BỘ GIÁO DỤC VÀ ĐÀO TẠO

TRƯỜNG ĐẠI HỌC MỎ - ĐỊA CHẤT

BÁO CÁO NGHIÊN CỨU KHOA HỌC

Đề tài: Thuật toán Kiến

Phần 3 ***Hệ kiến Max Min***

Giảng viên thực hiện: ThS. Dương Chí Thiện

Hà Nội, 12/2023

MỤC LỤC

1. Bài toán người chào hàng (TSP).

Bài toán người chào hàng (Traveling Salesman Problem - TSP) là bài toán TƯTH điển hình, được nghiên cứu nhiều và được xem là bài toán chuẩn để đánh giá hiệu quả các lược đồ giải bài toán TƯTH mới (xem [30,31]).

Bài toán được phát biểu như sau:

Có một tập gồm n thành phố (hoặc điểm tiêu thụ) $C = \{ci\}$ n, độ dài đường đi trực tiếp từ ci đến cj là di,j . Một người chào hàng muốn tìm một hành trình ngắn nhất từ nơi ở, đi qua mỗi thành phố đúng một lần để giới thiệu sản phẩm cho khách hàng, sau đó trở về thành phố xuất phát.

Như vậy, bài toán này chính là bài toán tìm chu trình Hamilton có độ dài ngắn nhất trên đồ thị đầy đủ có trọng số $G = (,)$, trong đó là tập đỉnh với nhãn là các thành phố trong C, là các cạnh nối các thành phố tương ứng, độ dài các cạnh chính là độ dài đường đi giữa các thành phố. Trong trường hợp này, tập S sẽ là các chu trình Hamilton trên G, f là độ dài của chu trình, Ω là ràng buộc đòi hỏi chu trình là chu trình Hamilton (qua tất cả các đỉnh, mỗi đỉnh đúng một lần), C là tập thành phố được x.

(trùng với), C_0 trùng với C, tập X là vectơ độ dài n: $x = (x_1, \dots , x_n)$ với x_i $\in C$ i ≤

n, còn X^* là các vectơ trong đó x_i khác x_j đối với mọi cặp (i, j).

Do đó, lời giải tối ưu của bài toán TSP là một hoán vị π của tập đỉnh

$\{c_1, c , \dots , c_n\}$ sao cho hàm độ dài $f(\pi)$ là nhỏ nhất, trong đó $f(\pi)$ được tính theo (1.1):

$f(\pi) = \sum_{i=1}^{n-1} d(\pi(i), \pi(i + 1)) + d(\pi(n), \pi(1))$ (1.1) ở đây $d(u, v)$ là khoảng cách từ u đến v.

Bài toán TSP được xem là bài toán chuẩn để kiểm định hiệu quả của các phương pháp giải bài toán TƯTH mới với thư viện dữ liệu chuẩn TSPLIB (Reinelt, 1991) tại địa chỉ [77] (Dữ liệu trong nó sẽ được sử dụng trong luận án này).

Bài toán này có nhiều ứng dụng thực tiễn, chẳng hạn như: khoan các lỗ trên bảng mạch in (Reinelt, 1994) hay định vị các thiết bị X-quang (Bland & Shallcross, 1989)… [31].

2. Từ những con kiến trong tự nhiên tới thuật toán ACO.

Thuật toán ACO lấy ý tưởng từ việc kiếm thức ăn của đàn kiến ngoài thực tế để giải quyết các bài toán tối ưu tổ hợp. Chúng dựa trên cơ sở một đàn kiến nhân tạo, chúng được tính toán tìm kiếm thức ăn nhờ mùi lạ nhân tạo.

Cấu trúc cơ bản của thuật toán ACO: trong mỗi thuật toán, tất cả kiến đi xây dựng cách giải quyết bài toán bằng cách xây dựng một đồ thị. Mỗi cạnh của đồ thị miêu tả các bước kiến có thể đi được kết hợp từ hai loại thông tin hướng dẫn kiến di chuyển:

Thông tin kinh nghiệm (heuristic information): giới hạn kinh nghiệm ưu tiên di chuyển từ nút r tới s…của cạnh a_{rs}. Nó được đánh dấu bởi η_{rs}. Thông tin này không được thay đổi bởi kiến trong suốt quá trình chạy thuật toán.

