create a traveling salesman app in c, with 1200 nodes, optimize it!

To create an optimized Traveling Salesman Problem (TSP) application for **1200 nodes** in C, we need to leverage efficient heuristics and algorithms given TSP's NP-hard nature (exact solutions are infeasible for large n). Below is a complete implementation using **Nearest Neighbor (NN)** for initial route generation, **2-Opt** for local optimization, and **Simulated Annealing (SA)** to escape local optima—providing a robust solution for 1200 nodes.

Key Optimizations for 1200 Nodes

1. **Nearest Neighbor (NN)**: Fast greedy heuristic to generate an initial route (sensitive to start node; we mitigate this with random restarts).

2. **2-Opt Local Search**: Improves the initial route by reversing segments to reduce total distance (fixes suboptimal edges).

3. **Simulated Annealing (SA)**: Probabilistic meta-heuristic that escapes local optima by allowing "bad" moves at high temperatures, then cooling down—critical for 1200 nodes where 2-Opt alone gets stuck.

Complete Code Implementation

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <string.h>

#define MAX_NODES 1200      // Target: 1200 nodes
#define INF 1e18            // Infinity for distance comparisons

// Struct to hold city coordinates (x, y) and distance matrix
typedef struct {
   double x, y;
} City;
```

```
City cities[MAX_NODES];
double dist_matrix[MAX_NODES][MAX_NODES]; // Precomputed distances between cities


// Compute Euclidean distance between two cities
double euclidean_distance(int i, int j) {
   double dx = cities[i].x - cities[j].x;
   double dy = cities[i].y - cities[j].y;
   return sqrt(dx*dx + dy*dy);
}

// Precompute distance matrix (symmetric: dist[i][j] = dist[j][i])
void compute_distance_matrix(int n) {
   for (int i = 0; i < n; i++) {
      dist_matrix[i][i] = 0.0; // Distance to self is zero
      for (int j = i + 1; j < n; j++) {
         double d = euclidean_distance(i, j);
         dist_matrix[i][j] = d;
         dist_matrix[j][i] = d; // Symmetry optimization
      }
   }
}

// Calculate total distance of a route (tour)
double route_distance(int *route, int n) {
   double total = 0.0;
   for (int i = 0; i < n; i++) {
      int j = (i + 1) % n; // Wrap around to start after last node
      total += dist_matrix[route[i]][route[j]];
   }
   return total;
}
```

```
// Nearest Neighbor heuristic: generate initial route starting from 'start'
void nearest_neighbor(int start, int *route, int n) {
    int visited[MAX_NODES] = {0}; // Track visited nodes
    route[0] = start;            // Start at the given city
    visited[start] = 1;          // Mark as visited
    int current = start;         // Current city in the tour

    for (int i = 1; i < n; i++) { // Add remaining n-1 cities
        double min_dist = INF;
        int next_city = -1;

        // Find nearest unvisited city
        for (int j = 0; j < n; j++) {
            if (!visited[j] && dist_matrix[current][j] < min_dist) {
                min_dist = dist_matrix[current][j];
                next_city = j;
            }
        }

        route[i] = next_city;     // Add to route
        visited[next_city] = 1;   // Mark as visited
        current = next_city;      // Move to next city
    }
}

// 2-Opt local search: optimize a route by reversing segments
void two_opt(int *route, int n) {
    double best_dist = route_distance(route, n);
    int improved = 1;

    while (improved) {
```

```
        improved = 0;
        // Iterate all possible segment reversals (i, j) where i < j-1
        for (int i = 0; i < n - 1 && !improved; i++) {
            for (int j = i + 2; j < n; j++) {
                // Current edges: (route[i], route[i+1]) and (route[j], route[j+1] mod n)
                double old_edge1 = dist_matrix[route[i]][route[(i+1)%n]];
                double old_edge2 = dist_matrix[route[j%n]][route[(j+1)%n]];

                // New edges after reversing i+1 to j: (route[i], route[j]) and (route[i+1], route[j+1] mod n)
                double new_edge1 = dist_matrix[route[i]][route[j]];
                double new_edge2 = dist_matrix[route[(i+1)%n]][route[(j+1)%n]];

                double delta = (new_edge1 + new_edge2) - (old_edge1 + old_edge2);

