UPgraded ICPC Notebook

Contents

1 C++

1.1 C++ Template

```
#include <bits/stdc++.h>
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
#pragma GCC optimize("Ofast")
#pragma GCC optimize ("unroll-loops")
#pragma GCC target("sse,sse2,sse3,ssse3,sse4,popcnt,abm,
    mmx,avx,tune=native")
#define ll long long
#define pb push_back
#define ld long double
#define nl '\n'
#define fast cin.tie(0), cout.tie(0), ios_base::
    sync_with_stdio(false)
#define fore(i,a,b) for(ll i=a;i<b;++i)
#define rofe(i,a,b) for(ll i=a-1;i>=b;--i)
#define ALL(u) u.begin(),u.end()
#define vi vector <ll>
#define vvi vector<vi>
#define sz(a) ((ll)a.size())
#define lsb(x) ((x)&(-x))
#define lsbpos(x) __builtin_ffs(x)
#define PI acos(-1.0)
#define pii pair<ll,ll>
#define fst first
#define snd second
#define eb emplace_back
#define ppb pop_back
#define i128 __int128_t

using namespace __gnu_pbds;
using namespace std;

typedef tree<pair<int, int>, null_type, less<pair<int,
    int>>, rb_tree_tag, tree_order_statistics_node_update>
     ordered_multiset;
typedef tree<int,null_type, less<int>, rb_tree_tag,
    tree_order_statistics_node_update> ordered_set;
```

1.2 Bits Manipulation

```
mask |= (1<<n) // prender bit-N
mask ^= (1<<n) // flippear bit-N
mask &= ~(1<<n) // apagar bit-N
if(mask&(1<<n)) // checkar bit-n
T = ((mask)&(-mask)) // LSO
__builtin_ffs(mask); // indice del LSO (indexado en 1)
//  Iterate over the subsets of S.
for(int subset= S; subset; subset= (subset-1) & S)
    for (int subset=0;subset=subset-S&S;) // Increasing
        order
```

1.3 Random

```
// Declare random number generator
mt19937_64 rng(0); // 64 bit, seed = 0
mt19937 rng(chrono::steady_clock::now().time_since_epoch
    ().count()); // 32 bit

// Use it to shuffle a vector
shuffle(all(vec), rng);

// Create int/real uniform dist. of type T in range [l, r
    ]
uniform_int_distribution<T> / uniform_real_distribution<T
    > dis(l, r);
dis(rng); // generate a random number in [l, r]

int rd(int l, int r) { return uniform_int_distribution<
    int>(l, r)(rng);}

srand(time(0)); //include this in main.
```

1.4 Other

```
#pragma GCC optimize("O3")
/*(UNCOMMENT WHEN HAVING LOTS OF RECURSIONS)\
#pragma comment(linker, "/stack:200000000")
(UNCOMMENT WHEN NEEDED) */
```

```
#pragma GCC optimize("Ofast,unroll-loops,no-stack-protector,fast-math")
#pragma GCC target("sse,sse2,sse3,ssse3,sse4,popcnt,abm,mmx,avx,tune=native")

// Custom comparator for set/map
struct comp {
  bool operator()(const double& a, const double& b) const
      {
    return a+EPS<b;}
};
set<double,comp> w; // or map<double,int,comp>

// double inf
const double DINF=numeric_limits<double>::infinity();
```

