Trabajo Práctico Nro 1
Inteligencia Artificial

23 de enero de 2008

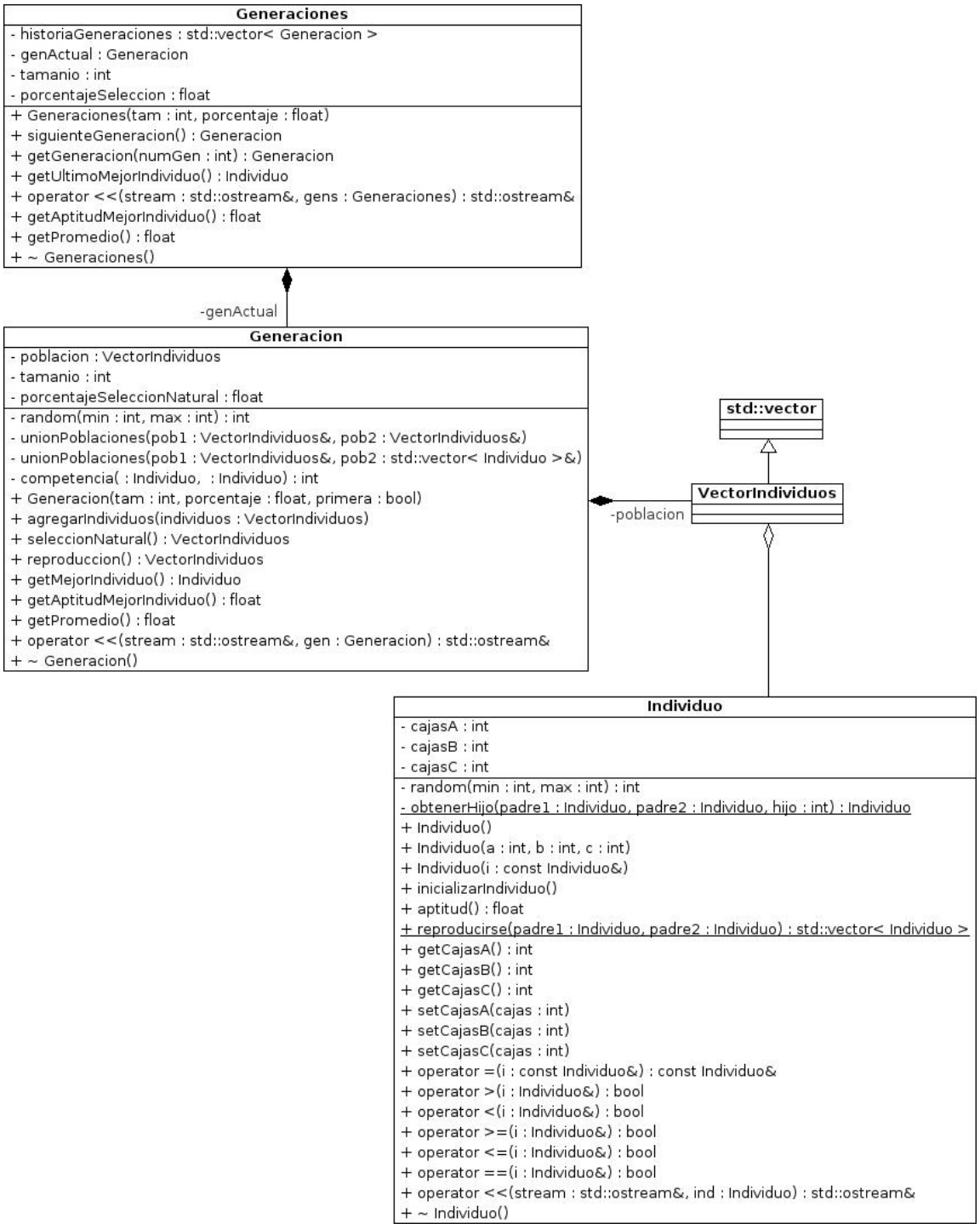
Índice

1. Enunciado:

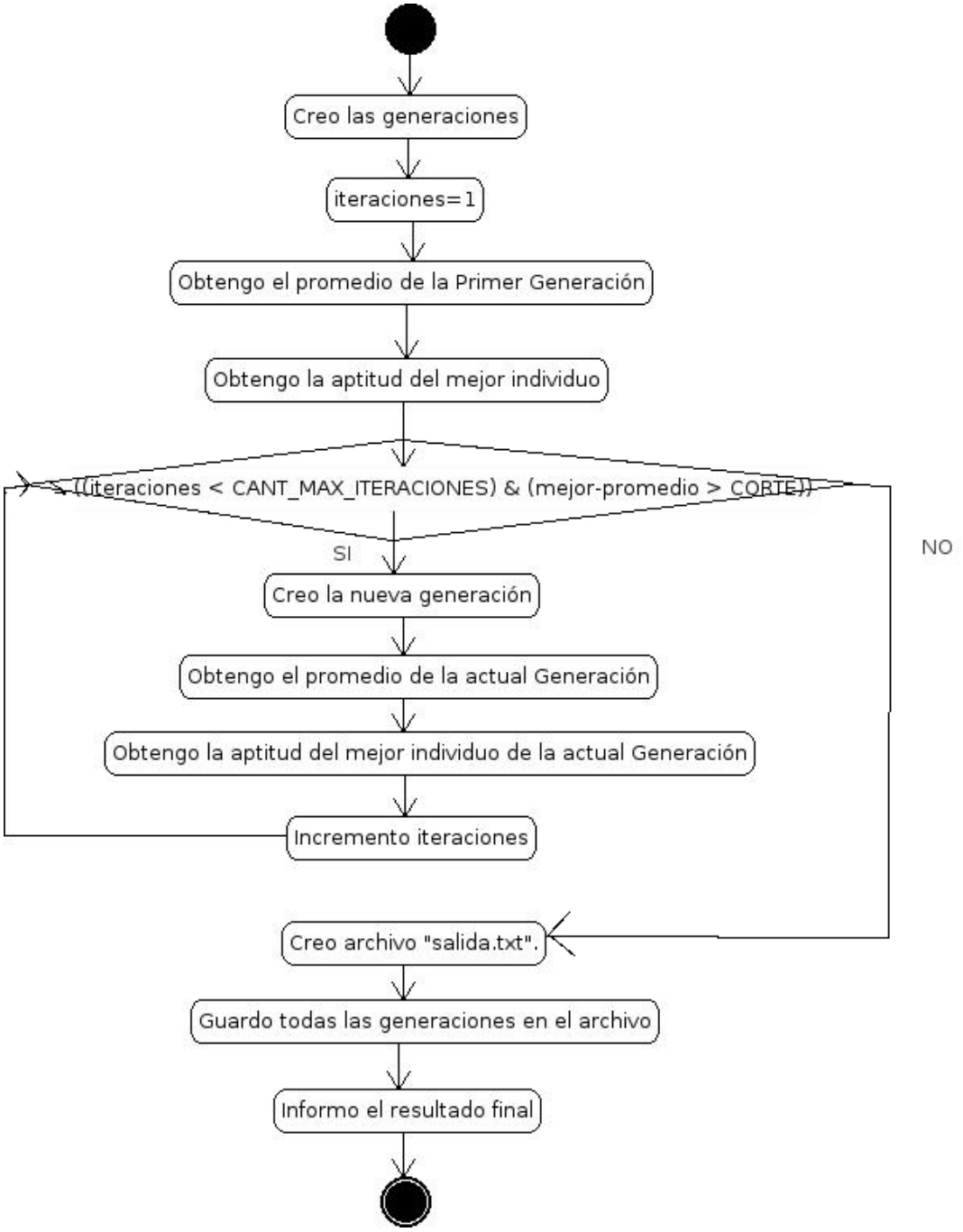
Realizar un programa que, utilizando algoritmos genéticos, resuelva el siguiente problema.
Se tiene un contenedor con un cierto ancho, alto y largo (parámetros del problema).
Se tienen 3 tipos de cajas, cada una con sus dimensiones (ancho, alto y largo, parámetros del problema), y una cantidad máxima de cada una de ellas.
El objetivo consiste en ubicar las cajas en el contenedor, minimizando el espacio libre en el mismo y maximizando la cantidad de cajas que se colocan (suma de las cantidades de cada tipo). Existe una cantidad mínima de cajas de cada tipo a colocar en el contenedor (parámetros del problema).
Se debe aplicar Selección, Reproducción y opcionalmente Mutación.
El TP se puede realizar en cualquier lenguaje.
Se debe entregar el código fuente y gráficos que muestren la evolución de las distintas generaciones.
El programa se deberá ejecutar en las máquinas del laboratorio (en el caso de requerir algún ambiente especial, este se debe traer el día de la exposición).

2. Diseño:

2.1. Diagrama de clases:



2.2. Diagrama de flujo:



2.3. Parámetros del Programa:

El programa tiene una gran cantidad de parámetros que no se reciben directamente por línea de comandos precisamente por ser muchos y para evitar confusiones en el orden en que se deberían pasar los mismos.
Hay dos tipos de parámetros en el programa: aquellos vinculados directamente con el problema del llenado del camión y los que están vinculados al funcionamiento del algoritmo genético.

2.3.1. Parámetros vinculados con el problema:

La forma en que se decidió parametrizar estos valores es a través de un archivo `constantes.h`, donde se agruparon todos los parámetros del problema.
