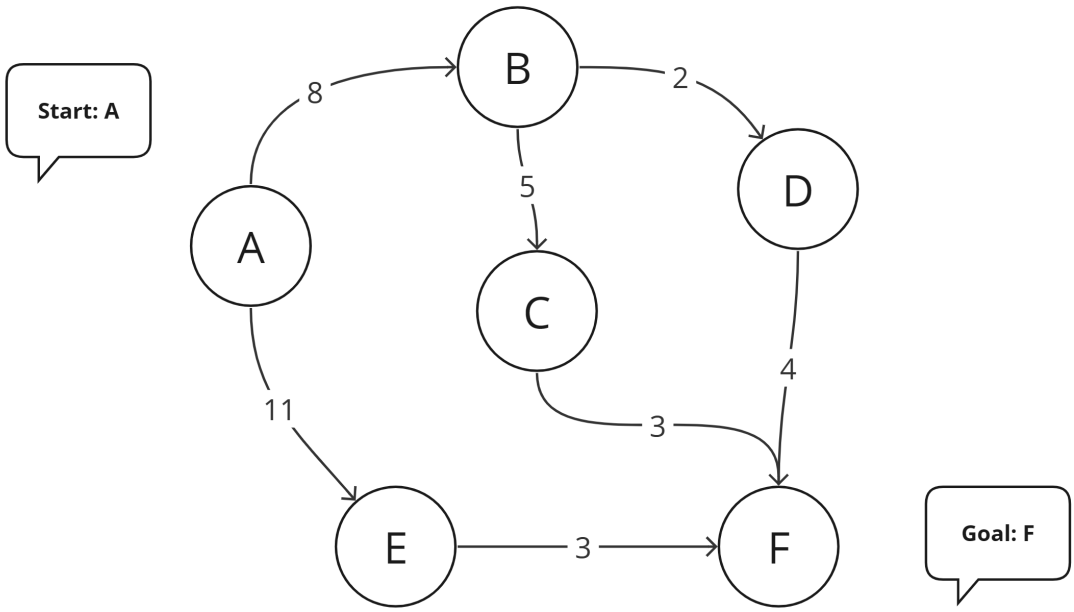
- Thuật toán DFS không đảm bảo tìm ra giải pháp tối ưu. Đặc biệt nếu mục tiêu là tìm đường đi ngắn nhất từ điểm bắt đầu tới điểm đích.

- DFS thường được xem là "thuật toán mù" vì nó không tính toán hoặc cân nhắc đến chi phí tìm kiếm. Nó chỉ đơn giản là theo đuổi một hướng cho đến khi không thể đi sâu hơn hoặc cho đến khi tìm thấy mục tiêu. Điều này có nghĩa là trong quá trình tìm kiếm, nó có thể bỏ qua các con đường ngắn hơn hoặc tối ưu hơn.

1.3.3 Độ Phức Tạp (Complexity):

- Độ Phức Tạp Thời Gian: O(V + E), trong đó V là số lượng đỉnh và E là số lượng cạnh trong đồ thị. Điều này phản ánh việc mỗi đỉnh và mỗi cạnh được xét một lần.
- Độ Phức Tạp Không Gian: O(V), trong đó V là số lượng đỉnh. Điều này do sử dụng ngăn xếp để lưu trữ các đỉnh trong quá trình duyệt. Trong trường hợp xấu nhất, toàn bộ đồ thị cần được lưu trong ngăn xếp.

1.4 Ví dụ



Current node	Stack	Visited
	A	
A	E, B	A
B	E, D, C	B
C	E, D, F	C
F	E, D	F

Đường đi: A-B-C-F

2 Thuật toán BFS

2.1 Ý tưởng chung

Thuật toán BFS (Breadth-First Search) được triển khai dựa trên các bước:

- Thuật toán BFS bắt đầu từ một nút gốc và đi theo chiều rộng, tức là nó sẽ khám phá tất cả các nút kề với nút gốc trước, sau đó mới chuyển sang các nút kề của các nút đã thăm.

- Sử Dụng Hàng Đợi (Queue): BFS sử dụng hàng đợi để quản lý các nút cần xử lý. Khi một nút được thăm, tất cả các nút kề chưa được thăm của nó sẽ được thêm vào cuối hàng đợi. Nút ở đầu hàng đợi sẽ được lấy ra và xử lý tiếp theo.

