Documento de Diseño de Prácticas de Metaheurísticas

Daniel Molina Cabrera

February 24, 2025

Contents

1 Sobre este documento

Este documento contiene guías de diseño que son **obligatorias** para las prácticas de la Asignatura de *Metaheurísticas* del *Grado en Ingeniería Informática*.

Aunque otros cursos se ha dado libertad absoluta sobre las decisiones de diseño de las prácticas, e incluso el Lenguaje de Programación, hay distintos motivos para cambiarlo.

La principal motivación es el hecho de que gran parte de las dudas en prácticas se debían a problemas en el código. Y dicho código era demasiado complejo debido a diseños inadecuados. La falta de guía se mostraba contraproducente si se valoraba tanto la falta de experiencia en diseñar código mantenible, como el hecho de que posiblemente era la primera vez que se pedía al estudiantado crear una práctica de cero. Pensaba que, dado que las prácticas no dependen prácticamente de librerías externas, y la relativa simplicidad de la mayoría de algoritmos permitía ese esquema, al considerarlo un paso interesante y conveniente para el aprendizaje. Sin embargo, tras varios cursos he comprobado que la falta de criterios para valorar un buen diseño hacía que ese aprendizaje no se daba, y que además, el coste ocasionado debido a un mal diseño aumentaba el número de horas en desarrollo, especialmente en depuración, perdiendo el foco en la parte relevante de la asignatura, como es el uso de metaheurísticas para resolver el problema. Y herramientas como CoPilot o similares no ayudan al diseño, al fomentar la redundancia en el código.

Por todo lo anterior, en este documento daré unas guías de diseño, que se materializarán en un repositorio github a usar como plantilla. De todas formas, con vista al aprendizaje, justificaré las decisiones tomadas.

La guía está planteada en C++, que es el lenguaje que más se ha usado en las prácticas, aunque si alguien quisiese utilizar otro lenguaje Orientado a Objeto (OO) como Java no habría problema, siempre que aplicase el mismo criterio (con los cambios de sintaxis correspondientes). Para lenguajes no OO daré también indicaciones alternativas.

2 Separación Algoritmo y Problema

Lo primero es que el diseño separe claramente la parte del problema de la parte del algoritmo, ya que tenemos un mismo problema y múltiples algoritmos.

2.1 Motivación

El principal problema observado es que se integraba demasiado el algoritmo y el problema. Aunque teóricamente eran diseños OO en la práctica no era tal, ya que había una clase P1 para la primera práctica, P2 para la segunda, ... y quizás alguna clase con utilidades (*Utils* o similar), pero sin una cohesión clara.

¿Eso que problemas presenta? De todo tipo:

- **La metaheurística depende del problema**: Lo primero, la metaheurística no se puede probar hasta tener el problema concreto implementado. Si se separase, se podría probar la metaheurística con un problema *de juguete* (como optimizar el número total de unos en un vector binario) que permitiese ver fácilmente si el algoritmo parece funcionar mucho más fácilmente.
- **Reutilizar código:** El problema es común para todos los algoritmos. Al no hacerlo así, se producían mucho copiar y pegar entre algoritmos y prácticas, en vez de poder utilizar el Problema *tal cual* entre

prácticas, sin tener que cambiarlo.

- **Aislar la complejidad del problema**: De esta forma la complejidad del problema y la función de evaluación se aisla, con lo que el código del algoritmo no se complica debido al problema, aislando la complejidad.
- **Pruebas independientes del problema:** Una mayor separación facilita el poder probar por separado si cada parte funciona. Es decir, se puede más fácilmente darle una solución a evaluar y confirmar si el valor obtenido es correcto.
- **Pruebas automáticas**: Las pruebas anteriores pueden automatizarse.

2.2 Aplicación

Lo principal es separar el código en clases.

2.3 Tipo tSolution

Tanto el algoritmo como la clase problem deben de trabajar, y operar con un tipo tSolution que representa la solución usando la codificación elegida para el problema. En vez de utilizar una clase abstract, es preferible en la medida de lo posible usar un tipo, ya que permite operar de forma más sencilla. El algoritmo puede suponer que es una colección de algún tipo.