Las constantes que se encuentran en este archivo son las siguientes:

- **Medidas del camión:**

```
ALTURA_CAMION
BASE_CAMION
PROFUNDIDAD_CAMION
```

- **Medidas de las cajas:**

```
ALTURA_CAJA_A
BASE_CAJA_A
PROFUNDIDAD_CAJA_A

ALTURA_CAJA_B
BASE_CAJA_B
PROFUNDIDAD_CAJA_B

ALTURA_CAJA_C
BASE_CAJA_C
PROFUNDIDAD_CAJA_C
```

- **Cantidad de cajas disponibles de cada tipo:**

```
MAX_CAJAS_A
MAX_CAJAS_B
MAX_CAJAS_C
```

- **Cantidad mínima de cajas de cada tipo que se deben colocar en el camión:**

```
MIN_CAJAS_A
MIN_CAJAS_B
MIN_CAJAS_C
```

Si el usuario quiere cambiar las condiciones del problema lo único que tiene que hacer es editar este archivo para reflejar los nuevos valores y volver a compilar el programa.

2.3.2. Parámetros vinculados con el algoritmo genético:

Hay otro tipo de parámetros que no están vinculados directamente con el problema del llenado del camión, sino que son parámetros del algoritmo genético que se usan para definir varios aspectos de su funcionamiento.
Para parametrizar este tipo de variables se usó otro archivo de constantes llamado `ctes_algoritmo.h`, que almacena los siguientes datos:

- `TAMANIO`: tamaño de las poblaciones.
- `PORCENTAJE_SELECCION`: es el porcentaje de la población actual que pasará a la siguiente generación por medio de la selección natural.
- `MAX_ITERACIONES`: es la cantidad máxima de generaciones que se calcularán, es una de las formas usadas para que el algoritmo corte.
- `CORTE`: Como se mostró en el diagrama de clases el algoritmo puede cortar su ciclo while por dos caminos: que se supere el máximo número de iteraciones o que la diferencia entre el mejor (medido por aptitud) individuo de una generación y el promedio sea menor a una constante, esa constante es `CORTE`.