- Đánh Dấu Các Nút Đã Thăm: Để tránh việc lặp lại những nút đã đi qua, mỗi nút được thăm sẽ được đánh dấu(sử dụng CTDL set). Khi một nút được lấy ra khỏi ngăn xếp để xử lý, nếu nó đã được đánh dấu là đã đi qua, thuật toán sẽ bỏ qua nó.

- Sử dụng vòng lặp: Thuật toán sẽ lặp lại các bước trên cho đến khi đi đến điểm đích hoặc ngăn xếp rỗng(không tìm được đường nào đến điểm đích).

2.2 Mã giả thuật toán

```
proceduce BFS(g: SearchSpace)
    // Khởi tạo hàng đợi cho các nút chưa xét và set cho các nút đã xét
    open_set = queue with start node
    closed_set = empty set

    // Tạo một dictionary để lưu trữ nút cha của mỗi nút
    father = dictionary with -1 for each node in grid_cells

    // Lặp cho đến khi hàng đợi trống
    while open_set is not empty
        // Lấy nút đầu tiên từ hàng đợi (BFS)
        current_id = dequeue from open_set

        // Đánh dấu nút là đã xét
        ADD current_id to closed_set

        // Lấy nút hiện tại từ grid_cells
        current_node = g.grid_cells[current_id]

        // Kiểm tra nếu đạt đích
        if g.is_goal(current_node) THEN
            // Vẽ đường đi và kết thúc
            draw_path(g, father)
            return
```

```
        // Đánh dấu nút hiện tại (trừ start và goal)

        // Xét các nút kề
        for each neighbor_node in g.get_neighbors(current_node)
            // Bỏ qua nếu nút kề đã xét
            if neighbor_node.id in closed_set THEN
                CONTINUE

            // Thêm nút kề vào hàng đợi
            enqueue neighbor_node.id in open_set

            // Đánh dấu nút kề là đã xét
            ADD neighbor_node.id to closed_set

            // Cập nhật nút cha cho nút kề
            father[neighbor_node.id] = current_id

    // Nếu không tìm thấy đường đi
    RAISE error 'not implemented'
end proceduce
```

2.3 Phân tích thuật toán

2.3.1 Tính Đầy Đủ (Completeness)

- Thuật toán BFS là đầy đủ. Nó đảm bảo tìm thấy một giải pháp nếu có tồn tại.
- Điều này được đảm bảo bởi cách BFS đi qua tất cả các nút ở cùng 1 bậc (đều là nút kề của các nút cùng bậc trước đó) trước khi chuyển sang bậc tiếp theo. Do đó, nếu có một đường đi từ nút gốc đến nút đích, BFS sẽ tìm thấy nó.

2.3.2 2. Tính Tối Ưu (Optimality):

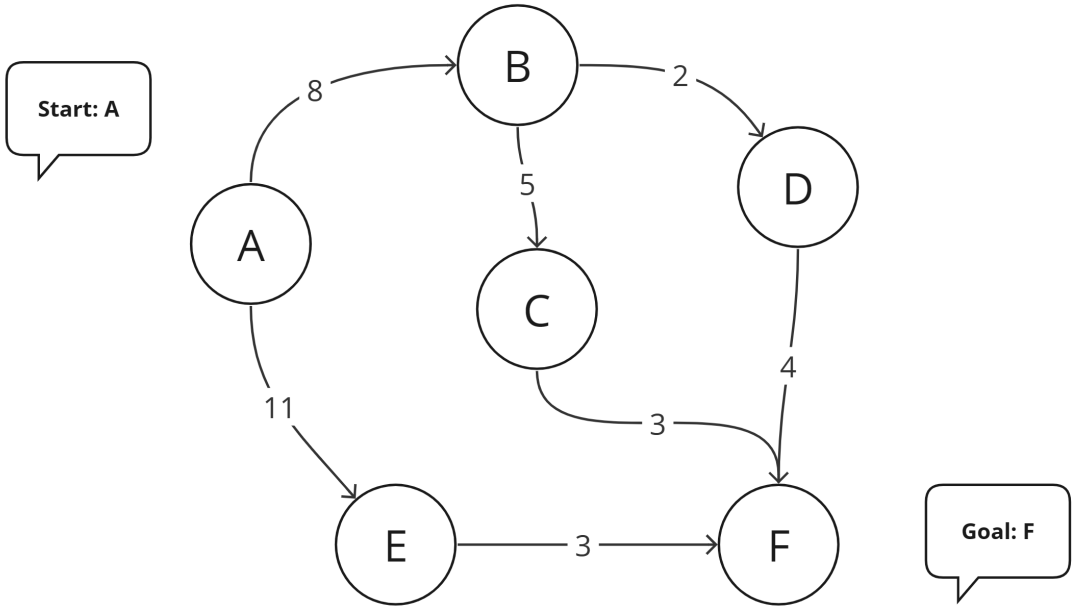
- Trong trường hợp mỗi bước di chuyển có cùng chi phí, BFS đảm bảo tìm ra giải pháp tối ưu, tức là đường đi ngắn nhất từ nút gốc đến nút đích. Ở đây ta xét mỗi bước di chuyển sang các nút kề đều có chi phí là 1.
- Trong trường hợp chi phí cho mỗi bước di chuyển là khác nhau, BFS không đảm bảo tìm ra giải pháp tối ưu.

