Thought for 180 minutes 14 seconds

Got it, let's tackle this. The user asked for a TSP tool with 3000 nodes. First, I need to remember that exact algorithms are impossible for 3000 nodes—they're NP-hard. So we need a heuristic or metaheuristic that can handle large N efficiently.

The previous response used 2-opt, which is good but maybe slow for 3000 nodes if not optimized. Let's think: 2-opt has O(N²) time per iteration, and with 3000 nodes, that's 9 million operations per iteration. If we do many iterations, that's going to be slow. Maybe add a faster initial heuristic first, like nearest neighbor (NN), which is O(N²) but gives a starting point better than random, so 2-opt can improve it faster.

Also, the user might want it to run in reasonable time. So:

Initialization: Generate random points (or let user input, but random is easier for example).
Initial Tour: Use Nearest Neighbor (NN) heuristic—starts at a random node, then each step goes to the closest unvisited node. Faster convergence than random order.
Local Search: 2-opt optimization, but maybe optimize the inner loop to be faster (like using integers instead of doubles where possible, precompute distances? Wait, no—distances are needed on the fly, but with 3000 nodes, storing a distance matrix would take 3000*3000 = 9 million doubles, which is ~72 MB (since each double is 8 bytes)—that's manageable. Precomputing distances can save time in 2-opt because we don't recalculate dist(a,b) every time.
Early Termination: Stop when no improvements for a number of iterations to save time.
Output: Print initial and final length, maybe the tour (but 3000 nodes is too long to print, so skip that or let user choose).
Wait, precomputing distance matrix: yes, for 3000 nodes, it's feasible. Let's add that. So create a 2D array (or vector of vectors) dist where dist[i][j] is the distance between node i and j. That way, in 2-opt, we just look up dist instead of calculating sqrt each time—much faster.

Also, NN with random start: sometimes starting at different nodes gives better initial tours. Maybe pick a few random starts (like 5) and take the best one—this is called "stochastic NN". But for simplicity, maybe just one random start first, or let the user set it.

Another thing: data types. Use double for coordinates and distances—precision is important. For integers, use int for node indices.

Let's outline steps:

Define constants: N=3000, MAX_COORD=10000 (so points are in 0-10000 range).
Struct for Point with x,y as doubles.
Precompute distance matrix: vector<vector<double>> dist(N, vector<double>(N)).
Generate N random points.
Fill distance matrix using Euclidean distance (sqrt((x_i-x_j)^2 + (y_i-y_j)^2)).
Initial tour: Nearest Neighbor starting at a random node (say, rand()%N).
2-opt optimization with early termination (e.g., stop after 100 iterations without improvement).

Print initial and final tour lengths.
Wait, but for N=3000, the distance matrix is 3000x3000 = 9 million entries. Each double is 8 bytes, so 72 MB—no problem for modern memory.

Now, code structure:

Include necessary headers: <iostream>, <vector>, <cmath>, <cstdlib>, <ctime>, <algorithm> (for reverse), <iomanip> (for setprecision).

Using namespace std to keep it simple (since it's an example).

Functions:

double distance(const Point& a, const Point& b): calculates Euclidean distance.
void nearest_neighbor(const vector<Point>& pts, vector<int>& tour, const vector<vector<double>>& dist): builds initial tour using NN.
bool two_opt(vector<int>& tour, const vector<vector<double>>& dist, double& current_len): performs one 2-opt iteration, returns true if improvement made, updates current_len.
int main(): generates points, precomputes distances, runs NN, then 2-opt, prints results.
Wait, in two_opt, instead of calculating the length every time, we can compute the delta for each swap (d1 - d0) and only reverse if delta < 0. That's more efficient than recalculating the entire length each time. Oh right! The previous 2-opt code calculated d0 and d1 as the sum of two edges, then compared. But to optimize, we can compute the change in length: delta = (dist[i-1][k] + dist[i][k+1]) - (dist[i-1][i] + dist[k][k+1]). If delta < 0, reversing improves the tour by |delta|. This is exactly what the previous code did, but explicit delta is clearer and maybe faster.