3. Gráficos evolutivos:

Hemos construido diferentes tipos de gráficos para poder analizar la evolución de las distintas generaciones en forma más amplia:

3.1. La función de aptitud:

Se decidió graficar el promedio de las aptitudes en función del número de generación.
La función de aptitud debía contemplar que el volumen libre del camión sea mínimo, y que la cantidad de cajas sea máxima, privilegiando el mínimo. Por la diferencia de escalas, se decidió normalizar esos valores y que sean comparables. Una vez que se decidió la forma de normalizar, se ponderaron esos valores y se sumaron.
Para normalizar la cantidad de cajas, se buscó la caja de menor volumen, se dividió el volumen del camión por el de ella y se obtubo la cantidad de cajas máximas. Finalmente se sumaron la cantidad de cajas de todos los tipos y se dividieron por la cantidad de cajas máximas.
Para normalizar el volúmen se calculo el volumen libre como el volumen del camión menos la suma de los volúmenes de cada caja por la cantidad de cajas de ese tipo; a eso se lo dividió por el volumen del camión. Por último como minimizar `N` es equivalente a maximizar `1-N`; y nosotros teníamos un funcional

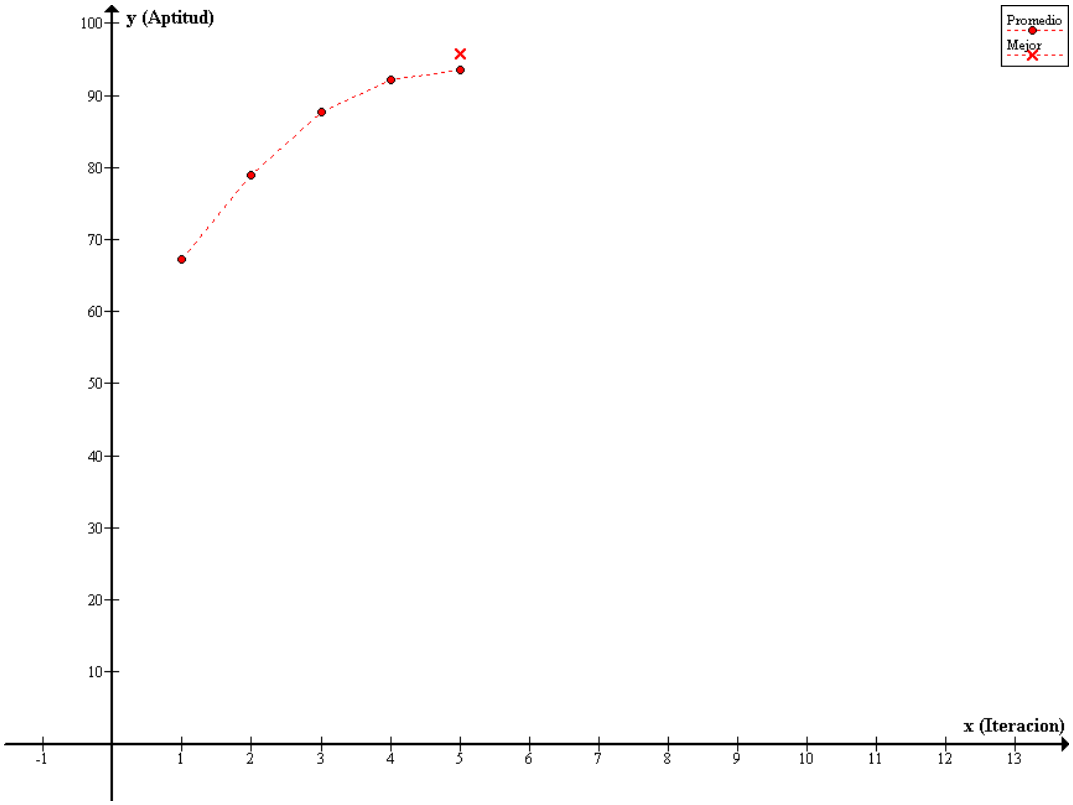


Figura 1: Aptitud Promedio en función de las iteraciones.

de máximizar el otro valor, la función aptitud nos quedó:
$aptitud = 60 * (1 - \frac{volumen\_libre}{volumen\_camion}) + 40 * \frac{cajasA + cajasB + cajasC}{cajasMaximas}$

3.2. Ejemplo 1:

A continuación puede observarse una tabla que muestra la aptitud promedio de cada generación, seguida por su gráfico.

Iteración	Aptitud promedio
1	67.3365
2	78.8895
3	87.715
4	92.238
5	93.472

En el gráfico puede observarse como mejora notablemente la solución acercandose en pocos pasos a la que fue, para esa corrida, la mejor solución.

