```cpp
#ifndef _TP1_PROBLEM_H_
#define _TP1_PROBLEM_H_ 1

#include <list>
#include <algorithm>
#include <map>
#include <cmath>
#include "Matrix.h"

enum Method {
    BAND_GAUSSIAN_ELIMINATION,
    LU_FACTORIZATION,
    SIMPLE_ALGORITHM,
    SHERMAN_MORRISON
};

typedef struct _Leech {
public:
    BDouble x;
    BDouble y;
    BDouble radio;
    BDouble temperature;
} Leech;

class Problem {
public:
    // Invariante:
    // h /= 0
    // height /= 0
    // width /= 0
    // h | height
    // h | width
    Problem(enum Method method,
            const BDouble &width,
            const BDouble &height,
            const BDouble &h,
            std::list<Leech> &leeches)
            : width(width), height(height), h(h), rows(round(height / h) + 1), columns(round(width / h) + 1),
              dims(rows * columns), leeches(leeches), method(method) {
        std::cerr << "Method: " << this->method << std::endl;
        std::cerr << "Width: " << this->width << std::endl;
        std::cerr << "Height: " << this->height << std::endl;
        std::cerr << "h: " << this->h << std::endl;
        std::cerr << "Discretization rows: " << this->rows << std::endl;
        std::cerr << "Discretization columns: " << this->columns << std::endl;
        std::cerr << "Total dimensions: " << this->dims << std::endl;
        std::cerr << "Leeches: " << this->leeches.size() << std::endl;
    }

    Matrix run() {
        Matrix system(this->dims, dims, this->columns, this->columns);
        BDouble *b = new BDouble[this->dims];
        Matrix temperatures(this->rows, this->columns);

        switch (method) {
            case BAND_GAUSSIAN_ELIMINATION:
                band_gaussian_elimination(system, b, temperatures);
                break;
            case LU_FACTORIZATION:
                lu_factorization(system, b, temperatures);
                break;
            case SIMPLE_ALGORITHM:
                simple_algorithm(system, b, temperatures);
                std::cout << "SLP: " << (double)singular_leeches_count()/(double)this->leeches.size() << std::endl;
                break;
            case SHERMAN_MORRISON:
                sherman_morrison_solution(system, b, temperatures);
                std::cout << "SLP: " << (double)singular_leeches_count()/(double)this->leeches.size() << std::endl;
                break;
        }

        delete[] b;
        return temperatures;
    }

private:
    void load_temperature_matrix(BDouble *x, Matrix &temperatures) {
        // Cargar los datos en la matriz
        for (int i = 0; i < temperatures.rows(); ++i) {
            for (int j = 0; j < temperatures.columns(); ++j) {
                temperatures(i, j) = x[(i * temperatures.columns()) + j];
            }
        }
    }

    void band_gaussian_elimination(Matrix &system, BDouble *b, Matrix &temperatures) {
        build_system(system, b, this->leeches);
        std::pair<BDouble *, enum Solutions> solution = gaussian_elimination(system, b);

        std::cout << "CPT: " << critic_point_temperature(system, solution.first) << std::endl;
        load_temperature_matrix(solution.first, temperatures);
        delete[] solution.first;
    }

    std::pair<BDouble *, enum Solutions> lu_resolution(Matrix &L, Matrix &U, BDouble *b) {
        //Resolvemos el sistema Ly = b
        std::pair<BDouble *, enum Solutions> partialSolution = forward_substitution(L, b);
        //Resolvemos el sistema Ux = y
        std::pair<BDouble *, enum Solutions> finalSolution = backward_substitution(U, partialSolution.first);

        delete[] partialSolution.first;
        return finalSolution;

    }

    void lu_factorization(Matrix &A, BDouble *b, Matrix &temperatures) {
        build_system(A, b, this->leeches);

        // Sea A la matriz del sistema de ecuaciones,
        // factorizamos A = LU con L, U triangulares inferior/superior
        std::pair<Matrix, Matrix> factors = LU_factorization(A);
        std::pair<BDouble *, enum Solutions> finalSolution = lu_resolution(factors.first, factors.second, b);

        std::cout << "CPT: " << critic_point_temperature(A, finalSolution.first) << std::endl;
        //Cargamos la solucion en la matriz de temperaturas
        load_temperature_matrix(finalSolution.first, temperatures);
        // Liberamos la memoria que usamos.
        delete[] finalSolution.first;
    }

    /**
    * Resuelve el problema por eliminacion gaussiana. En caso de que la temperatura del
    * punto critico sea mayor o igual a 235.0 grados de temperatura, resuelve el sistema
    * por cada sanguijuela, removiendo una de estas y se queda con la menor temperatura.
