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 Cài đặt và thực nghiệm

2.4.1 Khái quát

- Đầu vào:
  - Mảng 2 chiều $graph[][3]$ để biểu diễn đồ thị (kiểu dữ liệu *int*).
  - Số cạnh của đồ thị (kiểu dữ liệu *int*).
  - Đỉnh bắt đầu của chu trình (kiểu dữ liệu *char*)
- Đầu ra: Một chuỗi ký tự chứa các đỉnh để biểu diễn đường đi (kiểu dữ liệu string)

2.4.2 Ý tưởng thực hiện

Trong bài tập lớn này, em sử dụng thuật toán Backtracking kết hợp với kỹ thuật nhánh cận để có thể cắt những nhánh không thể tạo ra lời giải tốt hơn lời giải hiện tại và giúp giảm thời gian chạy đáng kể.

- **Tính cận dưới của nhánh:** Sắp xếp lại mảng theo chiều tăng dần của trọng số ($graph[i][2]$)
- **Cắt nhánh:** Sau khi tính cận dưới, nếu cận dưới đó có giá trị lớn hơn lời giải tốt nhất tìm được thì ngừng duyệt nhánh.
- **Duyệt đường đi:** Sử dụng kỹ thuật đệ quy để kiểm tra tất cả đường đi có thể từ một đỉnh bất kỳ.

2.4.3 Mã giả (Pseudocode)

Hàm bsort: Sắp xếp mảng bằng thuật toán Bubble Sort

Khai báo

$graph[][3]$
$numEdges$

Hiện thức

for $i \leftarrow 0$ to $numEdges - 1$ **do**
  for $j \leftarrow 0$ to $numEdges - 1$ **do**
    if $graph[j][2] > graph[j+1][2]$ **then**
      Đổi giá trị $graph[j][2]$ và $graph[j+1][2]$
    end if
  end for
end for

Hàm calculateBound: Tính cận dưới của nhánh

Khai báo

$visited[100]$
$sumWeight$ {Tổng chi phí của các cạnh đã đi}
$EdgePassed$ {Xét thứ tự của cạnh đang xét}

Hiện thực

for $i \leftarrow 0$ to $numEdges - 1$ **do**
  $v = graph[i][1]$
  if $!visited[v]$ **then** {Kiểm tra đã đi qua đỉnh v chưa}
    $sumWeight = sumWeight + graph[i][2]$
    Tăng biến $EdgePassed$ lên 1 đơn vị
  end if
  if $EdgePassed == numOfVertices$ **then**
    $bound = sumWeight$
  end if
end for

Hàm dfs: Duyệt đường đi

Khai báo

$visited[100]$
$currentPath[100]$
$bestPath[100]$
$minWeight$
$EdgePassed$ {Xét thứ tự của cạnh đang xét}

Hiện thực

if $EdgePassed == numVertices$ **then**
  if Tồn tại đường đi từ đỉnh cuối đến đỉnh bắt đầu của chu trình **then**
    Cập nhật đường đi nếu đó là lời giải được cải thiện
  end if
  Thoát hàm
end if
for $v \leftarrow 0$ to $numOfVertices - 1$ **do**
  if Tồn tại đường đi giữa đỉnh đang xét và các đỉnh khác **then**
    if Chưa đi qua đỉnh v **then**
      $visited[v] = true$ {Cập nhật đã đi rồi}
      Tính cận
      if cận nhỏ hơn giá trị của lời giải tốt nhất hiện tại **then**
        Cập nhật đỉnh vào đường đi
        Gọi đệ quy để tiếp tục tại đỉnh đang xét
      end if
      $visited[v] = false$ {Cập nhật lại thành chưa đi qua đỉnh này để xét nhánh khác}
    end if
  end if
end for

Hàm Traveling: Tạo ra chuỗi ký tự với lời giải tốt nhất

Khai báo

$minWeight = INT\_MAX$

Hiện thực

for $i \leftarrow 0$ to $NumOfVertices - 1$ **do**
  Xét từng nhánh lớn, sử dụng hàm dfs
end for
for $i \leftarrow 0$ to $NumOFVertices$ **do**
  Chuyển các giá trị trong currentPath thành ký tự và xâu vào chuỗi
end for

2.4.4 Chi tiết hóa

Đoạn mã 1. Khai báo biến dùng *extern*

```
extern int bestPath[100];
extern int currentPath[100];
extern bool visited[100];
extern int minWeight;
extern int bound;
```

Đoạn mã 2. Định nghĩa hàm *sort* để sắp xếp danh sách đỉnh theo thứ tự trong bảng *ASCII*

```
void sort(vector<char>& vectorOfVertices) {
    int size = vectorOfVertices.size();
    for (int i = 0; i < size; ++i) {
        for (int j = 0; j < size - i - 1; ++j) {
            if (vectorOfVertices[j] > vectorOfVertices[j + 1]) {
                char tmp = vectorOfVertices[j];
                vectorOfVertices[j] = vectorOfVertices[j + 1];
                vectorOfVertices[j + 1] = tmp;
            }
        }
    }
}
```