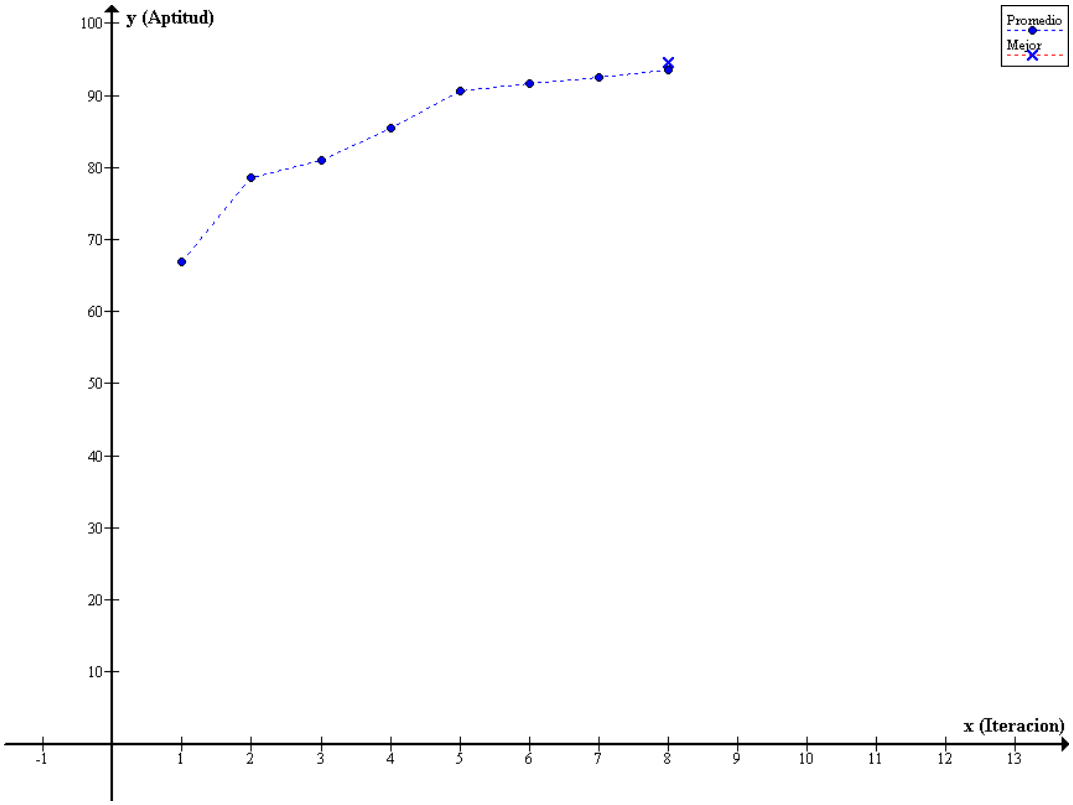


Figura 2: Aptitud Promedio en función de las iteraciones.

3.3. Ejemplo 2:

A continuación puede observarse una tabla que muestra la aptitud promedio de cada generación, seguida por su gráfico.

Iteración	Aptitud promedio
1	66.9565
2	78.6555
3	81.0085
4	85.4465
5	90.647
6	91.599
7	92.498
8	93.551

En el gráfico puede observarse como mejora notablemente la solución acercandose en pocos pasos a la que fue, para esa corrida, la mejor solución.
Para este segundo ejemplo el resultado final fue:

```
Cajas A: 370 Cajas B: 329 Cajas C: 17  Aptitud: 94.6
```

3.4. Comparación de los dos ejemplos

Como puede verse en las figuras 3.a y 3.b; el primer ejemplo logró un individuo de 95.75 de aptitud, en 5 generaciones, en cambio el segundo ejemplo logró en 8 generaciones tan solo un individuo de 94.6 de aptitud. Esto se debe a que para la reproducción y Selección Natural se usaron valores aleatorios, por lo que, mientras el primer ejemplo tubo un crecimiento logaritmico, el segundo comenzó bien, pero luego se estanco en su crecimiento hasta que retomo la curva original.

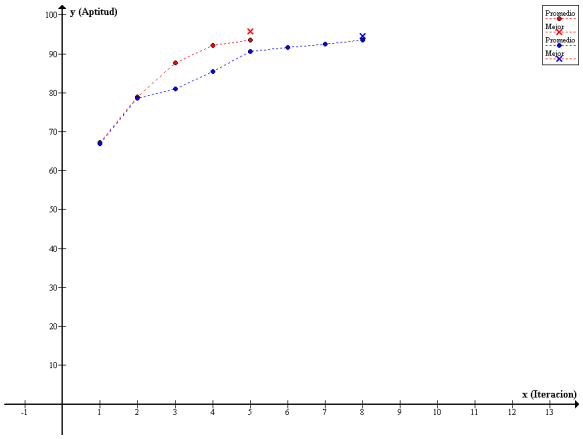


Figura 3.a

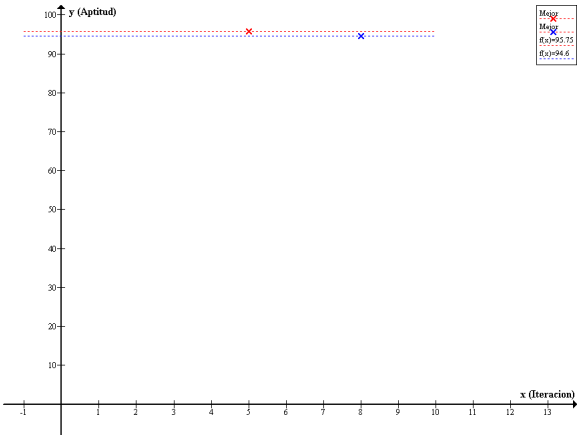


Figura 3.b

Figura 3: Comparación de las dos iteraciones.

