Vietnam National University, Ho Chi Minh City
University of Technology
Faculty of Computer Science and Engineering

DISCRETE STRUCTURE (CO1007)

Assignment

"BELLMAN-FORD ALGORITHM"

Instructor(s): Nguyễn Văn Minh Mẫn, Mahidol University
Nguyễn An Khương, CSE-HCMUT
Trần Tuấn Anh,CSE-HCMUT
Nguyễn Tiến Thịnh, CSE-HCMUT
Trần Hồng Tài, CSE-HCMUT
Mai Xuân Toàn, CSE-HCMUT
Class: L03
Student: Nguyễn Tiến Nam - 2412188

Ho Chi Minh City, May 2025

Contents

1 Đề Bài

Bài toán yêu cầu tìm lộ trình ngắn nhất cho một người bán hàng đi qua tất cả các thành phố trong danh sách, mỗi thành phố được ghé qua đúng một lần, và sau đó quay trở lại thành phố xuất phát.

- **Đầu vào:**
  - Danh sách các thành phố.
  - Khoảng cách giữa từng cặp thành phố (thường được biểu diễn dưới dạng ma trận khoảng cách hoặc đồ thị có trọng số).
- **Đầu ra:** Một lộ trình (hay chu trình Hamilton) có tổng chiều dài nhỏ nhất, thỏa mãn:
  - Người bán hàng đi qua tất cả các thành phố một lần.
  - Quay về thành phố xuất phát.

2 Giải thuật

```
Function Traveling(edges, numEdges, start):
    Xây dựng danh sách đỉnh và ma trận kề cạnh adj[][]
    Xác định vị trí startIdx trong danh sách đỉnh
    Kiểm tra tính đối xứng của adj[][]

    If số đỉnh <= 20:
        // Dùng Held-Karp
        Khởi tạo bảng dp và parent cho quy hoạch động
        Lặp qua các mask và đỉnh, cập nhật chi phí đi đến
        Tìm đường đi tốt nhất quay về startIdx
        Khôi phục lại đường đi bằng parent[][]

    Else:
        // Dùng ACO cho đồ thị lớn
        Khởi tạo ma trận pheromone[][]
        Lặp trong MAX_ITERATIONS:
            Cho mỗi con kiến, sinh tour dựa trên xác suất pheromone và
            ↪ heuristic
            Cập nhật tour tốt nhất nếu tìm được chi phí thấp hơn
            Bay mùi pheromone và cập nhật theo chất lượng tour
        Dùng 2-opt tối ưu tour cuối cùng

    Chuẩn hoá đường đi về chuỗi để xuất
    Trả về chuỗi ký tự biểu diễn đường đi Hamilton ngắn nhất

```

Figure 1: Hàm `Traveling` Tổng quát thực hiện giải bài toán người du lịch (TSP)

```
Function HeldKarpTSP(adj, startIdx):
    V := số lượng đỉnh
    dp[2^V][V] := INF
    parent[2^V][V] := -1

    dp[1 << startIdx][startIdx] := 0

    For mask from 0 to (1 << V) - 1:
        For k from 0 to V-1:
            If k không thuộc mask: continue
            For i from 0 to V-1:
                If i == k or i không thuộc mask: continue
                If adj[i][k] >= INF: continue
                newCost := dp[mask ^ (1 << k)][i] + adj[i][k]
                If newCost < dp[mask][k]:
                    dp[mask][k] := newCost
                    parent[mask][k] := i

    finalMask := (1 << V) - 1
    minCost := INF
    lastVertex := -1

    For k from 0 to V-1:
        If k == startIdx or adj[k][startIdx] >= INF: continue
        totalCost := dp[finalMask][k] + adj[k][startIdx]
        If totalCost < minCost:
            minCost := totalCost
            lastVertex := k

    If minCost >= INF: return ""

    path := []
    mask := finalMask
    curr := lastVertex
    While mask != (1 << startIdx):
        path.append(curr)
        prev := parent[mask][curr]
        If prev == -1: return ""
        mask := mask ^ (1 << curr)
        curr := prev
    path.append(startIdx)
    Reverse path
    path.append(startIdx) // tạo chu trình

    Return path
```