Đoạn mã 3. Định nghĩa hàm *bsort*

```
void bsort(int** graph, int num_edges) {
    for (int i = 0; i < num_edges; ++i) {
        for (int j = 0; j < num_edges - i - 1; ++j) {
            if (graph[j][2] > graph[j + 1][2]) {
                // Swap each subarray
                for (int k = 0; k < 3; ++k) {
                    int tmp = graph[j][k];
                    graph[j][k] = graph[j + 1][k];
                    graph[j + 1][k] = tmp;
                }
            }
        }
    }
}
```

Đoạn mã 4. Định nghĩa hàm *CopyGraph* và *Destroy* để tạo ra danh sách cạnh mới

```
int** CopyGraph(int graph[][3], int num_edges) {
    int** newGraph = new int*[num_edges];
    for (int i = 0; i < num_edges; ++i) {
        newGraph[i] = new int[3];
        for (int j = 0; j < 3; ++j) {
            newGraph[i][j] = graph[i][j];
        }
    }
    return newGraph;
}
void destroy(int** graph, int num_edges) {
    for (int i = 0; i < num_edges; ++i) {
        delete[] graph[i];
    }
    delete[] graph;
}
```

Đoạn mã 5. Định nghĩa hàm *CountVertices* để xác định số lượng đỉnh trong đồ thị

```
int countVertices(int graph[][3], int num_edges) {
    int exist[256] = {0};
    for (int i = 0; i < num_edges; ++i) {
        if (exist[graph[i][0]] == 0) {
            ++exist[graph[i][0]];
        }
        if (exist[graph[i][1]] == 0) {
            ++exist[graph[i][1]];
        }
    }
    int sum = 0;
    for (int i = 0; i < 256; ++i) {
        sum += exist[i];
    }
    return sum;
}
```

Đoạn mã 6. Định nghĩa hàm *GetVectorOfVertices* để lấy danh sách các đỉnh đã được sắp xếp theo thứ tự trong bảng ASCII

```
vector<char> getVectorOfVertices(int graph[][3], int num_edges) {
    int num_of_vertices = countVertices(graph, num_edges);
    vector<char> list;

    int index = 0;
    int exist[256] = {0};
    for (int i = 0; i < num_edges; ++i) {
        if (exist[graph[i][0]] == 0) {
            ++exist[graph[i][0]];
            list.push_back((char)graph[i][0]);
        }
        if (exist[graph[i][1]] == 0) {
            ++exist[graph[i][1]];
            list.push_back((char)graph[i][1]);
        }
    }
    sort(list);
    return list;
}
```

Đoạn mã 7. Định nghĩa hàm *findIndexInVector* để lấy chỉ số của đỉnh trong danh sách đỉnh

```
int findIndexInVector(vector<char> list, char vertex) {
    int size = list.size();
    for (int i = 0; i < size; ++i) {
        if (vertex == list[i]) {
            return i;
        }
    }
    return -1;
}
```

Đoạn mã 8. Định nghĩa hàm *getAdjacencyMatrix* (ma trận liền kề)

```
vector<vector<int>> getAdjacencyMatrix(int graph[][3], int num_edges) {
    int num_of_vertices = countVertices(graph, num_edges);
    vector<vector<int>> matrix(num_of_vertices, vector<int>(num_of_vertices, 0));

    vector<char> list_of_vertices = getVectorOfVertices(graph, num_edges);

    for (int i = 0; i < num_edges; ++i) {
        int u = findIndexInVector(list_of_vertices, graph[i][0]);
        int v = findIndexInVector(list_of_vertices, graph[i][1]);
        if (u >= 0 && v >= 0) {
            // Ensure that if the graph is a multigraph, only the edge with the
                smallest weight is kept
            if (matrix[u][v] == 0 || matrix[u][v] > graph[i][2])
                matrix[u][v] = graph[i][2];
        }
    }
    return matrix;
}
```

Đoạn mã 9. Định nghĩa hàm *dfs*

```
void dfs(vector<char>& list, int sumWeight, int numEdgePassed, int numVertices, int
    startIndex, vector<vector<int>>& matrix, int** graph, int num_edges) {
   // Verify the existence of a circuit
   if (numEdgePassed == numVertices) {
       if (matrix[startIndex][currentPath[0]]) {
           sumWeight += matrix[startIndex][currentPath[0]];
           // Update minWeight when a better result is found
           if (sumWeight < minWeight) {
               minWeight = sumWeight;
               for (int i = 0; i < numVertices; ++i) {
                   bestPath[i] = currentPath[i];
               }
           }
       }
       return;
   }

   for (int v = 0; v < numVertices; ++v) {
       if (matrix[startIndex][v]) {
           if (!visited[v]) {
               calculateBound(list, sumWeight, numEdgePassed, graph, num_edges,
                   numVertices);
               visited[v] = true;
               if (bound < minWeight) {
                   currentPath[numEdgePassed] = v;
                   dfs(list, sumWeight + matrix[startIndex][v], numEdgePassed + 1,
                       numVertices, v, matrix, graph, num_edges);
               }
               visited[v] = false;
           }
       }
   }
}
```

