ĐẠI HỌC QUỐC GIA TP.HCM
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN

Khoa Khoa học máy tính

Môn học Phân Tích Và Thiết Kế Thuật Toán

Bài tập Phương pháp thiết kế thuật toán gần đúng

Cao Lê Công Thành
MSSV: 23521437
Đặng Quang Vinh
MSSV: 23521786

Mục lục

1 Bài 1: Bài toán Set Cover

1.1 Mô tả bài toán

Cho một tập U, là một tập hợp các phần tử, và một tập hợp $S = \{S_1, S_2, \ldots, S_m\}$ gồm các tập con của U. Mỗi tập con S_i là một tập con của U, và mục tiêu là chọn một số ít các tập con sao cho tất cả các phần tử trong U đều được bao phủ, tức là mỗi phần tử của U phải xuất hiện ít nhất một lần trong các tập con đã chọn.
Cụ thể:

- Tập $U = \{u_1, u_2, \ldots, u_n\}$ là tập hợp các phần tử cần được bao phủ.
- Tập con $S = \{S_1, S_2, \ldots, S_m\}$ là tập hợp các tập con của U, mỗi tập con S_i chứa một số phần tử của U.
- **Mục tiêu:** Chọn một số ít các tập con từ S sao cho mỗi phần tử trong U xuất hiện ít nhất một lần trong các tập con được chọn. Tối thiểu hóa số lượng tập con được chọn.

Định nghĩa chính thức:

$$U = \{u_1, u_2, \ldots, u_n\}, \quad S = \{S_1, S_2, \ldots, S_m\} \text{ với } S_i \subseteq U \text{ cho mọi } i.$$

Mục tiêu là tìm một tập con các tập con $S' \subseteq S$ sao cho:

$$\bigcup_{S_i \in S'} S_i = U$$

và tối thiểu hóa số lượng các tập con được chọn, tức là:

$$\min |S'|.$$

1.2 Thuật toán Tham lam

1.2.1 Ý tưởng chính

Thuật toán tham lam (Greedy Algorithm) giải quyết bài toán Set Cover bằng cách:

- Ở mỗi bước, chọn tập con S_i bao phủ nhiều phần tử chưa được bao phủ nhất.
- Lặp lại cho đến khi tất cả các phần tử trong tập U đã được bao phủ.

1.2.2 Các bước chi tiết

1.2.3 Khởi tạo

- **Input:**
  - U: Tập các phần tử cần được bao phủ, ví dụ: $U = \{1, 2, 3, 4, 5\}$.
  - S: Tập các tập con của U, ví dụ: $S = \{\{1, 2\}, \{2, 3, 4\}, \{4, 5\}, \{5\}\}$.
- **Output:**
  - C: Tập hợp các tập con được chọn sao cho tất cả các tập trong C bao phủ toàn bộ U.
- **Khởi tạo:**
  - $C = \emptyset$: Tập hợp ban đầu rỗng (chưa chọn tập con nào).
  - $U_{\text{left}} = U$: Tập các phần tử chưa được bao phủ.

1.2.4 Vòng lặp chính

1. **Tìm tập con tốt nhất:**
   - Duyệt qua tất cả các tập con $S_i \in S$.
   - Tính số lượng phần tử trong S_i thuộc U_{left}, tức là $|S_i \cap U_{\text{left}}|$.
   - Chọn tập S_i có số lượng phần tử chung lớn nhất với U_{left}.
2. **Cập nhật:**
   - Thêm tập con S_i đã chọn vào C.
   - Loại bỏ các phần tử của S_i ra khỏi U_{left}, tức là:

$$U_{\text{left}} = U_{\text{left}} \setminus S_i$$

3. **Kết thúc vòng lặp:**
   - Dừng khi $U_{\text{left}} = \emptyset$, tức là tất cả các phần tử trong U đã được bao phủ.

1.2.5 Trả về kết quả

- C: Danh sách các tập con đã chọn.

1.2.6 Ví dụ minh họa

Input:

- Tập phần tử: $U = \{1, 2, 3, 4, 5\}$.
- Danh sách tập con: $S = \{\{1, 2\}, \{2, 3, 4\}, \{4, 5\}, \{5\}\}$.

