Aymane KABBA 19 00 00 30

CYCLE INGENIEUR EN GENIE INFORMATIQUE – S8

Intelligence artificielle

Rapport – Mini projet 02

Sous l’encadrement de :

Pr. Ayoub AIT LAHCEN

Table des matières

1. Présentation générale :

Berlin52 fait référence à une instance spécifique du Problème du Voyageur de Commerce (TSP), qui est un problème bien connu d'optimisation combinatoire. Le TSP implique de trouver le plus court chemin possible qui visite un ensemble de villes et retourne au point de départ, avec la contrainte supplémentaire que chaque ville doit être visitée exactement une fois.

L'instance Berlin52 du TSP se compose de 52 emplacements à Berlin, en Allemagne, et est souvent utilisée comme problème de référence pour évaluer les algorithmes qui résolvent le TSP. L'objectif est de trouver le chemin le plus court possible qui visite les 52 emplacements exactement une fois.

Il existe de nombreux algorithmes qui peuvent être utilisés pour résoudre le TSP, y compris des méthodes exactes telles que la méthode de branch-and-bound et des méthodes métaheuristiques auxquelles on s'intéresse telles que l'algorithme d'escalade simple, d'escalade complet, l'algorithme de recuit simulé et l'algorithme génétique. L'efficacité de ces algorithmes peut être évaluée en comparant leurs performances sur ce problème de référence : Berlin52.

2. Exécution du code :

2.1. Algorithme d'escalade simple :

```
Algorithme ESCALADE SIMPLE (SIMPLE HILL CLIMBING)

------------------Berlin52----------------------

Best route found: 46 12 13 51 10 50 32 9 8 7 40 18 44 31 21 17 2 16 41 6 1 29 20 30 22 19 49 28 15 0 48 38 39 37 36 35 3
4 33 43 45 47 23 4 5 14 42 3 24 11 27 26 25
Total distance: 8294.09

Process returned 0 (0x0)   execution time : 0.214 s
Press any key to continue.
```

```
Algorithme ESCALADE SIMPLE (SIMPLE HILL CLIMBING)

------------------Berlin52----------------------

Best route found: 17 30 21 31 44 18 40 7 8 9 42 32 50 10 51 13 12 26 46 25 27 11 3 5 14 37 39 38 36 47 23 4 24 45 15 43
33 34 35 48 0 22 19 49 28 29 1 6 41 20 16 2
Total distance: 8037.93

Process returned 0 (0x0)   execution time : 0.205 s
Press any key to continue.
```

2.2. Algorithme d'escalade complet :

```
Algorithme ESCALADE COMPLET (COMPLETE HILL CLIMBING)

------------------Berlin52----------------------

Best route found: 43 20 41 1 6 16 30 21 35 39 14 5 4 36 34 33 47 23 37 38 48 17 2 40 18 44 31 22 29 19 49 0 7 8 9 42 3 3
2 50 11 27 25 46 28 15 45 24 26 12 13 51 10
Total distance: 10929.3

Process returned 0 (0x0)   execution time : 0.096 s
Press any key to continue.
```

```
Algorithme ESCALADE COMPLET (COMPLETE HILL CLIMBING)

------------------Berlin52----------------------

Best route found: 8 9 32 42 39 38 33 43 15 22 20 30 21 19 49 34 35 36 4 14 5 3 27 25 26 10 50 11 24 0 17 16 2 40 18 44 3
1 48 45 46 13 51 12 1 6 41 29 28 47 23 37 7
Total distance: 11649.8

Process returned 0 (0x0)   execution time : 0.062 s
Press any key to continue.
```

2.3. Algorithme du recuit simulé :

```
Algorithme RECUIT SIMULE (SIMULATED ANNEALING)

------------------Berlin52----------------------

Best route: 11 50 3 5 37 33 34 21 0 48 39 42 32 2 17 20 30 35 38 36 26 10 51 13 12 41 1 6 16 8 9 7 31 49 19 43 45 24 23
44 40 18 14 4 47 15 22 29 28 46 25 27
Total distance : 12980.9

Process returned 0 (0x0)   execution time : 0.077 s
Press any key to continue.
```

```
Algorithme RECUIT SIMULE (SIMULATED ANNEALING)

------------------Berlin52----------------------

Best route: 9 42 32 27 24 45 43 48 31 44 40 8 3 50 10 51 13 46 28 35 39 37 14 4 47 33 38 36 34 0 21 30 41 1 6 16 2 22 49
 25 12 26 11 5 23 15 19 29 20 17 18 7
Total distance : 12170.4

Process returned 0 (0x0)   execution time : 0.032 s
Press any key to continue.
```