2.3.3 Độ Phức Tạp (Complexity):

- Độ Phức Tạp Thời Gian: O(V + E), trong đó V là số lượng đỉnh và E là số lượng cạnh trong đồ thị. Điều này bởi vì mỗi đỉnh và mỗi cạnh được xem xét ít nhất một lần.

- Độ Phức Tạp Không Gian: O(V), do cần lưu trữ thông tin về các đỉnh đã được thăm. Trong trường hợp tệ nhất, cần lưu trữ thông tin cho tất cả các đỉnh.

2.4 Ví dụ



Current node	Queue	Visited
	A	
A	E, B	A
B	E, D, C	B
E	D, C, F	E
D	C,F,F	D
C	F,F,F	C
F	F,F	F

Đường đi: A-B-E-D-C-F

3 Thuật toán A-Star

3.1 Ý tưởng chung

- Khởi tạo một dictionary lưu khoảng cách từ nút gốc tới nút hiện tại(cost) và tạo 1 hàm heuristic phù hợp để ước lượng chi phí từ nút hiện tại tới nút đích.

- Thuật toán A-Star bắt đầu từ một nút gốc. Đặt giá trị cost(start) là 0, vì nó là khoảng cách từ nút xuất phát đến chính nó. Sử dụng hàm heuristic để ước lượng g(start), là chi phí từ nút start đến nút đích. Tính f(start) = cost(start) + g(start) và đặt nút start vào hàng đợi ưu tiên dựa trên giá trị f.

- Sử Dụng Hàng Đợi Ưu Tiên (Priority Queue): Chọn current node từ hàng đợi có giá trị f nhỏ nhất. Nếu current node là nút đích, quá trình tìm kiếm kết thúc và trả về đường đi tìm được. Nếu không, mở rộng current node bằng cách xét tất cả các nút lân cận của nó.

- Đối với mỗi nút lân cận (neighbor) của current node: Tính cost(neighbor) mới, là tổng chi phí từ nút gốc đến neighbor qua current node (ở đây ta cho chi phí giữa mỗi nút với nhau đều là 1). Sử dụng hàm heuristic để ước lượng g(neighbor) sau đó tính giá trị f(neighbor). Nếu neighbor chưa được đi qua hoặc giá trị f mới nhỏ hơn giá trị f trước đó, cập nhật f, và đặt current node làm cha của neighbor (để truy vết lại lộ trình).

- Sử dụng vòng lặp: Thuật toán sẽ lặp lại các bước trên cho đến khi đi đến điểm đích hoặc ngăn xếp rỗng(không tìm được đường nào đến điểm đích).

3.2 Mã giả thuật toán

```
proceduce AStar(g: SearchSpace)
    // Khởi tạo hàng đợi ưu tiên cho các nút chưa xét và set cho các nút đã xét
    open_set = PrioQueue with start node
    closed_set = empty set

    // Tạo dictionaries để lưu trữ nút cha và chi phí tới mỗi nút
    father = dictionary with -1 for each node in grid_cells
    cost = dictionary with infinity for each node, except start node set to 0

    // Lặp cho đến khi hàng đợi ưu tiên trống
    while open_set is not empty
        // Lấy nút có độ ưu tiên cao nhất
        current_id = pop from open_set

        // Bỏ qua nếu nút đã xét
        if current_id in closed_set THEN
            CONTINUE

        // Đánh dấu nút là đã xét
```

```
        ADD current_id to closed_set

        // Lấy nút hiện tại từ grid_cells
        current_node = g.grid_cells[current_id]

        // Kiểm tra nếu đạt đích
        if g.is_goal(current_node) THEN
            // Vẽ đường đi và kết thúc
            draw_path(g, father)
            return

        // Đánh dấu nút hiện tại (trừ start và goal)

        // Xét các nút kề
        for each neighbor_node in g.get_neighbors(current_node)
            // Bỏ qua nếu nút kề đã xét
            if neighbor_node.id in closed_set THEN
                CONTINUE

            // Tính toán chi phí mới cho nút kề
            neighbor_cost = Euclidean_distance(current_node, neighbor_node)
            + cost[current_id]

            // Bỏ qua nếu chi phí mới không tốt hơn
            if neighbor_cost >= cost[neighbor_node.id] THEN
                CONTINUE

            // Tính toán heuristic (đánh giá tổng chi phí)
            heuristic = Euclidean_distance(neighbor_node, g.goal) + neighbor_cost

            // Thêm nút kề vào hàng đợi ưu tiên
            open_set.push(heuristic, neighbor_node.id)

            // Cập nhật nút cha và chi phí cho nút kề
            father[neighbor_node.id] = current_id
            cost[neighbor_node.id] = neighbor_cost

    // Nếu không tìm thấy đường đi
    RAISE error 'not implemented'
END FUNCTION
```

