BÀI TOÁN TREE EDIT DISTANCE

Toán Tổ Hợp và Lý Thuyết Đồ Thị

1 Lý thuyết cơ bản về Tree Edit Distance

1.1 Định nghĩa

Tree Edit Distance là khoảng cách tối thiểu để biến đổi một cây thành cây khác thông qua một chuỗi các phép biến đổi cơ bản.

Cho hai cây có gốc T_1 và T_2, Tree Edit Distance $\delta(T_1, T_2)$ là số phép biến đổi tối thiểu cần thiết để biến T_1 thành T_2.

1.2 Các phép biến đổi cơ bản

Tree Edit Distance sử dụng ba phép biến đổi cơ bản:

1. **Insertion (Chèn)**: Chèn một nút mới vào cây
2. **Deletion (Xóa)**: Xóa một nút khỏi cây
3. **Substitution (Thay thế)**: Thay đổi nhãn của một nút

Mỗi phép biến đổi có một chi phí (cost) được định nghĩa:

- $\gamma(a \to b)$: Chi phí thay thế nhãn a bằng nhãn b
- $\gamma(a \to \lambda)$: Chi phí xóa nút có nhãn a
- $\gamma(\lambda \to b)$: Chi phí chèn nút có nhãn b

Trong đó λ biểu diễn nút rỗng.

1.3 Tính chất quan trọng

- Tree Edit Distance thỏa mãn tính chất của một metric
- $\delta(T_1, T_2) \geq 0$ với đẳng thức xảy ra khi $T_1 = T_2$
- $\delta(T_1, T_2) = \delta(T_2, T_1)$ (tính đối xứng)
- $\delta(T_1, T_3) \leq \delta(T_1, T_2) + \delta(T_2, T_3)$ (bất đẳng thức tam giác)

1.4 Ứng dụng

- **Bioinformatics**: So sánh cấu trúc phân tử RNA/DNA
- **Compiler Design**: Phân tích sự khác biệt giữa các AST
- **Web Mining**: So sánh cấu trúc HTML/XML
- **Image Processing**: Phân tích cấu trúc hình ảnh
- **Natural Language Processing**: So sánh cấu trúc cú pháp

2 Mô tả bài toán

Đề bài: Cho hai cây có gốc T_1 và T_2 với các nút được gán nhãn. Tìm khoảng cách chỉnh sửa tối thiểu (Tree Edit Distance) giữa hai cây này bằng các phương pháp:

1. **Backtracking**: Duyệt tất cả các khả năng
2. **Branch-and-bound**: Cắt tỉa các nhánh không tối ưu
3. **Divide-and-conquer**: Chia nhỏ bài toán
4. **Dynamic programming - Quy hoạch động**: Tối ưu hóa bằng lưu trữ kết quả

Input:

- Hai cây T_1 và T_2 với các nút được gán nhãn
- Hàm chi phí cho các phép biến đổi

Output:

- Khoảng cách chỉnh sửa tối thiểu
- Chuỗi các phép biến đổi tối ưu (tùy chọn)

3 Ý tưởng và giải pháp

3.1 Phân tích bài toán

Tree Edit Distance là một bài toán tối ưu hóa có thể được giải quyết bằng nhiều cách tiếp cận khác nhau:

- **Độ phức tạp**: Bài toán có độ phức tạp cao do không gian tìm kiếm lớn
- **Cấu trúc con tối ưu**: Bài toán có tính chất cấu trúc con tối ưu
- **Chồng chéo bài toán con**: Các bài toán con có thể lặp lại nhiều lần

Phương pháp	Độ phức tạp	Bộ nhớ	Tối ưu
Backtracking	$O(3^{n+m})$	$O(h)$	Có
Branch-and-bound	$O(3^{n+m}/k)$	$O(h)$	Có
Divide-and-conquer	$O(n \cdot m \cdot (n+m))$	$O(h)$	Có
Dynamic Programming	$O(n^2 \cdot m^2)$	$O(n \cdot m)$	Có

Bảng 1: So sánh các phương pháp giải Tree Edit Distance

3.2 So sánh các phương pháp

Trong đó n, m là số nút của hai cây, h là chiều cao cây, k là hệ số cắt tỉa.

4 Thuật toán chi tiết

4.1 1. Phương pháp Backtracking

Backtracking duyệt tất cả các khả năng biến đổi có thể:

```
Algorithm: BacktrackingTED(T1, T2, i, j)
Input: T1, T2 - hai cay, i, j - chi so nut hien tai
Output: Khoang cach chinh sua toi thieu

1. If T1[i] is empty and T2[j] is empty:
2.     Return 0
3. If T1[i] is empty:
4.     Return cost_insert(T2[j]) + BacktrackingTED(T1, T2, i, j-1)
5. If T2[j] is empty:
6.     Return cost_delete(T1[i]) + BacktrackingTED(T1, T2, i-1, j)
7.
8. // Thu 3 phep bien doi
9. substitute = cost_substitute(T1[i], T2[j]) + BacktrackingTED(T1, T2,
   i-1, j-1)
10. delete_op = cost_delete(T1[i]) + BacktrackingTED(T1, T2, i-1, j)
11. insert_op = cost_insert(T2[j]) + BacktrackingTED(T1, T2, i, j-1)
12.
13. Return min(substitute, delete_op, insert_op)

Time Complexity: O(3^(n+m))
Space Complexity: O(h)  v i  h l  c h i u  cao  c  y
```