2.4. Algorithme génétique :

```
          Algorithme GENETIQUE (GENETIC)

------------------Berlin52----------------------

Best Path: 39 36 4 37 47 34 0 33 24 45 42 5 48 17 30 2 16 43 22 28 19 49 11 3 14 35 9 40 32 12 38 7 20 6 21 44 18 8 50 3
1 29 41 1 15 26 13 46 51 10 25 27 23
 Total distance: 18085.8


Process returned 0 (0x0)   execution time : 6.289 s
Press any key to continue.
```

```
          Algorithme GENETIQUE (GENETIC)

------------------Berlin52----------------------

Best Path: 19 27 3 5 4 36 8 9 32 11 29 31 34 14 23 38 21 28 22 30 15 48 24 0 2 18 40 47 44 17 16 7 33 37 12 42 35 39 41
20 1 6 49 13 10 50 25 46 26 51 45 43
 Total distance: 19208.4


Process returned 0 (0x0)   execution time : 6.327 s
Press any key to continue.
```

3. Implémentation et code source :

3.1. Algorithme d'escalade simple :

```cpp
#include <algorithm>
#include <random>
#include <ctime>

using namespace std;

// Définition de l'instance du problème
vector<pair<int, int>> berlin52 = {
    {565, 575}, {25, 185}, {345, 750}, {945, 685}, {845, 655}, {880, 660},
    {25, 230}, {525, 1000}, {580, 1175}, {650, 1130}, {1605, 620},
    {1220, 580}, {1465, 200}, {1530, 5}, {845, 680}, {725, 370},
    {145, 665}, {415, 635}, {510, 875}, {560, 365}, {300, 465},
    {520, 585}, {480, 415}, {835, 625}, {975, 580}, {1215, 245},
    {1320, 315}, {1250, 400}, {660, 180}, {410, 250}, {420, 555},
    {575, 665}, {1150, 1160}, {700, 580}, {685, 595}, {685, 610},
    {770, 610}, {795, 645}, {720, 635}, {760, 650}, {475, 960},
    {95, 260}, {875, 920}, {700, 500}, {555, 815}, {830, 485},
    {1170, 65}, {830, 610}, {605, 625}, {595, 360}, {1340, 725},
    {1740, 245}
};

//Calcul la distance totale d'un circuit
double total_distance(const vector<int>& route, const vector<pair<int, int>>& tsp_instance) {
    int n = route.size();
    double total_dist = 0.0;
    for (int i = 0; i < n; i++) {
        int j = (i + 1) % n;
        pair<int, int> city_i = tsp_instance[route[i]];
        pair<int, int> city_j = tsp_instance[route[j]];
        total_dist += sqrt(pow(city_i.first - city_j.first, 2) + pow(city_i.second - city_j.second, 2));
    }
    return total_dist;
}
```

```cpp
// La fct algo escalade simple
vector<int> simple_hill_climbing(const vector<int>& route, const vector<pair<int, int>>& tsp_instance) {
    vector<int> current_route = route;
    double current_distance = total_distance(current_route, tsp_instance);
    vector<int> best_route = current_route;
    double best_distance = current_distance;
    bool improved = true;

    while (improved) {
        improved = false;

        for (int i = 0; i < current_route.size() - 1; i++) {
            for (int j = i + 1; j < current_route.size(); j++) {
                vector<int> neighbor = current_route;
                reverse(neighbor.begin() + i, neighbor.begin() + j + 1); //2opt
                double neighbor_distance = total_distance(neighbor, tsp_instance);

                if (neighbor_distance < current_distance) {
                    current_distance = neighbor_distance;
                    current_route = neighbor;
                    improved = true;
                    break; // Retenir la première route améliorante
                }
            }
            if (improved) {
                break; // Vérifie si une amélioration est trouvé
            }
        }

        if (current_distance < best_distance) {
            best_distance = current_distance;
            best_route = current_route;
        }
    }
    return best_route; // Retourne la première route améliorante
}
```

La fonction simple hill climbing, est l'implémentation de l'algorithme d'escalade simple, elle prend comme paramètre une route initiale, et l'instance du problème berlin52.