Thông tin mùi lạ nhân tạo (artificial pheromone trail information), nó giới hạn "nghiên cứu sự thèm muốn" của chuyển động là kiến nhân tạo và bắt chước mùi lạ thực tế của đàn kiến tự nhiên. Thông tin này bị thay đổi trong suốt quá trình thuật toán chạy phụ thuộc vào cách giải quyết được tìm thấy bởi những con kiến. Nó được đánh dấu bởi τ_{rs}.

Giới thiệu các bước ảnh hưởng từ những con kiến thật vào ACO. Có hai vấn đề cần chú ý:

- Chúng trừu tượng hoá vài mô hình thức ăn của kiến ngoài thực tế để tìm ra đường đi tìm kiếm thức ăn ngắn nhất.

- Chúng bao gồm vài đặc điểm không giống với tự nhiên nhưng lại cho phép thuật toán phát triển chứa đựng cách giải quyết tốt tới bài toán bị cản (ví dụ: sử dụng thông tin kinh nghiệm để hướng dẫn chuyển động của kiến).

Cách thức hoạt động cơ bản của một thuật toán ACO như sau: m kiến nhân tạo di chuyển, đồng thời và không đồng bộ, qua các trạng thái liền kề của bài toán. Sự di chuyển này theo một tập quy tắc làm cơ sở từ những vùng thông tin có sẵn ở các thành phần (các nút). Vùng thông tin này bao gồm thông tin kinh nghiệm và thông tin mùi lạ để hướng dẫn tìm kiếm. Qua sự di chuyển trên đồ thị kiến xây dựng được cách giải quyết. Những con kiến sẽ giải phóng mùi lạ ở mỗi lần chúng đi qua một cạnh (kết nối) trong khi xây dựng cách giải quyết (cập nhật từng bước mùi lạ trực tuyến). Mỗi lần những con kiến sinh ra cách giải quyết, nó được đánh giá và nó có thể tạo luồng mùi lạ là hoạt động của chất lượng của cách giải quyết của kiến (cập nhật lại mùi lạ trực tuyến). Thông tin này sẽ hướng dẫn tìm kiếm cho những con kiến đi sau.

Hơn thế nữa, cách thức sinh hoạt động của thuật toán ACO bao gồm thêm hai thủ tục, sự bay hơi mùi lạ (*pheromone trail evaporation*) và hoạt động lạ (*daemon actions*). Sự bay hơi của mùi lạ được khởi sự từ môi trường và nó được sử dụng như là một kĩ thuật để tránh tìm kiếm bị dừng lại và cho phép kiến khảo sát vùng không gian mới. Daemon actions là những hoạt động tối ưu như một bản sao tự nhiên để thực hiện những nhiệm vụ từ một mục tiêu xa tới vùng của kiến.

3. Sơ đồ chung thuật toán đàn kiến

Procedure ACO
    Initial();
    While (!ĐK dừng) do
        ConstructSolutions();

```
            LocalSearch();  /*Tuỳ ý, có thể có hoặc không
            UpdateTrails();
        End;
    End;
```

4. Thuật toán MMAS

Thuật toán MMAS (Max-Min Ant System) do Stutzle và Hoos đề xuất năm 2000 [66] với bốn điểm thay đổi so với AS.

- Thứ nhất, để tăng cường khai thác lời giải tốt nhất tìm được: các cạnh thuộc vào lời giải I-best hoặc G-best được cập nhật mùi. Điều này có thể dẫn đến hiện tượng tắc nghẽn: tất cả các kiến sẽ cùng đi một hành trình, bởi vì lượng mùi trên các cạnh thuộc hành trình tốt được cập nhật quá nhiều, mặc dù hành trình này không phải là hành trình tối ưu.

- Thứ hai, để khắc phục nhược điểm trong thay đổi thứ nhất, MMAS đã đưa ra miền giới hạn cho vết mùi: vết mùi sẽ bị hạn chế trong khoảng [ ].

- Thứ ba là vết mùi ban đầu được khởi tạo bằng và hệ số bay hơi nhỏ nhằm tăng cường khám phá trong giai đoạn đầu.