Listing 1: Thuật toán Backtracking cho Tree Edit Distance

4.2 2. Branch-and-bound

Branch-and-bound cải thiện backtracking bằng cách cắt tỉa các nhánh không tối ưu:

```
Algorithm: BranchBoundTED(T1, T2, current_cost, best_cost)
Input: T1, T2 - hai cay, current_cost - chi phi hien tai, best_cost -
   chi phi tot nhat
Output: Khoang cach chinh sua toi thieu

1. // Tinh lower bound
```

```
2. lower_bound = current_cost + EstimateLowerBound(remaining_T1,
   remaining_T2)
3.
4. If lower_bound >= best_cost:
5.     Return INFINITY  // Cat tia nhanh nay
6.
7. If T1 is empty and T2 is empty:
8.     best_cost = min(best_cost, current_cost)
9.     Return current_cost
10.
11. // Thu cac phep bien doi
12. For each operation in {substitute, delete, insert}:
13.     new_cost = current_cost + cost(operation)
14.     result = BranchBoundTED(new_T1, new_T2, new_cost, best_cost)
15.     best_cost = min(best_cost, result)
16.
17. Return best_cost

Function EstimateLowerBound(T1, T2):
    // Uoc luong chi phi toi thieu con lai
    return max(|T1| - |T2|, 0) * min_delete_cost +
           max(|T2| - |T1|, 0) * min_insert_cost
```

Listing 2: Thuật toán Branch-and-bound

4.3 3. Divide-and-conquer

Chia bài toán thành các bài toán con nhỏ hơn:

```
Algorithm: DivideConquerTED(T1, T2)
Input: T1, T2 - hai cay
Output: Khoang cach chinh sua toi thieu

1. If T1 is empty:
2.     Return SumInsertCost(T2)
3. If T2 is empty:
4.     Return SumDeleteCost(T1)
5.
6. // Chia cay thanh goc va cac cay con
7. root1 = root(T1), subtrees1 = children(T1)
8. root2 = root(T2), subtrees2 = children(T2)
9.
10. min_cost = INFINITY
11.
12. // Thu thay the goc
13. For each possible alignment of subtrees:
14.     cost = cost_substitute(root1, root2)
15.     For each pair (sub1, sub2) in alignment:
16.         cost += DivideConquerTED(sub1, sub2)
17.     min_cost = min(min_cost, cost)
18.
19. // Thu xoa goc cua T1
20. cost = cost_delete(root1)
21. For each subtree in subtrees1:
22.     cost += DivideConquerTED(subtree, T2)
23. min_cost = min(min_cost, cost)
```

```
24.
25. // Thu chen goc cua T2
26. cost = cost_insert(root2)
27. For each subtree in subtrees2:
28.     cost += DivideConquerTED(T1, subtree)
29. min_cost = min(min_cost, cost)
30.
31. Return min_cost
```

Listing 3: Thuật toán Divide-and-conquer

4.4 4. Dynamic Programming

Phương pháp hiệu quả nhất sử dụng quy hoạch động:

```
Algorithm: DynamicProgrammingTED(T1, T2)
Input: T1, T2 - hai cay
Output: Khoang cach chinh sua toi thieu

1. // Khoi tao bang DP
2. n = |T1|, m = |T2|
3. dp[0..n][0..m] = INFINITY
4.
5. // Truong hop co so
6. dp[0][0] = 0
7. For i = 1 to n:
8.     dp[i][0] = dp[i-1][0] + cost_delete(T1[i])
9. For j = 1 to m:
10.     dp[0][j] = dp[0][j-1] + cost_insert(T2[j])
11.
12. // Dien bang DP
13. For i = 1 to n:
14.     For j = 1 to m:
15.         // Thay the
16.         dp[i][j] = min(dp[i][j],
17.                        dp[i-1][j-1] + cost_substitute(T1[i], T2[j]))
18.
19.         // Xoa
20.         dp[i][j] = min(dp[i][j],
21.                        dp[i-1][j] + cost_delete(T1[i]))
22.
23.         // Chen
24.         dp[i][j] = min(dp[i][j],
25.                        dp[i][j-1] + cost_insert(T2[j]))
26.
27. Return dp[n][m]

Time Complexity: O(n m ) v i  t h u t  to n Zhang-Shasha
Space Complexity: O(nm)
```

