ĐẠI HỌC QUỐC GIA, THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA KHOA HỌC VÀ KỸ THUẬT MÁY TÍNH

CẤU TRÚC RỜI RẠC CHO KHMT (CO1007)

BÁO CÁO BÀI TẬP LỚN

Traveling Salesman Problem

GVHD: Trần Hồng Tài
SV thực hiện: Mai Hồng Phước
MSSV: 2412803
Lớp: L01

TP Hồ Chí Minh, Tháng 6 Năm 2025

Mục lục

1 Cơ sở lý thuyết

1.1 Đồ thị

Trong toán học và tin học, đồ thị là đối tượng nghiên cứu cơ bản của lý thuyết đồ thị. Thông thường, đồ thị được vẽ dưới dạng một tập các điểm (đỉnh, nút) nối với nhau bởi các đoạn thẳng (cạnh) và được chia thành nhiều loại khác nhau tùy vào mục đích sử dụng:

- **Đồ thị có hướng:** Trong đồ thị có hướng, mỗi cạnh đều mang một chiều cụ thể, tức là có hướng đi từ một đỉnh này đến một đỉnh khác.
- **Đồ thị vô hướng:** Ngược lại, trong đồ thị vô hướng, các cạnh không có hướng, cho phép di chuyển tự do theo cả hai chiều giữa hai đỉnh nối với nhau.
- **Đa đồ thị:** là loại đồ thị cho phép tồn tại nhiều cạnh nối giữa cùng một cặp đỉnh. Điều này nghĩa là hai đỉnh có thể được nối với nhau bởi hơn một cạnh.
- **Đồ thị khuyên:** là đồ thị có ít nhất một cạnh nối từ một đỉnh trở lại chính nó. Cạnh như vậy được gọi là khuyên.

1.2 Ứng dụng

- **Tìm đường đi ngắn nhất:** Sử dụng trong các ứng dụng như GPS và lập kế hoạch hành trình, tìm kiếm tuyến đường nhanh nhất giữa các điểm.
- **Thiết kế mạch điện:** Chúng tôi đánh giá liệu một mạch điện có thể được triển khai trên một bảng mạch phẳng hay không bằng cách sử dụng lý thuyết đồ thị.
- **Tô màu bản đồ:** Đồ thị giúp tìm ra số màu tối thiểu cần dùng để tô các vùng trên bản đồ sao cho các vùng kề nhau không có cùng màu.
- **Lập lịch và phân bố kênh:** Đồ thị hỗ trợ trong việc lập lịch thi và phân bổ kênh cho các đài truyền hình sao cho không bị trùng lặp hay gây nhiễu.
- **Kết nối mạng:** Mô hình đồ thị được dùng để kiểm tra xem hai máy tính có được kết nối với nhau thông qua một liên kết truyền thông hay không.
- **Cấu trúc hóa học:** Đồ thị giúp phân biệt các hợp chất hóa học có cùng công thức phân tử nhưng khác nhau về cấu trúc.

2 Bài toán Traveling Salesman (TSP)

2.1 Giới thiệu

Traveling Salesman Problem là một bài toán NP-khó thuộc thể loại tối ưu rời rạc hay tổ hợp được nghiên cứu trong vận trù học hoặc lý thuyết khoa học máy tính. Bài toán được phát biểu như sau. Cho trước một danh sách các thành phố và khoảng cách giữa chúng, tìm chu trình ngắn nhất thăm mỗi thành phố đúng một lần.

Bài toán được nêu ra lần đầu tiên năm 1930 và là một trong những bài toán được nghiên cứu sâu nhất trong tối ưu hóa. Nó thường được dùng làm thước đo cho nhiều phương pháp tối ưu hóa. Mặc dù bài toán rất khó giải trong trường hợp tổng quát, có nhiều phương pháp giải chính xác cũng như heuristic đã được tìm ra để giải quyết một số trường hợp có tới hàng chục nghìn thành phố.

