

OR TEST REPORT

Quang Nguyen Van

Abstract—Báo cáo này bao gồm hai phần: phần 1 về Algorithms and Data Structures, và phần 2 về OR (Operation Research). Ở phần 2 sử dụng một thuật toán phát triển từ thuật toán tham lam để giải quyết, sau đó so sánh với các thuật toán như Ant Colony Optimization[4] [3], Evolutionary Algorithm[2] [5], để giải quyết bài toán NP-hard cụ thể là bài toán ATSP (Asymmetric Traveling Salesman Problem).
CODE: github.com/quangvan99/or_test

keywords—Algorithms and Data Structures, OR, NP-hard, ATSP

Contents

1	Phần 1: Algorithms and Data Structures	1
1.1	Giới thiệu đề bài	1
1.2	Bài 1: Count Islands	1
1.2.1	Ý tưởng:	1
1.2.2	Độ Phức Tạp	1
1.2.3	Pseudocode	2
1.2.4	Kết quả	2
1.3	Bài 2: Best Safe Cost	2
1.3.1	Ý Tưởng và độ phức tạp	2
1.3.2	Pseudocode	2
1.4	Kết luận	2
2	Phần 2: Operations Research	3
2.1	Phát biểu bài toán ATSP và TSP	3
2.2	Giới thiệu đề bài	3
2.3	Xác định các vấn đề	3
2.4	Xử lý vấn đề	3
2.4.1	Module 1: Histogram của khoảng cách giữa các điểm.	4
2.4.2	Module 2: Weighted Matrix	4
2.4.3	Module 3: Revisiting - tìm kiếm lại	5
2.4.4	Module 4: Local search với 2-otp	5
2.5	Bài 1: tìm đường đi chấp nhận được.	6
2.6	Bài 2	6
2.7	Bài 3	6
2.7.1	Evolutionary Algorithm (EA)	6
2.7.2	Ant Colony Optimization(ACO)	6
2.7.3	Kết quả thực nghiệm	7
2.8	Kết luận và hướng phát triển	8

References 8

1. Phần 1: Algorithms and Data Structures

1.1. Giới thiệu đề bài

Bài toán yêu cầu bạn điều khiển một con tàu từ San Diego về Việt Nam thông qua Đại Tây Dương. Đại Dương được biểu diễn dưới dạng ma trận A có kích thước N x M, trong đó N là số hàng và M là số cột ($N \leq 100$, $M \leq 100$).

- Nếu $A(i, j) = 's'$, đó là vị trí xuất phát của bạn. Nếu $A(i, j) = 'f'$, đó là đích đến của bạn. Chỉ có một điểm xuất phát và một điểm đến.
- Nếu $A(i, j) = 0$, đó là đất liền. Các khu vực đất liền kề nhau có thể tạo thành một hòn đảo hoặc lục địa. Chỉ xem xét 4 hướng: trái, phải, lên và xuống để xác định mối quan hệ hàng xóm giữa hai khu vực đất.
- Nếu $A(i, j) \geq 0$, biểu thị độ sâu của nước. Đất liền không có nước nhưng con tàu của bạn không thể vào đất. Càng sâu nước, nguy hiểm càng cao. Giả định rằng $0 \leq A(i, j) \leq 9$, ngoại trừ điểm xuất phát và điểm đến.

- Đầu vào là tệp "ocean.in". Dòng đầu tiên chứa N và M, cách nhau bởi một khoảng trắng. N dòng tiếp theo, mỗi dòng là một hàng của ma trận A. Ví dụ, chúng ta có ma trận 5 hàng và 6 cột. Điểm bắt đầu 's' nằm ở hàng đầu tiên và cột thứ ba. Điểm đến 'f' nằm ở hàng cuối/cột đầu tiên.

Table 1. Example test case

ocean.in	ocean.out1	ocean.out2
5 6 10s120 000201 402131 311541 f10002	4 1,1,3,5.	9

Bài 1

Hãy đếm số hòn đảo và liệt kê kích thước của từng hòn đảo theo thứ tự tăng dần.
ghi vào tệp "ocean.out1". Dòng đầu tiên: số lượng hòn đảo. Dòng thứ hai: danh sách kích thước của các hòn đảo, được phân tách bởi dấu phẩy "," và kết thúc bởi dấu chấm ".".
Với ví dụ ở table 1, với tệp đầu ra là "ocean.out1". Có 4 hòn đảo và Kích thước của chúng lần lượt là 1, 1, 3 và 5.

Bài 2

Hãy tìm con đường an toàn nhất để trở về nhà (từ 's' về 'f'), nghĩa là tổng độ sâu của nước trên con đường này là nhỏ nhất. Tàu không thể vào đất, tàu có thể đi lên, đi xuống, sang trái và sang phải (tức là 4 hướng)
Ghi vào tệp "ocean.out2". Chỉ có một dòng duy nhất biểu thị tổng mức độ sâu của con đường tốt nhất có thể. Nếu không có đường đi nào, hãy viết '-1'.
Với ví dụ ở table 1, với tệp đầu ra là "ocean.out2". Con đường tốt nhất là: phải, xuống, xuống, trái, xuống, trái, xuống, trái. Mức độ sâu của nước trên con đường này là: 1, 2, 1, 2, 1, 1, 1. Tổng của chúng là 9.

