Tìm đường đi ngắn nhất trên đồ thị

Nhóm sinh viên:

Nguyễn Duy Thành - 20102737
Vũ Văn Hiệp - 20101545

Nội dung chính

1. Phát biểu bài toán
2. Các thuật toán thông dụng tìm đường đi ngắn nhất trên đồ thị

1. Phát biểu bài toán

❖ Các khái niệm cơ bản:

- Xét đồ thị G=(V, E), nếu **P = $v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k$** là một đường đi trên G thì độ dài của nó là tổng các trọng số trên các cung của nó.
- Đường đi ngắn nhất từ đỉnh u đến đỉnh v là đường đi có độ dài ngắn nhất trong số các đường đi từ u đến v, ký hiệu là $\delta(u, v)$.

1. Phát biểu bài toán (tiếp)

- Phát biểu bài toán tổng quát:

- Trên đồ thị liên thông G=(V, E), tìm đường đi có độ dài nhỏ nhất từ một đỉnh đến một đỉnh $s \in V$ đến một đỉnh đích $t \in V$. Đường đi như vậy gọi là đường đi ngắn nhất từ s đến t, độ dài của nó ký hiệu là d(s, t).

1. Phát biểu bài toán (tiếp)

❖ Nhận xét:

- Nếu không tồn tại đường đi từ s đến t thì ta coi $d(s, t)=\infty$.
- Nếu mọi chu trình trên đồ thị đều có độ dài dương thì đường đi ngắn nhất không có đỉnh nào lặp lại (đường đi cơ bản).
- Nếu đồ thị có chu trình âm thì không thể xác định khoảng cách giữa một số cặp đỉnh.
- Đường đi ngắn nhất luôn có thể tìm được trong số các đường đi đơn.

1. Phát biểu bài toán (tiếp)

- Mọi đường đi ngắn nhất đều có không quá n-1 cạnh với n là số đỉnh.
- Ta có thể dùng tính chất "Mọi đường đi con của đường đi ngắn nhất cũng là đường đi ngắn nhất" để chứng minh tính đúng đắn của một thuật toán tìm đường đi ngắn nhất.
- Giả sử P là đường đi ngắn nhất từ s đến t: P=s $\rightarrow$ ... $\rightarrow$ u $\rightarrow$ ... $\rightarrow$ t, khi đó:

$$\delta(s, t) = \delta(s, u) + \delta(u, t)$$

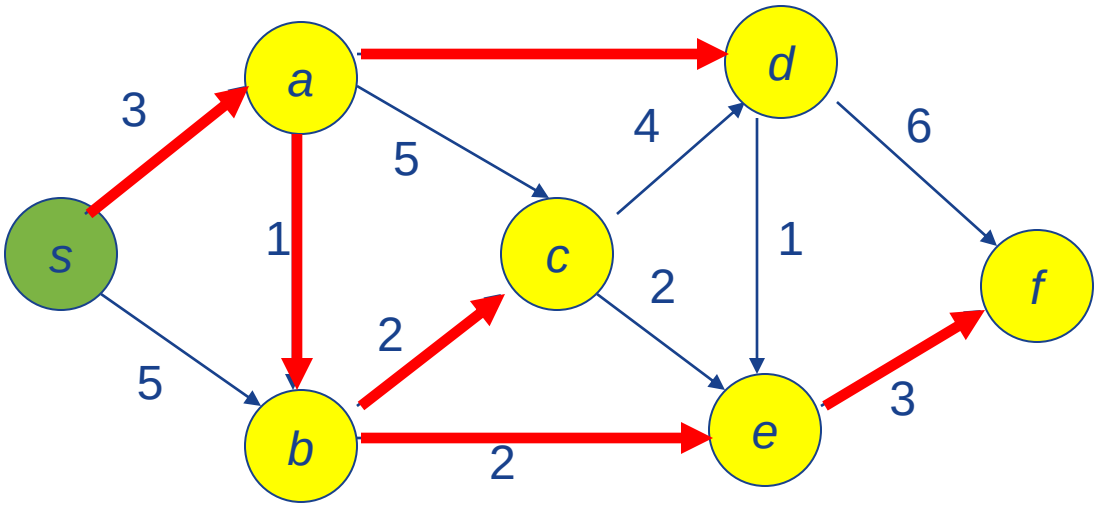
2. Các thuật toán thông dụng tìm đường đi ngắn nhất trên đồ thị