2.2 Mô tả bài toán

Bài toán đặt ra: Tìm hành trình ngắn nhất để có thể đi qua tất cả các đỉnh và trở về đỉnh ban đầu với chi phí thấp nhất.

Bản đồ đường đi được mô hình hóa bởi lý thuyết đồ thị và các đường đi được biểu diễn dưới dạng mảng gồm ba phần tử mang ý nghĩa như sau:

$$edge[a][b][c]$$

Trong đó: a là đỉnh đầu (hay đỉnh xuất phát)
b là đỉnh cuối (hay đỉnh đích)
c là trọng số của đường đi đó (có thể là chi phí, thời gian)

2.3 Các phương pháp giải

- **Brute Force** (Liệt kê hoán vị)
  - **Ý tưởng:** Duyệt qua tất cả hoán vị của các đỉnh, tính tổng chi phí của đường đi và chọn ra đường đi có chi phí thấp nhất.
  - Độ phức tạp: $O(n!)$
  - Ưu điểm: Dễ cài đặt, sử dụng
  - Nhược điểm: Thuật toán chạy chậm khi đồ thị có số đỉnh lớn hơn 10.
- **Quy hoạch động - Thuật toán Held-Karp**
  - **Ý tưởng:** Gọi $dp[mask][u]$ là chi phí nhỏ nhất để đi qua tập đỉnh $mask$ và kết thúc tại đỉnh u.
  - Độ phức tạp: $O(n^2 \times 2^n)$
  - Ưu điểm: Nhanh hơn Brute Force, hoạt động tốt khi số đỉnh < 20.

- **Backtracking kết hợp với kỹ thuật nhánh cận (Branch and Bound)** [1]
  - **Ý tưởng:** Duyệt theo phương pháp DFS (Depth-First Search), tìm chi phí thấp nhất và cắt những nhánh có chi phí dự kiến cao hơn lời giải tốt nhất tìm được.
  - Độ phức tạp: Tốt hơn $O(n!)$ trong thực tế
  - Ưu điểm: Thời gian thực hiện được cải thiện so với Brute Force
  - Nhược điểm: Vẫn còn chậm khi số đỉnh lớn hơn 20
- **Heuristics (Thuật toán kinh nghiệm)** [2]
  - **Ý tưởng:** Tìm lời giải gần đúng dựa trên kinh nghiệm hoặc chiến lược tìm kiếm nhanh.
  - Ưu điểm: Tốc độ rất nhanh (thường là tuyến tính hoặc gần tuyến tính), cho nghiệm đủ tốt trong thời gian thực
  - Nhược điểm: Không đảm bảo tối ưu

2.4 Ý tưởng thực hiện

Trong bài tập lớn này, em lựa chọn triển khai hai hướng tiếp cận để giải quyết bài toán *Travelling Salesman Problem (TSP)* với quy mô khác nhau: thuật toán Held-Karp và thuật toán Simulated Annealing.

Thuật toán **Held-Karp** là một phương pháp chính xác, sử dụng kỹ thuật quy hoạch động kết hợp với bitmask để tối ưu hoá việc duyệt tất cả các hoán vị của tập đỉnh. Phương pháp này có độ phức tạp tính toán là $O(n^2 \cdot 2^n)$ nên chỉ phù hợp cho các đồ thị nhỏ, thường có số đỉnh không vượt quá 20. Khi áp dụng cho các bài toán có kích thước nhỏ, thuật toán này đảm bảo tìm ra lời giải tối ưu tuyệt đối.

Tuy nhiên, với các trường hợp có số đỉnh lớn hơn 20, thuật toán Held-Karp trở nên không khả thi do giới hạn về thời gian và bộ nhớ. Do đó, em sử dụng thêm thuật toán **Simulated Annealing (SA)** – một thuật toán gần đúng (heuristic) dựa trên quá trình làm nguội trong vật lý, nhằm tìm ra lời giải gần tối ưu trong thời gian ngắn hơn. SA hoạt động dựa trên cơ chế chấp nhận lời giải tệ hơn trong giai đoạn đầu để tránh bị kẹt tại cực trị địa phương (local optimum), và dần dần giảm xác suất này khi hệ thống "nguội".