Cập nhật mùi Sau khi tất cả kiến xây dựng lời giải, vết mùi được cập nhật bằng thủ tục bay hơi giống như AS (công thức 2.4), và được thêm một lượng mùi cho tất cả các cạnh thuộc lời giải tốt như sau

Điểm thay đổi cuối cùng là vết mùi sẽ được khởi tạo lại khi có hiện tượng tắc nghẽn hoặc không tìm được lời giải tốt hơn sau một số bước

$$\tau_{ij} \leftarrow \tau_{ij} + \Delta\tau_{ij}^{best} \qquad (2.10)$$

trong đó $\Delta\tau_{ij}^{best} = \frac{1}{C^{G-best}}$ khi dùng G-best hoặc $\Delta\tau_{ij}^{best} = \frac{1}{C^{I-best}}$ khi dùng I-best để cập nhật mùi. Sau đó vết mùi sẽ bị giới hạn trong đoạn $[\tau_{min}, \tau_{max}]$ như sau:

$$\tau_{i,j} = \begin{cases} \tau_{max} & nếu\ \tau_{i,j} > \tau_{max} \\ \tau_{i,j} & nếu\ \tau_{i,j} \in [\tau_{min}, \tau_{max}] \\ \tau_{min} & nếu\ \tau_{i,j} < \tau_{min} \end{cases} \qquad (2.11)$$

Nói chung, MMAS dùng cả I-best và G-best thay phiên nhau. Rõ ràng, việc lựa chọn tần số tương đối cho hai cách cập nhật mùi ảnh hưởng đến hướng tìm kiếm. Nếu luôn cập nhật bằng G-best thì việc tìm kiếm sẽ sớm định hướng quanh G-

best còn khi cập nhật bằng I-best thì số lượng cạnh được cập nhật mùi nhiều do đó việc tìm kiếm giảm tính định hướng hơn.

Khi bắt đầu thuật toán, vết mùi trên tất cả các cạnh được thiết đặt bằng cận trên của vết mùi . Cách khởi tạo như vậy, kết hợp với tham số bay hơi nhỏ sẽ cho phép làm chậm sự khác biệt vết mùi giữa các cạnh. Do đó, giai đoạn đầu của MMAS mang tính khám phá.

Để tăng cường khả năng khám phá, MMAS khởi tạo lại vết mùi mỗi khi gặp tình trạng tắc nghẽn (thuật toán kiểm tra tình trạng tắc nghẽn dựa trên sự thống kê vết mùi trên các cạnh) hoặc sau một số bước lặp mà vẫn không tìm được lời giải tốt hơn.

MMAS là thuật toán được nghiên cứu nhiều nhất trong số các thuật toán ACO và nó có rất nhiều mở rộng. Một trong các cải tiến là khi khởi tạo lại vết mùi, cập nhật mùi dựa trên lời giải tốt nhất tìm được tính từ khi khởi tạo lại vết mùi thay cho G-best. Một cải tiến khác là sử dụng luật di chuyển theo kiểu ACS.