                // If reversing improves distance, do it and update
                if (delta < 0) {
                    // Reverse the segment from i+1 to j
                    for (int k = 0; k <= (j - i - 1)/2; k++) {
                        int temp = route[i + 1 + k];
                        route[i + 1 + k] = route[j - k];
                        route[j - k] = temp;
                    }

                    best_dist += delta; // Update total distance
                    improved = 1;       // Flag improvement
                    break;              // Break j-loop to restart i-loop
                }
            }
        }
    }
}
```

```
// Simulated Annealing: escape local optima with probabilistic moves
void simulated_annealing(int *route, int n, double initial_temp, double cooling_rate, int iterations_per_temp) {
    double T = initial_temp;
    int current_route[MAX_NODES];
    memcpy(current_route, route, n * sizeof(int));
    double current_dist = route_distance(current_route, n);

    int best_route[MAX_NODES];
    memcpy(best_route, current_route, n * sizeof(int));
    double best_dist = current_dist;

    srand(time(NULL)); // Seed random number generator

    while (T > 1e-4) { // Stop when temperature is very low
        for (int i = 0; i < iterations_per_temp; i++) {
            // Generate neighbor: swap two random distinct nodes
            int i1 = rand() % n;
            int i2 = rand() % n;
            while (i1 == i2) i2 = rand() % n;

            int neighbor_route[MAX_NODES];
            memcpy(neighbor_route, current_route, n * sizeof(int));
            int temp = neighbor_route[i1];
            neighbor_route[i1] = neighbor_route[i2];
            neighbor_route[i2] = temp;

            double neighbor_dist = route_distance(neighbor_route, n);
            double delta = neighbor_dist - current_dist;

            // Accept neighbor if better, or with probability exp(-delta/T) if worse
            if (delta < 0 || (rand() / (double)RAND_MAX) < exp(-delta / T)) {
```

```
            memcpy(current_route, neighbor_route, n * sizeof(int));
            current_dist = neighbor_dist;

            // Update best route if current is better
            if (current_dist < best_dist) {
               memcpy(best_route, current_route, n * sizeof(int));
               best_dist = current_dist;
            }
         }
      }
      T *= cooling_rate; // Cool down the system
   }

   memcpy(route, best_route, n * sizeof(int)); // Return best route found
}

// Read TSP file (TSPLIB format: nodes with x,y coordinates)
int read_tsp_file(const char *filename, int max_nodes) {
   FILE *file = fopen(filename, "r");
   if (!file) { fprintf(stderr, "Error opening file\n"); return -1; }

   char line[1024];
   int n = 0;

   // Skip header lines (e.g., NAME, TYPE, DIMENSION)
   while (fgets(line, sizeof(line), file)) {
      if (strstr(line, "DIMENSION")) {
         sscanf(line, "%*s %d", &n);
         if (n > max_nodes) { fprintf(stderr, "Too many nodes\n"); return -1; }
      } else if (strstr(line, "NODE_COORD_SECTION")) {
         break; // Start reading coordinates
      }
```

```
    }

    // Read node coordinates
    for (int i = 0; i < n && fgets(line, sizeof(line), file); i++) {
        int idx;
        if (sscanf(line, "%d %lf %lf", &idx, &cities[i].x, &cities[i].y) == 3) {
            // TSPLIB nodes are 1-indexed; we use 0-indexed here
            i--; // Re-read if idx doesn't match (some files have gaps)
        }
    }

    fclose(file);
    return n;
}

int main(int argc, char *argv[]) {
    if (argc != 2) { fprintf(stderr, "Usage: %s <tsp_file>\n", argv[0]); return 1; }

    // Read TSP file and compute distance matrix
    int n = read_tsp_file(argv[1], MAX_NODES);
    if (n <= 0) return 1;
    compute_distance_matrix(n);

    printf("TSP Instance: %d nodes\n", n);

    // Step 1: Generate initial route with random NN restarts (mitigate start-node bias)
    int best_initial_route[MAX_NODES];
    double best_initial_dist = INF;
    const int num_restarts = 5; // 5 random starts for NN

    for (int r = 0; r < num_restarts; r++) {
        int start = rand() % n; // Random starting city
```