Bằng cách kết hợp hai thuật toán này, em có thể xử lý hiệu quả bài toán TSP ở cả hai mức quy mô: sử dụng Held-Karp cho đồ thị nhỏ (dưới 20 đỉnh) để đảm bảo độ chính xác, và Simulated Annealing cho đồ thị lớn hơn (trên 20 đỉnh) để đảm bảo hiệu năng. Ý tưởng này nhằm khai thác điểm mạnh riêng của từng thuật toán, từ đó đưa ra giải pháp phù hợp với từng trường hợp đầu vào cụ thể.

2.5 Hiện thực thuật toán Held-Karp

2.5.1 Ý tưởng

- Sử dụng kỹ thuật bitmasking để biểu diễn những đỉnh đã đi qua bằng chuỗi bit với trạng thái 0, 1 biểu diễn những đỉnh đã đi qua.
- Dùng quy hoạch động để lưu lời giải những bài toán con
- Biến được sử dụng chính để quy hoạch động: $dp[mask][u]$

Trong đó: $mask$ là một số nguyên có dạng nhị phân (bitmask), thể hiện các đỉnh đã được đi qua
u là đỉnh kết thúc của hành trình hiện tại
$dp[mask][u]$ là chi phí nhỏ nhất để đi qua tất cả đỉnh trong $mask$ và kết thúc tại u

2.5.2 Mã giả (Pseudocode)

Hàm HKSolve: Sử dụng giải thuật Held-Karp để tìm đường đi ngắn nhất

```
numVertices
dp[1 << startIndex][startIndex] = 0
for mask ← 0 to 1 << numVertices − 1 do
  for u ← 0 to numVertices − 1 do
    if Đỉnh u tồn tại trong hành trình mask hoặc hành trình mask tới u tồn tại then
      for  dov ← 0 to numVertices − 1
        if Đỉnh v không tồn tại trong hành trình và tồn tại đường đi từ u đến v then
          Xác định trạng thái tiếp theo nextMask
          if Kết quả tốt hơn dp[nextMask][v] then
            Cập nhật giá trị
          end if
        end if
      end for
    end if
  end for
end for
Tìm đường đi dựa
```

2.5.3 Chi tiết hóa

Đoạn mã 1. Khai báo hằng số

```
#define INF 1000000000
```

Đoạn mã 2. Định nghĩa hàm *HKSolve*

```
string HKSolve(vector<vector<int>>& cost, int charToIndex[256], vector<char>&
    vertices, char start) {
    int numVertices = vertices.size();
    int states = 1 << numVertices;

    // Create spaces for "1 << numVertices" states
    vector<vector<int>> dp(states, vector<int>(numVertices, INF));
    vector<vector<int>> parent(states, vector<int>(numVertices, -1));

    int startIndex = charToIndex[start];

    dp[1 << startIndex][startIndex] = 0;

    for (int mask = 0; mask < states; ++mask) {
        for (int u = 0; u < numVertices; ++u) {
            // Inspect u's existence int state "mask"
            if (!(mask & (1 << u))) continue;

            if (dp[mask][u] >= INF) continue;
            for (int v = 0; v < numVertices; ++v) {
                if (mask & (1 << v)) continue;
                if (cost[u][v] == -1) continue;

                int nextMask = mask | (1 << v);
                int dis = dp[mask][u] + cost[u][v];

                if (dis < dp[nextMask][v]) {
                    dp[nextMask][v] = dis;
                    parent[nextMask][v] = u;
                }
            }
        }
    }

    int minWeight = INF, last = -1;
    int finalMask = states - 1;

    for (int u = 0; u < numVertices; ++u) {
        if(u == startIndex || cost[u][startIndex] == -1) continue;
        int curWeight = dp[finalMask][u] + cost[u][startIndex];
        if (curWeight < minWeight) {
            minWeight = curWeight;
            last = u;
```

```
        }
    }

    if (minWeight >= INF) {
        return "";
    }

    vector<int> path;
    int mask = finalMask, u = last;
    while (u != -1) {
        path.push_back(u);
        int previous = parent[mask][u];
        mask ^= (1 << u);
        u = previous;
    }
    reverse(path.begin(), path.end());
    path.push_back(charToIndex[start]);

    string result = "";
    for (int i = 0; i < path.size(); ++i) {
        result += vertices[path[i]];
        if (i < path.size() - 1) {
            result += ' ';
        }
    }
    return result;
}
```