2 Heuristics

2.1 Ant Colny Optimization

```
//Example with TSP problem.

// Parameters
const int NUM_ANTS = 10;
const int NUM_CITIES = 5;
const int MAX_ITERATIONS = 100;
const double ALPHA = 1.0;  // influence of pheromone
const double BETA = 2.0;   // influence of heuristic (1/distance)
const double EVAPORATION = 0.5;  // pheromone evaporation rate
const double Q = 100.0;     // pheromone deposited

// Distance matrix (example for 5 cities)
vector<vector<double>> distances = {
    {0, 2, 2, 3, 7},
    {2, 0, 4, 3, 6},
    {2, 4, 0, 5, 3},
    {3, 3, 5, 0, 6},
    {7, 6, 3, 6, 0}
};

// Pheromone matrix
vector<vector<double>> pheromones(NUM_CITIES, vector<double>(NUM_CITIES, 1.0));

random_device rd;
mt19937 rng(rd());

// Ant class
struct Ant {
    vi tour;
    double tourLength = 0.0;

    void visitCity(ll city) {
        tour.push_back(city);
    }

    bool visited(ll city) const {
        return find(ALL(tour), city) != tour.end();
    }

    ll lastCity() const {
        return tour.back();
    }

    void calculateTourLength() {
        tourLength = 0.0;
        for (int i = 0; i < sz(tour) - 1; ++i) {
            tourLength += distances[tour[i]][tour[i + 1]];
        }
        tourLength += distances[tour.back()][tour.front()];  // Return to start
    }

    void reset() {
        tour.clear();
        tourLength = 0.0;
    }
};

// Probability function for ant to choose next city
int selectNextCity(const Ant& ant) {
    int currentCity = ant.lastCity();
    vector<double> probabilities(NUM_CITIES, 0.0);

    double sum = 0.0;
    fore(nextCity,0,NUM_CITIES) {
        if (!ant.visited(nextCity)) {
            double pheromone = pow(pheromones[currentCity][nextCity], ALPHA);
            double heuristic = pow(1.0 / distances[currentCity][nextCity], BETA);
            probabilities[nextCity] = pheromone * heuristic;
            sum += probabilities[nextCity];
        }
    }

    // Normalize probabilities
    fore(i,0,NUM_CITIES){
        probabilities[i] /= sum;
    }

    // Roulette wheel selection
    double r = uniform_real_distribution<>(0, 1)(rng);
    double cumulative = 0.0;

    for (int i = 0; i < NUM_CITIES; ++i) {
        if (!ant.visited(i)) {
            cumulative += probabilities[i];
            if (r <= cumulative) {
```

```cpp
                    return i;
                }
            }
        }

        // Fallback (shouldn't happen)
        for (int i = 0; i < NUM_CITIES; ++i) {
            if (!ant.visited(i)) {
                return i;
            }
        }

        return -1;  // Error case (shouldn't reach here)
    }

    // Pheromone update
    void updatePheromones(vector<Ant>& ants) {
        // Evaporation
        fore(i,0,NUM_CITIES){
            fore(j,0,NUM_CITIES){
                pheromones[i][j] *= (1 - EVAPORATION);
            }
        }

        // Deposit pheromones
        for (const auto& ant : ants) {
            double contribution = Q / ant.tourLength;
            for (int i = 0; i < sz(ant.tour) - 1; ++i) {
                int from = ant.tour[i];
                int to = ant.tour[i + 1];
                pheromones[from][to] += contribution;
                pheromones[to][from] += contribution;
            }
            // Complete tour (back to start)
            pheromones[ant.tour.back()][ant.tour.front()] +=
                contribution;
            pheromones[ant.tour.front()][ant.tour.back()] +=
                contribution;
        }
    }

    // Main ACO loop
    void antColonyOptimization() {
        vector<Ant> ants(NUM_ANTS);

        double bestLength = numeric_limits<double>::max();
        vi bestTour;

        for (int iter = 0; iter < MAX_ITERATIONS; ++iter) {
            // Reset ants
            for (auto& ant : ants) {
                ant.reset();
                int startCity = uniform_int_distribution<>(0,
                    NUM_CITIES - 1)(rng);
                ant.visitCity(startCity);
            }

            // Build tours
            for (int step = 1; step < NUM_CITIES; ++step) {
                for (auto& ant : ants) {
                    int nextCity = selectNextCity(ant);
                    ant.visitCity(nextCity);
                }
            }

            // Calculate tour lengths
            for (auto& ant : ants) {
                ant.calculateTourLength();
                if (ant.tourLength < bestLength) {
                    bestLength = ant.tourLength;
                    bestTour = ant.tour;
                }
            }

            // Update pheromones
            updatePheromones(ants);
        }

        // Output best tour found
        cout << "Best Tour Length: " << bestLength << nl;
        cout << "Best Tour: ";
        for (int city : bestTour) {
            cout << city << " ";
        }
        cout << nl;
    }
```