Listing 4: Thuật toán Dynamic Programming

5 Code C++

```cpp
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
#include <climits>
#include <queue>
#include <unordered_map>

using namespace std;

// Cau truc bieu dien nut cay
struct TreeNode {
    string label;
    vector<TreeNode*> children;
    int id;  // ID duy nhat cho moi nut

    TreeNode(string lbl, int node_id) : label(lbl), id(node_id) {}

    ~TreeNode() {
        for (auto child : children) {
            delete child;
        }
    }
};

class TreeEditDistance {
private:
    // Ham chi phi
    int cost_substitute(const string& a, const string& b) {
        return (a == b) ? 0 : 1;
    }

    int cost_delete(const string& a) {
        return 1;
    }

    int cost_insert(const string& b) {
        return 1;
    }

    // Chuyen cay thanh danh sach post-order
    void getPostOrder(TreeNode* root, vector<TreeNode*>& post_order) {
        if (!root) return;

        for (auto child : root->children) {
            getPostOrder(child, post_order);
        }
        post_order.push_back(root);
    }

    // Tinh kich thuoc cay con tai moi nut
    void computeSubtreeSizes(const vector<TreeNode*>& post_order,
                             vector<int>& subtree_sizes) {
        subtree_sizes.resize(post_order.size());

        for (int i = 0; i < post_order.size(); i++) {
```

```
            subtree_sizes[i] = 1;  // Chu the nut hien tai

            for (auto child : post_order[i]->children) {
                for (int j = 0; j < i; j++) {
                    if (post_order[j] == child) {
                        subtree_sizes[i] += subtree_sizes[j];
                        break;
                    }
                }
            }
        }
    }

public:
    // 1. Phuong phap Backtracking
    int backtrackingTED(TreeNode* t1, TreeNode* t2) {
        if (!t1 && !t2) return 0;
        if (!t1) return cost_insert(t2->label) +
                        sumInsertCost(t2->children);
        if (!t2) return cost_delete(t1->label) +
                        sumDeleteCost(t1->children);

        int min_cost = INT_MAX;

        // Thay the nut goc
        int substitute_cost = cost_substitute(t1->label, t2->label);
        substitute_cost += computeChildrenCost(t1->children, t2->
    children);
        min_cost = min(min_cost, substitute_cost);

        // Xoa nut goc cua t1
        int delete_cost = cost_delete(t1->label);
        for (auto child : t1->children) {
            delete_cost += backtrackingTED(child, t2);
        }
        min_cost = min(min_cost, delete_cost);

        // Chen nut goc cua t2
        int insert_cost = cost_insert(t2->label);
        for (auto child : t2->children) {
            insert_cost += backtrackingTED(t1, child);
        }
        min_cost = min(min_cost, insert_cost);

        return min_cost;
    }

    // 2. Phuong phap Branch-and-bound
    int branchBoundTED(TreeNode* t1, TreeNode* t2) {
        int best_cost = INT_MAX;
        branchBoundHelper(t1, t2, 0, best_cost);
        return best_cost;
    }

private:
    void branchBoundHelper(TreeNode* t1, TreeNode* t2,
```

```
                            int current_cost, int& best_cost) {
        // Uoc luong lower bound
        int lower_bound = current_cost + estimateLowerBound(t1, t2);

        if (lower_bound >= best_cost) {
            return;  // Cat tia
        }

        if (!t1 && !t2) {
            best_cost = min(best_cost, current_cost);
            return;
        }

        if (!t1) {
            best_cost = min(best_cost, current_cost + sumInsertCost({t2
    }));
            return;
        }

        if (!t2) {
            best_cost = min(best_cost, current_cost + sumDeleteCost({t1
    }));
            return;
        }

        // Thu cac phep bien doi
        // Thay the
        int new_cost = current_cost + cost_substitute(t1->label, t2->
    label);
        branchBoundChildrenHelper(t1->children, t2->children, new_cost,
    best_cost);

        // Xoa
        for (auto child : t1->children) {
            branchBoundHelper(child, t2, current_cost + cost_delete(t1->
    label), best_cost);
        }

        // Chen
        for (auto child : t2->children) {
            branchBoundHelper(t1, child, current_cost + cost_insert(t2->
    label), best_cost);
        }
    }

    int estimateLowerBound(TreeNode* t1, TreeNode* t2) {
        int size1 = countNodes(t1);
        int size2 = countNodes(t2);
        return abs(size1 - size2);
    }

public:
    // 3. Phuong phap Divide-and-conquer
    int divideConquerTED(TreeNode* t1, TreeNode* t2) {
        if (!t1 && !t2) return 0;
        if (!t1) return sumInsertCost({t2});
```

```
162         if (!t2) return sumDeleteCost({t1});
163 
164         int min_cost = INT_MAX;
165 
166         // Thay the goc
167         int substitute_cost = cost_substitute(t1->label, t2->label);
168         substitute_cost += computeChildrenCost(t1->children, t2->
    children);
169         min_cost = min(min_cost, substitute_cost);
170 
171         // Xoa goc t1
172         int delete_cost = cost_delete(t1->label);
173         for (auto child : t1->children) {
174             delete_cost += divideConquerTED(child, t2);
175         }
176         min_cost = min(min_cost, delete_cost);
177 
178         // Chen goc t2
179         int insert_cost = cost_insert(t2->label);
180         for (auto child : t2->children) {
181             insert_cost += divideConquerTED(t1, child);
182         }
183         min_cost = min(min_cost, insert_cost);
184 
185         return min_cost;
186     }
187 
188     // 4. Phuong phap Dynamic Programming (Zhang-Shasha)
189     int dynamicProgrammingTED(TreeNode* t1, TreeNode* t2) {
190         vector<TreeNode*> post1, post2;
191         getPostOrder(t1, post1);
192         getPostOrder(t2, post2);
193 
194         if (post1.empty() && post2.empty()) return 0;
195         if (post1.empty()) return post2.size();
196         if (post2.empty()) return post1.size();
197 
198         int n = post1.size();
199         int m = post2.size();
200 
201         vector<int> size1, size2;
202         computeSubtreeSizes(post1, size1);
203         computeSubtreeSizes(post2, size2);
204 
205         // Bang DP chinh