4. Código Fuente:

4.1. tp1.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <fstream>
#include <iostream>

#include "generaciones.h"
#include "ctes_algoritmo.h"

using namespace std;

int main(int argc, char *argv[]){
    float promedio, mejor;

    srand(time(NULL));
```

```
    /* creo un objeto de la clase Generaciones. Contendra las sucesivas
    poblaciones que se iran formando en las iteraciones del algoritmo genetico.
    Se crea con la poblacion o generacion inicsi, tira algunosial. */
    Generaciones generaciones(TAMANIO, PORCENTAJE_SELECCION);

    int iteraciones = 1;
    promedio = generaciones.getPromedio();
    mejor = generaciones.getAptitudMejorIndividuo();

    while ((iteraciones < CANT_MAX_ITERACIONES) & (mejor-promedio > CORTE)){
        generaciones.siguienteGeneracion();
        promedio = generaciones.getPromedio();
        mejor = generaciones.getAptitudMejorIndividuo();
        iteraciones++;
    }

    cout << "Se termino el algoritmo genetico" << endl;


    // el constructor de ofstream abre el archvivo
    ofstream archivo ("salida.txt", ios::out);
    archivo << generaciones;

    cout << "El individuo final se obtuvo en la generacion numero " << iteraciones;
    cout << " y es:\n" << generaciones.getUltimoMejorIndividuo();

    return 0;
}
```

4.2. generaciones.h

```
#ifndef __GENERACIONES_H__
#define __GENERACIONES_H__

#include <vector>
#include "generacion.h"
#include "vectorindividuos.h"

class Generaciones{

    std::vector <Generacion> historiaGeneraciones;
    Generacion genActual;
    int tamanio;
    float porcentajeSeleccion;
public:
    /**
     * Construye la clase y crea la primer generacion.
     * @param tam ñ Tamao de las poblaciones.
```

```
     * @param porcentaje Porcentaje a la cual se le va a aplicar Seleccion Natural.
     */
    Generaciones(int tam, float porcentaje);

    /**
     * Metodo que genera, almacena y retorna la ógeneracin siguiente a la ultima almacenada.
     * @return Generacion siguiente a la actual.
     */
    Generacion siguienteGeneracion();

    /**
     * Metodo que retorna la generacion gen.
     * @param numGen Numero de generacion que genera, comenzando con 0.
     * @return La generacion elegida.
     */
    Generacion getGeneracion(int numGen);

    Individuo getUltimoMejorIndividuo();

    friend std::ostream& operator<< (std::ostream &stream, Generaciones gens);

    float getAptitudMejorIndividuo();

    float getPromedio();


    /**
     * Destructor de la clase.
     */
    ~Generaciones();

};

#endif
```

4.3. generaciones.cpp

```
#include "generaciones.h"

// constructor
Generaciones::Generaciones (int tam, float porcent):genActual(tam, porcent, true)
{
    tamanio = tam;
    porcentajeSeleccion = porcent;
}

// destructor
Generaciones::~Generaciones () {}
```

```
Generacion Generaciones::siguienteGeneracion ()
{
    Generacion nuevaGen(tamanio, porcentajeSeleccion);

    VectorIndividuos aux = genActual.seleccionNatural();

    nuevaGen.agregarIndividuos(aux);

    aux = genActual.reproduccion();

    nuevaGen.agregarIndividuos(aux);

    historiaGeneraciones.push_back(genActual);

    genActual = nuevaGen;

    return nuevaGen;
}

Generacion Generaciones::getGeneracion (int numGen)
{
    int size = historiaGeneraciones.size();

    if ((numGen < 0) || (numGen > size-1)) return genActual;
    else return historiaGeneraciones[numGen];
}

std::ostream& operator<< (std::ostream &stream, Generaciones gens)
{
    std::vector<Generacion>::iterator iterador;

    stream << "Imprimiendo TODAS las generaciones.\n";

    for (iterador=gens.historiaGeneraciones.begin(); iterador!=gens.historiaGeneraciones.end();
        iterador++)
        stream << *iterador << "\n";

    stream << "Final de TODAS las generaciones.\n";

    return stream;
}

float Generaciones::getAptitudMejorIndividuo()
{
    return genActual.getAptitudMejorIndividuo();
}

float Generaciones::getPromedio()
{
```

```
    return genActual.getPromedio();
}

Individuo Generaciones::getUltimoMejorIndividuo()
{
    return genActual.getMejorIndividuo();
}
```

4.4. generacion.h

```
#ifndef __GENERACION_H__
#define __GENERACION_H__

#include <vector>
#include <cstdlib>
#include <list>
#include <iostream>
#include <algorithm>
#include <cassert>
#include "individuo.h"
#include "vectorindividuos.h"

class Generacion{
    VectorIndividuos poblacion;
    int tamanio;
    float porcentajeSeleccionNatural;

    int random(int min, int max);
    void unionPoblaciones (VectorIndividuos & pob1, VectorIndividuos & pob2);
    void unionPoblaciones (VectorIndividuos & pob1, std::vector<Individuo> & pob2);
    int competencia(Individuo, Individuo);
public:

    /**
     * Constructor de la clase.
     * @param tam ñ Tamao final de la poblacion.
     * @param porcentaje Porcentaje a la cual se le va a aplicar Seleccion Natural.
     */
    Generacion(int tam, float porcentaje, bool primera=false);

    void agregarIndividuos (VectorIndividuos individuos);

    /**
     * Metodo que le aplica Seleccion Natural a la poblacion para obtener un nuevo conjunto de individuos.
     * @return Individuos que pertenecian a la poblacion0 y van a pertenecer a la siguiente generacion.
     */
    VectorIndividuos seleccionNatural();
```

```
    /**
     * Metodo que reproduce individuos a partir de la poblacion0 para obtener un nuevo conjunto de individ
     * @return Individuos que no pertenecian a la poblacion0 y van a pertenecer a la siguiente generacion.
     */
    VectorIndividuos reproduccion();

    /**
     * @return El mejor individuo de la generacion.
     */
    Individuo getMejorIndividuo();

    float getAptitudMejorIndividuo();

    float getPromedio();

    friend std::ostream& operator<< (std::ostream &stream, Generacion gen);