Bước thực hiện:

1. **Khởi tạo:**
   - $U_{\text{left}} = \{1, 2, 3, 4, 5\}$.
   - $C = \emptyset$.
2. **Lặp:**
   - **Lần 1:**
     - Chọn $S_2 = \{2, 3, 4\}$ (bao phủ nhiều nhất).
     - $C = \{\{2, 3, 4\}\}$, $U_{\text{left}} = \{1, 5\}$.
   - **Lần 2:**
     - Chọn $S_1 = \{1, 2\}$.
     - $C = \{\{2, 3, 4\}, \{1, 2\}\}$, $U_{\text{left}} = \{5\}$.
   - **Lần 3:**
     - Chọn $S_3 = \{4, 5\}$.
     - $C = \{\{2, 3, 4\}, \{1, 2\}, \{4, 5\}\}$, $U_{\text{left}} = \emptyset$.

Kết quả: $C = \{\{2, 3, 4\}, \{1, 2\}, \{4, 5\}\}$.

1.2.7 Code Python

```python
def greedy_set_cover(U, S):

    U_left = set(U)  # Các phần tử chưa được bao phủ
    C = []  # Tập các tập con được chọn

    while U_left:
        # Tìm tập con bao phủ nhiều phần tử nhất trong U_left
        best_set = max(S, key=lambda subset: len(set(subset) & U_left))
        C.append(best_set)
        U_left -= set(best_set)  # Loại bỏ các phần tử đã được bao phủ

    return C

# Ví dụ sử dụng
U = [1, 2, 3, 4, 5]
S = [[1, 2], [2, 3, 4], [4, 5], [5]]
result = greedy_set_cover(U, S)
print("Các tập con được chọn:", result)

```

1.3 Cách giải bằng lập trình phi tuyến

1.3.1 Ý tưởng và cách giải

Lập trình phi tuyến được sử dụng để tìm lời giải gần đúng cho bài toán Set Cover bằng cách chuyển các biến nhị phân thành các biến liên tục. Các bước thực hiện như sau:

1.3.2 Biến đổi bài toán Set Cover

- **Hàm mục tiêu**: Thay vì chỉ tối thiểu hóa số lượng tập S_i được chọn, thêm một thành phần phi tuyến vào chi phí để phản ánh mức độ ưu tiên hoặc tác động của mỗi tập:

$$\min \sum_{i=1}^{m} w_i \cdot x_i^2$$

với w_i là chi phí của tập S_i và $x_i \in [0, 1]$.

- **Ràng buộc**: Đảm bảo mỗi phần tử trong tập U được bao phủ:

$$\sum_{i:u_j \in S_i} x_i \geq 1 \quad \forall u_j \in U.$$

1.3.3 Cài đặt Python

Dưới đây là mã Python minh họa cách giải bài toán:

Mô tả bài toán cụ thể

- **Tập các phần tử cần phủ**: $U = \{1, 2, 3, 4, 5\}$
- **Các tập con**:

$$S_1 = \{1, 2\},\ S_2 = \{2, 3\},\ S_3 = \{3, 4, 5\}$$

- **Trọng số các tập con**: $w_1 = 1, w_2 = 2, w_3 = 3$

Mã Python

```python
import numpy as np
from scipy.optimize import minimize

# Dữ liệu bài toán
U = {1, 2, 3, 4, 5}  # Tập các phần tử cần phủ
S = [{1, 2}, {2, 3}, {3, 4, 5}]  # Các tập con
weights = [1, 2, 3]  # Trọng số các tập con
m = len(S)

# Hàm mục tiêu: f(x) = sum(w_i * x_i^2)
def objective(x):
    return sum(weights[i] * x[i]**2 for i in range(m))

# Ràng buộc: mọi phần tử u_j phải được phủ
def coverage_constraint(x):
    constraints = []
    for u in U:
        # Tổng x_i cho các S_i chứa u phải >= 1
        constraints.append(sum(x[i] for i in range(m) if u in S[i]) - 1)
    return np.array(constraints)

# Ràng buộc trong scipy.optimize yêu cầu dạng (g(x) >= 0 hoặc <= 0)
constraints = [{'type': 'ineq', 'fun': lambda x, idx=i: coverage_constraint(x)[idx]} for i in range(len(U))]

# Ràng buộc x_i ∈ [0, 1]
bounds = [(0, 1) for _ in range(m)]

# Khởi tạo giá trị ban đầu cho x
x0 = np.random.rand(m)

# Giải bài toán tối ưu phi tuyến
result = minimize(objective, x0, bounds=bounds, constraints=constraints)

# Kết quả
if result.success:
    print("Optimal solution found:")
    for i, val in enumerate(result.x):
        print(f"x_{i+1} = {val:.4f}")
    print(f"Total cost: {result.fun:.4f}")
else:
    print("Optimization failed.")