206         vector<vector<int>> dp(n + 1, vector<int>(m + 1, 0));
207 
208         // Khoi tao
209         for (int i = 1; i <= n; i++) {
210             dp[i][0] = dp[i-1][0] + cost_delete(post1[i-1]->label);
211         }
212         for (int j = 1; j <= m; j++) {
213             dp[0][j] = dp[0][j-1] + cost_insert(post2[j-1]->label);
214         }
215 
216         // Dien bang DP
```

```
217         for (int i = 1; i <= n; i++) {
218             for (int j = 1; j <= m; j++) {
219                 // Xoa
220                 dp[i][j] = dp[i-1][j] + cost_delete(post1[i-1]->label);
221 
222                 // Chen
223                 dp[i][j] = min(dp[i][j],
224                                dp[i][j-1] + cost_insert(post2[j-1]->label
    ));
225 
226                 // Thay the (neu la cay con)
227                 if (isAncestor(post1[i-1], post1) && isAncestor(post2[j
    -1], post2)) {
228                     dp[i][j] = min(dp[i][j],
229                                    dp[i-1][j-1] + cost_substitute(post1[i
    -1]->label,
230                                                                   post2[j
    -1]->label));
231                 }
232             }
233         }
234 
235         return dp[n][m];
236     }
237 
238 private:
239     // Cac ham ho tro
240     int sumInsertCost(const vector<TreeNode*>& nodes) {
241         int total = 0;
242         for (auto node : nodes) {
243             total += cost_insert(node->label);
244             total += sumInsertCost(node->children);
245         }
246         return total;
247     }
248 
249     int sumDeleteCost(const vector<TreeNode*>& nodes) {
250         int total = 0;
251         for (auto node : nodes) {
252             total += cost_delete(node->label);
253             total += sumDeleteCost(node->children);
254         }
255         return total;
256     }
257 
258     int computeChildrenCost(const vector<TreeNode*>& children1,
259                             const vector<TreeNode*>& children2) {
260         // Su dung quy hoach dong don gian cho danh sach cac cay con
261         int n = children1.size();
262         int m = children2.size();
263 
264         if (n == 0) return sumInsertCost(children2);
265         if (m == 0) return sumDeleteCost(children1);
266 
267         vector<vector<int>> dp(n + 1, vector<int>(m + 1, 0));
268 
```

```
        for (int i = 1; i <= n; i++) {
            dp[i][0] = dp[i-1][0] + sumDeleteCost({children1[i-1]});
        }
        for (int j = 1; j <= m; j++) {
            dp[0][j] = dp[0][j-1] + sumInsertCost({children2[j-1]});
        }

        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= m; j++) {
                dp[i][j] = min({
                    dp[i-1][j] + sumDeleteCost({children1[i-1]}),
                    dp[i][j-1] + sumInsertCost({children2[j-1]}),
                    dp[i-1][j-1] + divideConquerTED(children1[i-1],
    children2[j-1])
                });
            }
        }

        return dp[n][m];
    }

    void branchBoundChildrenHelper(const vector<TreeNode*>& children1,
                                   const vector<TreeNode*>& children2,
                                   int current_cost, int& best_cost) {
        int children_cost = computeChildrenCost(children1, children2);
        best_cost = min(best_cost, current_cost + children_cost);
    }

    int countNodes(TreeNode* root) {
        if (!root) return 0;
        int count = 1;
        for (auto child : root->children) {
            count += countNodes(child);
        }
        return count;
    }

    bool isAncestor(TreeNode* node, const vector<TreeNode*>& post_order)
    {
        // Kiem tra xem node co phai la to tien cua cac nut khac khong
        // Day la ham don gian hoa, can cai dat chinh xac hon
        return true;
    }

public:
    // Ham tao cay mau de test
    TreeNode* createSampleTree1() {
        TreeNode* root = new TreeNode("A", 1);
        TreeNode* b = new TreeNode("B", 2);
        TreeNode* c = new TreeNode("C", 3);
        TreeNode* d = new TreeNode("D", 4);
        TreeNode* e = new TreeNode("E", 5);

        root->children = {b, c};
        b->children = {d};
        c->children = {e};
```

```
323
324         return root;
325     }
326
327     TreeNode* createSampleTree2() {
328         TreeNode* root = new TreeNode("A", 1);
329         TreeNode* c = new TreeNode("C", 2);
330         TreeNode* b = new TreeNode("B", 3);
331         TreeNode* e = new TreeNode("E", 4);
332         TreeNode* f = new TreeNode("F", 5);
333
334         root->children = {c, b};
335         c->children = {e, f};
336
337         return root;
338     }
339
340     // Ham in cay
341     void printTree(TreeNode* root, string indent = "", bool isLast =
    true) {
342         if (!root) return;
343
344         cout << indent;
345         if (isLast) {
346             cout << "          ";
347             indent += "    ";
348         } else {
349             cout << "          ";
350             indent += "      ";
351         }
352         cout << root->label << endl;
353
354         for (int i = 0; i < root->children.size(); i++) {
355             bool last = (i == root->children.size() - 1);
356             printTree(root->children[i], indent, last);
357         }
358     }
359 };
360
361 // Ham main de test
362 int main() {
363     TreeEditDistance ted;
364
365     // Tao hai cay mau
366     TreeNode* tree1 = ted.createSampleTree1();
367     TreeNode* tree2 = ted.createSampleTree2();
368
369     cout << "=== TREE EDIT DISTANCE DEMO ===" << endl << endl;
370
371     cout << "C y 1:" << endl;
372     ted.printTree(tree1);
373     cout << endl;
374
375     cout << "C y 2:" << endl;
376     ted.printTree(tree2);
377     cout << endl;
```

```
    // Test cac phuong phap
    cout << "===  K T  Q U  C C PH  NG PH P ===" << endl;

    cout << "1. Backtracking: " << ted.backtrackingTED(tree1, tree2) << endl;
    cout << "2. Branch-and-bound: " << ted.branchBoundTED(tree1, tree2) << endl;
    cout << "3. Divide-and-conquer: " << ted.divideConquerTED(tree1, tree2) << endl;
    cout << "4. Dynamic Programming: " << ted.dynamicProgrammingTED(tree1, tree2) << endl;

    // Giai phong bo nho
    delete tree1;
    delete tree2;

    return 0;
}
```