2.6 Hiện thực thuật toán Simulated Annealing

2.6.1 Ý tưởng

Thuật toán Simulated Annealing (SA) là một phương pháp tối ưu hóa ngẫu nhiên mô phỏng quá trình ủ nhiệt trong vật lý. Ý tưởng chính của SA là tìm kiếm lời giải tối ưu bằng cách cho phép chấp nhận tạm thời những lời giải kém hơn, nhằm tránh bị kẹt tại các cực trị địa phương (local optimum).

Mô phỏng từ vật lý: Trong luyện kim, annealing là quá trình nung vật liệu ở nhiệt độ cao rồi làm nguội từ từ để các phân tử sắp xếp lại thành cấu trúc ổn định nhất, có năng lượng thấp nhất. Tương tự như trong SA, nhiệt độ cao cho phép hệ thống thử nghiệm các lời giải khác biệt lớn, kể cả lời giải xấu. Khi nhiệt độ thấp dần, hệ thống dần ổn định, chỉ chấp nhận lời giải tốt hơn.

Áp dụng vào bài toán Traveling Salesman Problem

- Coi trọng số của đường đi giữa 2 cạnh là năng lượng
- Mỗi hành trình là một trạng thái
- Simulated Annealing hoán đổi đổi các đỉnh trong hành trình để tạo lời giải mới

2.6.2 Mã giả

Hàm SASolve

Khởi tạo hành trình ban đầu S
Khởi tạo $T = 1000$, $T_{min} = 0.01$
while $T > T_{min}$ **do**
  S' là biến thể của S
  $\Delta E = cost(S') - cost(S)$
  if $\Delta E < 0$ hoặc *random probability* $< e^{-\frac{\Delta E}{T}}$ **then**
    Chấp nhận S'
  end if
end while
Hiện thức

2.6.3 Chi tiết hóa

Đoạn mã 3. Định nghĩa hàm *calculatePath* tính độ dài hành trình

```
int calculatePath(vector<int>& currentPath, vector<vector<int>>& cost) {
    int numVertices = cost.size();
    int result = 0;
    for (int i = 0; i < numVertices; ++i) {
        if (cost[currentPath[i]][currentPath[i + 1]] > 0) {
            result += cost[currentPath[i]][currentPath[i + 1]];
        }
        else return INF;
    }
    return result;
}
```

Đoạn mã 4. Định nghĩa hàm *getNeighbor* tạo ra hành trình mới

```
vector<int> getNeighbor(const vector<int>& currentPath, mt19937& rng) {
    vector<int> neighbor = currentPath;
    int numVertices = neighbor.size() - 1;

    // Swap 2 vertex randomly except the first and the last vertex
    uniform_int_distribution<int> dist(1, numVertices - 1);
    int i = dist(rng);
    int j = dist(rng);
    while (i == j) {
        j = dist(rng);
    }
    swap(neighbor[i], neighbor[j]);
    return neighbor;
}
```