Figure 2: Hàm `HeldKarpTSP` thể hiện giải thuật Held-Karp (DP with Bitmasking)

```
Function ACO_TSP(adj, startIdx):
    V := số lượng đỉnh
    Khởi tạo pheromone[i][j] = 1.0 cho mọi cạnh (i, j)
    bestTour := rỗng, bestCost := INF
    noImprovementCount := 0

    For iter from 1 to MAX_ITERATIONS:
        For mỗi ant:
            - tour := [startIdx], visited[startIdx] := true
            - Lặp V - 1 bước:
                + Tính xác suất chọn đỉnh tiếp theo dựa vào pheromone và 1/distance
                + Quay roulette để chọn đỉnh tiếp theo chưa đi qua
                + Nếu không có đỉnh hợp lệ: dừng tour
                + Ngược lại: cập nhật tour và đánh dấu đã đi qua
            - Nếu tour hợp lệ (đóng vòng):
                + Tính chi phí tour
                + Nếu tốt hơn bestCost: cập nhật bestTour và bestCost

        Nếu có cải thiện: đặt noImprovementCount := 0
        Ngược lại: tăng noImprovementCount
        Nếu noImprovementCount >= NO_IMPROVEMENT_LIMIT: dừng sớm

        Bay mùi pheromone: pheromone[i][j] *= (1 - EVAPORATION)
        Với mỗi ant có tour hợp lệ:
            + Cộng pheromone theo chất lượng tour: pheromone[u][v] += Q / cost

    Sau lặp: áp dụng 2-opt cho bestTour nếu có thể
    Nếu tour có chi phí < THRESHOLD: return bestTour
    Ngược lại: return rỗng

```

Figure 3: Hàm `ACO_TSP` thể hiện giải thuật ACO (Ant Colony Optimization)

```
Function Apply2opt(tour, adj):
    V := độ dài của tour
    improved := true
    While improved:
        improved := false
        For i from 1 to V - 2:
            For j from i + 1 to V - 2:
                a := tour[i - 1], b := tour[i]
                c := tour[j], d := tour[j + 1]

                // Kiểm tra hai kết nối mới (a-c và b-d) có hợp lệ không
                If adj[a][c] == INF or adj[b][d] == INF:
                    continue

                // Tính chi phí đoạn ban đầu giữa i và j
                originalSegmentCost := 0
                For k from i to j - 1:
                    If adj[tour[k]][tour[k + 1]] == INF:
                        originalSegmentCost := INF
                        break
                    originalSegmentCost += adj[tour[k]][tour[k + 1]]
                If originalSegmentCost == INF:
                    continue
                // Tính chi phí đoạn đảo ngược giữa i và j
                reversedSegmentCost := 0
                For k from i to j - 1:
                    If adj[tour[k + 1]][tour[k]] == INF:
                        reversedSegmentCost := INF
                        break
                    reversedSegmentCost += adj[tour[k + 1]][tour[k]]
                If reversedSegmentCost == INF:
                    continue
                // Tính chi phí kết nối cũ và mới
                oldConnectionCost := adj[a][b] + adj[c][d]
                newConnectionCost := adj[a][c] + adj[b][d]
                // Tính sự thay đổi chi phí toàn phần nếu đảo đoạn
                delta := (newConnectionCost - oldConnectionCost)
                         + (reversedSegmentCost - originalSegmentCost)

                // Nếu tổng chi phí giảm thì thực hiện đảo đoạn
                If delta < 0:
                    Đảo ngược đoạn tour[i..j]
                    improved := true

    Return tour đã được cải thiện
```

Figure 4: Hàm `Apply2opt` thể hiện giải thuật Local Search

3 Code

3.1 Mô tả

Đối với trường hợp từ 20 đỉnh trở xuống

Với bài toán Traveling Salesman Problem, nếu số đỉnh (number of cities) là 7 trở xuống thì ta có thể sử dụng vét cạn (Brute Force) vì thuật toán này có độ phức tạp là $O(n!)$ và 7! vẫn chưa đủ lớn để thành vấn đề. Nhưng nếu số đỉnh là 8 trở lên thì ta nên tiếp cận bằng phương pháp khác, ở đây ta sẽ sử dụng Dynamic Programming với thuật toán Held-Karp[1]. Với độ phức tạp $O(n^2 \times 2^n)$, thuật toán này trở nên nhanh hơn rất nhiều so với Brute Force đối với số đỉnh tương đối lớn hơn. Vậy ta sẽ chọn Held-Karp luôn cho trường hợp dưới 20 đỉnh thay vì hybrid giữa Brute Force và Held-Karp vì lợi thế của Brute Force đối với số đỉnh dưới 8 so với Held-Karp là không nhiều.