5. Code

```
#include <algorithm>
#include <cassert>
#include <chrono>
#include <cstdint>
#include <fstream>
#include <iostream>
#include <random>
#include <sstream>
#include <stdexcept>
#include <vector>
        using namespace std;
        std::default_random_engine &get_rng() {
          unsigned seed = std::chrono::system_clock::now().time_since_epoch().count();
          static default_random_engine instance(seed);
          return instance;
        }
        uint32_t get_random_uint32(uint32_t min, uint32_t max_inclusive) {
          uniform_int_distribution<uint32_t> distribution(min, max_inclusive);
          return distribution(get_rng());
        }
        double get_random_double(double from = 0.0, uint32_t to_exclusive = 1.0) {
          uniform_real_distribution<double> distribution(from, to_exclusive);
          return distribution(get_rng());
        }
        struct ProblemInstance {
          uint32_t dimension_;
          bool is_symmetric_ = true;
          vector<double> distance_matrix_;
          vector<vector<uint32_t>> nearest_neighbor_lists_;
          ProblemInstance(uint32_t dimension, const vector<double> &distance_matrix,
                          bool is_symmetric)
              : dimension_(dimension), is_symmetric_(is_symmetric),
                distance_matrix_(distance_matrix) {
            assert(dimension >= 2);
          }
          void initialize_nn_lists(uint32_t nn_list_size) {
            assert(dimension_ > 1);
            nn_list_size = min(dimension_ - 1, nn_list_size);
```

```
    nearest_neighbor_lists_.resize(dimension_);
    vector<uint32_t> neighbors(dimension_);
    for (uint32_t i = 0; i < dimension_; ++i) {
      neighbors[i] = i;
    }
    for (uint32_t node = 0; node < dimension_; ++node) {
      sort(neighbors.begin(), neighbors.end(),
          [this, node](uint32_t a, uint32_t b) {
            return get_distance(node, a) < get_distance(node, b);
          });
      assert(get_distance(node, neighbors.at(0)) <=
          get_distance(node, neighbors.at(1)));
      auto &nn_list = nearest_neighbor_lists_.at(node);
      nn_list.clear();
      nn_list.reserve(nn_list_size);
      uint32_t count = 0;
      for (uint32_t i = 0; count < nn_list_size; ++i) {
        if (neighbors[i] != node) { // node is not its own neighbor
          nn_list.push_back(neighbors[i]);
          ++count;
        }
      }
    }
}
double get_distance(uint32_t from, uint32_t to) const {
    assert((from < dimension_) && (to < dimension_));
    return distance_matrix_[from * dimension_ + to];
}
const vector<uint32_t> &get_nearest_neighbors(uint32_t node) const {
    assert(node < nearest_neighbor_lists_.size());
    return nearest_neighbor_lists_[node];
}
double calculate_route_length(const vector<uint32_t> &route) const {
    double distance = 0;
    if (!route.empty()) {
      auto prev_node = route.back();
      for (auto node : route) {
        distance += get_distance(prev_node, node);
        prev_node = node;
```

```
            }
        }
        return distance;
    }
};
ProblemInstance load_tsplib_instance(const char *path) {
    enum EdgeWeightType { EUC_2D, EXPLICIT };
    ifstream in(path);
    if (!in.is_open()) {
        throw runtime_error(string("Cannot open TSP instance file: ") + path);
    }
    string line;
    uint32_t dimension = 0;
    vector<double> distances;
    EdgeWeightType edge_weight_type{EUC_2D};
    bool is_symmetric = true;
    while (getline(in, line)) {
        cout << "Read line: " << line << endl;
        if (line.find("TYPE") == 0) {
            if (line.find(" TSP") != string::npos) {
                is_symmetric = true;
            } else if (line.find(" ATSP") != string::npos) {
                is_symmetric = false;
            } else {
                throw runtime_error("Unknown problem type");
            }
        } else if (line.find("DIMENSION") != string::npos) {
            istringstream line_in(line.substr(line.find(':') + 1));
            if (!(line_in >> dimension)) {
                throw runtime_error(string("Cannot read instance dimension"));
            }
        } else if (line.find("EDGE_WEIGHT_TYPE") != string::npos) {
            if (line.find(" EUC_2D") != string::npos) {
                edge_weight_type = EUC_2D;
            } else if (line.find(" EXPLICIT") != string::npos) {
                edge_weight_type = EXPLICIT;
            } else {
                throw runtime_error(string("Unsupported edge weight type"));
            }
```

```
} else if (line.find("NODE_COORD_SECTION") != string::npos) {
    vector<pair<double, double>> coords;
    while (getline(in, line)) {
        if (line.find("EOF") == string::npos) {
            istringstream line_in(line);
            uint32_t id;
            pair<double, double> point;
            line_in >> id >> point.first >> point.second;
            if (line_in.bad()) {
                cerr << "Error while reading coordinates";
            }
            coords.push_back(point);
        } else {
            break;
        }
    }
    distances.resize(dimension * dimension, 0);
    for (uint32_t i = 0; i < dimension; ++i) {
        auto from = coords.at(i);
        for (uint32_t j = 0; j < dimension; ++j) {
            if (i != j) {
                auto to = coords.at(j);
                auto dx = to.first - from.first;