1.2. Bài 1: Count Islands

1.2.1. Ý tưởng:

- Ta duyệt từng ô trong ma trận. Nếu ô là hòn đảo, ta sử dụng Thuật toán loang (Flood Fill) để ghé thăm và đánh dấu những vùng đất lân cận của hòn đảo đó.
- Với thuật toán Flood Fill ta có thể sử dụng phương pháp duyệt theo chiều sâu (DFS) hoặc duyệt theo chiều rộng (BFS). Ở trong báo cáo, sử dụng DFS để code trông ngắn gọn hơn.
- Cuối cùng, đếm số lượng các hòn đảo và trả về kích thước của mỗi hòn đảo.

1.2.2. Độ Phức Tạp

- Thời Gian: Độ phức tạp thời gian của thuật toán là $O(n * m)$, trong đó n là số hàng và m là số cột của ma trận. Do phải duyệt qua từng ô trong ma trận một lần và thực hiện DFS cho mỗi ô.
- Không Gian: Độ phức tạp không gian của thuật toán cũng là $O(n * m)$, do cần lưu trữ ma trận đầu vào và một tập hợp các ô đã được ghé thăm.

1.2.3. Pseudocode

Algorithm 1: Count Islands Algorithm

```

Input: ocean: the input ocean
Output: number of islands and their sizes
1 Function dfs(cur):
    // Mark the current cell as visited
2     visited.add(cur);
3     size = 1;
4     for (dx, dy) ← [(0, 1), (0, -1), (1, 0), (-1, 0)] do
5         x, y ← cur[0] + dx, cur[1] + dy;
6         if (0 ≤ x < n and 0 ≤ y < m and
            ocean[x][y] == '0' and (x, y) ∉ visited)
            then
7             size += dfs((x, y));
8     return size;
9 Function countIslands(ocean):
10    visited ← empty set;
11    lands ← sorted list of islands sizes;
12    for i ← 0 to n do
13        for j ← 0 to m do
14            if ocean[i][j] == 0 and (i, j) ∉ visited
            then
15                lands.append(dfs((i, j)));
16    return number of islands, list of island sizes;

```

1.2.4. Kết quả

Table 2. Result of test case 2

ocean.in2	ocean.out1	ocean.out2
5 6 10s120 000001 400131 311541 f10002	3 1,3,8.	-1

Table 3. Result of test case 3

ocean.in3	ocean.out1	ocean.out2
5 6 11s721 197351 111131 981541 f13672	0	11

1.3. Bài 2: Best Safe Cost

1.3.1. Ý Tưởng và độ phức tạp

Bài toán được đặt ra để tìm đường đi từ một ô bắt đầu đến một ô kết thúc trong một ma trận với chi phí nhỏ nhất từ ô bắt đầu đến ô kết thúc.

- Brute Force Approach:**

Ta có thể sử dụng phương pháp Brute Force để giải quyết bài toán này. Đối với mỗi ô trên ma trận, ta có thể thử tất cả các hướng di chuyển có thể (trên, dưới, trái, phải) để xác định các đường đi có thể từ ô bắt đầu đến ô kết thúc. Tuy nhiên, độ phức tạp của phương pháp này sẽ là $O(4^{nm})$, với *n* là số hàng của ma trận và *m* là số cột của ma trận.

- Dynamic Programming Approach:**

Để giảm độ phức tạp, ta có thể áp dụng kỹ thuật Dynamic Programming. Trong quá trình tìm kiếm, ta lưu trữ một bảng lưu trữ chi phí nhỏ nhất để đến từ ô bắt đầu đến mỗi ô trên ma trận. Mỗi khi ta di chuyển đến một ô mới, ta sẽ cập nhật giá trị chi phí nhỏ nhất cho ô đó. Quá trình này được lặp lại cho tất cả các ô trên ma trận cho đến khi đến ô kết thúc. Độ phức tạp ở đây là $O(nm)$.

<i>n</i>	Big O Times		Thời gian xử lý (s) trên chip 3.2GHz	
	$4^{n \times n}$	$n \times n$	$4^{n \times n}$	$n \times n$
1	1	4	1.25×10^{-9}	3.1×10^{-10}
2	4	256	8×10^{-8}	1.25×10^{-9}
3	9	262144	8.192×10^{-5}	2.8×10^{-9}
...
17	289	9.8E+173	3.0×10^{164}	9.0×10^{-8}
18	324	1.1E+195	3.6×10^{185}	1.0×10^{-7}
19	361	2.2E+217	6.8×10^{207}	1.1×10^{-7}
20	400	6.6E+240	2.0×10^{231}	1.2×10^{-7}
21	441	3.2E+265	1.0×10^{256}	1.3×10^{-7}

Table 4. So sánh thời gian tương đối xử lý cho bài toán Count Islands với Brute Force và Dynamic Programming. ($= \text{BigOtimes}/(3.2 \times 10^9)$)

⇒ Sử dụng Dynamic Programming giúp giảm đáng kể độ phức tạp của bài toán từ $O(4^{nm})$ của Brute Force xuống còn $O(nm)$, trong việc tìm kiếm đường đi an toàn với chi phí nhỏ nhất trên ma trận.