Trước tiên, vì input là danh sách kề nên để tối ưu cho thuật toán này (vốn cần lookup cạnh khá nhiều, nếu sử dụng danh sách kề thì độ phức tạp sẽ là $O(V)$ cho mỗi lần duyệt), ta sẽ dùng ma trận kề vì có độ phức tạp khi duyệt thấp hơn rất nhiều, chỉ là $O(1)$[2].

Trong Held-Karp, có 2 phương pháp là Top-Down và Bottom-Up. Vì Top-Down sử dụng đệ quy nên đối với các đồ thị có kích thước lớn thì có thể gây recursion stack overhead, hay tệ hơn là stack overflow. Nên trong bài này ta sử dụng Bottom-Up, với việc sử dụng Tabulation và xây dựng giải pháp tăng dần.

Ta sẽ sử dụng bảng (2D array), giả sử phần tử DP[S][v] sẽ thể hiện chi phí tối thiểu để đến thành phố v, với S là tập hợp các đỉnh hay thành phố đã đi qua. Trước tiên, chúng ta sẽ định nghĩa base case (starting city) với giá trị 0 (DP[start][start] = 0), các ô còn lại mang giá trị vô cùng. Sau đó, chúng ta sẽ lặp và xây dựng kết quả với các kích thước từ bé đến lớn, cụ thể là các tập con với kích thước 2, 3, 4,... một cách tuần tự. Đối với mỗi trường hợp, chúng ta sẽ tìm chi phí tối ưu từ các tập con nhỏ hơn. Chúng ta sẽ điền vào bảng DP với công thức tổng quát như sau[3]:

$$DP[S][v] = \min_{\substack{u \in S' \\ u \neq v}} \{DP[S'][u] + \text{cost}(u, v)\}$$

Trong hàng cuối của bảng DP, mỗi ô DP[S][v] chứa chi phí tối thiểu để đi qua tất cả các thành phố có trong S, kết thúc ở v. Để tìm chi phí tối ưu của TSP, ta lấy giá trị nhỏ nhất từ hàng cuối cùng:

$$TSP\_Cost = \min_{v \neq \text{start}} \{DP[S_{\text{all cities}}][v] + \text{cost}(v, \text{start})\}$$

Song song với DP, ta cũng sẽ sử dụng một bảng Prev để lưu thành phố u tốt nhất đã đi qua trước khi đến v mỗi khi cập nhật DP[S][v] để phục vụ cho việc trace back Path của chu trình Hamilton. Khi đã tìm ra thành phố v cuối cùng trước khi quay về Start, ta sẽ sử dụng giá trị trong tại Prev[S][v] để tìm ra thành phố trước đó, và cứ như thế cho đến khi tìm thấy Start, ta sẽ được con đường hoàn chỉnh.

Đối với trường hợp từ 20 đỉnh trở lên

Khi số đỉnh là 20 trở lên thì độ phức tạp của Held-Karp trở thành vấn đề, thuật toán hoạt động rất chậm và thậm chí nếu trên 23 thì nó dễ dàng bị TLE (Time Limit Exceeded) trên các môi trường test. Do đó ta sẽ chuyển từ thuật toán chính xác (Exact Algorithm) sang thuật toán xấp xỉ (Approximation Algorithm), hay còn gọi là Heuristic Algorithm. Các phương pháp heuristic thông thường tuy đơn giản nhưng dễ rơi vào bẫy tối ưu cục bộ. Do đó, ta chuyển hướng sang metaheuristic — một lớp thuật toán cấp cao cho phép khai thác cấu trúc lời giải và khám phá thông minh hơn. Ta sẽ chọn Ant Colony Optimization (ACO) — một thuật toán metaheuristic mô phỏng hành vi bầy kiến — với khả năng tìm lời giải gần tối ưu với hiệu quả

cao[4]. Đồng thời chúng ta cũng sẽ áp dụng Local Search (2-opt) vào để cải thiện chu trình nếu có thể.