2.2 Genetic Algorithm

```cpp
// Random number generator
mt19937 rng(time(0));

// Parameters
const int POPULATION_SIZE = 100;
const double MUTATION_RATE = 0.1;
const int GENERATIONS = 100;

// Chromosome type (can be adjusted for different
    problems)
using Chromosome = vi;

// Problem-specific settings (binary string length)
const int CHROMOSOME_LENGTH = 20;

// Function to create a random chromosome
Chromosome randomChromosome() {
    Chromosome chrom(CHROMOSOME_LENGTH);
    for (ll &gene : chrom) {
        gene = rng() % 2;  // Binary chromosome (0 or 1)
    }
    return chrom;
}

// Example fitness function (number of 1s in the
    chromosome)
```

```
ld fitness(const Chromosome &chrom) {
    return std::count(ALL(chrom), 1);
}

// Selection (tournament selection)
Chromosome tournamentSelection(const std::vector<
   Chromosome> &population) {
    int bestIdx = rng() % POPULATION_SIZE;
    fore(i,0,3) {
        int idx = rng() % POPULATION_SIZE;
        if (fitness(population[idx]) > fitness(population
           [bestIdx])) {
            bestIdx = idx;
        }
    }
    return population[bestIdx];
}

// Crossover (single point crossover)
Chromosome crossover(const Chromosome &parent1, const
   Chromosome &parent2) {
    ll point = rng() % CHROMOSOME_LENGTH;
    Chromosome offspring = parent1;
    for (int i = point; i < CHROMOSOME_LENGTH; i++) {
        offspring[i] = parent2[i];
    }
    return offspring;
}

// Mutation (flip bits)
void mutate(Chromosome &chrom) {
    for (ll &gene : chrom) {
        if ((rng() % 100) < (MUTATION_RATE * 100)) {
            gene = 1 - gene;
        }
    }
}

// Genetic Algorithm main loop
void geneticAlgorithm() {
    // Initialize population
    vector<Chromosome> population(POPULATION_SIZE);
    for (Chromosome &chrom : population) {
        chrom = randomChromosome();
    }

    ld bestFitness;
    Chromosome bestChromosome;

    for (int gen = 0; gen < GENERATIONS; gen++) {
        vector<Chromosome> newPopulation;

        for (int i = 0; i < POPULATION_SIZE; i++) {
            Chromosome parent1 = tournamentSelection(
               population);
            Chromosome parent2 = tournamentSelection(
               population);
            Chromosome offspring = crossover(parent1,
               parent2);
            mutate(offspring);

            newPopulation.push_back(offspring);
        }

        population = newPopulation;

        for (const Chromosome &chrom : population) {
            ld this_fitness = fitness(chrom);
            if (bestFitness < this_fitness){
                bestFitness = this_fitness;
                bestChromosome = chrom;
            }
        }
        std::cout << "Generation " << gen << " - Best
           fitness: " << bestFitness << "\n";
    }

    // Print the best chromosome (best solution) and its
       fitness.
}

// Main
int main() {
    geneticAlgorithm();
    return 0;
}
```

2.3 GRASP

```
// Example structure for a generic GRASP
struct Solution {
    vi elements;
    double cost;

    bool operator<(const Solution& other) const {
        return cost < other.cost;
    }
};

// Random number generator
std::mt19937 rng(std::random_device{}());

// Function to evaluate solution cost (problem-specific)
double evaluate(const Solution& solution) {
    // Replace with actual evaluation logic
    return 1.0;
}

// Greedy randomized construction
Solution greedyRandomizedConstruction(const vi&
   candidates, double alpha) {
    Solution solution;
```

```cpp
    vi available = candidates;

    while (sz(available)) {
        // Evaluate candidate costs
        vector<pair<int, double>> candidateCosts;
        for (int c : available) {
            // In a real problem, you would evaluate
                adding 'c' to the solution
            double cost = static_cast<double>(c);  //
                Example cost (problem-specific)
            candidateCosts.emplace_back(c, cost);
        }

        // Sort candidates by cost
        sort(ALL(candidateCosts), [](const auto& a, const
             auto& b) { return a.snd < b.snd; });

        // Restricted Candidate List (RCL)
        double minCost = candidateCosts.front().snd;
        double maxCost = candidateCosts.back().snd;
        double threshold = minCost + alpha * (maxCost -
            minCost);

        vi RCL;
        for (const auto& [candidate, cost] :
            candidateCosts) {
            if (cost <= threshold) {
                RCL.push_back(candidate);
            }
        }

        // Select a random candidate from the RCL
        uniform_int_distribution<int> dist(0, sz(RCL) -
            1);
        int selected = RCL[dist(rng)];

        // Add to solution
        solution.elements.push_back(selected);

        // Remove selected from available
        available.erase(remove(ALL(available), selected),
             available.end());
    }

    solution.cost = evaluate(solution);
    return solution;
}

// Local search (problem-specific)
Solution localSearch(const Solution& initial) {
    Solution best = initial;

    // Example: Simple swap local search (problem-
        dependent)
    for (int i = 0; i < sz(initial.elements); ++i) {
        for (int j = i + 1; j < sz(initial.elements); ++j
            ) {
            Solution neighbor = initial;
            swap(neighbor.elements[i], neighbor.elements[
                j]);
            neighbor.cost = evaluate(neighbor);

            if (neighbor.cost < best.cost) {
                best = neighbor;
            }
        }
    }

    return best;
}

// GRASP algorithm
Solution grasp(const vi& candidates, int maxIterations,
    double alpha) {
    Solution best;
    best.cost = numeric_limits<double>::infinity();

    for (int iteration = 0; iteration < maxIterations; ++
        iteration) {
        Solution constructed =
            greedyRandomizedConstruction(candidates, alpha
            );
        Solution improved = localSearch(constructed);

        if (improved.cost < best.cost) {
            best = improved;
        }
    }

    return best;
}

// Example problem
int main() {
    vi candidates = {1, 2, 3, 4, 5, 6, 7, 8, 9};

    int maxIterations = 100;
    double alpha = 0.3; // Controls greediness/randomness
         balance

    Solution best = grasp(candidates, maxIterations,
        alpha);

    cout << "Best cost: " << best.cost << "\nElements: ";
    for (int e : best.elements) {
        std::cout << e << " ";
    }
    cout << "\n";

    return 0;
}
```