3.3 Phân tích thuật toán

3.3.1 Tính Đầy Đủ (Completeness)

- Thuật toán A-Star là đầy đủ. Nó đảm bảo tìm thấy một giải pháp nếu có tồn tại.

- Tính đầy đủ này phụ thuộc vào việc hàng đợi ưu tiên được xử lý đúng cách và giả định rằng chi phí di chuyển giữa các nút không âm.

3.3.2 2. Tính Tối Ưu (Optimality):

- Trong trường hợp hàm heuristic được chọn là admissible (không bao giờ ước lượng quá cao chi phí thực tế), Thuật toán A-Star đảm bảo tìm thấy đường đi ngắn nhất từ điểm gốc tới điểm đích.

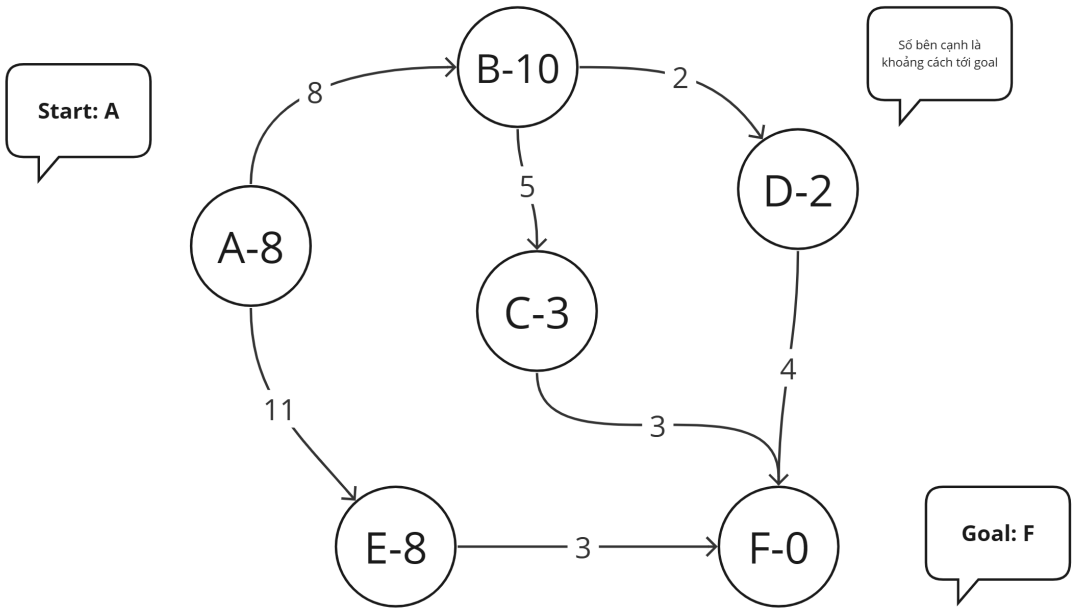
- Điều này xảy ra bởi vì A-Star luôn xét đến nút có tổng chi phí ước lượng f(n) thấp nhất.

3.3.3 Độ Phức Tạp (Complexity):

- Độ Phức Tạp Thời Gian: phụ thuộc nhiều vào cách chọn hàm heuristic. Trong trường hợp tốt nhất, nếu hàm heuristic rất chính xác thì độ phức tạp thời gian có thể đạt tới O(V) . Trong trường hợp xấu nhất thì độ phức tạp thời gian là O((V+E)*logV)

- Độ Phức Tạp Không Gian: O(V), do cần lưu trữ thông tin về các đỉnh đã được thăm. Trong trường hợp tệ nhất, cần lưu trữ thông tin cho tất cả các đỉnh.