Also, in the two_opt loop: the standard 2-opt loops i from 1 to N-2, k from i+1 to N-1 (since tour[N] = tour[0], so k+1 mod N is just k+1 when k < N). Wait, the tour is of size N+1 with tour[N] = tour[0], so for i in 0..N-2? No, usually 2-opt loops i from 1 to N-2 (to avoid adjacent pairs) and k from i+1 to N-1. Let's confirm: the edges are (i-1,i) and (k,k+1). Reversing the subpath from i to k gives edges (i-1,k) and (i,k+1). So yes, i starts at 1 (since i-1 is 0), k starts at i+1 up to N-1 (since k+1 is N, which is tour[0]).

Another optimization: in 2-opt, after each reversal, we can break out of the inner loop and start over with i=1—this is called the "fast 2-opt" heuristic because reversing changes the tour so much that continuing the current i,k loops is useless. The standard approach is to iterate i from 1 to N-2, and for each i, find the best k > i that gives the maximum improvement (not just any improvement). That's called the "greedy 2-opt" and is faster than checking all k for each i. Oh right! Instead of checking all k for each i and reversing whenever an improvement is found (which can lead to many small reversals), we can for each i find the k that gives the largest delta (most negative, i.e., biggest improvement), then reverse that pair. This reduces the number of iterations needed.

So let's adjust the two_opt function to use greedy 2-opt: for each i from 1 to N-2, find the k > i that minimizes (dist[i-1][k] + dist[i][k+1]) - (dist[i-1][i] + dist[k][k+1]). If the minimal delta is negative, reverse i..k and return true (improvement made). This is more efficient than checking all pairs and reversing

immediately.

Yes, that's a better approach for larger N. Because it finds the best possible k for each i in one pass, then does one reversal per outer loop iteration—faster convergence.

So updating the two_opt function: instead of nested loops with reverse on every improvement, we have an outer loop over i, inner loop over k to find best delta for i, if any delta is negative, reverse and return true (since we can restart the outer loop after a big reversal). Wait, no—some implementations do multiple passes until no more improvements. Let's go with:

In each 2-opt iteration:

Initialize best_delta to 0, best_i and best_k to -1.
Loop i from 1 to N-2: a. Loop k from i+1 to N-1: i. Compute delta = (dist[i-1][k] + dist[i][k+1]) - (dist[i-1][i] + dist[k][k+1]). ii. If delta < best_delta (since we want the most negative, i.e., biggest improvement): - Update best_delta to delta. - Set best_i = i, best_k = k.
After checking all i,k: a. If best_delta < -1e-9 (improvement possible): i. Reverse the subpath from best_i to best_k in the tour. ii. Subtract best_delta from the current length (since delta is negative, this reduces the length). iii. Return true (improvement made). b. Else: i. Return false (no improvement).
This way, each iteration of two_opt does one big reversal (the best possible) instead of many small ones—much faster for large N.

Also, precomputing the distance matrix is crucial here—it saves us from calculating sqrt(N²) times (which is slow). So in main:

Generate points.
Fill dist[i][j] = sqrt((pts[i].x - pts[j].x)^2 + (pts[i].y - pts[j].y)^2) for all i,j. But wait, since dist[i][j] = dist[j][i], we can compute only half of it and copy, but for N=3000, even computing all is manageable—9 million operations, each with a few arithmetic steps and one sqrt. It's worth it for the speedup in 2-opt.
Now, initial tour: nearest neighbor. Let's implement that properly:

Start at node s (random). Mark s as visited. Then, for each step from 1 to N-1:

Find the unvisited node j with the smallest dist[current][j]. Add j to tour, mark as visited, set current = j.

Yes—O(N²) time, but for N=3000, that's 9 million operations—manageable once.

Another thing: random seed—use srand(time(0)) to get different points each run.

Output: print initial length, then after each 2-opt iteration (say, every 10 iterations) print the current length, then final length. Don't print the tour since it's 3000 nodes—too long.

Potential issues:

Memory: For N=3000, vector<vector<double>> dist(N, vector<double>(N)) is 3000*3000 = 9e6 doubles—72 MB. No problem.

