Traveling Salesman Problem (TSP) Genetic Algorithm

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <random>
#include <cmath>
#include <numeric>
#include <chrono>
#include <iomanip>

// Configuration constants (user can modify these)
constexpr int POP_SIZE = 100;
constexpr int MAX_GENERATIONS = 500;
constexpr double MUTATION_RATE = 0.02;
constexpr double CROSSOVER_RATE = 0.9;
constexpr int TOURNAMENT_SIZE = 5;
constexpr int ELITISM_COUNT = 2;

// Global variables that will be set based on user input
int NUM_CITIES;
std::vector<std::vector<double>> distance_matrix;

// Individual representing a solution
struct Individual {
    std::vector<int> path;
    double fitness = 0.0;
```

```cpp
    double distance = 0.0;

    Individual() = default;

    explicit Individual(int size) : path(size) {
        std::iota(path.begin(), path.end(), 0);
    }
};

// Function to get user input for number of cities
int get_number_of_cities() {
    int num;
    std::cout << "Enter number of cities: ";
    while (!(std::cin >> num) || num < 2) {
        std::cin.clear();
        std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
        std::cout << "Invalid input. Please enter an integer >= 2: ";
    }
    return num;
}

// Function to get distance matrix from user
void get_distance_matrix() {
    distance_matrix.resize(NUM_CITIES, std::vector<double>(NUM_CITIES, 0.0));

    std::cout << "\nEnter distances between cities (use 0 for same city):\n";
    for (int i = 0; i < NUM_CITIES; ++i) {
        for (int j = 0; j < NUM_CITIES; ++j) {
            if (i == j) {
```

```
                distance_matrix[i][j] = 0.0;
                continue;
            }

            if (j > i) { // Only ask for upper triangular matrix
                double dist;
                std::cout << "Distance from city " << i+1 << " to city " << j+1 << ": ";
                while (!(std::cin >> dist) || dist <= 0) {
                    std::cin.clear();
                    std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
                    std::cout << "Invalid input. Please enter a positive number: ";
                }
                distance_matrix[i][j] = dist;
                distance_matrix[j][i] = dist; // Symmetric TSP
            }
        }
    }
}

// Calculate total distance of a path
double calculate_path_distance(const std::vector<int>& path) {
    double total = 0.0;
    for (size_t i = 0; i < path.size() - 1; ++i) {
        total += distance_matrix[path[i]][path[i+1]];
    }
    total += distance_matrix[path.back()][path.front()]; // Return to start
    return total;
}
```

```
// Initialize population with random permutations
void initialize_population(std::vector<Individual>& population) {
    std::random_device rd;
    std::mt19937 gen(rd());

    population.resize(POP_SIZE);
    for (auto& individual : population) {
        individual = Individual(NUM_CITIES);
        std::shuffle(individual.path.begin(), individual.path.end(), gen);
        individual.distance = calculate_path_distance(individual.path);
        individual.fitness = 1.0 / individual.distance;
    }
}

// Tournament selection
int tournament_selection(const std::vector<Individual>& population) {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> dis(0, POP_SIZE - 1);

    int best_idx = dis(gen);
    double best_fitness = population[best_idx].fitness;

    for (int i = 1; i < TOURNAMENT_SIZE; ++i) {
        int candidate = dis(gen);
        if (population[candidate].fitness > best_fitness) {
            best_idx = candidate;
            best_fitness = population[candidate].fitness;
        }
```

```
    }
    return best_idx;
}

// Ordered crossover (OX)
void ordered_crossover(const Individual& parent1, const Individual& parent2, Individual& child) {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> dis(0, NUM_CITIES - 1);

    int start = dis(gen);
    int end = dis(gen);
    if (start > end) std::swap(start, end);

    std::vector<bool> used(NUM_CITIES, false);

    // Copy segment from parent1
    for (int i = start; i <= end; ++i) {
        child.path[i] = parent1.path[i];
        used[child.path[i]] = true;
    }

    // Fill remaining from parent2
    int current_pos = (end + 1) % NUM_CITIES;
    for (int i = 0; i < NUM_CITIES; ++i) {
        int candidate = parent2.path[(end + 1 + i) % NUM_CITIES];
        if (!used[candidate]) {
            child.path[current_pos] = candidate;
            current_pos = (current_pos + 1) % NUM_CITIES;
```

```cpp
        }
    }
}

// Inversion mutation (more effective than swap for TSP)
void invert_mutation(Individual& individual) {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> dis(0, NUM_CITIES - 1);
    std::uniform_real_distribution<> prob(0.0, 1.0);

    if (prob(gen) < MUTATION_RATE) {
        int start = dis(gen);
        int end = dis(gen);
        if (start > end) std::swap(start, end);

        while (start < end) {
            std::swap(individual.path[start], individual.path[end]);
            start++;
            end--;
        }

        // Update fitness
        individual.distance = calculate_path_distance(individual.path);
        individual.fitness = 1.0 / individual.distance;
    }
}

// Main genetic algorithm
```

```
void run_genetic_algorithm() {
  std::vector<Individual> population;
  initialize_population(population);

  std::vector<Individual> new_population(POP_SIZE);

  auto start_time = std::chrono::high_resolution_clock::now();

  for (int generation = 0; generation < MAX_GENERATIONS; ++generation) {
    // Sort by fitness (descending)
    std::sort(population.begin(), population.end(),
      [](const Individual& a, const Individual& b) {
        return a.fitness > b.fitness;
      });

    // Elitism: keep top individuals
    for (int i = 0; i < ELITISM_COUNT; ++i) {
      new_population[i] = population[i];
    }

    // Create new population
    for (int i = ELITISM_COUNT; i < POP_SIZE; ++i) {
      if (static_cast<double>(rand()) / RAND_MAX < CROSSOVER_RATE) {
        int parent1 = tournament_selection(population);
        int parent2 = tournament_selection(population);

        Individual child(NUM_CITIES);
        ordered_crossover(population[parent1], population[parent2], child);
        invert_mutation(child);
```

```
            new_population[i] = child;
        } else {
            // Copy without crossover
            new_population[i] = population[tournament_selection(population)];
            invert_mutation(new_population[i]);
        }
    }

    population = new_population;

    // Print progress
    if (generation % 50 == 0) {
        std::cout << "Generation " << std::setw(3) << generation
                  << "  Best distance: " << std::fixed << std::setprecision(2)
                  << 1.0 / population[0].fitness << std::endl;
    }
}

auto end_time = std::chrono::high_resolution_clock::now();
auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(end_time - start_time);

std::cout << "\nOptimization complete!\n";
std::cout << "Best solution distance: " << std::fixed << std::setprecision(2)
          << 1.0 / population[0].fitness << "\n";
std::cout << "Execution time: " << duration.count() << " ms\n";

// Print best path
std::cout << "\nBest path: ";
for (int city : population[0].path) {
```

```cpp
        std::cout << city+1 << " "; // Display as 1-based index
    }
    std::cout << population[0].path[0]+1 << "\n"; // Return to start
}

int main() {
    std::cout << "Traveling Salesman Problem Solver using Genetic Algorithm\n";
    std::cout << "---------------------------------------------------\n";

    NUM_CITIES = get_number_of_cities();
    get_distance_matrix();

    run_genetic_algorithm();
    return 0;
}
```