L'algorithme itère sur toutes les routes voisines possibles en inversant une sous-séquence de la route courante (à l'aide de l'algorithme 2opt) et sélectionne la première route améliorante.

```
int main() {

    srand(time(nullptr));
    cout << "\033[32mAlgorithme ESCALADE SIMPLE (SIMPLE HILL CLIMBING)\n\033[0m" << endl;
    cout << "\033[32m--------------------Berlin52----------------------\n\033[0m" << endl;
    // Generate a random initial route
    vector<int> route(berlin52.size());
    std::iota(route.begin(), route.end(), 0);
    std::random_shuffle(route.begin(), route.end());

    // Call the hill climbing function
    std::vector<int> solution = simple_hill_climbing(route, berlin52);

    // Print the best route found
    std::cout << "Best route found:";
    for (int i : solution) {
        std::cout << " " << i;
    }
    std::cout << "\n";
    std::cout << "Total distance: \033[32m" << total_distance(solution, berlin52) << "\033[0m\n";

    return 0;
}
```

3.2. Algorithme d'escalade complet :

```
#include <iostream>
#include <vector>
#include <utility>
#include <algorithm>
#include <random>
#include <cmath>
#include <ctime>
using namespace std;

//Instance de berlin52
vector<pair<int, int>> berlin52 = {
    {565, 575}, {25, 185}, {345, 750}, {945, 685}, {845, 655}, {880, 660},
    {25, 230}, {525, 1000}, {580, 1175}, {650, 1130}, {1605, 620},
    {1220, 580}, {1465, 200}, {1530, 5}, {845, 680}, {725, 370},
    {145, 665}, {415, 635}, {510, 875}, {560, 365}, {300, 465},
    {520, 585}, {480, 415}, {835, 625}, {975, 580}, {1215, 245},
    {1320, 315}, {1250, 400}, {660, 180}, {410, 250}, {420, 555},
    {575, 665}, {1150, 1160}, {700, 580}, {685, 595}, {685, 610},
    {770, 610}, {795, 645}, {720, 635}, {760, 650}, {475, 960},
    {95, 260}, {875, 920}, {700, 500}, {555, 815}, {830, 485},
    {1170, 65}, {830, 610}, {605, 625}, {595, 360}, {1340, 725},
    {1740, 245}
};

// Calcul la distance d'un circuit
double total_distance(const vector<int>& route, const vector<pair<int, int>>& tsp_instance) {
    int n = route.size();
    double total_dist = 0;
    for (int i = 0; i < n; i++) {
        int j = (i + 1) % n;
        auto city_i = tsp_instance[route[i]];
        auto city_j = tsp_instance[route[j]];
        total_dist += sqrt(pow(city_i.first - city_j.first, 2) + pow(city_i.second - city_j.second, 2));
    }
    return total_dist;
}
```

```cpp
// Fonction d'escalade complet
vector<int> hill_climbing(vector<int> route, const vector<pair<int, int>>& tsp_instance, int max_iterations) {
    // Initialisation avec la route donnée en entrée
    vector<int> current_route = route;
    // Calcul de la distance totale de la route courante
    double current_distance = total_distance(current_route, tsp_instance);
    // Variable pour suivre si une amélioration a été trouvée
    bool improved = true;
    // Compteur pour le nombre d'itérations effectuées
    int iteration_count = 0;
    // Initialisation de la meilleure route et de la meilleure distance
    vector<int> best_route = current_route;
    double best_distance = current_distance;

    // Boucle jusqu'à ce qu'aucune amélioration ne soit trouvée ou que le nombre maximal d'itérations soit atteint
    while (improved && iteration_count < max_iterations) {
        // Réinitialisation du booléen pour l'itération en cours
        improved = false;

        // Boucle à travers chaque paire de villes dans la route courante
        for (int i = 0; i < current_route.size() - 1; i++) {
            for (int j = i + 1; j < current_route.size(); j++) {
                // Création d'une nouvelle route en inversant l'ordre des villes entre les indices i et j
                vector<int> neighbor = current_route;
                reverse(neighbor.begin() + i, neighbor.begin() + j + 1); //2opt
                // Calcul de la distance totale de la nouvelle route
                double neighbor_distance = total_distance(neighbor, tsp_instance);

                // Si la nouvelle route est meilleure que la route courante
                if (neighbor_distance < current_distance) {
                    // Mise à jour de la route courante et de la distance totale
                    current_distance = neighbor_distance;
                    current_route = neighbor;
                    // Indiquer qu'une amélioration a été trouvée pour cette itération
                    improved = true;
                }
```

```cpp
            }
        }

        // Si la distance de la route courante est meilleure que la meilleure distance trouvée jusqu'à présent
        if (current_distance < best_distance) {
            // Mettre à jour la meilleure route et la meilleure distance
            best_distance = current_distance;
            best_route = current_route;
        }

        // Incrémenter le compteur d'itérations
        iteration_count++;
    }

    // Retourner la meilleure route trouvée
    return best_route;
}
```