                auto dy = to.second - from.second;
                double distance = int(sqrt(dx * dx + dy * dy) + 0.5);
                distances.at(i * dimension + j) = distance;
            }
        }
    }
} else if (line.find("EDGE_WEIGHT_SECTION") != string::npos) {
    assert(dimension > 0);
    if (edge_weight_type != EXPLICIT) {
        throw runtime_error("Expected EXPLICIT edge weight type");
    }
    distances.reserve(dimension * dimension);
    while (getline(in, line)) {
        if (line.find("EOF") != string::npos) {
            break;
        }
```

```
            istringstream line_in(line);
            double distance;
            while (line_in >> distance) {
              distances.push_back(distance);
            }
          }
          assert(distances.size() == dimension * dimension);
        }
      }
      in.close();
      assert(dimension > 2);
      return ProblemInstance(dimension, distances, is_symmetric);
    }
    struct Ant {
      vector<uint32_t> visited_; // A list of visited nodes, i.e. a route
      vector<uint8_t> is_visited_;
      double cost_ = std::numeric_limits<double>::max();
      void initialize(uint32_t dimension) {
        visited_.clear();
        visited_.reserve(dimension);
        is_visited_.clear();
        is_visited_.resize(dimension, false);
      }
      void visit(uint32_t node) {
        assert(!is_visited_.at(node));
        visited_.push_back(node);
        is_visited_.at(node) = true;
      }
      bool is_visited(uint32_t node) const {
        assert(node < is_visited_.size());
        return is_visited_[node];
      }
      bool all_visited() const {
        return find(is_visited_.begin(), is_visited_.end(), false) ==
            is_visited_.end();
      }
    };
    struct PheromoneMemory {
      uint32_t dimension_;
```

```
    vector<double> pheromone_values_; // For every edge (a,b),
          // where 0 <= a, b < dimension_double min_pheromone_value_;
    PheromoneMemory(uint32_t dimension, double min_pheromone_value = 0)
      : dimension_(dimension), min_pheromone_value_(min_pheromone_value) {
      pheromone_values_.resize(dimension * dimension, min_pheromone_value);
    }
    double get(uint32_t from, uint32_t to) const {
      assert((from < dimension_) && (to < dimension_));
      return pheromone_values_[from * dimension_ + to];
    }
    void evaporate_from_all(double evaporation_rate,
                            double min_pheromone_value) {
      for (auto &value : pheromone_values_) {
        value = max(min_pheromone_value, value * (1 - evaporation_rate));
      }
    }
    void increase(uint32_t from, uint32_t to, double deposit,
                  double max_pheromone_value, bool is_symmetric) {
      assert((from < dimension_) && (to < dimension_));
      auto &value = pheromone_values_[from * dimension_ + to];
      value = min(max_pheromone_value, value + deposit);
      if (is_symmetric) {
        pheromone_values_[to * dimension_ + from] = value;
      }
    }
};

Ant create_solution_nn(const ProblemInstance &instance,
                       uint32_t start_node = 0) {
    Ant ant;
    ant.initialize(instance.dimension_);
    uint32_t current_node = start_node;
    ant.visit(current_node);
    for (uint32_t i = 1; i < instance.dimension_; ++i) {
      uint32_t next_node = current_node;
      const auto &candidates = instance.get_nearest_neighbors(current_node);
      for (auto node : candidates) {
        if (!ant.is_visited(node)) {
          next_node = node;
```

```
                break;
            }
        }
        if (next_node == current_node) { // All closest nodes were visited,
                                         // we have to check the rest
            double min_distance = numeric_limits<double>::max();
            for (uint32_t node = 0; node < instance.dimension_; ++node) {
                if (!ant.is_visited(node)) {
                    auto distance = instance.get_distance(current_node, node);
                    if (distance < min_distance) {
                        min_distance = distance;
                        next_node = node;
                    }
                }
            }
        }
        assert(next_node != current_node);
        ant.visit(next_node);
        current_node = next_node;
    }
    return ant;
}
struct MMASParameters {
    double rho_ = 0.98;
    uint32_t ants_count_ = 10;
    double beta_ = 2;
    uint32_t cand_list_size_ = 15;
    double p_best_ = 0.05; // Prob. that the constructed sol. will contain only
    double get_evaporation_rate() const { return 1 - rho_; }
};
const uint32_t MaxCandListSize = 64;
uint32_t move_ant_mmas(const ProblemInstance &instance,
                       const PheromoneMemory &pheromone,
                       const vector<double> &heuristic, Ant &ant) {
    assert(!ant.visited_.empty());
    const auto dimension = instance.dimension_;
    const auto current_node = ant.visited_.back();
    const uint32_t offset = current_node * dimension;
    uint32_t cand_list[MaxCandListSize];
```

```
uint32_t cand_list_size = 0;
for (auto node : instance.get_nearest_neighbors(current_node)) {
    if (!ant.is_visited(node)) {
        cand_list[cand_list_size] = node;
        ++cand_list_size;
    }
}
uint32_t chosen_node = current_node;
if (cand_list_size > 0) { // Select from the closest nodes
    double products_prefix_sum[MaxCandListSize] = {0};