    /**
     * Destructor de la clase.
     */
    ~Generacion();

};

#endif
```

4.5. generacion.cpp

```
#include "generacion.h"

Generacion::Generacion(int tam, float porcentaje, bool primera){
    tamanio = tam;
    porcentajeSeleccionNatural = porcentaje;
    if (primera) {
        for (int cont=0; cont<tamanio; cont++) {
            Individuo aux;
            aux.inicializarIndividuo();
            poblacion.push_back(aux);
        }
    }
}

void Generacion::unionPoblaciones (VectorIndividuos & pob1, std::vector<Individuo> & pob2){
    std::vector<Individuo>::iterator iterador;
    for (iterador=pob2.begin();iterador!=pob2.end();iterador++)
        if (pob1.size()<tamanio) pob1.push_back(*iterador);
}
```

```
void Generacion::unionPoblaciones (VectorIndividuos & pob1, VectorIndividuos & pob2){
    VectorIndividuos::iterator iterador;
    for (iterador=pob2.begin();iterador!=pob2.end();iterador++)
        if (pob1.size()<tamanio) pob1.push_back(*iterador);
}

void Generacion::agregarIndividuos (VectorIndividuos individuos){
    unionPoblaciones(poblacion,individuos);
}


int Generacion::random(int min, int max){
    int rango = max - min, randomFinal;
    float randomNormalizado;

    randomNormalizado = rand()/(RAND_MAX + 1.0);
    randomFinal = min + (int)rango*randomNormalizado;

    return randomFinal;
}

/**
 * Metodo que compara la aptitud de cada los individuos y retorna el numero del mejor.
 * @param i1 Individuo 1
 * @param i2 Individuo 2
 */
int Generacion::competencia(Individuo i1, Individuo i2){
    if (i1.aptitud() > i2.aptitud()) return 1;
    else return 2;
}

/**
 * @param i1 Individuo 1.
 * @param i2 Individuo 2.
 * @return true si individuo 1 es mayor a individuo 2.
 */
bool comparador(Individuo i1, Individuo i2){
    return (i1.aptitud() > i2.aptitud());
}

/**
 * Metodo que le aplica Seleccion Natural a la poblacion para obtener un nuevo conjunto de individuos.
 * @param poblacion0 Poblacion original o generacion anterior.
 * @return Individuos que pertenecian a la poblacion y van a pertenecer a la siguiente generacion.
 */
VectorIndividuos Generacion::seleccionNatural(){
    int ind1, ind2, contador, cantidad = tamanio*(porcentajeSeleccionNatural/2);
    VectorIndividuos::iterator iterador;
    VectorIndividuos returnGen;
```

```
    contador=0;
    // Elijo el primer cuarto del total para competencia.
    while (contador<cantidad){
        ind1 = random(0, poblacion.size());
        while ((ind2=random(0, poblacion.size())) != ind1);
        int indiceMejor = competencia(poblacion[ind1],poblacion[ind2]);
        returnGen.push_back(poblacion[indiceMejor]);
        returnGen.push_back(poblacion[indiceMejor]);
        contador += 2;
    }

    // Ordeno el vector en forma decreciente.
    sort(poblacion.begin(), poblacion.end(), comparador);

    iterador=poblacion.begin();
    for (contador=0; contador<cantidad; contador++) {
        returnGen.push_back(*iterador);
        iterador++;
    }

    return returnGen;
}

/**
 * Metodo que reproduce individuos a partir de la poblacion0 para obtener un nuevo conjunto de individuos.
 * @param poblacion0 Poblacion original o generacion anterior.
 * @return Individuos que no pertenecian a la poblacion0 y van a pertenecer a la siguiente generacion.
 */
VectorIndividuos Generacion::reproduccion(){
    int ind1, ind2, contador, cantidad = tamanio-2*cantidaSN;
    int cantidaSN = tamanio*(porcentajeSeleccionNatural/2);
    contador=0;
    VectorIndividuos returnGen;

    while (contador<cantidad){
        ind1 = random(0, poblacion.size());
        while ((ind2=random(0, poblacion.size())) != ind1);
        std::vector<Individuo> hijos = Individuo::reproducirse(poblacion[ind1],poblacion[ind2]);
        unionPoblaciones(returnGen, hijos);
        contador += hijos.size();
    }
    return returnGen;
}

/**
 * @return El mejor individuo de la generacion.
 */
Individuo Generacion::getMejorIndividuo(){
    VectorIndividuos::iterator iterador;
```

```
    sort(poblacion.begin(), poblacion.end(), comparador);

    iterador = poblacion.begin();
    return (*iterador);
}

float Generacion::getAptitudMejorIndividuo(){
    VectorIndividuos::iterator iterador;

    sort(poblacion.begin(), poblacion.end(), comparador);

    iterador = poblacion.begin();
    return (*iterador).aptitud();
}

float Generacion::getPromedio(){
    VectorIndividuos::iterator iterador;
    float suma=0;

    for (iterador=poblacion.begin();iterador!=poblacion.end();iterador++)
        suma += iterador->aptitud();

    return suma/tamanio;
}

std::ostream& operator<< (std::ostream &stream, Generacion gen){
    VectorIndividuos::iterator iterador;

    stream << "Imprimiendo una nueva generacion\n";
    for (iterador=gen.poblacion.begin(); iterador!=gen.poblacion.end(); iterador++)
        stream << *iterador;
    stream << "Promedio: " << gen.getPromedio() << " Mejor Aptitud: " << gen.getAptitudMejorIndi
    stream << gen.getMejorIndividuo() << std::endl;
    stream << "Fin de la generacion\n\n";

    return stream;
}

Generacion::~Generacion(){ }
```

4.6. vectorindividuos.h

```
#ifndef __VECTORINDIVIDUOS_H__
#define __VECTORINDIVIDUOS_H__

#include <vector>
#include "individuo.h"
```

```
class VectorIndividuos : public std::vector<Individuo>{};