1.3.2. Pseudocode

Algorithm 2: Best Safe Cost Algorithm

```

Input: ocean: the input ocean, start: the starting point,
        end: the destination
Output: the safest cost of reaching the destination
1 Function bestSafeCost(ocean, start, end):
2     costSoFar ← {start: 0}; queue ← [(start, 0)];
3     while queue is not empty do
4         cur, cost ← queue.pop(0);
5         for (dx, dy) ← [(0, 1), (0, -1), (1, 0), (-1, 0)] do
6             x, y ← cur[0] + dx, cur[1] + dy;
7             if (0 ≤ x < n and 0 ≤ y < m) and
                (ocean[x][y] ∉ [0, 's']) then
8                 newCost ← cost + ocean[x][y];
9                 if (x, y) ∉ costSoFar or newCost <
                    costSoFar[(x, y)] then
10                    costSoFar[(x, y)] ← newCost;
11                    queue.append(((x, y), newCost));
12    return costSoFar[end];

```

1.4. Kết luận

Chúng ta đã giải quyết 2 bài toán trên. Đặc điểm chung của 2 bài toán này là sử dụng ma trận để biểu diễn và từ 1 ô của ma trận chỉ di chuyển được theo bốn hướng: trái, phải, trên và dưới.

Trong phần tiếp theo ta sẽ tìm hiểu bài toán TSP. Đây là bài toán có thể sử dụng ma trận để biểu diễn, nhưng độ phức tạp sẽ cao hơn bởi mỗi ô của ma trận, ta có thể di chuyển qua các đỉnh trong đồ thị. Mỗi bước di chuyển giữa hai đỉnh có thể được biểu diễn bằng một cạnh trong đồ thị, và giá trị của cạnh này thể hiện chi phí di chuyển giữa hai thành phố tương ứng. Do đó khi liệt kê tất cả các trường hợp độ phức tạp thay vì $O(4^{nm})$, nó sẽ là $O(n!)$

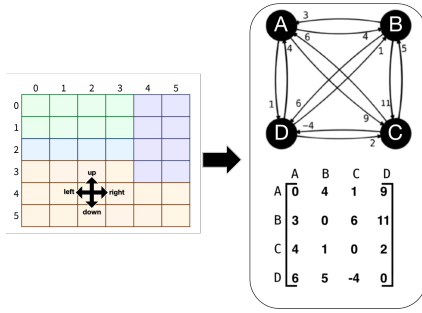


Figure 1. From matrix to graph

2. Phần 2: Operations Research

2.1. Phát biểu bài toán ATSP và TSP

Định nghĩa:

- **ATSP** (Asymmetric Traveling Salesman Problem): Cho một đồ thị có hướng $G = (V, E)$, trong đó V là tập hợp các đỉnh và E là tập hợp các cạnh có hướng, ATSP yêu cầu tìm một chu trình Hamilton có tổng trọng số nhỏ nhất.
- **TSP** (Traveling Salesman Problem): Cho một đồ thị đầy đủ có trọng số $G = (V, E, w)$, trong đó V là tập hợp các đỉnh và E là tập hợp các cạnh, TSP yêu cầu tìm một chu trình Hamilton có tổng trọng số nhỏ nhất.

Trọng số của cạnh:

- **ATSP**: Trọng số của cạnh e_{ij} có thể khác với trọng số của cạnh e_{ji} .
- **TSP**: Trọng số của cạnh e_{ij} phải bằng trọng số của cạnh e_{ji} .

Ma trận trọng số:

- **ATSP**: Ma trận trọng số W có thể không đối xứng.
- **TSP**: Ma trận trọng số W là ma trận đối xứng.

Ràng buộc về hướng đi:

- **ATSP**: Có ràng buộc về hướng đi trong một số trường hợp.
- **TSP**: Không có ràng buộc về hướng đi.

Sự khác biệt chính giữa ATSP và TSP nằm ở tính không đối xứng của các cạnh.

Để tìm nghiệm tối ưu cho bài toán. Ta có thể sử dụng Dynamic Programming với độ phức tạp $O(n^2 * 2^n)$. Tuy nhiên khi số thành phố > 30 , nhìn vào bảng dưới đây, mất quá nhiều thời gian xử lý. Khi đó ta cần sử dụng những thuật toán khác để khả thi về mặt thời gian hơn.

Big O Time			Time xử lý (s) chip 3.2GHz	
n	$n!$	$n^2 \times 2^n$	$n!$	$n^2 \times 2^n$
1	1	2	3.1×10^{-10}	6.2×10^{-10}
2	2	16	6.25×10^{-10}	5×10^{-9}
3	6	72	1.8×10^{-9}	2.2×10^{-8}
4	24	256	7.5×10^{-9}	8×10^{-8}
...
28	3.0×10^{29}	2.1×10^{11}	9.5×10^{19}	65.7
29	8.8×10^{30}	4.5×10^{11}	2.7×10^{21}	141.0
30	2.6×10^{32}	9.6×10^{11}	8.2×10^{22}	301.9
31	8.2×10^{33}	2.0×10^{12}	2.5×10^{24}	644.9

Table 5. So sánh thời gian tương đối xử lý cho bài toán TSP với Brute Force và Dynamic Programming. ($= \text{BigOtime} / (3.2 * 10^9)$)

2.2. Giới thiệu đề bài

Cho một tập hợp các nút $V = \{0, 1, \dots, n+1\}$ và khoảng cách giữa mỗi cặp nút $i \in V$ và $j \in V$ được cho bởi d_{ij} . Một đường đi khả thi phải thỏa mãn các ràng buộc sau:

- Đường đi phải bắt đầu tại nút 0 và kết thúc tại nút $n+1$
- Mỗi nút được ghé thăm đúng một lần
- Việc di chuyển từ nút $i \in V \setminus \{0, n+1\}$ đến nút $j \in V \setminus \{0, n+1\}$ bị cấm nếu (i) i là một số chẵn, (ii) j là một số lẻ, và (iii) $i < n/2$
- Việc di chuyển từ nút $i \in V \setminus \{0, n+1\}$ đến nút $j \in V \setminus \{0, n+1\}$ bị cấm nếu (i) i là số lẻ, (ii) j là số chẵn, và (iii) $i \geq n/2$

Tìm đường đi khả thi mà làm giảm thiểu hàm $(n \Delta \max_{i,j \in V} d_{ij} + D)$, trong đó D là tổng khoảng cách của tuyến đường, và Δ là

sự khác biệt giữa khoảng cách giữa cặp nút xa nhất và cặp nút gần nhất của một đường đi đã cho.

Ví dụ, xét đường đi $0 - 3 - 1 - 2 - 4$ với các khoảng cách giữa các cặp nút tương ứng $d_{03} = 10, d_{31} = 5, d_{12} = 8, d_{24} = 7, D = 30$ và $\Delta = d_{03} - d_{31} = 5$.

Bài 1

Xây dựng chương trình đọc từ file csv theo format sau:

Node_ID	X-cor	Y-cor
0	40.8	24.1
1	82	10
2	34.12	23.1

Tính khoảng cách Euclidean giữa các node và làm tròn đến số thập phân thứ nhất. Cung cấp 1 đường đi thỏa mãn điều kiện vào file txt sử dụng format sau:

Path : 0 - 1 - 2 - 3 - 4 - 5

Total distances : 2323.12

Bài 2

- Q1) How is your method scaling?
 Q2) What is the required time by your method to get a good solution?
 Q3) How would you change your method if you were asked to significantly reduce the computational time?
 Q4) How would you change your method to deal with dynamic arrival of requests?
 Q5) What aspects of the problem could be enhanced to include stochastic information?

Bài 3 Q6) Explore the potential of **Evolutionary Algorithm** (EA) and **Ant Colony Optimization** (ACO)! Try to implement them for this problem and show us the comparison (in table) among baselines!

2.3. Xác định các vấn đề

Đây là bài toán ATSP. Ta phải cố gắng xử lý những vấn đề sau đây:

- Số lượng đường cấm trong đồ thị tương đối lớn.
- Thuật toán không hoạt động tốt khi số lượng đỉnh lớn.
- Hàm mục tiêu có yếu tố mới.

2.4. Xử lý vấn đề

Thuật toán đề xuất mà báo cáo phát triển sẽ dựa vào 4 module nhỏ. Trong đó module 3 để tạo đường đi chấp nhận được cho

bài 1. Và 3 module còn lại để giải quyết bài toán ATSP, tóm tắt các bước xử lý như sau:

B1. Chọn các thresholds.

Từ ma trận khoảng cách, tính histogram dựa trên $n_{bin} = 0.7 \times n$. Chọn ngưỡng từ min d_{ij} đến $bins[j_{max} + \frac{n_{bin}-j_{max}}{3}]$ với bước $step = 1$. Trong đó $bins$ là giá trị của các bin của histogram, j_{max} là chỉ số vị trí của đỉnh trong histogram.

B2. Khởi tạo đường đi qua $m\%$ đỉnh đầu tiên.

Với mỗi *threshold*, sử dụng thuật toán tham lam để tìm đỉnh kế tiếp *next* trên đường đi hiện tại. Với điều kiện đỉnh *next* là đỉnh ngẫu nhiên thuộc danh sách 3 phần tử nhỏ nhất dựa vào công thức (*), và xác suất chọn đỉnh *next* thuộc danh sách đó dựa vào công thức softmax (**), *i* là thành phố hiện tại, *j* là các thành phố láng giềng của *i*:

$$(*) \text{ cost}_j = |d_{ij} - \text{threshold}|, \text{ next}_j = \text{argsort}(\text{cost}_j)[: 3]$$

$$(**) P(i) = \frac{e^{-\frac{1}{\text{cost}_j}}}{\sum_{j=1}^3 e^{-\frac{1}{\text{cost}_j}}}$$

Tiếp tục tìm cho đến khi đủ $m = \alpha * n$ (trong thử nghiệm $\alpha = 0.2$) thành phố đầu tiên. Lưu $m\%$ thành phố đầu tiên này làm đường đi khởi tạo cho bước tiếp theo.

B3. Hoàn thiện đường đi.

Với $m\%$ thành phố đầu tiên, tiếp tục sử dụng thuật toán tham lam để tìm $(100\% - m\%)$ đỉnh còn lại

Với điều kiện đỉnh kế tiếp $\text{next} = \text{argmin}((n\Delta \max_{ij} d_{ij} + D))$,

với:

- *i* và *j* là các cặp đỉnh của các cạnh trên đường đi hiện tại.
- *D* là tổng khoảng cách tích lũy của đường đi hiện tại.
- Δ là sự khác biệt giữa khoảng cách lớn nhất và nhỏ nhất của cặp cạnh trên đường đi hiện tại.

Tiếp tục đi hết các thành phố còn lại.

B4. Hiệu chỉnh đường đi

Áp dụng thuật toán tìm kiếm địa phương 2-opt cho mỗi đường đi, để tìm ra đường đi tốt hơn.