    double total = 0;
    for (uint32_t i = 0; i < cand_list_size; ++i) {
        const auto node = cand_list[i];
        const auto product =
            pheromone.get(current_node, node) * heuristic[offset + node];
        total += product;
        products_prefix_sum[i] = total;
    }
    chosen_node = cand_list[cand_list_size - 1];
    const auto r = get_random_double() * total;
    for (uint32_t i = 0; i < cand_list_size; ++i) {
        if (r < products_prefix_sum[i]) {
            chosen_node = cand_list[i];
            break;
        }
    }
} else { // Select from the rest of the unvisited nodes the one with the
    double max_product = 0;
    for (uint32_t node = 0u; node < dimension; ++node) {
        if (!ant.is_visited(node)) {
            const auto product = pheromone.get(current_node, node) *
                                 heuristic[offset + node];
            if (product > max_product) {
                max_product = product;
                chosen_node = node;
            }
        }
    }
}
```

```
    assert(chosen_node != current_node);
    ant.visit(chosen_node);
    return chosen_node;
}
pair<double, double> calc_trail_limits_mmas(const MMASParameters &params,
                                            uint32_t instance_dimension,
                                            double solution_cost) {
    const auto tau_max = 1 / (solution_cost * (1. - params.rho_));
    const auto avg = instance_dimension / 2.;
    const auto p = pow(params.p_best_, 1. / instance_dimension);
    const auto tau_min = min(tau_max, tau_max * (1 - p) / ((avg - 1) * p));
    return make_pair(tau_min, tau_max);
}
Ant run_mmas(const ProblemInstance &instance, const MMASParameters &params,
             uint32_t iterations) {
    const auto greedy_sol = create_solution_nn(instance);
    const auto greedy_cost =
        instance.calculate_route_length(greedy_sol.visited_);
    const auto initial_limits =
        calc_trail_limits_mmas(params, instance.dimension_, greedy_cost);
    auto min_pheromone = initial_limits.first;
    auto max_pheromone = initial_limits.second;
    PheromoneMemory pheromone(instance.dimension_, max_pheromone);
    vector<double> heuristic;
    heuristic.reserve(instance.dimension_ * instance.dimension_);
    for (auto distance : instance.distance_matrix_) {
        heuristic.push_back(1 / pow(distance, params.beta_));
    }
    vector<Ant> ants(params.ants_count_);
    Ant best_ant;
    for (uint32_t iteration = 0; iteration < iterations; ++iteration) {
        for (auto &ant : ants) {
            ant.initialize(instance.dimension_);
            auto start_node = get_random_uint32(0, instance.dimension_ - 1);
            ant.visit(start_node);
            for (uint32_t j = 1; j < instance.dimension_; ++j) {
                move_ant_mmas(instance, pheromone, heuristic, ant);
            }
            ant.cost_ = instance.calculate_route_length(ant.visited_);
```

```
            }
            auto &iteration_best = ants.front();
            bool new_best_found = false;
            for (auto &ant : ants) {
                if (ant.cost_ < best_ant.cost_) {
                    best_ant = ant;
                    new_best_found = true;
                    cout << "New best solution found with the cost: "
                        << best_ant.cost_ << " at iteration " << iteration << endl;
                }
                if (ant.cost_ < iteration_best.cost_) {
                    iteration_best = ant;
                }
            }
            if (new_best_found) {
                auto limits = calc_trail_limits_mmas(params, instance.dimension_,
                                                     best_ant.cost_);
                min_pheromone = limits.first;
                max_pheromone = limits.second;
            }
            pheromone.evaporate_from_all(params.get_evaporation_rate(),
                                         min_pheromone);
            const auto &update_ant = iteration_best;
            const double deposit = 1.0 / update_ant.cost_;
            auto prev_node = update_ant.visited_.back();
            for (auto node : update_ant.visited_) {
            pheromone.increase(prev_node, node, deposit, max_pheromone,
                               instance.is_symmetric_);
                prev_node = node;
            }
        }
        return best_ant;
    }
    int main(int argc, char *argv[]) {
        string path = "kroA100.tsp";
        if (argc >= 2) {
            path = argv[1];
        }
        try {
```

```
        MMASParameters params;
        auto instance = load_tsplib_instance(path.c_str());
        instance.initialize_nn_lists(params.cand_list_size_);
        params.ants_count_ = instance.dimension_;
        run_mmas(instance, params, 1000000 / params.ants_count_);
    } catch (runtime_error e) {
        cout << "An error has occurred: " << e.what() << endl;
    }
    return 0;
}
```

6. Tài liệu tham khảo

1. M. Dorigo, V. Maniezzo and A. Colorni (1991), The Ant System: An autocatalytic optimizing process, Technical Report 91-016 Revised, Dipartimento di Elettronica, Politecnico di Milano, Milano, Italy.
2. M. Dorigo (1992), Optimization, learning and natural algorithms, PhD. dissertation, Milan Polytechnique, Italy.
3. M. Dorigo, and T. Stützle (2004), Ant Colony Optimization, The MIT Press, Cambridge, Masachusetts.