La fonction hill climbing, est l'implémentation de l'algorithme d'escalade simple, elle prend comme paramètre une route initiale, l'instance du problème berlin52 et le nombre d'itérations maximales.

L'algorithme itère sur toutes les routes voisines possibles en inversant une sous-séquence de la route courante (à l'aide de l'algorithme 2opt) jusqu'à ce qu'aucune amélioration ne soit trouvée ou que le nombre maximal d'itérations soit atteint.

Si une amélioration est trouvée, la meilleure route est mise à jour.

```
int main() {

    srand(time(nullptr));
    cout << "\033[32mAlgorithme ESCALADE COMPLET (COMPLETE HILL CLIMBING)\n\033[0m" << endl;
    cout << "\033[32m--------------------Berlin52----------------------\n\033[0m" << endl;
    // Generate a random initial route
    vector<int> route(berlin52.size());
    for (int i = 0; i < route.size(); i++) {
        route[i] = i;
    }
    std::random_shuffle(route.begin(), route.end());

    // Call the hill climbing function
    int max_iterations = 1000;
    std::vector<int> solution = hill_climbing(route, berlin52, max_iterations);

    // Print the best route found
    std::cout << "Best route found:";
    for (int i : solution) {
        std::cout << " " << i;
    }
    std::cout << "\n";
    std::cout << "Total distance: \033[32m" << total_distance(solution, berlin52) << "\033[0m\n";

    return 0;
}
```