1. Ứng dụng BFS tìm đường đi ngắn nhất
2. Thuật toán Ford-Bellman
3. Thuật toán Dijkstra
4. Thuật toán A*
5. Thuật toán tìm đường đi ngắn nhất trên DAG
6. Thuật toán Floyd-Warshall

2. Các thuật toán thông dụng tìm đường đi ngắn nhất trên đồ thị

- Ví dụ: cho đồ thị sau, đường đi ngắn nhất từ s tới f là: s → a → b → e → f

Đỉnh nguồn



2.1 Ứng dụng BFS tìm đường đi ngắn nhất trên đồ thị theo số cạnh

- Mô tả thuật toán:
- Quy ước w(u, v) = 1 nếu có cạnh nối 2 đỉnh u và v và w(u, v) = 0 nếu ngược lại. Khi đó ta có thể dùng thuật toán tìm kiếm theo chiều rộng để thăm tất cả các đỉnh thuộc thành phần liên thông với s.
- Các đỉnh của đồ thị lần lượt được duyệt bằng cách cho vào hàng đợi queue.
- Mảng p[u] dùng để ghi vết các đỉnh đã đi qua.

2.1 Ứng dụng BFS tìm đường đi ngắn nhất trên đồ thị theo số cạnh

- Mã giả thuật toán:

```
def BFS(s, V):
        Queue = deque([])
        for v in V:
          In_queue[v] = False;
        Queue.append(v);
        In_queue[v] = True;

        while Queue:
            v = Queue.popleft()
          for u in Adjacency[v]:
             if Not In_queue[u]:
                Queue.append(u)
                d[u] = d[u] + 1
                   p[u] = v
                In_queue[u] = True;
```

2.1 Ứng dụng BFS tìm đường đi ngắn nhất trên đồ thị theo số cạnh

❖ Nhận xét:

- Nếu muốn chỉ tìm đường đi từ s đến t thì có thể dừng ngay khi có u=t.
- Độ phức tạp tính toán của thuật toán là O(n+m).

2.2 Thuật toán Ford-Bellman

- Mô tả thuật toán:

- Giả sử có đồ thị G = (V, E) không có chu trình âm, a[u, v] là ma trận trọng số.
- Ta gọi d[v] là khoảng cách từ s đến v. Ta khởi tạo d[s] = 0 và d[v] = ∞.
- Ta lần lượt xét các cạnh (u, v) nếu cạnh nào thỏa mãn d[u] + a[u, v] < d[v] thì ta tối ưu hóa bằng cách gán d[v] = d[u] + a[u, v]. Thuật toán kết thúc khi không còn cạnh nào cần tối ưu nữa.

2.2 Thuật toán Ford-Bellman

- Mã giả thuật toán:

```
def Ford_Bellman(s):
    for v in V:
        d[v] = a[s, v];
    p[v] = s;
    d[s] = 0;

    for k in range(1, n):
        for v in V \ {s}:
            for u in V:
                if d[v] > d[u] + a[u, v]
                    d[v] = d[u] + a[u, v]
                    p[v] = u
```

2.2 Thuật toán Ford-Bellman

❖ Nhận xét:

- Thuật toán có thể tìm đường đi từ một đỉnh đến tất cả các đỉnh còn lại và làm việc được với cả đồ thị có trọng số cung tùy ý.
- Độ phức tạp thuật toán là O(n^3).
- Ta thể tăng hiệu quả thuật toán bằng cách chấm dứt vòng lặp theo k khi phát hiện trong quá trình thực hiện hai vòng lặp trong không có biến d[v] nào bị thay đổi giá trị.

2.2 Thuật toán Ford-Bellman

- Với đồ thị thưa, nếu dùng ma trận kề thì độ phức tạp thuật toán chỉ còn O(n.m).

```
for u in Adjacency[v]:
    if d[v] > d[u] + a[u, v]:
        d[v] = d[u] + a[u, v]
        p[v] = u
```

- Ta có thể dùng hàng đợi lưu các đỉnh rồi lần lượt lấy ra, tính d[v] rồi tối ưu. Đây chính là ý tưởng của thuật toán cải tiến More-Bellman.