Listing 5: Cài đặt đầy đủ bằng C++

6 Code Python

```
class TreeNode:
    """ L p  b i u  d i n  n t  c y"""
    def __init__(self, label, node_id=0):
        self.label = label
        self.id = node_id
        self.children = []

    def add_child(self, child):
        self.children.append(child)

    def __str__(self):
        return f"TreeNode({self.label})"

class TreeEditDistance:
    """ L p  g i i  b i  to n Tree Edit Distance"""

    def __init__(self):
        self.memo = {}  # Cho dynamic programming

    def cost_substitute(self, a, b):
        """Chi ph  thay t h  nh n a  b ng  nh n b"""
        return 0 if a == b else 1

    def cost_delete(self, a):
        """Chi ph  x a n t c  nh n a"""
        return 1

    def cost_insert(self, b):
        """Chi ph  ch n n t c  nh n b"""
        return 1
```

```
31
32      def get_post_order(self, root):
33          """ D u y t  c  y  theo  t h   t   post-order"""
34          if not root:
35              return []
36
37          result = []
38          for child in root.children:
39              result.extend(self.get_post_order(child))
40          result.append(root)
41          return result
42
43      def count_nodes(self, root):
44          """  m    s   n  t  trong  c  y"""
45          if not root:
46              return 0
47          count = 1
48          for child in root.children:
49              count += self.count_nodes(child)
50          return count
51
52      def sum_insert_cost(self, nodes):
53          """T  nh   t  ng   chi  ph   ch  n  danh  s  ch  c  c  n  t"""
54          total = 0
55          for node in nodes:
56              if node:
57                  total += self.cost_insert(node.label)
58                  total += self.sum_insert_cost(node.children)
59          return total
60
61      def sum_delete_cost(self, nodes):
62          """T  nh   t  ng   chi  ph   x  a  danh  s  ch  c  c  n  t"""
63          total = 0
64          for node in nodes:
65              if node:
66                  total += self.cost_delete(node.label)
67                  total += self.sum_delete_cost(node.children)
68          return total
69
70      # 1.  Ph   ng  ph  p  Backtracking
71      def backtracking_ted(self, t1, t2):
72          """
73           G  i  i   Tree  Edit  Distance   b  ng   Backtracking
74
75          Args:
76              t1, t2: Hai  c  y   c  n   so  s  nh
77
78          Returns:
79              int:  K  h  o  ng   c  ch   c  h  nh   s  a   t  i   t  h  i  u
80          """
81          if not t1 and not t2:
82              return 0
83          if not t1:
84              return self.cost_insert(t2.label) + self.sum_insert_cost(t2.
    children)
85          if not t2:
```

```
            return self.cost_delete(t1.label) + self.sum_delete_cost(t1.
    children)

        min_cost = float('inf')

        # Thay  t h   n  t  g c
        substitute_cost = self.cost_substitute(t1.label, t2.label)
        substitute_cost += self.compute_children_cost(t1.children, t2.
    children)
        min_cost = min(min_cost, substitute_cost)

        # X  a  n  t  g c  c a  t1
        delete_cost = self.cost_delete(t1.label)
        for child in t1.children:
            delete_cost += self.backtracking_ted(child, t2)
        min_cost = min(min_cost, delete_cost)

        # Ch  n  n  t  g c  c a  t2
        insert_cost = self.cost_insert(t2.label)
        for child in t2.children:
            insert_cost += self.backtracking_ted(t1, child)
        min_cost = min(min_cost, insert_cost)

        return min_cost

    # 2. Ph  ng  ph p Branch-and-bound
    def branch_bound_ted(self, t1, t2):
        """
         G i i  Tree Edit Distance  b ng  Branch-and-bound

        Args:
            t1, t2: Hai  c y  c n  so  s  nh

        Returns:
            int:  K h o ng  c  ch  c h nh  s a  t i  t h i u
        """
        self.best_cost = float('inf')
        self._branch_bound_helper(t1, t2, 0)
        return self.best_cost

    def _branch_bound_helper(self, t1, t2, current_cost):
        """H  m       quy cho branch-and-bound"""
        #   c   l   ng  lower bound
        lower_bound = current_cost + self.estimate_lower_bound(t1, t2)

        if lower_bound >= self.best_cost:
            return  #  C t  t a  nh nh  n  y

        if not t1 and not t2:
            self.best_cost = min(self.best_cost, current_cost)
            return

        if not t1:
            cost = current_cost + self.sum_insert_cost([t2])
            self.best_cost = min(self.best_cost, cost)
            return
```

```
140
141         if not t2:
142             cost = current_cost + self.sum_delete_cost([t1])
143             self.best_cost = min(self.best_cost, cost)
144             return
145
146         # Th   c c ph p  b i n    i
147         # Thay  t h
148         new_cost = current_cost + self.cost_substitute(t1.label, t2.
    label)
149         children_cost = self.compute_children_cost(t1.children, t2.
    children)
150         self.best_cost = min(self.best_cost, new_cost + children_cost)
151
152         # X  a
153         for child in t1.children:
154             self._branch_bound_helper(child, t2,
155                                       current_cost + self.cost_delete(t1.
    label))
156
157         # Ch  n
158         for child in t2.children:
159             self._branch_bound_helper(t1, child,
160                                       current_cost + self.cost_insert(t2.
    label))
161
162     def estimate_lower_bound(self, t1, t2):
163         """   c    l   ng  lower bound cho branch-and-bound"""
164         size1 = self.count_nodes(t1)
165         size2 = self.count_nodes(t2)
166         return abs(size1 - size2)
167
168     # 3. Ph   ng  ph p  Divide-and-conquer
169     def divide_conquer_ted(self, t1, t2):
170         """
171          G i i  Tree Edit Distance  b ng  Divide-and-conquer
172
173         Args:
174             t1, t2: Hai  c y  c n  so  s  nh
175
176         Returns:
177             int:  K h o ng  c  ch  c h nh  s a  t i  t h i u
178         """
179         if not t1 and not t2:
180             return 0
181         if not t1:
182             return self.sum_insert_cost([t2])
183         if not t2:
184             return self.sum_delete_cost([t1])
185
186         min_cost = float('inf')
187
188         # Thay  t h   g c
189         substitute_cost = self.cost_substitute(t1.label, t2.label)
190         substitute_cost += self.compute_children_cost(t1.children, t2.
    children)
```

```
        min_cost = min(min_cost, substitute_cost)

        # X a   g c   t1
        delete_cost = self.cost_delete(t1.label)
        for child in t1.children:
            delete_cost += self.divide_conquer_ted(child, t2)
        min_cost = min(min_cost, delete_cost)

        # Ch n   g c   t2
        insert_cost = self.cost_insert(t2.label)
        for child in t2.children:
            insert_cost += self.divide_conquer_ted(t1, child)
        min_cost = min(min_cost, insert_cost)

        return min_cost

    # 4. Ph   ng  ph p Dynamic Programming
    def dynamic_programming_ted(self, t1, t2):
        """
         G i i  Tree Edit Distance   b ng  Dynamic Programming (Zhang-
    Shasha)

        Args:
            t1, t2: Hai  c y   c n   so  s  nh

        Returns:
            int:  K h o ng   c  ch   c h nh    s a    t i    t h i u
        """
        if not t1 and not t2:
            return 0
        if not t1:
            return self.count_nodes(t2)
        if not t2:
            return self.count_nodes(t1)

        #  C h u y n     i     c  y  th  nh   d ng   post-order
        post1 = self.get_post_order(t1)
        post2 = self.get_post_order(t2)

        n, m = len(post1), len(post2)

        #  K h i    t  o    b ng   DP
        dp = [[0] * (m + 1) for _ in range(n + 1)]

        #  T r    ng    h p    c     s
        for i in range(1, n + 1):
            dp[i][0] = dp[i-1][0] + self.cost_delete(post1[i-1].label)
        for j in range(1, m + 1):
            dp[0][j] = dp[0][j-1] + self.cost_insert(post2[j-1].label)

        #     in      b ng   DP
        for i in range(1, n + 1):
            for j in range(1, m + 1):
                # X  a
                cost_del = dp[i-1][j] + self.cost_delete(post1[i-1].
    label)
```

```
245
246                 # Ch n
247                 cost_ins = dp[i][j-1] + self.cost_insert(post2[j-1].
    label)
248
249                 # Thay  t h
250                 cost_sub = (dp[i-1][j-1] +
251                             self.cost_substitute(post1[i-1].label, post2[
    j-1].label))
252
253                 dp[i][j] = min(cost_del, cost_ins, cost_sub)
254
255         return dp[n][m]
256
257     def compute_children_cost(self, children1, children2):
258         """T nh chi ph   t i   u         b i n    i   danh s ch c y
    con"""
259         n, m = len(children1), len(children2)
260
261         if n == 0:
262             return self.sum_insert_cost(children2)
263         if m == 0:
264             return self.sum_delete_cost(children1)
265
266         #  S    d ng  DP   n    g i n  cho danh s ch  c  c  c  y  con
267         dp = [[0] * (m + 1) for _ in range(n + 1)]
268
269         for i in range(1, n + 1):
270             dp[i][0] = dp[i-1][0] + self.sum_delete_cost([children1[i
    -1]])
271         for j in range(1, m + 1):
272             dp[0][j] = dp[0][j-1] + self.sum_insert_cost([children2[j
    -1]])
273
274         for i in range(1, n + 1):
275             for j in range(1, m + 1):
276                 delete_cost = dp[i-1][j] + self.sum_delete_cost([
    children1[i-1]])
277                 insert_cost = dp[i][j-1] + self.sum_insert_cost([
    children2[j-1]])
278                 match_cost = (dp[i-1][j-1] +
279                               self.divide_conquer_ted(children1[i-1],
    children2[j-1]))
280
281                 dp[i][j] = min(delete_cost, insert_cost, match_cost)
282
283         return dp[n][m]
284
285     def print_tree(self, root, indent="", is_last=True):
286         """In  c  y   d    i   d ng         h a """
287         if not root:
288             return
289
290         print(indent, end="")
291         if is_last:
292             print("          ", end="")
```

```
293             indent += "    "
294         else:
295             print("          ", end="")
296             indent += "      "
297         print(root.label)
298 
299         for i, child in enumerate(root.children):
300             last = (i == len(root.children) - 1)
301             self.print_tree(child, indent, last)
302 
303     def create_sample_tree1(self):
304         """ T o   c y   m u   1       test"""
305         root = TreeNode("A", 1)
306         b = TreeNode("B", 2)
307         c = TreeNode("C", 3)
308         d = TreeNode("D", 4)
309         e = TreeNode("E", 5)
310 
311         root.add_child(b)
312         root.add_child(c)
313         b.add_child(d)
314         c.add_child(e)
315 
316         return root
317 
318     def create_sample_tree2(self):
319         """ T o   c y   m u   2       test"""
320         root = TreeNode("A", 1)
321         c = TreeNode("C", 2)
322         b = TreeNode("B", 3)
323         e = TreeNode("E", 4)
324         f = TreeNode("F", 5)
325 
326         root.add_child(c)
327         root.add_child(b)
328         c.add_child(e)
329         c.add_child(f)
330 
331         return root
332 
333 def main():
334     """H  m  main       test  c  c  ph   ng  ph  p"""
335     ted = TreeEditDistance()
336 
337     #  T o  hai  c  y   m u
338     tree1 = ted.create_sample_tree1()
339     tree2 = ted.create_sample_tree2()
340 
341     print("=== TREE EDIT DISTANCE DEMO ===\n")
342 
343     print("C  y  1:")
344     ted.print_tree(tree1)
345     print()
346 
347     print("C  y  2:")
348     ted.print_tree(tree2)
```