Đoạn mã 10. Định nghĩa hàm Traveling

```
    string Traveling(int graph[][3], int num_edges, char start) {
    minWeight = INT_MAX;
    int** tmp_graph = CopyGraph(graph, num_edges);
    bsort(tmp_graph, num_edges);

    vector<char> listVertices = getVectorOfVertices(graph, num_edges);
    int numVertices = listVertices.size();
    vector<vector<int>> matrix = getAdjacencyMatrix(graph, num_edges);

    int startIndex = findIndexInVector(listVertices, start);
    currentPath[0] = startIndex;
    for (int i = 0; i < numVertices; ++i) {
        // Traverse each major branch
        visited[startIndex] = true;
        dfs(listVertices, matrix[startIndex][i], 1, numVertices, i ,matrix,
            tmp_graph, num_edges);
    }

    string result = "";
    for (int i = 0; i <= numVertices; ++i) {
        result += listVertices[bestPath[i]];
        if (i < numVertices ) {
            result += ' ';
        }
    }
    // Reset the value of visited array
    for (int i = 0; i < numVertices; ++i) {
        visited[i] = false;
    }
    // Free the memory allocated for tmp_graph
    destroy(tmp_graph, num_edges);
    if (minWeight < INT_MAX) return result;
    else return "";
}
```

2.5 Phân tích độ phức tạp

- Độ phức tạp thời gian
  - n là cạnh, m là số đỉnh của đồ thị
  - Với hàm **getAdjacencyMatrix** và **calculateBound**, ta cần thực hiện $m \times n$ lần vì cần duyệt hết tất cả các cạnh trong đồ thị kết hợp với việc tìm ra chỉ số của đỉnh tương ứng trong danh sách đỉnh, nên độ phức tạp của hàm là $O(m \times n)$.
  - Với hàm **bsort**, ta sử dụng thuật toán Bubble Sort để sắp xếp nên cần thực hiện $\frac{n(n-1)}{2}$ bước. Vậy độ phức tạp của hàm là $O(n^2)$.
  - Với hàm **dfs** (Depth-First Search), với n cạnh, trong trường hợp xấu nhất ta phải duyệt nhiều nhất $(n-2)!$ đường đi vì hàm bắt đầu xét ở đường đi bậc 2. Vậy đồ phức tạp của hàm là $O(n!)$.

  → Với hàm Traveling, sử dụng tất cả hàm trên khiến cho hàm có độ phức tạp thời gian là:

$$\boxed{O(n!)}$$

- Độ phức tạp không gian: $O(n)$ vì chỉ lưu trạng thái của n đỉnh ở các mảng $visited[100]$, $currentPath[100]$, $bestPath[100]$

2.6 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

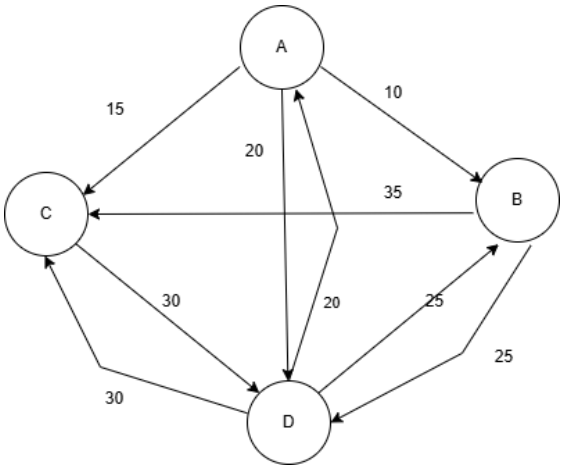

Figure 1: Minh họa đồ thị

Figure 2: Kết quả chạy chương trình

3 Kết luận

Trong bài báo cáo, em đã trình bày thuật toán Backtracking kết hợp với kỹ thuật nhánh cận để đưa ra lời giải cho 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 *Branch and Bound* nằm ở khả năng loại bỏ sớm các nhánh không tiềm năng bằng cách sử dụng các cận dưới. Nhờ đó, không gian tìm kiếm được rút gọn đáng kể, giúp cải thiện hiệu suất và giảm thời gian thực thi so với phương pháp vét cạn.

Mặc dù *Branch and Bound* giúp rút gọn không gian tìm kiếm so với phương pháp vét cạn, thuật toán vẫn gặp khó khăn khi số đỉnh vượt quá 20 do không gian tìm kiếm vẫn tăng theo cấp số nhân. Ngoài ra, việc sử dụng đệ quy sâu dễ dẫn đến tràn bộ nhớ ngăn xếp (stack overflow) trên các hệ thống có giới hạn tài nguyên.

Hiện nay, 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.