2.3 Thuật toán Dijkstra

- Mô tả thuật toán:

- Ta gán nhãn tạm thời cho các đỉnh, các nhãn đó cho biết cận trên của độ dài đường đi ngắn nhất từ s đến đỉnh đó.
- Các nhãn được biến đổi theo một thủ tục lặp. Mỗi bước lặp sẽ có một nhãn tạm thời trở thành nhãn cố định. Nếu nhãn của một đỉnh nào đó trở thành nhãn cố định thì nó sẽ cho ta không phải là cận trên mà là độ dài đường đi ngắn nhất từ đỉnh s đến đỉnh nó.

2.3 Thuật toán Dijkstra

- Mã giả thuật toán:

```
def Dijkstra(s):
    for v in V:
        d[v] = a[s,v]
        p[v] = s
    d[s] = 0
    S = {s}
    T = V \ {s}
    while T:
        u = z : d[z] = min {d[z] | z in T}
        T = T \ {u}
        S.add(u)
        for v in T:
          if d[v] > d[u] + a[u, v]:
             d[v] = d[u] + a[u, v]
             p[v] = u
```

2.3 Thuật toán Dijkstra

❖ Nhận xét:

- Trong trường hợp tìm kiếm u bằng tìm kiếm tuyến tính thì độ phức tạp thuật toán là O(n^2).
- Với đồ thị thưa, do phải liên tục lấy phần tử u có d[u] nhỏ nhất trong tập T, nên tốt nhất T nên là một hàng đợi ưu tiên (priority queue). Nếu cài đặt hàng đợi ưu tiên bằng Binary Heap thì độ phức tạp tính toán chỉ còn O((m + n)log(n)), với hàng đợi ưu tiên cài đặt bằng Fibonacci Heap thì độ phức tạp tính toán còn tốt hơn: O(m + nlog(n)).

2.4 Thuật toán A*

Mô tả thuật toán:

- Thuật toán A* sử dụng một hàm heuristic ước lượng khoảng cách từ nút hiện tại đến nút đích, rồi duyệt theo thứ tự ước lượng này.
- Ta xây dựng một tập mở OPEN SET ban đầu chỉ chứa đỉnh nguồn s, những đỉnh khác ở trong tập này có nghĩa là nó đã được phát hiện (discovered) bởi thuật toán (do được nối với một đỉnh thuộc OPEN SET trước đó) nhưng chưa phân tích tiếp các đỉnh kề với nó. Khi đã phân tích hết thì đỉnh này được loại ra khỏi OPEN SET.

2.4 Thuật toán A*

- Ở mỗi lần lặp thì chọn một đỉnh v trong OPEN SET sao cho khoảng cách đến đích theo dự đoán của nó là nhỏ nhất. Tức f(v) = g(v) + h(v) nhỏ nhất, trong đó g(v) là chi phí nhỏ nhất từ nguồn s đến v (tính được cho đến thời điểm hiện tại), h(v) là đánh giá heuristic về chi phí từ v đến đích t. Sau khi đã chọn được đỉnh v, tối ưu hoá đường đi ngắn nhất của các đỉnh kề với v (như trong thuật toán Dijkstra), nếu đỉnh kề đó không thuộc OPEN SET thì cho vào. Sau đó loại v ra khỏi OPEN SET. Thuật toán dừng lại khi OPEN SET rỗng.

2.4 Thuật toán A*

Mã giả thuật toán:

```
def AStar(s, t)
    for v in V:
        f[v] = d[v] = Infinity
        p[v] = v
    d[s] = 0
    f[s] = h(s)
    openset = {s}
    while Not openset.empty():
        v = openset.min()
        if v == t: #target found return
      for u in Adjacency[v]:
          if d[u] > d[v] + a[v,u]:
              d[u] = d[v] + a[v,u]
              p[u] = v
              f[u] = d[u] + h(u)
              if u in openset:
                openset.insert(u)
      openset.remove(v)
    return # target not found
```

2.4 Thuật toán A*

❖ Nhận xét:

- Nên cài đặt OPEN SET dưới dạng hàng đợi ưu tiên để lấy được đỉnh v có f(v) nhỏ nhất.
- Có thể dùng bảng màu ghi nhận trạng thái mỗi đỉnh: ban đầu tất cả màu trắng, khi vào OPEN SET có màu xám, khi ra khỏi OPEN SET có màu đen.
- Thuật toán Dijkstra là một trường hợp đặc biệt của thuật toán A* với h(v) luôn bằng 0.