3.3. Algorithme de recuit simulé :

```
#include <iostream>
#include <vector>
#include <utility>
#include <cmath>
#include <random>
#include <iostream>
#include <algorithm>
#include <ctime>

using namespace std;

// Instance berlin52
vector<pair<int, int>> berlin52 = {
    {565, 575}, {25, 185}, {345, 750}, {945, 685}, {845, 655}, {880, 660},
    {25, 230}, {525, 1000}, {580, 1175}, {650, 1130}, {1605, 620},
    {1220, 580}, {1465, 200}, {1530, 5}, {845, 680}, {725, 370},
    {145, 665}, {415, 635}, {510, 875}, {560, 365}, {300, 465},
    {520, 585}, {480, 415}, {835, 625}, {975, 580}, {1215, 245},
    {1320, 315}, {1250, 400}, {660, 180}, {410, 250}, {420, 555},
    {575, 665}, {1150, 1160}, {700, 580}, {685, 595}, {685, 610},
    {770, 610}, {795, 645}, {720, 635}, {760, 650}, {475, 960},
    {95, 260}, {875, 920}, {700, 500}, {555, 815}, {830, 485},
    {1170, 65}, {830, 610}, {605, 625}, {595, 360}, {1340, 725},
    {1740, 245}
};
// calcul de la distance d'un circuit
double cost(const vector<int>& route) {
    double total_distance = 0.0;
    for (int i = 0; i < route.size(); i++) {
        int x1 = berlin52[route[i]].first;
        int y1 = berlin52[route[i]].second;
        int x2 = berlin52[route[(i+1)%route.size()]].first;
        int y2 = berlin52[route[(i+1)%route.size()]].second;
        double distance = sqrt((x2-x1)*(x2-x1) + (y2-y1)*(y2-y1));
        total_distance += distance;
    }
    return total_distance;
}
```

```cpp
// Implementation de l'algorithme
vector<int> simulated_annealing(double temperature, double cooling_rate) {
    // Initialiser la route courante à une route aléatoire
    vector<int> current_route(berlin52.size());
    for (int i = 0; i < current_route.size(); i++) {
        current_route[i] = i;
    }
    random_shuffle(current_route.begin(), current_route.end());
    // Initialiser la meilleur route avec la courante
    vector<int> best_route(current_route);
    // Paramètre du générateur de nombre aléatoires
    random_device rd;
    mt19937 rng(rd());
    uniform_int_distribution<int> uni(0, current_route.size()-1);
    uniform_real_distribution<double> unif(0.0, 1.0);
    // Répéter jusqu'à que la température est très faible
    while (temperature > 1e-6) {
        // la route généré voisine
        int i = uni(rng);
        int j = uni(rng);
        vector<int> new_route(current_route);
        swap(new_route[i], new_route[j]);
        // Variation du cout entre la nouvelle route et celle courante
        double delta_cost = cost(new_route) - cost(current_route);
        // Quand retenir une route
        if (delta_cost < 0.0 || exp(-delta_cost/temperature) > unif(rng)) {
            current_route = new_route;
        }
        // Mise à jour de la meilleur route
        if (cost(current_route) < cost(best_route)) {
            best_route = current_route;
        }
        // Diminuer la temperature
        temperature *= cooling_rate;
    }
    return best_route;
}
```

La fonction implémente l'algorithme du recuit simulé. Elle commence par générer une solution aléatoire pour représenter une route. Ensuite, elle génère une nouvelle solution candidate en échangeant deux villes choisies aléatoirement dans la solution courante et calcule son coût. La fonction décide alors si elle accepte ou rejette la nouvelle solution en fonction de la différence de coût et de la température. Si la nouvelle solution est acceptée, la solution courante est mise à jour. Si la nouvelle solution est rejetée, la solution courante n'est pas mise à jour. La température diminue progressivement jusqu'à ce qu'elle soit suffisamment basse. Enfin, la fonction renvoie la meilleure solution trouvée.

```cpp
int main() {

    srand(time(nullptr));
    cout << "\033[32mAlgorithme RECUIT SIMULE (SIMULATED ANNEALING)\n\033[0m" << endl;
    cout << "\033[32m--------------------Berlin52----------------------\n\033[0m" << endl;
    // Set the initial temperature and cooling rate
    double temperature = 100.0;
    double cooling_rate = 0.99;
    // Run the simulated annealing algorithm
    vector<int> best_route = simulated_annealing(temperature, cooling_rate);
    // Print the best route and its cost
    cout << "Best route: ";
    for (int i = 0; i < best_route.size(); i++) {
        cout << best_route[i] << " ";
    }
    cout << endl;
    cout << "Total distance : \033[32m" << cost(best_route) << "\033[0m\n";
    return 0;
}
```