```
    int route[MAX_NODES];
    nearest_neighbor(start, route, n);
    double dist = route_distance(route, n);

    if (dist < best_initial_dist) {
        best_initial_dist = dist;
        memcpy(best_initial_route, route, n * sizeof(int));
    }
}

printf("Initial NN Route Distance: %.2f\n", best_initial_dist);

// Step 2: Optimize with 2-Opt local search
int two_opt_route[MAX_NODES];
memcpy(two_opt_route, best_initial_route, n * sizeof(int));
two_opt(two_opt_route, n);
double two_opt_dist = route_distance(two_opt_route, n);

printf("2-Opt Optimized Distance: %.2f\n", two_opt_dist);

// Step 3: Escape local optima with Simulated Annealing
int sa_route[MAX_NODES];
memcpy(sa_route, two_opt_route, n * sizeof(int));
simulated_annealing(
    sa_route,         // Route to optimize
    n,                // Number of nodes
    1000.0,           // Initial temperature (higher = more exploration)
    0.95,             // Cooling rate (slower cooling = better results)
    100               // Iterations per temperature level
);
double sa_dist = route_distance(sa_route, n);
```

```
    printf("Simulated Annealing Distance: %.2f\n", sa_dist);

    // Optional: Print the final route (comment out for large n)
    /*printf("\nFinal Route:\n");
    for (int i = 0; i < n; i++) {
        printf("%d ", sa_route[i] + 1); // Convert back to 1-indexed for TSPLIB compatibility
    }
    printf("\n");*/

    return 0;
}
```

How to Use the Code

1. **Input Format**

The code reads TSP files in the standard **TSPLIB format**, which includes:

- Header lines (e.g., `NAME`, `TYPE`, `DIMENSION`).
- A `NODE_COORD_SECTION` with node indices and (x, y) coordinates.

Example TSP file snippet:

```
NAME: ulysses16.tsp
TYPE: TSP
DIMENSION: 16
EDGE_WEIGHT_TYPE: EUC_2D
NODE_COORD_SECTION
1 38.24 20.42
2 39.57 26.15
...
```

2. **Compile and Run**

Use a C compiler (e.g., GCC) to build the app:

```bash
gcc -O3 tsp_optimized.c -o tsp_optimized -lm
```

Run with a TSP file (ensure it has ≤1200 nodes):

```bash
./tsp_optimized ulysses16.tsp  # Example with 16 nodes; replace with your 1200-node file
```

Key Optimizations Explained

1. **Distance Matrix Symmetry**

The distance matrix is computed once and stored symmetrically (`dist[i][j] = dist[j][i]`), avoiding redundant calculations during route evaluation—critical for speed with 1200 nodes (reduces computations from $O(n^2)$ to $O(n^2/2)$).

2. **Random Restarts in NN**

The Nearest Neighbor heuristic is prone to poor initial routes if starting at a bad node. We use 5 random restarts to select the best initial route, reducing bias.

3. **2-Opt Efficiency**

The 2-Opt algorithm optimizes by reversing subpaths that reduce total distance. It runs in $O(n^2)$ time but is highly effective for local optimization—often cutting the initial NN distance by 10–30%.

4. **Simulated Annealing Parameters**

- **Initial Temperature**: Set to `1000.0` (high enough to allow large moves early on).
- **Cooling Rate**: `0.95` (slow cooling to balance exploration and exploitation).
- **Iterations per Temp**: `100` (enough samples at each temperature to avoid local optima traps).

Performance for 1200 Nodes

For a 1200-node TSP instance:

- **NN Initialization**: ~O(n^2) time (fast, as it's greedy).

- **2-Opt**: ~O(n^2) time (optimizes the initial route quickly).

- **Simulated Annealing**: ~O(k * n^2), where `k` is the number of temperature levels—still feasible for 1200 nodes with reasonable parameters (e.g., 100 iterations/temp and 50 temp levels: 1200^2 * 50 = ~72 million operations, manageable on modern CPUs).

Expected Results

For a 1200-node TSP instance:

- The initial NN route will be far from optimal (e.g., 2–3x the optimal distance for random instances).

- 2-Opt will reduce this by ~30–50%.

- Simulated Annealing will push it closer to the known optimal (or proven lower bound) by escaping local optima.

This implementation balances speed and quality, making it suitable for large TSP instances like 1200 nodes.