3.4 Ví dụ



A	B	C	D	E	F
8*	∞	∞	∞	∞	∞
-	(16;A)*	∞	∞	(19;A)	∞
-	-	(31;B)	(28;B)	(19;A)*	∞
-	-	(31;B)	(28;B)	-	(22;E)*

Đường đi: A - B - E - F

4 Thuật toán Dijkstra

4.1 Ý tưởng chung

Thuật toán Dijkstra được triển khai dựa trên các bước:

- Thuật toán Dijkstra bắt đầu từ một nút gốc và khởi tạo khoảng cách từ đỉnh nguồn đến tất cả các đỉnh khác là vô cùng. Khoảng cách từ đỉnh nguồn đến chính nó là 0.

- Sử Dụng Hàng Đợi Ưu Tiên (Priority Queue): Bằng cách lựa chọn đỉnh chưa được đi qua ở mỗi bước mà có khoảng cách tối thiểu từ đỉnh nguồn. Đỉnh này được coi là "đỉnh hiện tại".
- Cập Nhật Khoảng Cách: Thuật toán sau đó xét tất cả các đỉnh lân cận của "đỉnh hiện tại" và cập nhật khoảng cách của chúng. Nếu khoảng cách từ đỉnh nguồn đến một đỉnh lân cận qua "đỉnh hiện tại" ngắn hơn khoảng cách hiện tại đến đỉnh lân cận đó, thuật toán sẽ cập nhật khoảng cách mới.
- Đánh Dấu Đỉnh: Sau khi đã xét tất cả các đỉnh lân cận, "đỉnh hiện tại" được đánh dấu là đã được xử lý, nghĩa là khoảng cách từ đỉnh nguồn đến đỉnh này đã là ngắn nhất và sẽ không được cập nhật nữa.
- Sử dụng vòng lặp: Thuật toán sẽ lặp lại các bước trên cho đến khi tất cả các đỉnh trong đồ thị đều đã được xử lý.

4.2 Mã giả thuật toán

```
proceduce Dijkstra(g: SearchSpace)
    // Khởi tạo hàng đợi ưu tiên cho các nút chưa xét và set cho các nút đã xét
    open_set = PrioQueue with start node
    closed_set = empty set

    // Tạo dictionaries để lưu trữ nút cha và chi phí tới mỗi nút
    father = dictionary with -1 for each node in grid_cells
    cost = dictionary with infinity for each node, except start node set to 0

    // Lặp cho đến khi hàng đợi ưu tiên trống
    while open_set is not empty
        // Lấy nút có chi phí thấp nhất
        current_id = pop from open_set

        // Bỏ qua nếu nút đã xét
        if current_id in closed_set THEN
            CONTINUE

        // Đánh dấu nút là đã xét
        ADD current_id to closed_set

        // Lấy nút hiện tại từ grid_cells
        current_node = g.grid_cells[current_id]

        // Kiểm tra nếu đạt đích
        if g.is_goal(current_node) THEN
            // Vẽ đường đi và kết thúc
            draw_path(g, father)
            return
```

```
        // Đánh dấu nút hiện tại (trừ start và goal)

        // Xét các nút kề
        for each neighbor_node in g.get_neighbors(current_node)
            // Tính toán chi phí mới cho nút kề
            neighbor_cost = cost[current_id] +
            Euclidean_distance(current_node, neighbor_node)

            // Bỏ qua nếu nút kề đã xét hoặc chi phí mới không tốt hơn
            if neighbor_node.id in closed_set OR neighbor_cost
            >= cost[neighbor_node.id] THEN
                CONTINUE

            // Thêm nút kề vào hàng đợi ưu tiên
            open_set.push(neighbor_cost, neighbor_node.id)

            // Cập nhật nút cha và chi phí cho nút kề
            father[neighbor_node.id] = current_id
            cost[neighbor_node.id] = neighbor_cost

    // Nếu không tìm thấy đường đi
    RAISE error 'not implemented'
end proceduce
```

4.3 Phân tích thuật toán

4.3.1 Tính Đầy Đủ (Completeness)

- Thuật toán Dijkstra là đầy đủ trong đồ thị có trọng số không âm. Nó sẽ tìm được một lộ trình từ điểm xuất phát đến mọi điểm khác trong đồ thị (nếu tồn tại đường đi).