Đầu tiên, chúng ta sẽ khởi tạo một ma trận pheromone với kích thước như ma trận kề của đồ thị và gán cho một giá trị ban đầu như nhau. Rồi ta khởi tạo một máy chạy số ngẫu nhiên với xác suất như nhau, phục vụ cho việc ra quyết định đối với mỗi con kiến.

Phần chính của thuật toán sẽ là việc khởi tạo vòng lặp cho đến giá trị `MAX_ITERATIONS`. Trong mỗi lần lặp đó, chúng ta lại cho `NUM_ANTS` con kiến hoạt động. Ta tạo một ma trận 2 chiều để theo dõi đường đi của các con kiến trong mỗi lần lặp và ma trận 1 chiều để theo dõi trạng thái `visited` của từng đỉnh cho mỗi con kiến. Trong `V` bước (với `V` là số lượng đỉnh), ta sẽ lần lượt cân nhắc `V` đỉnh trong đồ thị như là đỉnh kế tiếp. Nếu đường đi đến đỉnh đó (edge) có tồn tại và đỉnh đó chưa được đi qua trước đó, ta sẽ tính xác suất đến mỗi đỉnh này bằng công thức sau[5]:

$$p_{ij}^k = (\tau_{ij})^\alpha (\eta_{ij})^\beta, \quad j \in N_i^k$$

Trong đó:

- p_{ij}^k: xác suất để con kiến k đi từ đỉnh i đến đỉnh j
- τ_{ij}: lượng pheromone trên cạnh nối từ i đến j
- η_{ij}: thông tin heuristic trên cạnh (i, j), là $\eta_{ij} = \frac{1}{d_{ij}}$, với d_{ij} là khoảng cách giữa hai đỉnh
- α: hệ số điều chỉnh ảnh hưởng của pheromone (τ)
- β: hệ số điều chỉnh ảnh hưởng của heuristic (η)
- N_i^k: tập các đỉnh mà con kiến k có thể đi đến từ đỉnh i (tức là các đỉnh chưa được thăm)

Sau khi tính được xác suất đến từng đỉnh và lấy tổng lại (`totalProb`), để tìm khoảng của đỉnh mà con số sinh ngẫu nhiên thuộc về (con số sinh ngẫu nhiên nằm trong khoảng [0,1], ta sẽ nhân cho `totalProb` để chuẩn hoá giá trị đó nằm trong khoảng [0, `totalProb`]), ta dùng một vòng lặp khởi tạo xác suất cộng dồn (`cumulativeProb`) bằng 0 và cộng dần xác suất của từng đỉnh (p_{ij}^k) vào, một khi giá trị của `cumulativeProb` vượt qua giá trị ngẫu nhiên được chuẩn hoá đó, nghĩa là giá trị ngẫu nhiên được chuẩn hoá đó đã lọt vào phân phối xác suất của một đỉnh. Đỉnh đó sẽ là đỉnh tiếp theo mà con kiến sẽ đi qua.

Sau khi con kiến đi hết chu trình, ta kiểm tra tính hợp lệ của chu trình và nếu có, ta sẽ tính tổng chi phí của chu trình đó. Đồng thời ta cũng sẽ so sánh để chọn ra chu trình tốt nhất trong những chu trình của tất cả các con kiến. Nếu chu trình tốt nhất không thay đổi (hay không cải thiện), ta sẽ tăng chỉ số `improvedThisIter` lên 1 đơn vị. Còn nếu có cải thiện thì ta reset chỉ số này về 0. Nếu chỉ số này bằng hoặc vượt quá `NO_IMPROVEMENT_LIMIT` thì thoát khỏi vòng lặp ngay lập tức (early stopping) để giảm số lần lặp không cần thiết.