    **/
    void simple_algorithm(Matrix &system, BDouble *b, Matrix &temperatures) {
        // Observamos que sucede con el caso que no borramos sanguijuelas
        build_system(system, b, this->leeches);
        std::pair<BDouble *, enum Solutions> solution = gaussian_elimination(system, b);

        // Si no hace falta borrar ninguna, terminamos antes.
        BDouble minT = critic_point_temperature(system, solution.first);

        if (minT < 235.0) {
            std::cout << "CPT: " << minT << std::endl;
            std::cout << "REMOVED_LEECH: -1" << std::endl;
            load_temperature_matrix(solution.first, temperatures);
            delete[] solution.first;
            return;
        }

        delete[] solution.first;

        // Valores de salida por defecto
        BDouble *minX = NULL;
        minT = 0.0;
        long taken = -1;

        for (std::list<Leech>::iterator itLeech = leeches.begin(); itLeech != leeches.end(); ++itLeech) {
            //Armamos una lista sin la sanguijuela
            std::list<Leech> curLeeches(leeches);
            auto curLeechesIterator = curLeeches.begin();
            std::advance(curLeechesIterator, std::distance(leeches.begin(), itLeech));
            curLeeches.erase(curLeechesIterator);

            //Inicializamos el sistema sin la sanguijuela
            //Matrix curSystem(system.rows(), system.columns(), system.lower_bandwidth(), system.upper_bandwidth());
            BDouble *curB = new BDouble[system.columns()];
            clean_system(system);
            build_system(system, curB, curLeeches);

            //Resolvemos el sistema
            std::pair<BDouble *, enum Solutions> curSolution = gaussian_elimination(system, curB);

            BDouble curT = critic_point_temperature(system, curSolution.first);

            std::cerr << "Removing leech (" << itLeech->x << ", " << itLeech->y << ", " << itLeech->radio << ", " <<
            itLeech->temperature << ") gives a critic point temperature of " << curT << std::endl;

            // Nos quedamos con la solucion si es mejor que la anterior
            delete[] curB;

            if (minX == NULL || curT <= minT) {
                if (minX != NULL) {
                    delete[] minX;
                }

                minX = curSolution.first;
                minT = curT;
                taken = std::distance(leeches.begin(), itLeech);
            } else {
                delete[] curSolution.first;
            }
        }

        // Critic point temperature
        std::cout << "REMOVED_LEECH: " << taken << std::endl;
        std::cout << "CPT: " << minT << std::endl;
        load_temperature_matrix(minX, temperatures);

        delete[] minX;
    }

    int singular_leeches_count() {
        int singular_count = 0;

        for (std::list<Leech>::iterator itLeech = leeches.begin(); itLeech != leeches.end(); ++itLeech) {
            Leech leech = *itLeech;

            if (is_singular_leech(leech)) {
                singular_count++;
            }
        }
        return singular_count;
    }

    std::pair<BDouble *, enum Solutions> singular_leech_resolution(Matrix &system, Matrix &L, Matrix &U, BDouble *b,
                                                                   std::list<Leech> &leeches, const Leech &removed_leech) {
        //Nos fijamos si otra sanguijuela afecta la posicion de esta
        int i = round(removed_leech.y / h);
        int j = round(removed_leech.x / h);

        //Tratamiento para sanguijuelas singulares (afectan una sola ecuacion)
        std::map<std::pair<int, int>, BDouble> affected_positions = generate_affected_positions(leeches);
        bool affected_position = affected_positions.count(std::pair<int, int>(i, j)) >= 1;

        std::pair<BDouble *, enum Solutions> solution;

        if (affected_position) {
            //Otra sanguijuela afecta la posicion => No podemos aprovechar sherman-morrison.
            //Utilizamos unicamente la factorizacion LU.