2.5 Thuật toán tìm đường đi ngắn nhất trên đồ thi không có chu trình

- Mô tả thuật toán:

- Ta đánh số lại các đỉnh của đồ thị sao cho chỉ tồn tại những cung (order[i], order[j]) sao cho i < j. Do ta chỉ tìm đường đi từ s đến t nên chỉ cần đánh số cho s và các đỉnh có thể tìm thấy đường đi từ s.
- Sau khi đánh số ta làm tương tự các thuật toán đã đề cập tức là lần lượt tối ưu các cặp đỉnh nếu thỏa mãn hệ thức d[u] > d[v] + a[v, u].

2.5 Thuật toán tìm đường đi ngắn nhất trên đồ thi không có chu trình

- Mã giả thuật toán đánh số:

```
def DFS(v, T):
    for u in Adjacency[v]:
        if Not u in T:
            DFS(u, T)
    T.add(v)

def TopologySort(s):
    T = []
    v = [0] * |V|
    DFS(s, T)
    order = 0
    for u in reverse(T):
        map[order] = u
        order = order + 1
    return order
```

2.5 Thuật toán tìm đường đi ngắn nhất trên đồ thi không có chu trình

- Mã giả thuật toán tìm đường đi ngắn nhất:

```
def DAGShortestPath(s):
    for v in V:
        d[v] = Infinity
        p[v] = -1
    d[s] = 0
    p[s] = 0
    count = TopologySort(s)
    for i in range(0, count):
        v = map[i]
        for u in Adjacency[v]:
            if d[u] > d[v] + a[v, u]:
                d[u] = d[v] + a[v, u]
                p[u] = v
```

2.5 Thuật toán tìm đường đi ngắn nhất trên đồ thi không có chu trình

❖ Nhận xét:

- Thuật toán đánh số có độ phức tạp O(m).
- Nếu trong khi thực hiện DFS phát hiện được một cạnh ngược (back edge) thì đồ thị có chu trình, ta có thể kết thúc bài toán ngay tại đây.
- Độ phức tạp tính toán của phần tìm đường đi ngắn nhất là O(m+n) (do mỗi cung được duyệt tối đa 1 lần). Tổng hợp lại, độ phức tạp tính toán của cả hai bước là O(m + n).

2.6 Thuật toán Floyd-Warshall

- Mô tả thuật toán:

- Với mọi đỉnh k của đồ thị được xét theo thứ tự từ 1 đến n, xét mọi cặp đỉnh (u, v), ta làm cực tiểu hóa d[u, v] theo công thức: d[u, v] = min(d[u, v], d[u, k] + d[k, v]). Tức là nếu đường đi từ u đến v đang có lại dài hơn đường đi từ u đến k cộng với đường đi từ k đến v thì ta hủy bỏ đường đi từ u đến v hiện tại và thay vào đó là hai đường từ u đến k và từ k đến v

2.6 Thuật toán Floyd-Warshall

- Mã giả thuật toán:

```
def  Floyd_Warshall
    for u  in  V
       for u  in  V
          d[i,  j]  =  a[i,  j]
          Before[i,  j]  =  i

    for  k  in V  do
       for u  in  V
          for u  in  V
             if  (d[i,  j]  >  d[i,  k]  +  d[k,  j])  then
                d[i,  j]  =  d[i,  k]  +  d[k,  j]
                Before[i,  j]  =  Before[k,  j]
```

2.6 Thuật toán Floyd-Warshall

❖ Nhận xét:

- Thuật toán Floyd-Warshall dùng để tìm đường đi ngắn nhất giữa tất cả các đỉnh trên đồ thị
- Độ phức tạp của thuật toán là O(n^3).


Thank You !