Báo cáo bài tập về nhà - LongestPath

Nhóm thực hiện: Nhóm 7
Danh sách thành viên:

- Vũ Minh Phương
- Võ Lê Ngọc Thịnh

1. Ý tưởng và phương pháp giải bài toán

1.1 Phát biểu bài toán

Bài toán yêu cầu tìm đường đi đơn dài nhất trong một đồ thị vô hướng có N đỉnh và M cạnh, trong đó đường đi phải bắt đầu tại đỉnh 1 và kết thúc tại đỉnh N, đồng thời không được phép lặp lại đỉnh. Độ dài của đường đi được tính theo số cạnh mà đường đi đó đi qua. Đầu vào bao gồm danh sách các cạnh của đồ thị, và đầu ra gồm hai phần: độ dài lớn nhất của một đường đi đơn từ 1 đến N, và dãy các đỉnh thể hiện một đường đi đơn đạt độ dài tối đa đó.

1.2 Hướng tiếp cận

Ta sẽ duyệt tất cả các đường đi đơn có thể từ $1 \rightarrow N$ nhưng ta sẽ loại bỏ hay cắt tỉa (prune) sớm các nhánh không thể tạo ra lời giải tốt hơn lời giải hiện tại và kết hợp với tìm kiếm ưu tiên (best-first search).

1.3 Phương pháp & Chi tiết ý tưởng

Ý tưởng của lời giải dựa trên phương pháp Branch and Bound kết hợp với tìm kiếm ưu tiên (best-first search). Thay vì duyệt toàn bộ các đường đi đơn có thể từ 1 đến n một không gian tìm kiếm có độ lớn theo giai thừa thuật toán khai thác cấu trúc của bài toán để cắt tỉa mạnh các nhánh không cần thiết.

Mỗi trạng thái tìm kiếm bao gồm đỉnh hiện tại, tập các đỉnh đã thăm và đường đi hiện thời. Với mỗi lần mở rộng, thuật toán tính một cận trên (upper bound) cho độ dài tối đa có thể đạt được từ trạng thái đó bằng cách chạy BFS để đếm số đỉnh chưa thăm còn có thể tiếp cận. Cận trên này được dùng để loại bỏ những nhánh mà ngay cả

trong trường hợp tốt nhất cũng không thể vượt qua lời giải hiện tại. Các trạng thái được đưa vào một hàng đợi ưu tiên, trong đó trạng thái có cận trên lớn nhất được mở rộng trước, giúp nhanh chóng tìm thấy đường đi dài và từ đó tăng hiệu quả cắt tỉa.

Bằng cách kết hợp heuristic ước lượng cận trên với chiến lược duyệt có ưu tiên, thuật toán giảm đáng kể số lượng trạng thái cần khảo sát, từ đó tìm được đường đi đơn dài nhất từ 1 đến n trong thời gian phù hợp với giới hạn của đề bài.

2. Phân tích và đánh giá thuật toán sử dụng

2.1. Chiến lược Branch and Bound

Branch and Bound là kỹ thuật tìm kiếm có hệ thống trong một không gian nghiệm lớn, trong đó mỗi trạng thái đại diện cho một nghiệm riêng biệt hoặc nghiệm đang được xây dựng. Hai thành phần cốt lõi của phương pháp này là: (1) mở rộng trạng thái (branching), và (2) loại bỏ trạng thái không cần thiết dựa trên cận (bounding).

Trong lời giải, mỗi trạng thái được mô tả bởi:

- Đỉnh hiện tại
- Tập các đỉnh đã thăm (`visited`)
- Đường đi hiện tại (`path`)
- Độ dài hiện tại (`current_length`)
- Cận trên ước lượng độ dài tối đa có thể đạt được từ trạng thái đó.

Khi mở rộng, thuật toán lần lượt thử tất cả các đỉnh kề chưa được thăm. Tuy nhiên, trước khi đưa trạng thái mới vào hàng đợi, thuật toán so sánh cận trên của trạng thái đó với độ dài tốt nhất hiện tại. Nếu cận trên không thể vượt qua lời giải tốt nhất, trạng thái này sẽ bị loại bỏ mà không cần mở rộng thêm. Điều này giúp giảm đáng kể số nhánh cần xét.

2.2. Tính cận trên bằng BFS

Cận trên đóng vai trò quan trọng trong việc quyết định xem một nhánh có đáng để tiếp tục phát triển hay không. Trong thuật toán, cận trên được tính bằng cách chạy BFS từ đỉnh hiện tại để đếm xem còn bao nhiêu đỉnh chưa được thăm có thể tiếp cận.

Hàm `calculate_upper_bound()` thực hiện:

1. Khởi tạo BFS từ đỉnh hiện tại.
2. Di chuyển qua tất cả các đỉnh có thể đi tới mà chưa thuộc `visited`.
3. Đếm số lượng các đỉnh này, tức là số đỉnh còn tiềm năng nằm trên đường đi tương lai.
4. Cận trên được tính bằng:

$$\text{upper\_bound} = \text{current\_length} + |\text{reachable\_nodes}|$$

Ở đây, reachable nodes chính là tập các đỉnh không vi phạm tính chất đường đi đơn, do đó việc cộng thêm số lượng này vào độ dài hiện tại cho ta một ước lượng hợp lý cho độ dài tối đa của nhánh hiện tại.

Ưu điểm của cách ước lượng này:

- sử dụng thông tin cấu trúc đồ thị,
- cho cận trên chặt hơn so với các cách đếm đơn giản,
- giúp việc cắt tỉa (pruning) hiệu quả hơn.