            BDouble newTemperature = affected_positions.at(std::pair<int, int>(i, j));

            // Inicializamos la solucion del sistema
            BDouble *b2 = new BDouble[system.columns()];
            std::copy(b, b + system.columns(), b2);
            b2[(i * this->columns) + j] = newTemperature;

            // Resolvemos utilizando LU
            solution = lu_resolution(L, U, b2);
            delete[] b2;

        } else {
            //Podemos aprovechar sherman-morrison!!
            std::pair<BDouble *, BDouble *> uv = generate_sherman_morrison_uv(system, i, j);
            BDouble *u = uv.first;
            BDouble *v = uv.second;
            BDouble *b2 = generate_sherman_morrison_b(system, b, i, j);

            //Resolvemos utilizando sherman-morrison
            solution = sherman_morrison(L, U, u, v, b2);

            //Liberamos memoria
            delete[] u;
            delete[] v;
            delete[] b2;

        }
        return solution;

    }


    /**
    * En caso de que la cantidad de sanguijuelas singulares (afectan una sola ecuacion del sistema discretizado)
    * sea menor o igual a 1 resuelve el problema usando simple_algorithm.
    * En caso contrario obtiene la factorizacion LU del sistema y separa el tratamiento de sanguijuelas normales
    * de las sanguijuelas singulares.
    * - Si la sanguijuela no es singular resuelve rehaciendo el sistema sin la sanguijuela como en simple_algorithm.
    * - Si la sanguijuela es singular a su vez separa en dos casos:
    *     - Si la posicion se encuentra afectada por otra sanguijuela, simplemente modifica el valor del vector
    *    correspondiente por el de mayor temperatura y resuelve utilizando la factorizacion LU.
    *    - Si la posicion no se encuentra afectada por otra sanguijuela, resuelve utilizando
    **/
    void sherman_morrison_solution(Matrix &system, BDouble *b, Matrix &temperatures) {

        if (singular_leeches_count() < 2) {
            std::cerr << "Haciendo fallback al algoritmo simple" << std::endl;

            // Si la cantidad de sanguijuelas singulares es menor es 0 o 1
            // no tiene sentido obtener la factorizacion LU de la matriz.
            // Basta con utilizar la version simple del metodo
            simple_algorithm(system, b, temperatures);
            return;
        }

        std::list<Leech> singularLeeches;
        build_system(system, b, this->leeches);

        //Calculamos la factorizacion LU para aprovechar en las sanguijuelas singulares
        std::pair<Matrix, Matrix> factors = LU_factorization(system);
        Matrix &L = factors.first;
        Matrix &U = factors.second;

        //Solucion sin sacar sanguijuela
        std::pair<BDouble *, enum Solutions> solution = lu_resolution(L, U, b);

        // Si no hace falta borrar ninguna, terminamos antes.
        BDouble minT = critic_point_temperature(system, solution.first);

        if (minT < 235.0) {
            std::cout << "CPT: " << minT << std::endl;
            std::cout << "REMOVED_LEECH: -1" << std::endl;
            load_temperature_matrix(solution.first, temperatures);
            delete[] solution.first;
            return;
        }

        delete[] solution.first;

        long taken = -1;
        BDouble *minX = NULL;
        minT = 0.0;

        for (std::list<Leech>::iterator itLeech = leeches.begin(); itLeech != leeches.end(); ++itLeech) {
            //Armamos una lista sin la sanguijuela
            std::list<Leech> curLeeches(leeches);
            auto curLeechesIterator = curLeeches.begin();
            std::advance(curLeechesIterator, std::distance(leeches.begin(), itLeech));
            curLeeches.erase(curLeechesIterator);

            if (is_singular_leech(*itLeech)) {
                //Tratamos a las sanguijuelas singulares aparte
                solution = singular_leech_resolution(system, L, U, b, curLeeches, *itLeech);
            } else {
                //Inicializamos el sistema sin la sanguijuela
                BDouble *curB = new BDouble[system.columns()];
                clean_system(system);
                build_system(system, curB, curLeeches);

                // Resolvemos el sistema por eliminación gaussiana
                solution = gaussian_elimination(system, curB);

                //Liberamos memoria
                delete[] curB;
            }

            BDouble curT = critic_point_temperature(system, solution.first);

            std::cerr << "Removing leech (" << itLeech->x << ", " << itLeech->y << ", " << itLeech->radio << ", " <<
            itLeech->temperature << ") gives a critic point temperature of " << curT << std::endl;

            if (curT <= minT || minX == NULL) {
                if (minX != NULL) {
                    delete[] minX;
                }

                minX = solution.first;
                minT = curT;
                taken = std::distance(leeches.begin(), itLeech);
            } else {
                delete[] solution.first;
            }
        }

        // Critic point temperature
        std::cout << "REMOVED_LEECH: " << taken << std::endl;
        std::cout << "CPT: " << minT << std::endl;

        load_temperature_matrix(minX, temperatures);
        delete[] minX;
```

```cpp
    }

    /**
    * Devuelve true si la sanguijuela solo afecta una ecuacion del sistema.