Time: Nearest neighbor is O(N²) (9e6 ops). 2-opt with greedy approach: each iteration is O(N²), but in practice, it converges quickly (like 100-500 iterations for 3000 nodes, depending on the initial tour). So total time is acceptable for a demo.
Now, code:

First, includes and using namespace:

```
#include <iostream>
#include <vector>
#include <cmath>
#include <cstdlib>
#include <ctime>
#include <algorithm>
#include <iomanip>

using namespace std;
```

Constants:

```
const int N = 3000;
const double MAX_COORD = 10000.0; // Random points in [0, MAX_COORD]
const int NN_STARTS = 5;          // Number of random starts for NN (take best)
const int TWO_OPT_ITERATIONS = 1000; // Max iterations without improvement to
stop
```

Struct Point:

```
struct Point {
double x, y;
};
```

Distance function:

```
inline double distance(const Point& a, const Point& b) {
double dx = a.x - b.x;
double dy = a.y - b.y;
return sqrt(dxdx + dydy);
}
```

Nearest Neighbor heuristic (returns tour length):

```
double nearest_neighbor(const vector<Point>& points, const
vector<vector<double>>& dist_matrix, vector<int>& tour) {
int n = points.size();
vector<bool> visited(n, false);
tour.resize(n + 1); // Tour ends with start node

// Random start node
int current = rand() % n;
tour[0] = current;
visited[current] = true;

double total_len = 0.0;
```

```
for (int i = 1; i < n; ++i) {
    double min_dist = INFINITY;
    int next_node = -1;

    for (int j = 0; j < n; ++j) {
        if (!visited[j] && dist_matrix[current][j] < min_dist) {
            min_dist = dist_matrix[current][j];
            next_node = j;
        }
    }

    tour[i] = next_node;
    visited[next_node] = true;
    total_len += min_dist;
    current = next_node;
}

// Close the tour (add start node at end)
tour[n] = tour[0];
total_len += dist_matrix[tour[n-1]][tour[n]];

return total_len;
}
```

Stochastic Nearest Neighbor: run NN multiple times with random starts, take the best tour:

```
double stochastic_nearest_neighbor(const vector<Point>& points, const
vector<vector<double>>& dist_matrix, vector<int>& best_tour) {
double best_len = INFINITY;
best_tour.resize(N + 1);

for (int start = 0; start < NN_STARTS; ++start) {
    vector<int> tour;
    double len = nearest_neighbor(points, dist_matrix, tour);

    if (len < best_len) {
        best_len = len;
        best_tour.swap(tour); // Efficient swap
    }
}

return best_len;
}
```

2-Opt optimization function: returns new length if improved, else original. Returns true if improvement made.

```
bool two_opt(vector<int>& tour, const vector<vector<double>>& dist_matrix,
double& current_len) {
int n = tour.size() - 1; // Exclude the closing node (tour[n] = tour[0])
bool improved = false;
double best_delta = 0.0;
```

```
int best_i = -1, best_k = -1;

// Find the best i,k pair to reverse
for (int i = 1; i < n - 1; ++i) {
    for (int k = i + 1; k < n; ++k) {
        double delta = (dist_matrix[tour[i-1]][tour[k]] +
dist_matrix[tour[i]][tour[k+1]]) -
                       (dist_matrix[tour[i-1]][tour[i]] +
dist_matrix[tour[k]][tour[k+1]]);

        if (delta < best_delta) {
            best_delta = delta;
            best_i = i;
            best_k = k;
        }
    }
}

// If improvement possible, reverse the subpath
if (best_delta < -1e-9) {
    reverse(tour.begin() + best_i, tour.begin() + best_k + 1);
    current_len += best_delta; // Delta is negative, so length decreases
    improved = true;
}

return improved;
}
```

Main function:

```
int main() {
srand(time(nullptr)); // Seed random number generator

// Step 1: Generate N random points
vector<Point> points(N);
for (int i = 0; i < N; ++i) {
    points[i].x = static_cast<double>(rand()) / RAND_MAX * MAX_COORD;
    points[i].y = static_cast<double>(rand()) / RAND_MAX * MAX_COORD;
}

// Step 2: Precompute distance matrix
vector<vector<double>> dist_matrix(N, vector<double>(N));
for (int i = 0; i < N; ++i) {
    for (int j = 0; j < N; ++j) {
        if (i == j) {
            dist_matrix[i][j] = 0.0;
        } else {
            dist_matrix[i][j] = distance(points[i], points[j]);
        }
    }
}

// Step 3: Generate initial tour using Stochastic Nearest Neighbor
vector<int> best_tour;
```

```
double current_len = stochastic_nearest_neighbor(points, dist_matrix,
best_tour);

cout << "Initial tour length (Stochastic NN): " << fixed << setprecision(2) <<
current_len << endl;

// Step 4: Optimize with 2-Opt
int iterations = 0;
while (iterations < TWO_OPT_ITERATIONS) {
    if (two_opt(best_tour, dist_matrix, current_len)) {
        iterations = 0; // Reset counter if improvement made
        if (iterations % 10 == 0) {
            cout << "Iteration " << iterations << ": Length = " << fixed <<
setprecision(2) << current_len << endl;
        }
    } else {
        iterations++;
    }
}

// Step 5: Output results
cout << "\nFinal tour length (after 2-Opt): " << fixed << setprecision(2) <<
current_len << endl;

return 0;
}
```