```

2 Bài tập 2: Bài toán TSP (Travelling Salesman Problem)

Mô tả bài toán:
Bài toán TSP (Travelling Salesman Problem - Bài toán người du lịch) là một bài toán tối ưu hóa cổ điển trong lý thuyết đồ thị và tổ hợp. Mục tiêu của bài toán này là tìm một chu trình Hamilton (có thể hiểu là một con đường) trong đồ thị sao cho:

- Mỗi đỉnh được thăm đúng một lần.
- Tổng chi phí (hoặc tổng quãng đường) đi qua tất cả các đỉnh là nhỏ nhất.

Định nghĩa chính thức: Cho một đồ thị $G = (V, E)$, trong đó:

- $V = \{v_1, v_2, \ldots, v_n\}$ là tập hợp các đỉnh, mỗi đỉnh tương ứng với một thành phố.
- E là tập hợp các cạnh nối các đỉnh.
- Mỗi cạnh $(v_i, v_j) \in E$ có một trọng số $c(v_i, v_j)$, có thể là chi phí, thời gian hoặc khoảng cách di chuyển giữa các thành phố v_i và v_j.

Mục tiêu: Tìm một chu trình Hamiltonian (một chuỗi các đỉnh mà bắt đầu từ một thành phố và quay lại thành phố ban đầu) sao cho tổng trọng số của các cạnh trong chu trình này là nhỏ nhất.

2.1 Thuật toán tham lam (Greedy Algorithm)

Ý tưởng:

- Bắt đầu từ một đỉnh bất kỳ trong đồ thị.
- Ở mỗi bước, chọn cạnh có trọng số nhỏ nhất để kết nối với đỉnh chưa được thăm.
- Tiếp tục thực hiện cho đến khi tất cả các đỉnh được thăm và quay lại đỉnh khởi đầu.

Ưu điểm:

- Dễ hiểu và dễ triển khai.

- Hoạt động hiệu quả trên đồ thị nhỏ.

Nhược điểm:

- Không đảm bảo tìm được lời giải tối ưu.
- Kết quả có thể khác nhau nếu thay đổi đỉnh bắt đầu.

Thuật toán:

1. **Khởi tạo:**
   - Chọn đỉnh khởi đầu v_1.
   - Tập các đỉnh chưa thăm ban đầu $U = V \setminus \{v_1\}$.
   - Chu trình ban đầu $P = [v_1]$.
2. **Lặp lại:**
   - Tìm đỉnh $v \in U$ sao cho trọng số $c(P[-1], v)$ nhỏ nhất.
   - Thêm v vào chu trình P và loại bỏ v khỏi U.
3. **Kết thúc:**
   - Thêm cạnh quay về đỉnh khởi đầu: P.append(v_1).
   - Tính tổng trọng số của chu trình.

Code:

```
import numpy as np

def tsp_greedy(cost_matrix):
    n = len(cost_matrix)
    visited = [False] * n
    path = [0]  # Bat dau tu dinh 0
    visited[0] = True
    total_cost = 0

    for _ in range(n - 1):
        last = path[-1]
        next_city = np.argmin([
            cost_matrix[last][j] if not visited[j] else float
                ('inf')
            for j in range(n)
        ])
        total_cost += cost_matrix[last][next_city]
        path.append(next_city)
        visited[next_city] = True
```

```
19
20      # Quay lai dinh dau tien
21      total_cost += cost_matrix[path[-1]][path[0]]
22      path.append(path[0])
23
24      return path, total_cost
```

2.2 Thuật toán Nearest Neighbor Heuristic (NNH)

Ý tưởng:

- Bắt đầu từ một đỉnh ngẫu nhiên.
- Ở mỗi bước, tìm đỉnh gần nhất (theo trọng số nhỏ nhất) chưa được thăm để đi.
- Kết thúc khi tất cả các đỉnh đã được thăm, và quay lại đỉnh khởi đầu.