    */
    bool is_singular_leech(Leech leech) {
        // Ponemos el rango que vamos a chequear
        BDouble topJ = std::min(leech.x + leech.radio, this->width - this->h) / h;
        BDouble bottomJ = std::max(leech.x - leech.radio, this->h) / h;
        BDouble topI = std::min(leech.y + leech.radio, this->height - this->h) / h;
        BDouble bottomI = std::max(leech.y - leech.radio, this->h) / h;

        int coordinates_count = 0;
        for (int i = std::ceil(bottomI); BDouble(double(i)) <= topI; ++i) {
            BDouble iA = BDouble(double(i));

            for (int j = std::ceil(bottomJ); BDouble(double(j)) <= topJ; ++j) {
                BDouble iJ = BDouble(double(j));
                BDouble coef = std::pow(iA * this->h - leech.y, 2) + std::pow(iJ * this->h - leech.x, 2);

                if (coef <= std::pow(leech.radio, 2)) {
                    coordinates_count++;
                }
            }
        }
        return coordinates_count == 1;
    }


    BDouble *generate_sherman_morrison_b(const Matrix &system, BDouble *b, int leech_y, int leech_x) {
        int columns = system.upper_bandwidth();
        BDouble *b2 = new BDouble[system.columns()];
        std::copy(b, b + system.columns(), b2);
        //std::cerr << "b2[" << (leech_y * columns) + leech_x << "] = " << b2[(leech_y * columns) + leech_x] << std::endl;
        b2[(leech_y * columns) + leech_x] = 0.0;
        return b2;
    }


    std::pair<BDouble *, BDouble *> generate_sherman_morrison_uv(const Matrix &system, int leech_y, int leech_x) {
        //Construimos el vector columna con un vector canonico
        //especificando la fila que corresponde a la ecuacion
        //donde hay una sanguijuela
        BDouble *u = new BDouble[system.rows()];

        for (int ijEq = 0; ijEq < this->dims; ijEq++) {
            int i = ijEq / this->columns;
            int j = ijEq % this->columns;

            if (i == leech_y && j == leech_x) {
                u[ijEq] = 1.0;
            } else {
                u[ijEq] = 0.0;
            }
        }

        //Armamos el vector fila con un vector especificando
        //las columnas donde colocaremos las componentes
        //que corresponden a las diferencias finitas
        BDouble *v = new BDouble[system.rows()];

        for (int ijEq = 0; ijEq < this->dims; ijEq++) {
            v[ijEq] = 0.0;
        }

        int i = leech_y;
        int j = leech_x;

        if (j-1 >= 0) {
            v[(i * this->columns) + j - 1] = -0.25;
        }

        if (j+1 < this->columns) {
            v[(i * this->columns) + j + 1] = -0.25;
        }

        if (i-1 >= 0) {
            v[((i - 1) * this->columns) + j] = -0.25;
        }

        if (i+1 < this->rows) {
            v[((i + 1) * this->columns) + j] = -0.25;
        }

        return std::pair<BDouble *, BDouble *>(u, v);

    }

    double critic_point_temperature(const Matrix &system, BDouble *solution) {
        double centerJ = this->width / 2.0;
        double centerI = this->height / 2.0;

        double topJ = std::min(centerJ / double(this->h) + 1.0, double(this->width) / double(this->h) - 1.0);
        double bottomJ = std::max(centerJ / double(this->h) - 1, 1.0);

        double topI = std::min(centerI / double(this->h) + 1.0, double(this->height) / double(this->h) - 1.0);
        double bottomI = std::max(centerI / double(this->h) - 1.0, 1.0);

        double output = 0.0;
        double k = 0;

        for (int i = std::ceil(bottomI); i <= std::floor(topI); ++i) {
            for (int j = std::ceil(bottomJ); j <= std::floor(topJ); ++j) {
                output += solution[i * this->columns + j];
                ++k;
            }
        }

        output /= k;

        return output;
    }

    void clean_system(Matrix &system) {
        for (int ijEq = 0; ijEq < this->dims; ijEq++) {
            system(ijEq, ijEq) = 0.0;

            int bound = std::min(system.upper_bandwidth(), system.lower_bandwidth());

            for (int l = 1; l <= bound; l++) {
                if (ijEq > l) {
                    system(ijEq, ijEq - l) = 0.0;
                }

                if (ijEq + l < this->dims) {
                    system(ijEq, ijEq + l) = 0.0;
                }
            }
        }
    }


    std::map<std::pair<int, int>, BDouble> generate_affected_positions(const std::list<Leech> &leeches) const {
        std::map<std::pair<int, int>, BDouble> associations;

        for (auto &leech : leeches) {
            // Ponemos el rango que vamos a generar
            BDouble topJ = std::min(leech.x + leech.radio, this->width - this->h) / h;
            BDouble bottomJ = std::max(leech.x - leech.radio, this->h) / h;

            BDouble topI = std::min(leech.y + leech.radio, this->height - this->h) / h;
            BDouble bottomI = std::max(leech.y - leech.radio, this->h) / h;

            // Seteamos las temperaturas en la matriz.