2.4 Simulated Annealing

```cpp
// Objective function - you can change this to your problem
ld objective(ld x) {
    return x * x;  // Minimize x^2 (parabola)
}

// Generates a neighbor (for continuous space problems)
ld random_neighbor(ld x, ld step_size) {
    return x + ((rand() / (ld)RAND_MAX) * 2 - 1) * step_size;  // Move x left or right
}

// Cooling schedule
ld reduce_temperature(ld T, ld alpha = 0.99) {
    return T * alpha;  // Geometric cooling
}

// Simulated Annealing
ld simulated_annealing(ld initial_x, ld initial_temperature, ld final_temperature) {
    ld x = initial_x;
    ld best_x = x;
    ld T = initial_temperature;

    while (T > final_temperature) {
        ld next_x = random_neighbor(x, 1.0);  // step size 1.0
        ld delta = objective(next_x) - objective(x);

        if (delta < 0 || exp(-delta / T) > ((ld)rand() / RAND_MAX)) {
            x = next_x;
        }

        if (objective(x) < objective(best_x)) {
            best_x = x;
        }

        T = reduce_temperature(T);  // Cool down
    }
    return best_x;
}

/*
objective(x) is the function we want to minimize.
random_neighbor() gives a new candidate state near the current state.
exp(-delta / T) gives the probability of accepting worse solutions.
reduce_temperature() gradually cools the system.
*/
```

2.5 Tabu Search

```cpp
// Example using the TSP problem.

// Random number generator
mt19937 rng(random_device{}());

// Example distance matrix for TSP
const int N = 5;
int dist[N][N] = {
    {0, 10, 15, 20, 10},
    {10, 0, 35, 25, 15},
    {15, 35, 0, 30, 20},
    {20, 25, 30, 0, 25},
    {10, 15, 20, 25, 0}
};

// Evaluate the cost of a tour
int evaluate(const vi& tour) {
    int cost = 0;
    fore(i,0,N){
        cost += dist[tour[i]][tour[(i + 1) % N]];
    }
    return cost;
}

// Generate a random initial tour
vi randomTour() {
    vi tour(N);
    for (int i = 0; i < N; i++) tour[i] = i;
    shuffle(ALL(tour), rng);
    return tour;
}

// Tabu Search for TSP
vi tabuSearch(int maxIterations, int tabuTenure){ // amount of iterations, size of the tabu list.
    vi bestTour = randomTour();
    int bestCost = evaluate(bestTour);
    vi currentTour = bestTour;
    int currentCost = bestCost;

    // Tabu list stores pairs of cities that should not be swapped
    set<pii> tabuList;

    for (int iter = 0; iter < maxIterations; iter++) {
        vi bestNeighbor = currentTour;
        int bestNeighborCost = numeric_limits<int>::max();
        pii bestMove = {-1, -1};

        // Explore all pairwise swaps (2-opt neighborhood)
        for (int i = 0; i < N - 1; i++) {
            for (int j = i + 1; j < N; j++) {
                vi neighbor = currentTour;
                swap(neighbor[i], neighbor[j]);
                int neighborCost = evaluate(neighbor);

                pii move = {min(neighbor[i], neighbor[j]), max(neighbor[i], neighbor[j])};
```

```cpp
            // Check if move is tabu
            bool isTabu = tabuList.count(move) > 0;

            // Aspiration criterion: accept move if
                it's better than global best
            if (neighborCost < bestNeighborCost && (!
                isTabu || (neighborCost < bestCost)))
                {
                bestNeighbor = neighbor;
                bestNeighborCost = neighborCost;
                bestMove = move;
            }
        }
    }

    // Update current tour
    currentTour = bestNeighbor;
    currentCost = bestNeighborCost;

    // Update global best if improved
    if (currentCost < bestCost) {
        bestCost = currentCost;
        bestTour = currentTour;
    }

        // Update Tabu List: add move and enforce tabu
            tenure (recent moves are tabu for 'tabuTenure'
             iterations)
        tabuList.insert(bestMove);
        if (sz(tabuList) > tabuTenure) {
            tabuList.erase(tabuList.begin());  // Simple
                FIFO expiration policy
        }

        cout << "Iteration " << iter + 1 << ": Best cost
            so far = " << bestCost << endl;
    }

    return bestTour;
}

/*
It used to solve optimization problems, particularly
    combinatorial problems like the
Traveling Salesman Problem (TSP) or scheduling problems.
*/
```