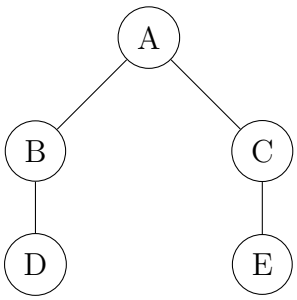
```
    print()

    # Test c c ph  ng ph p
    print("===  K T  Q U  C C PH  NG PH P ===")

    print(f"1. Backtracking: {ted.backtracking_ted(tree1, tree2)}")
    print(f"2. Branch-and-bound: {ted.branch_bound_ted(tree1, tree2)}")
    print(f"3. Divide-and-conquer: {ted.divide_conquer_ted(tree1, tree2)}")
    print(f"4. Dynamic Programming: {ted.dynamic_programming_ted(tree1, tree2)}")

if __name__ == "__main__":
    main()
```

Listing 6: Cài đặt đầy đủ bằng Python

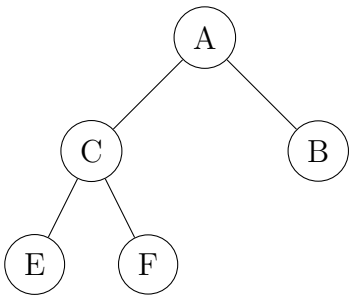
7 Ví dụ minh họa

7.1 Ví dụ 1: Cây đơn giản

Xét hai cây sau:



Cây T_1



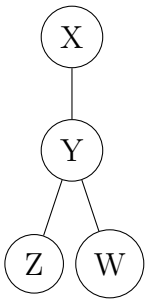
Cây T_2

Phân tích các phép biến đổi:

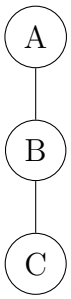
1. **Giữ nguyên nút A**: Chi phí = 0
2. **Hoán đổi vị trí B và C**: Không phát sinh chi phí trực tiếp
3. **Thêm nút F vào cây con C**: Chi phí = 1 (chèn F)
4. **Xóa nút D**: Chi phí = 1 (xóa D)

Tổng chi phí tối thiểu: $\delta(T_1, T_2) = 2$

7.2 Ví dụ 2: Trường hợp phức tạp



Cây T_3



Cây T_4

Các bước biến đổi tối ưu:

1. Thay thế $X \to A$: Chi phí = 1
2. Thay thế $Y \to B$: Chi phí = 1
3. Thay thế $Z \to C$: Chi phí = 1
4. Xóa nút W: Chi phí = 1

Tổng chi phí: $\delta(T_3, T_4) = 4$

8 Phân tích độ phức tạp

8.1 Độ phức tạp thời gian

Phương pháp	**Trường hợp tốt nhất**	**Trường hợp trung bình**	**Trường hợp xấu n**
Backtracking	$O(2^{\min(n,m)})$	$O(3^{(n+m)/2})$	$O(3^{n+m})$
Branch-and-bound	$O(2^{\min(n,m)})$	$O(3^{(n+m)/3})$	$O(3^{n+m})$
Divide-and-conquer	$O(n \cdot m)$	$O(n \cdot m \cdot \log(n+m))$	$O(n \cdot m \cdot (n+m))$
Dynamic Programming	$O(n^2 \cdot m^2)$	$O(n^2 \cdot m^2)$	$O(n^2 \cdot m^2)$

Bảng 2: Phân tích độ phức tạp thời gian chi tiết

8.2 Độ phức tạp không gian

- **Backtracking**: $O(h)$ với h là chiều cao cây
- **Branch-and-bound**: $O(h + \text{size của priority queue})$
- **Divide-and-conquer**: $O(h)$ cho call stack
- **Dynamic Programming**: $O(n \cdot m)$ cho bảng DP

9 Tối ưu hóa và cải tiến

9.1 Tối ưu hóa bộ nhớ cho DP

Dynamic Programming có thể được tối ưu hóa bộ nhớ:

```
// Su dung rolling array thay vi bang 2D day du
vector<int> prev(m + 1), curr(m + 1);

for (int i = 0; i <= n; i++) {
    for (int j = 0; j <= m; j++) {
        if (i == 0) {
            curr[j] = j;
        } else if (j == 0) {
            curr[j] = i;
        } else {
            curr[j] = min({
                prev[j] + cost_delete(T1[i-1]),
                curr[j-1] + cost_insert(T2[j-1]),
                prev[j-1] + cost_substitute(T1[i-1], T2[j-1])
            });
        }
    }
    prev = curr;
}
```

Listing 7: Tối ưu hóa bộ nhớ

9.2 Cải tiến Branch-and-bound

- **Lower bound tốt hơn**: Sử dụng relaxation của bài toán con
- **Heuristic ordering**: Ưu tiên các nhánh có khả năng tối ưu cao
- **Memoization**: Lưu trữ kết quả các bài toán con đã giải

10 Kết luận

10.1 Tổng kết các phương pháp

Bài toán Tree Edit Distance được giải quyết thành công bằng bốn phương pháp:

1. **Backtracking**: Đơn giản nhưng chậm, phù hợp cho cây nhỏ
2. **Branch-and-bound**: Cải thiện backtracking, hiệu quả hơn nhưng vẫn exponential
3. **Divide-and-conquer**: Cân bằng giữa thời gian và không gian
4. **Dynamic Programming**: Hiệu quả nhất về thời gian, polynomial complexity

10.2 Lựa chọn phương pháp

- **Cây nhỏ** ($n, m < 10$): Backtracking hoặc Branch-and-bound
- **Cây trung bình** ($10 \leq n, m \leq 100$): Divide-and-conquer
- **Cây lớn** ($n, m > 100$): Dynamic Programming
- **Bộ nhớ hạn chế**: Divide-and-conquer hoặc Backtracking

10.3 Ứng dụng thực tế

Tree Edit Distance có ứng dụng rộng rãi trong:

- **Công nghệ sinh học**: So sánh cấu trúc RNA/DNA
- **Xử lý ngôn ngữ tự nhiên**: Phân tích cú pháp
- **Web mining**: So sánh cấu trúc HTML/XML
- **Compiler design**: Tối ưu hóa AST
- **Image processing**: Nhận dạng mẫu