B5. Chọn lời giải tốt nhất

Sau khi đã thực hiện 2-opt cho tất cả các đường đi, chọn đường đi có giá trị hàm mục tiêu nhỏ nhất từ danh sách các kết quả.

2.4.1. Module 1: Histogram của khoảng cách giữa các điểm.

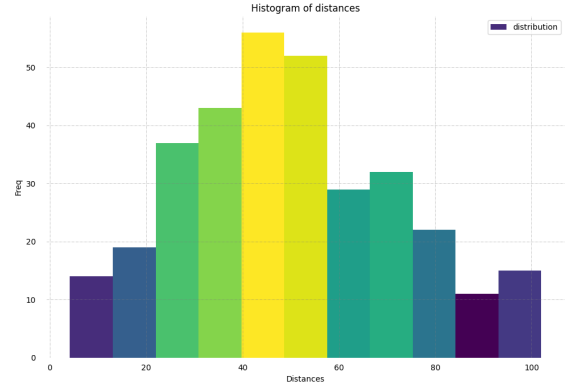


Figure 2. Histogram of distances

Vì đây khoảng cách Euclid, nên khi vẽ histogram khoảng cách giữa các điểm, thì hình dạng của nó giống hình chuông. Ta chọn khoảng cách xuất hiện nhiều nhất trong histogram làm *threshold*. (đỉnh chuông)

- Nếu trong mỗi lần di chuyển, sử dụng thuật toán tham lam để tìm đỉnh kế tiếp *next* trên đường đi hiện tại, với điều kiện đỉnh $\text{next} = \text{argmin}(d_{ij} - \text{threshold})$, *i* là thành phố hiện tại, *j* là các thành phố láng giềng của *i* mà khoảng cách $d_{ij} > \text{threshold}$. Thì thực nghiệm sử dụng công thức chặn dưới ở trên cho ra kết quả nghiệm không tốt. Cho nên thuật toán đề xuất sử dụng công thức $\text{next} = \text{argmin}(\text{abs}(d_{ij} - \text{threshold}))$, với *i* là thành phố hiện tại, *j* là các thành phố láng giềng của *i*. Ta sẽ chọn khoảng cách gần với *threshold* nhất, tại đây có số lượng đường đi lớn hơn từ đó Δ có xu hướng nhỏ hơn, và cũng giảm tỷ lệ đường đi là không chấp nhận được.
- Khi *threshold* dịch về phía bên trái của histogram, $\max d_{ij}$ sẽ càng nhỏ.

So sánh histogram của đường đi bất kì và đường đi mong muốn như Figure 3:

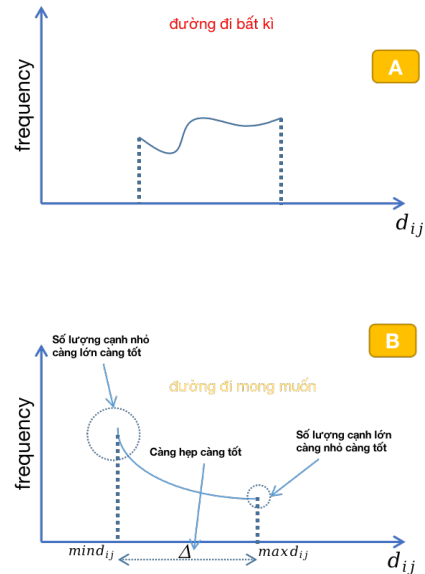


Figure 3. Phân bố khoảng cách của các cạnh trên 1 đường đi.

2.4.2. Module 2: Weighted Matrix

Ví dụ $n = 20$, bắt đầu từ 0 và kết thúc tại 21. Ở nhóm 1 ta có: các node 0, 1, 3, 5, 7, 9, 10, 12, 14, 16, 18, 20, 21 sẽ đi được tất

cả các nút. Ở nhóm 2 ta có: các node 2,4,6,8,11,13,15,17,19 chỉ đi được một số node.

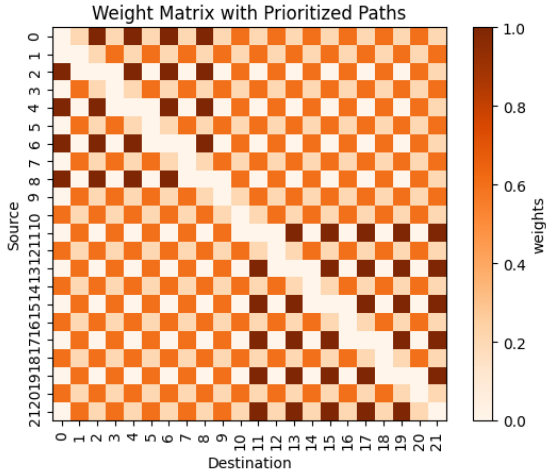


Figure 4. Weight Matrix with Prioritized Paths

- Nhóm 1: node $i=0, 1, 3, 5, 7, 9, 10, 12, 14, 16, 18, 20, 21$ đến được tất cả các node $V \setminus \{i\}$
- Nhóm 2: node $i=2, 4, 6, 8$ đến các nút j chẵn $\in V \setminus \{0, i\}$, node $i=11, 13, 15, 17, 19$ đến các nút j lẻ $\in V \setminus \{0, i\}$

Cho ví dụ sau:

0	14	1	9	20	3	4	2	8	16	12	6	18	11	15	5	10	13	7	17	19	21

ta để ý, chẵn sẽ thường sang chẵn, và lẻ sẽ thường sang với lẻ.