#endif
```

4.7. individuo.h

```
#ifndef __INDIVIDUO_H__
#define __INDIVIDUO_H__

#include <cstdlib>
#include <stdio.h>
#include <iostream>
#include <sstream>
#include <vector>
#include "constantes.h"

class Individuo{
    int cajasA;
    int cajasB;
    int cajasC;

    int random(int min, int max);
    static Individuo obtenerHijo(Individuo padre1, Individuo padre2, int hijo);
public:
    /**
     * Constructor de la clase que verifica que se cumplan todas las condiciones.
    */
    Individuo();

    /**
     * Constructor de la clase que verifica que se cumplan todas las condiciones.
     */
    Individuo(int a, int b, int c);

    /**
     * Constructor de copia.
     */
    Individuo(const Individuo & i);

    /**
     * Metodo que inicializa automaticamente el Individuo.
     */
    void inicializarIndividuo();

    /**
     * Meotodo que calcula la aptitud de un individuo.
     * @return aptitud del individuo.
```

```
     */
    float aptitud();

    /**
     * Metodo que crea un nuevo individuo a partir del individuo actual y un padre.
     * @param padre1 Individuo que, sumado al padre2, crea un nuevo individuo.
     * @param padre2 Individuo que, sumado al padre1, crea un nuevo individuo.
     * @return Hijo del padre y este individuo.
     */
    static std::vector<Individuo> reproducirse(Individuo padre1, Individuo padre2);

    int getCajasA() {return cajasA;}

    int getCajasB() {return cajasB;}

    int getCajasC() {return cajasC;}

    void setCajasA(int cajas) { cajasA = cajas;}

    void setCajasB(int cajas) { cajasB = cajas;}

    void setCajasC(int cajas) { cajasC = cajas;}

    const Individuo &operator= (const Individuo &i);

    bool operator> (Individuo &i);

    bool operator< (Individuo &i);

    bool operator>= (Individuo &i);

    bool operator<= (Individuo &i);

    bool operator== (Individuo &i);

    friend std::ostream& operator<< (std::ostream &stream, Individuo ind);


    /**
     * Destructor de la clase.
     */
    ~Individuo();

};

#endif
```

4.8. individuo.cpp

```
#include "individuo.h"

Individuo::Individuo(){
    cajasA =0;
    cajasB =0;
    cajasC =0;
}

Individuo::Individuo(int a, int b, int c){
    cajasA = a;
    cajasB = b;
    cajasC = c;
}

Individuo::Individuo(const Individuo & i){
    this->cajasA = i.cajasA;
    this->cajasB = i.cajasB;
    this->cajasC = i.cajasC;
}


Individuo Individuo::obtenerHijo(Individuo padre1, Individuo padre2, int hijo)
{
    Individuo indHijo;
    switch (hijo)
    {
        case 0:
            indHijo.setCajasA(padre1.getCajasA());
            indHijo.setCajasB(padre2.getCajasB());
            indHijo.setCajasC(padre1.getCajasC());
            break;
        case 1:
            indHijo.setCajasA(padre1.getCajasA());
            indHijo.setCajasB(padre1.getCajasB());
            indHijo.setCajasC(padre2.getCajasC());
            break;
        case 2:
            indHijo.setCajasA(padre2.getCajasA());
            indHijo.setCajasB(padre1.getCajasB());
            indHijo.setCajasC(padre1.getCajasC());
            break;
        case 3:
            indHijo.setCajasA(padre1.getCajasA());
            indHijo.setCajasB(padre2.getCajasB());
            indHijo.setCajasC(padre2.getCajasC());
            break;
        case 4:
            indHijo.setCajasA(padre2.getCajasA());
            indHijo.setCajasB(padre2.getCajasB());
```

```
            indHijo.setCajasC(padre1.getCajasC());
            break;
        case 5:
            indHijo.setCajasA(padre2.getCajasA());
            indHijo.setCajasB(padre1.getCajasB());
            indHijo.setCajasC(padre2.getCajasC());
            break;
    }
    return indHijo;
}


void Individuo::inicializarIndividuo(){
    float volumen_camion = ALTURA_CAMION * BASE_CAMION * PROFUNDIDAD_CAMION;
    float volumen_cajas_A, volumen_cajas_B, volumen_cajas_C, volumen_total;

    do{
        cajasA = random (MIN_CAJAS_A, MAX_CAJAS_A+1);
        cajasB = random (MIN_CAJAS_B, MAX_CAJAS_B+1);
        cajasC = random (MIN_CAJAS_C, MAX_CAJAS_C+1);

        volumen_cajas_A = cajasA*ALTURA_CAJA_A*BASE_CAJA_A*PROFUNDIDAD_CAJA_A;
        volumen_cajas_B = cajasB*ALTURA_CAJA_B*BASE_CAJA_B*PROFUNDIDAD_CAJA_B;
        volumen_cajas_C = cajasC*ALTURA_CAJA_C*BASE_CAJA_C*PROFUNDIDAD_CAJA_C;

        volumen_total = volumen_cajas_A + volumen_cajas_B + volumen_cajas_C;

    }while (volumen_total > volumen_camion);