Một khi tất cả các con kiến đã hoàn thành và ta đã cập nhật chu trình tốt nhất cho 1 lần lặp (`iter`), ta tiến hành cập nhật ma trận mùi (pheromone) như thể hiện trong công thức sau[6]:

$$\tau_{ij} \leftarrow (1 - \rho) \cdot \tau_{ij} + \sum_{k=1}^{m} \Delta\tau_{ij}^{(k)}$$

$$\Delta\tau_{ij}^{(k)} = \begin{cases} \frac{Q}{L^{(k)}} & \text{nếu kiến } k \text{ đi qua cạnh } (i, j) \\ 0 & \text{ngược lại} \end{cases}$$

Trong đó:

- ρ: hệ số tỷ lệ bay hơi của pheromone (τ)
- Q: lượng pheromone mà kiến để lại trên tour
- $L^{(k)}$: chi phí để đi hết 1 tour của con kiến k

Sau cùng, ta áp dụng giải thuật Local Search (2-opt) (*) cho chu trình (tour) tốt nhất trong tất cả các lần lặp (`iter`) và nếu có cải thiện, đó sẽ là chu trình cuối cùng mà ta trả về.

Dây là các tham số mặc định cho giải thuật ACO:

```
const int NUM_ANTS = 60;     //nhiều kiến -> đa dạng tour hơn
const int MAX_ITERATIONS = 500; //lặp nhiều -> cơ hội tìm ra tour optimal hơn
const double ALPHA = 1.1;  //càng cao -> tin vào dấu vết cũ nhiều hơn
const double BETA = 2.0;   //càng cao -> thiên về chọn cạnh ngắn hơn
const double EVAPORATION = 0.45;//càng cao -> giảm trí nhớ cũ, khám phá hơn
const double Q = 100.0;    //càng lớn -> tour tốt ảnh hưởng mạnh hơn
const int NO_IMPROVEMENT_LIMIT = 100; //ngăn lặp nhiều mà không cải thiện
```

Tóm lại, ta chọn thuật toán ACO cho trường hợp trên 20 đỉnh vì nó là một thuật toán metaheuristic, tức là hướng đến toàn cục thay vì cục bộ dẫn đến dễ bị mắc kẹt và bị bẫy như các thuật toán tham lam; và nó cũng có tính chất ngẫu nhiên dựa trên xác suất (mặc dù ta có sử dụng seed cố định để đảm bảo tính ổn định của kết quả khi test), thay vì tất định (deterministic) như các thuật toán tham lam đơn giản. Với độ phức tạp là $O(A \times I \times V^2)$ (V là số đỉnh, A là số kiến `NUM_ANTS` và I là số lần lặp `MAX_ITERATIONS`), thuật toán này trở thành một công cụ hữu hiệu hứa hẹn những lời giải chất lượng cho bài toán TSP với đồ thị lớn.

(*) 2-opt là một kĩ thuật tối ưu hoá, nó cải thiện một tour được cho bằng cách liên tục xoá 2 cạnh bất kì (trừ cạnh đầu vì nó nối với đỉnh bắt đầu để khép kín chu trình) và nối lại sao cho tổng chi phí của tour giảm đi. Ví dụ, đối với 2 cạnh (A $\rightarrow$ B) và (C $\rightarrow$ D) ta có cách xếp lại duy nhất là (A $\rightarrow$ C) và (B $\rightarrow$ D) [7]. Nếu thay đổi cải thiện chi phí của tour thì ta thực hiện. Để thực hiện thay đổi ta chỉ cần đảo ngược đoạn từ B đến C như trong ví dụ. Thuật toán 2-opt cơ bản chỉ phù hợp cho STSP (Symmetric TSP); đối với ATSP (Asymmetric TSP) thì mọi thứ không còn đúng theo cách đó nữa, vì 2 chiều giữa 2 đỉnh có khoảng cách khác nhau. Do đó bên cạnh sự thay đổi chi phí của 2 cạnh được "cắt và nối", ta sẽ kiểm tra thêm sự thay đổi chi phí của đoạn bị đảo ngược lại để xem liệu tour có thật sự được cải thiện không. Nếu cải thiện thì ta mới thật sự đảo được lại để hoàn thành một lần lặp của thuật toán. Cứ thế lặp lại đối với tất cả các cặp cạnh cho đến khi không cải thiện được nữa (đạt đến tối ưu hoá cục bộ). Thuật toán có độ phức tạp thời gian là $O(V^2)$ [8].