3.4. Algorithme génétique :

```cpp
#include <iostream>
#include <cmath>
#include <vector>
#include <ctime>
#include <algorithm>

using namespace std;
const int POPULATION_SIZE = 100;

// Define the TSP instance
vector<pair<int, int>> berlin52 = {
        {565, 575}, {25, 185}, {345, 750}, {945, 685}, {845, 655}, {880, 660},
        {25, 230}, {525, 1000}, {580, 1175}, {650, 1130}, {1605, 620},
        {1220, 580}, {1465, 200}, {1530, 5}, {845, 680}, {725, 370},
        {145, 665}, {415, 635}, {510, 875}, {560, 365}, {300, 465},
        {520, 585}, {480, 415}, {835, 625}, {975, 580}, {1215, 245},
        {1320, 315}, {1250, 400}, {660, 180}, {410, 250}, {420, 555},
        {575, 665}, {1150, 1160}, {700, 580}, {685, 595}, {685, 610},
        {770, 610}, {795, 645}, {720, 635}, {760, 650}, {475, 960},
        {95, 260}, {875, 920}, {700, 500}, {555, 815}, {830, 485},
        {1170, 65}, {830, 610}, {605, 625}, {595, 360}, {1340, 725},
        {1740, 245}
};

double total_distance(const vector<int>& route, const vector<pair<int, int>>& tsp_instance) {
    int n = route.size();
    double total_dist = 0.0;
    for (int i = 0; i < n; i++) {
        int j = (i + 1) % n;
        pair<int, int> city_i = tsp_instance[route[i]];
        pair<int, int> city_j = tsp_instance[route[j]];
        total_dist += sqrt(pow(city_i.first - city_j.first, 2) + pow(city_i.second - city_j.second, 2));
    }
    return total_dist;
}

bool cmp(const vector<int>& route1, const vector<int>& route2) {
    return total_distance(route1, berlin52) < total_distance(route2, berlin52);
}


vector<vector<int>> selection(const vector<vector<int>>& population, const string& option) {
    if (option == "random") {
        int r = rand() % POPULATION_SIZE;
        return {population[r]};
    }
    if (option == "roulette") {
        double total_fitness = 0;
        for (int i = 0; i < POPULATION_SIZE; ++i) {
            total_fitness += total_distance(population[i], berlin52);
        }
        double r = rand() / (double) RAND_MAX * total_fitness;
        double partial_sum = 0.0;
        for (int i = 0; i < POPULATION_SIZE; i++) {
            partial_sum += total_distance(population[i], berlin52);
            if (partial_sum >= r) {
                return {population[i]};
            }
        }
        return {population[POPULATION_SIZE - 1]};
    }
}
```

Cette fonction prend en entrée une population de solutions représentées par des vecteurs d'entiers, ainsi qu'une option de sélection ("random" ou "roulette"). Si l'option est "random", la fonction choisit une solution aléatoire dans la population et la retourne sous forme de vecteur de vecteurs. Si l'option est "roulette", la fonction commence par calculer la somme des performances de tous les individus de la population. Ensuite, un nombre aléatoire est généré entre 0 et cette somme. La fonction itère ensuite sur tous les individus de la population, en accumulant leur performance jusqu'à ce que la somme accumulée

soit supérieure ou égale au nombre aléatoire. L'individu correspondant à la somme accumulée est alors sélectionné. Si la boucle itère sur tous les individus sans en sélectionner aucun, la fonction renvoie l'individu ayant la performance la plus élevée. La fonction retourne la solution sélectionnée.

```
vector<vector<int>> crossover(vector<int> x, vector<int> y) {
    vector<vector<int>> child(2, vector<int>(x.size()));

    int index1 = rand() % (x.size()-1) ;
    int index2 = rand() % (x.size()-1) ;
    int startIndex = min(index1,index2);
    int endIndex = max(index1,index2);

    vector<int> takenCities_ids;

    // Recopier la partie du croisement
    for (int i = startIndex; i <= endIndex; i++) {
        child[0][i] = x[i];
        takenCities_ids.push_back(x[i]);
    }

    int j = 0;
    for (int i = 0; i < y.size(); i++) {
        if (j == startIndex) {
            j = endIndex + 1;
        }

        if (find(takenCities_ids.begin(), takenCities_ids.end(), y[i]) == takenCities_ids.end()) {
            child[0][j] = y[i];
            j++;
        }
    }
    child[1] = y;
    return child;
}
```

L'objectif est de cette fonction et de créer deux nouveaux individus enfants en combinant les caractéristiques de leurs parents.

Tout d'abord, deux indices d'emplacements sont choisis au hasard dans les deux parents x et y. Ces indices représentent la plage de villes qui seront échangées entre les deux parents pour créer les enfants. Les villes entre ces deux indices sont copiées du parent x pour le premier enfant, et les villes restantes du parent y sont ajoutées dans l'ordre dans lequel elles apparaissent, à partir de l'indice endIndex + 1.