Wait, let's check the two_opt function: the tour has size N+1 (since tour[N] = tour[0]). So in two_opt, n is tour.size() - 1 = N. Then i runs from 1 to n-2 (since i < n-1—yes, because i goes up to n-2). k runs from i+1 to n-1 (since k < n—yes). That's correct.

Also, in the stochastic_nearest_neighbor, we use swap to avoid copying the entire tour vector each time—efficient.

The random points: using rand() / RAND_MAX gives a value between 0 and 1, multiplied by MAX_COORD to get points in [0, MAX_COORD]. That's better than modulo because it uses the full range of rand().

In the nearest_neighbor function, after building the tour (size N), we add tour[N] = tour[0] to close the loop, and add the distance between the last node and the start to the total length—correct.

In two_opt, when we reverse from best_i to best_k+1: because tour.begin() + best_i is the iterator to tour[best_i], and tour.begin() + best_k + 1 is one past tour[best_k], so reverse reverses [best_i, best_k]—correct.

Testing for delta < -1e-9 instead of < 0 to avoid floating-point errors (since due to precision, delta might be a tiny negative number that's not actually an improvement).

The TWO_OPT_ITERATIONS constant: if no improvement for 1000 iterations, stop—prevents infinite loops.

Output: prints initial length, then every 10 iterations (wait, in the code above, it says if (iterations % 10 == 0), but iterations is reset to 0 each time an improvement is made. So actually, it will print when iterations is 0, 10, 20, etc.—but since iterations is reset to 0 on improvement, it's printing every 10 improvements. Maybe better to count total iterations instead of resets: let's adjust that.

Ah, right—current code has a bug in the iteration counter. Let's fix that:

In main(), change the two_opt loop to:

```
int total_iterations = 0;
while (total_iterations < TWO_OPT_ITERATIONS) {
if (two_opt(best_tour, dist_matrix, current_len)) {
cout << "Iteration " << total_iterations << ": Length = " << fixed <<
setprecision(2) << current_len << endl;
} else {
total_iterations++;
}
total_iterations++; // Increment every iteration, not just on no improvement
}
```

Wait, no—let's think: we want to run two_opt until either we've done TWO_OPT_ITERATIONS iterations (total) or no improvement for, say, 100 consecutive iterations. Let's adjust the code to track both total iterations and consecutive non-improvements:

```
const int MAX_TOTAL_ITER = 2000;
const int MAX_CONSECUTIVE_NO_IMPROVE = 100;
```

Then in main():

```
int total_iterations = 0;
int consecutive_no_improve = 0;

while (total_iterations < MAX_TOTAL_ITER && consecutive_no_improve <
MAX_CONSECUTIVE_NO_IMPROVE) {
if (two_opt(best_tour, dist_matrix, current_len)) {
consecutive_no_improve = 0;
cout << "Iteration " << total_iterations << ": Length = " << fixed <<
setprecision(2) << current_len << endl;
} else {
consecutive_no_improve++;
}
total_iterations++;
}
```

That's better—stops if either too many total iterations or too many consecutive no improvements.

Yes, that's a more robust termination condition.

Updating the main() code accordingly.

Another thing: in C++, when using vector::swap, it's O(1) because it just swaps the internal pointers—so stochastic_nearest_neighbor uses swap to assign the best tour efficiently.

Also, the Point struct is simple with x and y as doubles—correct.

Potential optimizations:

Use faster random number generation (but for example purposes, rand() is fine).
Parallelize distance matrix computation (but again, example purposes—no need).
Use integer coordinates (if applicable) to avoid sqrt in distance calculation—but the problem says "example", so Euclidean distance is standard.
Testing: when you run this code, it should generate 3000 random points, compute a stochastic NN initial tour, then optimize with 2-opt, printing the length at each iteration until convergence. The final length should be significantly shorter than the initial one (depending on the points).

Yes—this should work.