Đoạn mã 5. Định nghĩa hàm *SASolve* hiện thực giải thuật Simulated Annealing

```
string SASolve(vector<vector<int>>& cost,int charToIndex[256], vector<char>&
    vertices, char start) {
   // initiate the first path
   vector<int> currentPath;
   int startIndex = charToIndex[start];
   for (int i = 0; i < cost.size(); ++i) {
       if (i != startIndex) currentPath.push_back(i);
   }
   // Shuffle the order of vertices
   random_device rd;
   mt19937 rng(rd());
   shuffle(currentPath.begin(), currentPath.end(),rng);
   currentPath.insert(currentPath.begin(), startIndex);
   currentPath.push_back(startIndex);
   int bestCost = calculatePath(currentPath, cost);
   vector<int> bestPath = currentPath;
   // Initialize parameters
   double T = 1000;
   double T_min = 0.01;
   double alpha = 0.995;
   int iteration = 1000;
   uniform_real_distribution<double> prob(0.0, 1.0);
   while (T > T_min) {
       for (int i = 0; i < iteration; ++i) {
           vector<int> neighbor = getNeighbor(currentPath, rng);
           int currentCost = calculatePath(currentPath, cost);
           int neighborCost = calculatePath(neighbor, cost);
           int delta = neighborCost - currentCost;

           if (delta < 0 || prob(rng) < exp(-delta / T)) {
               currentPath = neighbor;
               if (neighborCost < bestCost) {
                   bestCost = neighborCost;
                   bestPath = neighbor;
               }
           }
       }
       T *= alpha;
   }
   string result = "";
   for (int i = 0; i < bestPath.size(); ++i) {
       result += vertices[bestPath[i]];
       if (i < bestPath.size() - 1) {
           result += ' ';
       }
   }
   if (bestCost < INF) return result;
   return "";
}
```

2.7 Hiện thực hàm Traveling

Đoạn mã 4. Định nghĩa hàm *getNeighbor* tạo ra hành trình mới

```
string Traveling(int graph[][3], int num_edges, char start) {
    int numVertices = 0;
    vector<char> vertices;
    for (int i = 0; i < num_edges; ++i) {
        if (find(vertices.begin(), vertices.end(), static_cast<char>(graph[i][0]))
            == vertices.end()) {
            vertices.push_back(graph[i][0]);
            ++numVertices;
        }
        if (find(vertices.begin(), vertices.end(), static_cast<char>(graph[i][1]))
            == vertices.end()) {
            vertices.push_back(graph[i][1]);
            ++ numVertices;
        }
    }
    sort(vertices.begin(), vertices.end());
    int charToIndex[256];
    for (int i = 0; i < 256; ++i) {
        charToIndex[i] = -1;
    }
    for (int i = 0; i < numVertices; ++i) {
        charToIndex[vertices[i]] = i;
    }
    vector<vector<int>> cost(numVertices, vector<int>(numVertices, -1));
    for (int i = 0; i < num_edges; ++i) {
        int u = charToIndex[static_cast<char>(graph[i][0])];
        int v = charToIndex[static_cast<char>(graph[i][1])];
        if (cost[u][v] == -1 || cost[u][v] > graph[i][2])
            cost[u][v] = graph[i][2];
    }
    for (int i = 0; i < numVertices; ++i) {
        cost[i][i] = 0;
    }

    if (numVertices >= 20) {
        return SASolve(cost, charToIndex, vertices, start);
    }
    return HKSolve(cost, charToIndex, vertices, start);
}
```

2.8 Ví dụ

File EdgeList.txt

65 66 10
65 67 15
65 68 20
66 67 35
66 68 25
67 68 30
68 65 20
68 66 7
68 67 30
Đường đi: A B C D A
Chi phí: 95