Ưu điểm:

- Hoạt động nhanh chóng.
- Hiệu quả trên các đồ thị đầy đủ.

Nhược điểm:

- Không đảm bảo lời giải tối ưu.
- Kết quả phụ thuộc nhiều vào đỉnh xuất phát.

Thuật toán:

1. **Khởi tạo:**
   - Chọn đỉnh xuất phát v_1.
   - Tập các đỉnh chưa được thăm $U = V \setminus \{v_1\}$.
   - Khởi tạo chu trình $P = [v_1]$.
2. **Lặp lại:**
   - Tìm đỉnh gần nhất $v \in U$.
   - Thêm đỉnh này vào chu trình P và loại nó khỏi U.
3. **Kết thúc:**

- Thêm cạnh quay lại đỉnh khởi đầu.
- Tính tổng chi phí.

Code:

```
def tsp_nearest_neighbor(cost_matrix):
    n = len(cost_matrix)
    visited = [False] * n
    path = [0]  # Bat dau tu dinh 0
    visited[0] = True
    total_cost = 0

    for _ in range(n - 1):
        last = path[-1]
        min_distance = float('inf')
        next_city = -1
        for j in range(n):
            if not visited[j] and cost_matrix[last][j] <
               min_distance:
                min_distance = cost_matrix[last][j]
                next_city = j
        path.append(next_city)
        visited[next_city] = True
        total_cost += min_distance

    # Quay lai dinh dau tien
    total_cost += cost_matrix[path[-1]][path[0]]
    path.append(path[0])

    return path, total_cost
```

2.3 Thuật toán 2-Approximation dựa trên Cây khung nhỏ nhất (MST)

Ý tưởng:

- Dựa trên cây khung nhỏ nhất (Minimum Spanning Tree) để tạo chu trình gần đúng:
  1. Duyệt Preorder trên cây MST để xây dựng chu trình.
  2. Loại bỏ các đỉnh lặp lại trong kết quả để tạo thành chu trình Hamilton.

Ưu điểm:

- Tìm lời giải với chi phí không vượt quá 2 lần lời giải tối ưu.

- Hiệu quả trên đồ thị lớn.

Nhược điểm:

- Cần tính toán cây khung nhỏ nhất trước khi xây dựng chu trình.

Thuật toán:

1. **Tạo cây khung nhỏ nhất (MST):**
   - Sử dụng thuật toán Prim hoặc Kruskal để tạo cây MST từ đồ thị đầu vào.
2. **Duyệt cây khung:**
   - Sử dụng thuật toán duyệt Preorder để thăm các đỉnh trong cây MST.
3. **Loại bỏ đỉnh lặp lại:**
   - Tạo chu trình Hamilton từ chu trình Euler bằng cách bỏ qua các đỉnh thăm lại.
4. **Hoàn tất:**
   - Tính tổng chi phí và thêm cạnh quay về đỉnh ban đầu.

Code:

```
from scipy.sparse.csgraph import minimum_spanning_tree
from collections import defaultdict

def tsp_mst_approximation(cost_matrix):
    n = len(cost_matrix)
    mst = minimum_spanning_tree(cost_matrix).toarray().astype
        (float)

    # Xay dung do thi MST
    mst_graph = defaultdict(list)
    for i in range(n):
        for j in range(n):
            if mst[i][j] > 0:
                mst_graph[i].append(j)
                mst_graph[j].append(i)

    # Duyet Preorder tren MST
    visited = [False] * n
    path = []

```

```
20      def preorder(node):
21          visited[node] = True
22          path.append(node)
23          for neighbor in mst_graph[node]:
24              if not visited[neighbor]:
25                  preorder(neighbor)
26
27      preorder(0)
28
29      # Loai bo cac dinh lap lai va quay lai dinh dau
30      path = list(dict.fromkeys(path))
31      path.append(path[0])
32
33      # Tinh tong chi phi
34      total_cost = sum(cost_matrix[path[i]][path[i + 1]] for i
            in range(len(path) - 1))
35
36      return path, total_cost
```