- Thuật toán sẽ xét lần lượt từng đỉnh, cập nhật khoảng cách ngắn nhất đến các đỉnh lân cận chưa được xét. Quá trình này tiếp tục cho đến khi tất cả các đỉnh trong đồ thị đều đã được xét. Điều này đảm bảo rằng mọi đường đi khả thi đều được xem xét.

4.3.2 2. Tính Tối Ưu (Optimality):

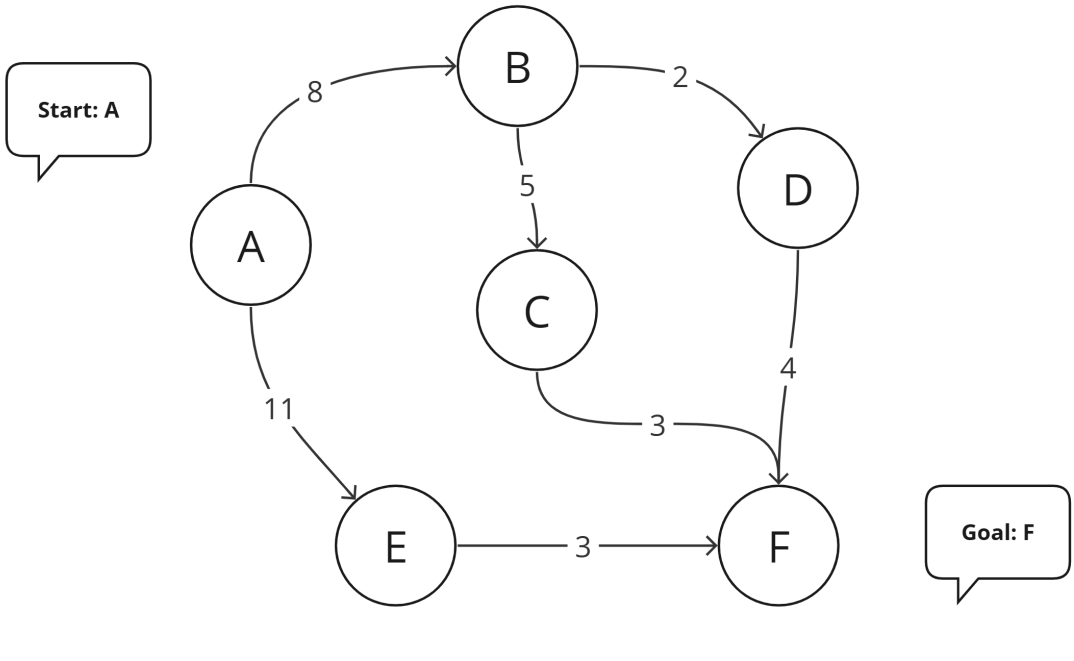
- Thuật toán Dijkstra đảm bảo tìm ra lộ trình ngắn nhất từ một nút đến tất cả các nút khác trong đồ thị có trọng số không âm.

- Tính tối ưu này đạt được nhờ việc thuật toán liên tục chọn đỉnh có khoảng cách ước lượng ngắn nhất từ nút nguồn, đồng thời cập nhật liên tục khoảng cách của các nút lân cận dựa trên khoảng cách đã tìm được.

4.3.3 Độ Phức Tạp (Complexity):

- Độ Phức Tạp Thời Gian: O((V+E)*logV), trong đó V là số lượng đỉnh và E là số lượng cạnh trong đồ thị. Điều này đến từ việc mỗi đỉnh được xử lý một lần (O(V*logV)) và mỗi cạnh được xem xét một lần trong quá trình cập nhật khoảng cách (O(E*logV)).
- Độ Phức Tạp Không Gian: O(V), do cần lưu trữ thông tin khoảng cách và trạng thái của mỗi đỉnh.

4.4 Ví dụ



A	B	C	D	E	F
0*	∞	∞	∞	∞	∞
-	(8;A)*	∞	∞	(11;A)	∞
-	-	(13;B)	(10;B)*	(11;A)	∞
-	-	(13;B)	-	(11;A)	(14;D)*

Đường đi: A - B - D - F

5 Tài liệu tham khảo

Depth First Search or DFS for a Graph
Depth-first search
Breadth First Search or BFS for a Graph
Breadth-first search A* Search Algorithm
A* search algorithm
A* search algorithm
How to find Shortest Paths from Source to all Vertices using Dijkstra's Algorithm
Dijkstra's algorithm