Ta sẽ **ưu tiên hơn** (nhân với hệ số $\alpha < 1$):

- Cặp (i, j) chẵn, và (i, j) lẻ đi với nhau.
- Đặc biệt cặp (i, j) lẻ với $i \geq n/2$ và cặp (i, j) chẵn với $i < n/2$ ta sẽ ưu tiên nhiều hơn.

sử dụng **weighted matrix** khác ma trận khoảng cách ban đầu để tìm đường đi. Sử dụng module này sẽ **giảm được tỷ lệ đường đi không chấp nhận được**.

2.4.3. Module 3: Revisiting - tìm kiếm lại

Khác với kỹ thuật Backtracking, vì Backtracking là một kỹ thuật thiết kế giải thuật dựa trên đệ quy trong chiến thuật tìm kiếm theo chiều sâu. Ở đây sử dụng chiến lược tìm kiếm theo chiều rộng, khi gặp đường cụt nó sẽ quay lại, nên tạm gọi là: Revisiting. Ở kỹ thuật này, ta chỉ cần lùi lại node $i-1$, lưu một dictionary *forbidden* (key là path, value là đỉnh bị chặn) để chặn path trước đó sang node i . Khi gặp những lần đi tới nó sẽ né đường sang i và đi hướng khác chấp nhận được.

Ví dụ: ta đã đi được $path = [0, 2, 4, 5]$

Tại $i = 5$ đường đi kế tiếp bị chặn, ta lùi lại node 4. Tại đây $path = (0, 2, 4)$, $forbidden = \{(0, 2, 4) : [5]\}$. Khi đó những đường đi từ $(0, 2, 4)$ sang 5 sẽ bị chặn, để đi qua đường khác.

2.4.4. Module 4: Local search với 2-otp

Bắt đầu từ một đường đi chấp nhận được, ta cố liên tục cố gắng cải thiện nó bằng cách thay đổi trong đường đi đó và kiểm tra xem nếu thay đổi đó có làm giảm chi phí hay không? Nếu có thì cập nhật. Ở báo cáo này, sử dụng kỹ thuật 2-otp[1], trình bày ở **Algorithm 3**. Với kỹ thuật này, ta tưởng tượng như tâm của đồng cát histogram sẽ dịch về phía bên trái so với ban đầu, $\min d_{ij}$ sẽ nâng lên hơn, miêu tả như hình vẽ dưới đây:

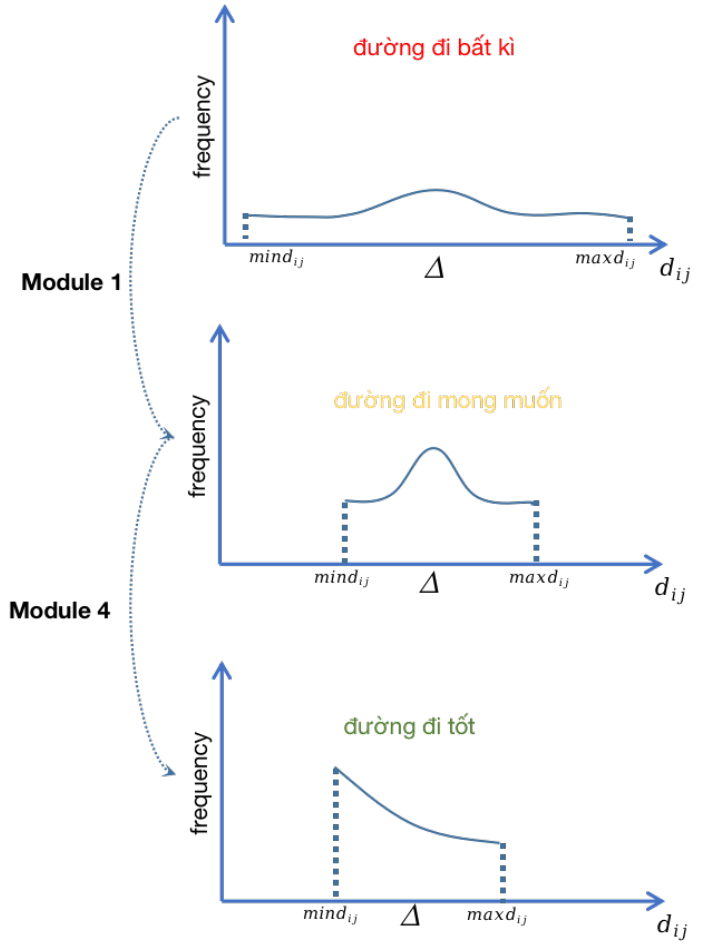


Figure 5. Distribution of a solution when using 2-otp

Algorithm 3: 2-opt Local Search Algorithm

Input: tour: list representing the tour, startNode: index of the starting node, obj_fn : objective function