3.2 Cách chạy code

Chạy file bellman.h và bellman.cpp

Kiểm tra file cần thiết Đảm bảo có các file:

- `main.cpp`: File chính chứa hàm `main` để thực thi.
- `bellman.h`: File tiêu đề khai báo các hàm, lớp liên quan đến thuật toán Bellman-Ford.
- `bellman.cpp`: File nguồn chứa định nghĩa của các hàm đã khai báo trong `bellman.h`.

Câu lệnh biên dịch Mở terminal và chạy lệnh sau để biên dịch:

```
g++ -DUSE_BELLMAN main.cpp bellman.cpp -o bellman
```

- `-DUSE_BELLMAN`: Là định nghĩa macro (giả sử cần kích hoạt các phần mã liên quan đến `USE_BELLMAN`).
- `main.cpp` và `bellman.cpp`: Là các file nguồn được biên dịch.
- `-o bellman`: Tạo ra tệp thực thi có tên là `bellman`.

Chạy chương trình Sau khi biên dịch thành công, chạy chương trình bằng lệnh:

```
./bellman
```

Chạy file tsm.h và tsm.cpp

Kiểm tra file cần thiết Đảm bảo có các file:

- `main.cpp`: File chính chứa hàm `main`.
- `tsm.h`: File tiêu đề khai báo các hàm, lớp liên quan đến bài toán TSP.
- `tsm.cpp`: File nguồn chứa định nghĩa các hàm trong `tsm.h`.

Câu lệnh biên dịch Mở terminal và chạy lệnh sau:

```
g++ main.cpp tsm.cpp -o tsm
```

- `main.cpp` và `tsm.cpp`: Là các file nguồn được biên dịch.
- `-o tsm`: Tạo ra tệp thực thi có tên là `tsm`.

Chạy chương trình Sau khi biên dịch thành công, chạy chương trình bằng lệnh:

```
./tsm
```

References

[1] Sahil Bansal, *Held-Karp Algorithm for TSP*, CompGeeks, 2020. Available at: https://compgeek.co.in/held-karp-algorithm-for-tsp/. Accessed: June 3, 2025.

[2] GeeksforGeeks, *Comparison between Adjacency List and Adjacency Matrix Representation of Graph*, October 8, 2021. Available at: https://www.geeksforgeeks.org/comparison-between-adjacency-list-and-adjacency-matrix-representation-of-graph/. Accessed: June 4, 2025.

[3] Quang Nhat Nguyen, *Travelling Salesman Problem and Bellman-Held-Karp Algorithm*, May 2020. Available at: https://www.math.nagoya-u.ac.jp/~richard/teaching/s2020/Quang1.pdf. Accessed: June 5, 2025.

[4] Marco Dorigo, Mauro Birattari, and Thomas Stützle, *Ant Colony Optimization*, IEEE Computational Intelligence Magazine, November 2006. Available at: https://www.cs.princeton.edu/courses/archive/fall24/cos597C/bib2/Ant_colony_optimization.pdf. Accessed: June 16, 2025.

[5] Lê Ngọc Quang, *Giải bài toán tìm đường đi ngắn nhất bằng thuật toán song song meta-heuristic*, Luận văn Thạc sĩ Kỹ thuật, Đại học Dà Nẵng, 2012. Available at: https://luanvan.net.vn/luan-van/luan-van-giai-bai-toan-tim-duong-di-ngan-nhat-bang-thuat-toan-song-song-meta-heuristic-51 Accessed: June 16, 2025.

[6] Anirudh Shekhawat, Pratik Poddar, Dinesh Boswal, *Ant Colony Optimization Algorithms: Introduction and Beyond*, Artificial Intelligence Seminar, Indian Institute of Technology Bombay, 2009. Available at: https://mat.uab.cat/~alseda/MasterOpt/ACO_Intro.pdf. Accessed: June 17, 2025.

[7] Matej Gazda, *TSP Algorithms: 2-opt, 3-opt in Python*, February 8, 2019. Available at: http://matejgazda.com/tsp-algorithms-2-opt-3-opt-in-python/. Accessed: June 17, 2025.

[8] KeiruaProd, *Traveling Salesman Problem with 2-opt*, September 15, 2021. Available at: https://www.keiruaprod.fr/blog/2021/09/15/traveling-salesman-with-2-opt.html. Accessed: June 18, 2025.