2.3. Tìm kiếm ưu tiên

Mỗi trạng thái được đưa vào một hàng đợi ưu tiên dạng max-heap. Khóa sắp xếp bao gồm:

1. Cận trên của trạng thái,
2. Độ dài hiện tại của trạng thái.

Chiến lược này đảm bảo rằng:

- Trạng thái có cận trên lớn nhất sẽ được mở rộng trước,
- Thuật toán sẽ nhanh chóng tìm được một lời giải dài, từ đó tăng khả năng cắt tỉa những trạng thái còn lại.

Đây là một chiến thuật quen thuộc trong best-first search, tương tự như A* hoặc Beam Search, nhưng trong bối cảnh bài toán Longest Path không có điều kiện tối ưu hóa đơn giản, việc dùng cận trên giúp định hướng tìm kiếm tốt hơn.

2.4. Cơ chế cắt tỉa trạng thái (Pruning)

Cắt tỉa xảy ra khi:

$$\text{upper\_bound} \leq \text{best\_length}.$$

Nghĩa là, ngay cả trong kịch bản tốt nhất, độ dài của đường đi từ trạng thái hiện tại vẫn không thể vượt qua đường đi tốt nhất đã tìm thấy. Khi đó, mở rộng trạng thái này là không cần thiết.

Cơ chế này làm giảm số trạng thái cần duyệt từ mức độ theo giai thừa (như trong tìm kiếm toàn phần) xuống mức có thể xử lý được trong thực tế.

2.5. Đánh giá mức độ hiệu quả của các kỹ thuật

- **Branch and Bound:** giúp loại bỏ số lượng lớn các nhánh tiềm năng mà không cần duyệt đến cuối, phù hợp cho bài toán có không gian nghiệm lớn như Longest Simple Path.
- **Cận trên bằng BFS:** cận trên khá chặt vì được xây dựng dựa trên cấu trúc thật của đồ thị, giúp hạn chế duyệt các nhánh tồi.
- **Ưu tiên bằng heap:** khiến thuật toán tìm ra lời giải tốt rất sớm, từ đó tăng hiệu quả cắt tỉa.
- **Pruning:** là yếu tố quyết định giúp thuật toán chạy trong thời gian khả thi dù bản chất bài toán là NP-hard.

Tổng thể, các kỹ thuật trên kết hợp với nhau tạo thành một thuật toán tìm kiếm có định hướng, vừa hiệu quả trong thực tế, vừa bảo đảm không bỏ sót lời giải tối ưu.

2.6. Mã nguồn cài đặt tham khảo

```
from collections import defaultdict, deque
import heapq

class LongestPathBranchAndBound:
    def __init__(self, n, m):
        self.n = n
        self.m = m
        self.graph = defaultdict(list)
        self.best_path = []
        self.best_length = 0

    def add_edge(self, u, v):
        self.graph[u].append(v)

    def calculate_upper_bound(self, current_vertex, visited, current_length):
        reachable = set()
        queue = deque([current_vertex])
        temp_visited = set([current_vertex])

        while queue:
            u = queue.popleft()
            for v in self.graph[u]:
                if v not in visited and v not in temp_visited:
                    reachable.add(v)
                    temp_visited.add(v)
                    queue.append(v)

        return current_length + len(reachable)

    def branch_and_bound(self):
        initial_visited = frozenset([1])
        initial_path = [1]
        initial_upper_bound = self.calculate_upper_bound(1, initial_visited, 1)

        heap = [(-initial_upper_bound, -1, 1, initial_visited, initial_path)]

        while heap:
            neg_ub, neg_length, current_vertex, visited, path = heapq.heappop(heap)
            current_length = -neg_length
            upper_bound = -neg_ub

            if upper_bound <= self.best_length:
                continue

            if current_length > self.best_length:
                self.best_length = current_length
                self.best_path = path[:]
```

```
            for next_vertex in self.graph[current_vertex]:
                if next_vertex not in visited:
                    new_visited = visited | frozenset([next_vertex])
                    new_path = path + [next_vertex]
                    new_length = current_length + 1
                    new_upper_bound = self.calculate_upper_bound(next_vertex, new_vi

                    if new_upper_bound > self.best_length:
                        heapq.heappush(heap, (-new_upper_bound, -new_length, next_ve

        return self.best_length, self.best_path

n, m = map(int, input().split())
solver = LongestPathBranchAndBound(n, m)

for _ in range(m):
    u, v = map(int, input().split())
    solver.add_edge(u, v)

length, path = solver.branch_and_bound()
print(length - 1)
print(' '.join(map(str, path)))
```

3. Phân tích độ phức tạp

Chi phí thời gian

- Tính cận trên bằng BFS cho mỗi trạng thái: $O(n + m)$.
- Mở rộng trạng thái và tạo trạng thái con (tối đa theo bậc của đỉnh): $O(1)$ đến $O(n)$ thao tác nhẹ.
- Tổng thời gian phụ thuộc số trạng thái được mở rộng K: $O(K \cdot (n + m))$.

Trên dữ liệu thực tế: Nhờ cắt tỉa mạnh nên K nhỏ → thuật toán chạy tốt trong giới hạn thời gian.

Chi phí không gian

- Lưu đồ thị theo danh sách kề: $O(n + m)$.
- Lưu các trạng thái trong hàng đợi ưu tiên (heap), mỗi trạng thái tốn $O(n)$ bộ nhớ. Tổng cộng: $O(K \cdot n)$.
- Tổng không gian: $O(K \cdot n + n + m)$ hay $O(K \cdot n)$