}

int Individuo::random(int min, int max)
{
    int rango = max - min, randomFinal;
    float randomNormalizado;

    randomNormalizado = rand()/(RAND_MAX + 1.0);
    randomFinal = min + (int)rango*randomNormalizado;

    return randomFinal;
}

std::vector<Individuo> Individuo::reproducirse(Individuo padre1, Individuo padre2){
    int hijo1=0, hijo2=5;
    Individuo indHijo1, indHijo2;
    std::vector<Individuo> hijos;

    indHijo1 = obtenerHijo(padre1, padre2, hijo1);
    indHijo2 = obtenerHijo(padre1, padre2, hijo2);
```

```
    // Trato de devolver un hijo1
    while ((indHijo1.aptitud() == 0) && (hijo1<6)) {
        hijo1++;
        indHijo1 = obtenerHijo(padre1, padre2, hijo1);
    }

    if (indHijo1.aptitud() != 0) hijos.push_back(indHijo1);

    // Trato de devolver un hijo2, no importa si son gemelos.
    while ((indHijo2.aptitud() == 0) && (hijo2>=0)) {
        hijo2--;
        indHijo2 = obtenerHijo(padre1, padre2, hijo2);
    }

    if (indHijo2.aptitud() != 0) hijos.push_back(indHijo2);

    return hijos;
}

float Individuo::aptitud(){
    float apt;
    float volumen_camion = ALTURA_CAMION * BASE_CAMION * PROFUNDIDAD_CAMION;
    float volumen_cajas_A, volumen_cajas_B, volumen_cajas_C, volumen_total, volumen_libre;
    float minimo, cajasMaximas;

    // Calculo el volumen de las cajas.
    volumen_cajas_A = ALTURA_CAJA_A*BASE_CAJA_A*PROFUNDIDAD_CAJA_A;
    volumen_cajas_B = ALTURA_CAJA_B*BASE_CAJA_B*PROFUNDIDAD_CAJA_B;
    volumen_cajas_C = ALTURA_CAJA_C*BASE_CAJA_C*PROFUNDIDAD_CAJA_C;

    //Calculo el volumen total de las cajas.
    volumen_total = cajasA*volumen_cajas_A + cajasB*volumen_cajas_B + cajasC*volumen_cajas_C;
    if (volumen_total > volumen_camion) return 0;
    if ((cajasA>MAX_CAJAS_A)||(cajasA<MIN_CAJAS_A)) return 0;
    if ((cajasB>MAX_CAJAS_B)||(cajasB<MIN_CAJAS_B)) return 0;
    if ((cajasC>MAX_CAJAS_C)||(cajasC<MIN_CAJAS_C)) return 0;

    if (volumen_cajas_A<volumen_cajas_B) minimo = volumen_cajas_A;
    else minimo = volumen_cajas_B;

    if (volumen_cajas_C<minimo) minimo = volumen_cajas_C;

    cajasMaximas = volumen_camion/minimo;

    volumen_libre = volumen_camion - volumen_total;

    // Normalizo los valores de Volumne Libre, y cantidad de cajas para luego ponderar
    // mas el volumen libre y que esa sea la aptitud final.
    apt = 60*(1-volumen_libre/volumen_camion)+40*(cajasA+cajasB+cajasC)/cajasMaximas;
```

```
    return apt;
}

bool Individuo::operator> (Individuo &i){
    if (this->aptitud()>i.aptitud()) return true;
    else return false;
}

bool Individuo::operator< (Individuo &i){
    if (this->aptitud()<i.aptitud()) return true;
    else return false;
}

bool Individuo::operator>= (Individuo &i){
    if (this->aptitud()>=i.aptitud()) return true;
    else return false;
}

bool Individuo::operator<= (Individuo &i){
    if (this->aptitud()<=i.aptitud()) return true;
    else return false;
}

bool Individuo::operator== (Individuo &i){
    if (this->aptitud()==i.aptitud()) return true;
    else return false;
}

const Individuo &Individuo::operator= (const Individuo &i){
    if (&i != this) {
        this->cajasA = i.cajasA;
        this->cajasB = i.cajasB;
        this->cajasC = i.cajasC;
    }
    return (*this);
}

std::ostream& operator<< (std::ostream &stream, Individuo ind){
    stream << "Cajas A: " << ind.cajasA << "\t";
    stream << "Cajas B: " << ind.cajasB << "\t";
    stream << "Cajas C: " << ind.cajasC << "\t\t Aptitud: " << ind.aptitud() << "\n";
    return stream;
}

Individuo::~Individuo(){}
```

4.9. constantes.h

```
#ifndef __CONSTANTES_H__
#define __CONSTANTES_H__

// En ídecmetros.
#define ALTURA_CAMION 40
#define BASE_CAMION 20
#define PROFUNDIDAD_CAMION 180

#define ALTURA_CAJA_A 5
#define BASE_CAJA_A 2
#define PROFUNDIDAD_CAJA_A 18

#define ALTURA_CAJA_B 15
#define BASE_CAJA_B 3
#define PROFUNDIDAD_CAJA_B 4

#define ALTURA_CAJA_C 5
#define BASE_CAJA_C 30
#define PROFUNDIDAD_CAJA_C 6

#define MAX_CAJAS_A 2000
#define MAX_CAJAS_B 3200
#define MAX_CAJAS_C 5250
#define MIN_CAJAS_A 5
#define MIN_CAJAS_B 2
#define MIN_CAJAS_C 3

#endif
```

4.10. ctes_algoritmo.h

```
#ifndef __CONSTANTES_H__
#define __CONSTANTES_H__

#define TAMANIO 50
#define PORCENTAJE_SELECCION 0.50
#define CANT_MAX_ITERACIONES 20
#define CORTE 1

#endif
```