Kết quả

Đường đi: A B C D A
Chi phí: AB + BC + CD + DA = 10 + 35 + 30 + 20 = 95

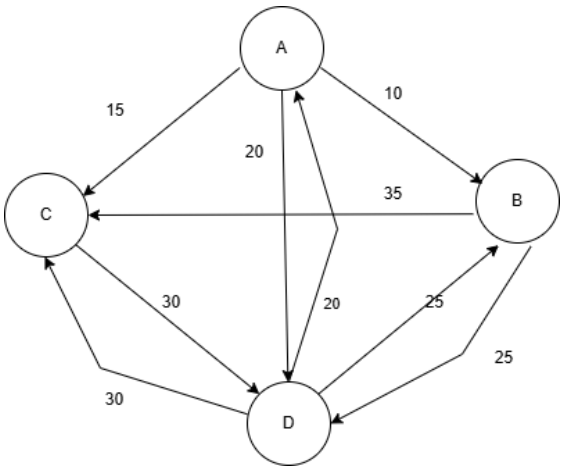

Figure 1: Minh họa đồ thị

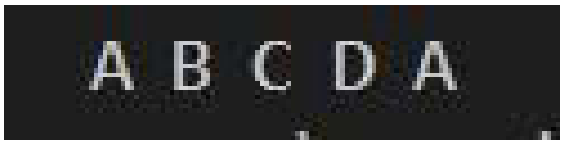

Figure 2: Kết quả chạy chương trình

3 Kết luận

Trong bài báo cáo này, em đã tìm hiểu và triển khai hai thuật toán Held-Karp – một thuật toán chính xác dựa trên kỹ thuật quy hoạch động kết hợp bitmask, và Simulated Annealing – một thuật toán gần đúng thuộc nhóm metaheuristic, lấy cảm hứng từ quá trình ủ nhiệt trong vật lý để giải bài toán *Traveling Salesman Problem* (TSP), một bài toán khá phức tạp trong các lĩnh vực như logistics, microchip, tạo tuyến đường đi cho xe bus, vận chuyển tiền.[3], v.v.

Điểm mạnh của thuật toán *Held-Karp* nằm ở khả năng đưa ra lời giải chính xác tuyệt đối, tuy nhiên lại bị hạn chế bởi chi phí tính toán tăng theo cấp số mũ. Điều này khiến nó chỉ khả thi với các đồ thị nhỏ (thường dưới 20–22 đỉnh). Trong khi đó, Simulated Annealing tuy không đảm bảo tìm ra lời giải tối ưu, nhưng cho kết quả gần đúng với chi phí tính toán thấp hơn nhiều và có thể áp dụng cho các trường hợp lớn hơn (25 đỉnh trở lên), rất phù hợp với bài toán thực tế khi cần đáp ứng thời gian và tài nguyên hạn chế.

Qua quá trình thực hiện, em đã hiểu rõ hơn về cách tư duy giải bài toán tổ hợp, biết cách lựa chọn thuật toán phù hợp tùy theo yêu cầu bài toán (độ chính xác, tốc độ), cũng như cách đánh đổi giữa tính tối ưu và hiệu năng.

Hiện nay, đã có nhiều hướng giải quyết để giải bài toán tối ưu hơn, ví dụ như thuật toán *Branch And Bound* đã được cải thiện bằng cách kết hợp với chiến lược *Heuristics* nhằm tăng hiệu quả xử lý, giảm thời gian thực thi nhưng vẫn cho kết quả sát với nghiệm chính xác. Một trong những hướng đi triển vọng là tích hợp với thuật toán $A*$ sử dụng hàm đánh giá để đánh giá và ưu tiên mở rộng các nhánh có tiềm năng trước, từ đó giúp thu hẹp không gian tìm kiếm một cách thông minh hơn. Bên cạnh đó, các kỹ thuật trong lĩnh vực trí tuệ nhân tạo, điển hình như *Genetic Algorithm* cũng có thể được sử dụng để tìm ra lời giải tốt trong thời gian ngắn, đặc biệt là với những bài toán có không gian tìm kiếm lớn như *Traveling Salesman Problem*. Việc kết hợp giữa phương pháp truyền thống và hiện đại hứa hẹn sẽ mở ra những hướng giải quyết tối ưu trong các bài toán tối ưu với các tổ hợp phức tạp.

Nguồn tham khảo

[1] GeeksforGeeks. *Traveling Salesman Problem using Branch And Bound*. https://www.geeksforgeeks.org/traveling-salesman-problem-using-branch-and-bound-2/. 2023.

[2] Kenneth H Rosen. *Discrete mathematics & applications*. McGraw-Hill, 1999.

[3] Karla L Hoffman, Manfred Padberg, Giovanni Rinaldi, et al. "Traveling salesman problem". In: *Encyclopedia of operations research and management science* 1 (2013), pp. 1573–1578.