Le deuxième enfant est créé en échangeant les rôles des parents, donc les villes copiées de y et les villes restantes de x sont combinées.

```
void mutation(vector<vector<int>>& x, string option) {
    int index1 = rand() % x.size();
    int index2 = rand() % x.size();
    if (option == "singleSwap") {
        std::swap(x[index1], x[index2]);
    }
    if (option == "2optSwap") {
        reverse(begin(x) + index1 + 1, begin(x) + index2 + 1);
    }
}

double averagePopulationFitness(const vector<vector<int>>& population){
    double total = 0.0;
    for (int i = 0; i < POPULATION_SIZE; ++i) {
        total += total_distance(population[i], berlin52);
    }
    double trueAvg = total / POPULATION_SIZE;
    return trueAvg;
}
```

La fonction mutation implémente deux options de mutation différentes pour la population x passée en paramètre. Si l'option est "singleSwap", elle échange deux éléments choisis aléatoirement dans un membre de la population. Si l'option est "2optSwap", elle effectue un "2-opt swap" qui consiste à inverser l'ordre des villes entre deux positions choisies aléatoirement dans un membre de la population.

La fonction averagePopulationFitness calcule la moyenne des coûts des solutions dans la population passée en paramètre.

```
void genetic_algorithm() {
    const int NUM_GENERATIONS = 1000;
    const double MUTATION_RATE = 0.1;


    vector<vector<int>> population(POPULATION_SIZE, vector<int>(berlin52.size()));
    for (int i = 0; i < POPULATION_SIZE; i++) {
        for (int j = 0; j < berlin52.size(); j++) {
            population[i][j] = j;
        }
        random_shuffle(population[i].begin(), population[i].end());
    }

    vector<int> best_individual;
    double best_fitness = numeric_limits<double>::max();

    for (int g = 0; g < NUM_GENERATIONS; g++) {
        // Selection
        vector<vector<int>> parents;
        for (int i = 0; i < 2; i++) {
            parents.push_back(selection(population, "roulette")[0]);
        }

        vector<vector<int>> offspring = crossover(parents[0], parents[1]);

        if (rand() / (double)RAND_MAX < MUTATION_RATE) {
            mutation(offspring, "2optSwap");
        }

        population.insert(population.end(), offspring.begin(), offspring.end());

        sort(population.begin(), population.end(), cmp);

        double current_fitness = total_distance(population[0], berlin52);
        if (current_fitness < best_fitness) {
            best_individual = population[0];
            best_fitness = current_fitness;
        }

        population.resize(POPULATION_SIZE);

        double avg_fitness = averagePopulationFitness(population);
        //cout << "Generation " << g + 1 << ", Average Fitness: " << avg_fitness << ", Best Fitness: " << best_fitness << endl;
    }

    cout << "Best Path: ";
    for (int i = 0; i < best_individual.size(); i++) {
        cout << best_individual[i] << " ";
    }
    std::cout << "\n Total distance: \033[32m" << total_distance(best_individual, berlin52) << "\033[0m\n";
    cout << endl;

}
```

L'algorithme commence par initialiser une population de chemins aléatoires et procède ensuite par itérations sur un nombre prédéfini de générations. À chaque itération, il sélectionne les parents les plus performants à l'aide de la méthode de sélection de la roulette, utilise le croisement pour générer des descendants, effectue une mutation avec une certaine probabilité et évalue la qualité de la nouvelle population en termes de la distance totale parcourue. Le processus de sélection et de reproduction est répété jusqu'à ce que le nombre maximal de générations soit atteint ou jusqu'à ce qu'une solution satisfaisante soit trouvée.

Le meilleur chemin trouvé et sa distance totale sont ensuite affichés à la fin du processus.

```cpp
int main() {
    srand(time(nullptr));

    cout << "\033[32m           Algorithme GENETIQUE (GENETIC)        \n\033[0m" << endl;
    cout << "\033[32m-------------------Berlin52---------------------\n\033[0m" << endl;
    genetic_algorithm();

    return 0;
}
```