Output: optimized tour after local search

```

1 Function LocalSearch(tour, startNode):
    // Perform local search on the given tour
    // using 2-opt algorithm
2     stable ← False;
3     best ← obj_fn(tour);
4     while not stable do
5         stable ← True;
6         for i ← startNode to n do
7             for j ← i + 1 to n+1 do
8                 newTour ← copy of tour;
9                 swap(newTour[i], newTour[j]);
10                sub ← newTour[i-1:j+2];
11                // The sub satisfies the conditions
12                // of the problem
13                if not isValid(sub) then
14                    continue;
15                nNewTour ← obj_fn(newTour);
16                if best > nNewTour then
17                    tour, best ← newTour, nNewTour;
18                stable ← False;
19 return tour;

```


- Tạo các kiến con: Mỗi con kiến sẽ di chuyển qua các đỉnh trong đồ thị, và cập nhật mùi pheromone trên mỗi con đường mà nó đi qua.
- Lựa chọn con đường: Các con kiến sẽ lựa chọn con đường đi dựa trên một số tiêu chí như mức pheromone và chi phí của đường đi.
- Cập nhật mùi pheromone: Sau khi tất cả các con kiến đã hoàn thành hành trình, mùi pheromone trên các con đường sẽ được cập nhật dựa trên hiệu suất của các con kiến.
- Lặp lại quá trình: Quá trình lựa chọn và cập nhật pheromone được lặp lại cho đến khi điều kiện dừng được đạt đến, chẳng hạn như số lượng vòng lặp tối đa hoặc khi đạt được giải pháp tối ưu mong muốn.

Sau khi có ma trận pheromone, nhờ ma trận này ta có thể tìm feasible solutions mà không tốn quá nhiều thời gian để có thể dự đoán một cách tốt nhất các nút đi tiếp theo, thay vào đó ta có thể sử dụng bất kỳ thuật toán tìm kiếm nào cho đồ thị luôn như: Greedy search.

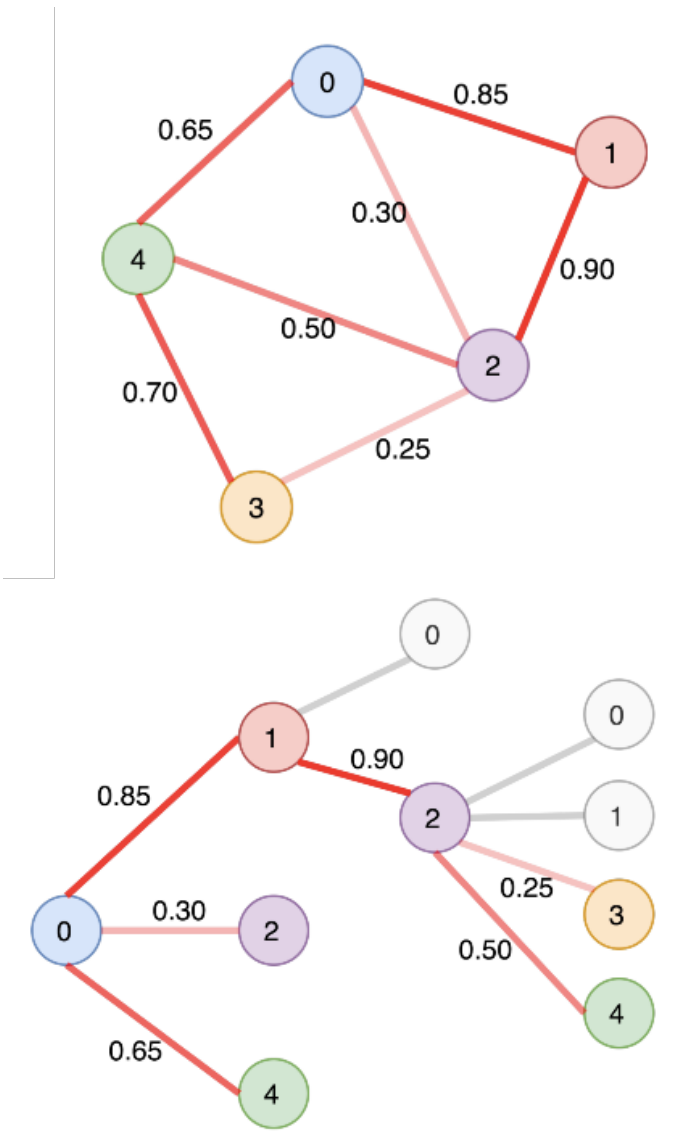


Figure 7. Pheromone graph

Thuật toán mà báo cáo sử dụng là Elitist ant system[4], một biến thể của ACO. Trong thuật toán này, giải pháp tốt nhất ở mỗi vòng lặp sẽ gửi pheromone vào dấu vết của nó sau mỗi lần lặp lại.

2.7.3. Kết quả thực nghiệm

Bảng dưới đây là kết quả thực nghiệm 3 thuật toán khác nhau: MyTSP, Evolutionary Algorithm(EA), Ant Colony Optimization(ACO). Kết hợp local search với 2-otp ở mỗi thuật toán.

Thông số cài đặt:

- Kết quả của mỗi thuật toán là nhỏ nhất của 5 lần chạy. Mỗi lần chạy lấy kết quả nhỏ nhất của nhiều epoch.
- Thuật toán EA mặc định với các thông số sau: n_population=100, n_epoch=100, tournament_size=4, mutation_rate=0.5, crossover_rate=0.9.
- Thuật toán ACO mặc định với các thông số sau: n_ant=50, n_epoch=100, alpha=1.0, beta=1.0, rho=0.5, del_tau=1.0, k=2.
- Sử dụng chip M1, ram 8GB.