            // Cabe destacar, la temperatura de cada sanguijuela es igual para todos los puntos que cubre.
            for (int i = std::ceil(bottomI); BDouble(double(i)) <= topI; ++i) {
                BDouble iA = BDouble(double(i));

                for (int j = std::ceil(bottomJ); BDouble(double(j)) <= topJ; ++j) {
                    BDouble iJ = BDouble(double(j));
                    BDouble coef = std::pow(iA * this->h - leech.y, 2) + std::pow(iJ * this->h - leech.x, 2);

                    if (coef <= std::pow(leech.radio, 2)) {
                        try {
                            if (associations.at(std::pair<int, int>(i, j)) < leech.temperature) {
                                associations[std::pair<int, int>(i, j)] = leech.temperature;
                            }
                        } catch (...) {
                            associations[std::pair<int, int>(i, j)] = leech.temperature;
                        }
                    }
                }
            }
        }

        return associations;
    }

    /**
     * Construimos:
     * - system, la matriz de ecuaciones que representa la relación de las temperaturas.
     * - b, el vector de resultados que representa las condiciones del sistema.
     * */
    void build_system(Matrix &system, BDouble *b, const std::list<Leech> &leeches) const {
        int columns = system.upper_bandwidth();
        int rows = system.rows() / columns;
        int limit = columns * rows;

        std::map<std::pair<int, int>, BDouble> associations = generate_affected_positions(leeches);

        for (int ijEq = 0; ijEq < limit; ijEq++) {
            system(ijEq, ijEq) = 1.0;
            int i = ijEq / columns;
            int j = ijEq % columns;

            if (i == 0 || j == 0 || i == rows - 1 || j == columns - 1) {
                //Si esta en el borde el valor esta fijo en -100.0 y no hay que usar
                //la ecuacion de laplace
                b[ijEq] = -100.0;

            } else {
                try {
                    //Si la posicion se encuentra en el radio de una sanguijuela
                    //la temperatura que afecta la posicion es la de la sanguijuela
                    //y no hay que usar la ecuacion de laplace
                    b[ijEq] = associations.at(std::pair<int, int>(i, j));

                } catch (...) {
                    //Finalmente si no es borde ni sanguijuela, hay que usar la
                    //ecuacion de laplace.
                    //Las posiciones de los bordes se ignoran porque figuran con -100.0
                    //y fija el valor.
                    b[ijEq] = 0.0;

                    // t[i-1][j] + t[i, j-1] - 4*t[i, j] + t[i+1, j] + t[i, j+1] = 0
                    // - t[i-1][j] - t[i, j-1] - t[i+1, j] - t[i, j+1] = 0 con t[i, j] = 0
                    system(ijEq, (i * columns) + j - 1) = -0.25;
                    system(ijEq, (i * columns) + j + 1) = -0.25;
                    system(ijEq, ((i - 1) * columns) + j) = -0.25;
                    system(ijEq, ((i + 1) * columns) + j) = -0.25;
                }
            }
        }
    }

    BDouble width;
    BDouble height;
    BDouble h;
    int rows;
    int columns;
    int dims;
    std::list<Leech> leeches;
    enum Method method;
};


#endif //_TP1_PROBLEM_H_
```