Sự hội tụ của EA và ACO+2otp được mô tả ở **Figure 8** và **Figure 9**

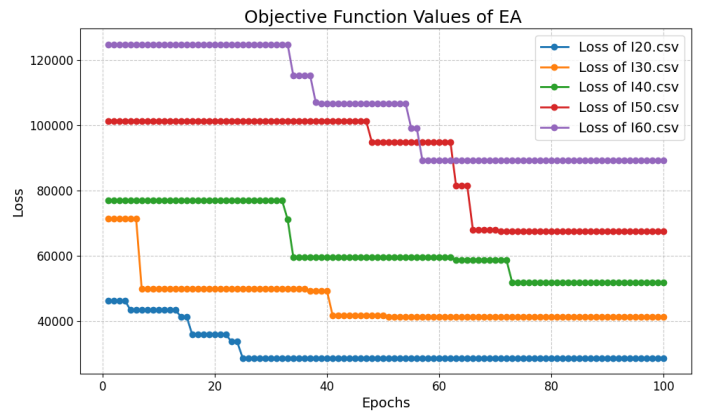


Figure 8. Objective Function Values of EA

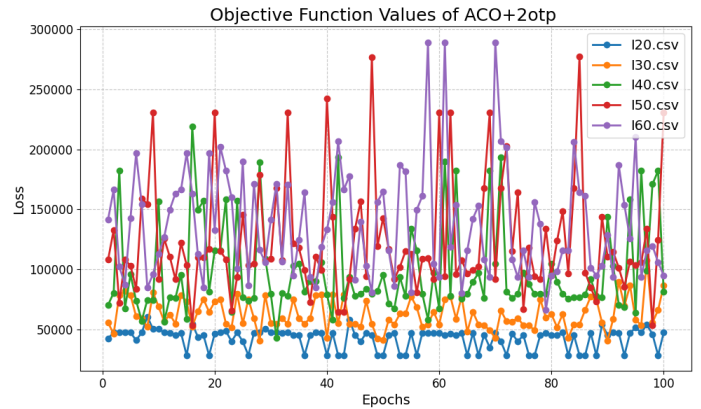


Figure 9. Objective Function Values of ACO+2otp

Method	I20.csv	I30.csv	I40.csv	I50.csv	I60.csv
EA	28548.8	41241.2	51787.4	67397.6	89259.8
ACO	46454.8	63223.8	71796.3	102143	149264.4
ACO+2otp	28331.3	40543.6	43016.6	55508.9	65978.5
MyTSP	21922.6	39726.4	33246.6	57868.1	57567.0

Table 6. Giá trị nhỏ nhất hàm mục tiêu của các thuật toán, trên các tập test

Về giá trị hàm mục tiêu: Kết quả thống kê ở **Table 6**. Thuật toán myTSP cho ra đường đi tốt hơn ACO+2otp ở đa số tập test. Và đánh bại hoàn toàn EA và ACO trên tất cả tập test. Điều này cho ta thấy tính hiệu quả về việc tìm nghiệm của

Method	I20.csv	I30.csv	I40.csv	I50.csv	I60.csv
EA	0.63	1.29	2.1	3.7	5.5
ACO	6.9	16	29	46	73
ACO+2otp	7.46	18.8	34.2	58	81
MyTSP	0.045	0.146	0.25	0.6	0.84

Table 7. So sánh về mặt thời gian của các thuật toán

MyTSP. Đối với thuật toán ACO, dường như phải có thêm local search để thuật toán tốt hơn.

Về mặt thời gian: Được mô tả ở **Table 7**. MyTSP vượt trội hơn các thuật toán khác. **Nhanh hơn từ 90 đến 100 lần** so với thuật toán tốn nhiều thời gian nhất trong bảng là ACO+2otp,

2.8. Kết luận và hướng phát triển


Thuật toán MyTSP ứng dụng rất tốt cho những bài toán đòi hỏi thời gian chạy ngắn để tìm đường đi đủ tốt.

Trong tương lai chúng ta có thể phát triển thêm dựa vào Q2, Q5 để cải thiện nghiệm cho MyTSP. Ngoài ra có thể sử dụng những phương pháp cải thiện đường đi nhờ tìm lời giải trong quá trình vòng lặp, từ đó sẽ tiếp cận tốt hơn cho các bài toán thực tế.

References

- [1] G. A. Croes, “A method for solving traveling-salesman problems,” *Operations Research*, vol. 6, pp. 791–812, 1958. [Online]. Available: <https://api.semanticscholar.org/CorpusID:123646893>.
- [2] J. H. Holland, “Adaptation in natural and artificial systems,” 1975.
- [3] M. Dorigo and L. M. Gambardella, “Ant colony system: a cooperative learning approach to the traveling salesman problem,” *IEEE Transactions on Evolutionary Computation*, vol. 1, no. 1, pp. 53–66, Apr. 1997. DOI: [10.1109/4235.585892](https://doi.org/10.1109/4235.585892).
- [4] S. Negulescu, C. Oprean, C. Kifor, and I. Carabulea, “Elitist ant system for route allocation problem,” Jan. 2008.
- [5] T. Alam, S. Qamar, A. Dixit, and M. Benaïda, “Genetic algorithm: Reviews, implementations, and applications,” 2020. arXiv: [2007.12673](https://arxiv.org/abs/2007.12673) [cs.OH].

Contact:

 vanquang@gmail.com