Ejemplo para representación binaria:

```
typedef vector<bool> tSolution
```

y para una representación de números reales:

```
typedef vector<float> tSolution
```

No se almacena el fitness de forma asociada directamente, ya que puede haber casos en los que no tenga sentido (una solución no válida) o bien porque se quiere postergar la evaluación de la solución.

El uso de un *typedef* en vez de un *class* facilita el procesarlo de forma más clara: acceder a elementos, conocer su longitud, ..., que de la otra forma implicaría el tener que definir muchos métodos triviales y redundantes.

2.4 Tipo tFitness

Para facilitar el uso de fitness, se define el tipo de dato. Valores posibles son *float*, *double* o *int*.

2.5 Clase Problem

Esta clase se define para representar el problema. La clase es abstract, lo cual quiere decir que se debe heredar y crear los métodos correspondientes para ofrecer el interfaz a los algoritmos.

El interfaz es el mínimo posible:

```
virtual class Problem {
public:
  virtual float fitness(tSolution);
```

```
  virtual tSolution createSolution();
  virtual int getSize();
}
```

- **getSize** permite conocer la longitud de cada variable.
- **createSolution** permite crear una solución de forma totalmente aleatoria.
- **fitness** es para evaluar una solución dada. Siempre ante la misma solución se de deberá de ontener el mismo valor *fitness*. De esta manera, dada una solución, generará un valor fitness que se debe de minimizar. Que solo minimicen los algoritmos no se pierde generalidad. Si es un problema de minimizar se puede calcular el fitness y devolver -$\text{fitness}_{\text{maximizar}}$, así se convierte en un problema de maximización,

El problema deberá de implementarse heredando de la clase *Problem* implementando los tres métodos anteriores.

Todas las variables que se requieran para evaluar la solución se podrán guardar como variables de instancia, y procesarlo.

Solo se usará una variable de tipo Problem compartido entre algoritmos, por lo que no es necesario el uso de variables de clase.

2.6 Clase Random y generación de Números aleatorios

Los algoritmos que se probarán suelen depender de factores aleatorios. Para poder hacer eso se utilizan generadores de números seudoaleatorios. Los generadores de números seudoaleatorios producen secuencias de números que parecen aleatorias. A diferencia de los generadores de números verdaderamente aleatorios, utilizan un estado interno y una fórmula matemática para generar los números, lo que significa que la secuencia se puede repetir dada la misma semilla.

La semilla es el valor inicial utilizado para iniciar la generación de la secuencia de números seudoaleatorios. Dos secuencias serán idénticas si se utilizan las mismas semillas. Al establecer una semilla específica, se puede reproducir la misma secuencia de números seudoaleatorios, lo que resulta útil para las pruebas y la reproducibilidad ya que permite reproducir resultados y depurar problemas. Se ejecutarán los distintos algoritmos con las mismas semillas.

Se recomienda mirar el programa $\text{example}_{\text{random.cpp}}$ para aprender cómo utilizar la clase random de la plantilla.

2.7 Algoritmo MH

Todo algoritmo implementado deberá de cumplir el siguiente interfaz:

```
class MH {
  public:
    ...
    virtual pair<tSolution, tFitness> optimize(Problem *problem,
                                               int maxevals) = 0;
  };
```

Como se puede observar, lo principal es implementar el método optimize, que:

- Recibe un objeto de tipo Problem*, con el que podrá evaluar las soluciones, entre otras cosas.
- Recibe el número máximo de evaluaciones de soluciones permitido.

El criterio de parada debe ser igual a que se haya alcanzado el número máximo de evaluaciones. A no ser que en el algoritmo se indique otra cosa, es el único criterio para parar. No se sabe el valor óptimo de *fitness*, por lo que cualquier valor es adecuado.

El método deberá de devolver la mejor solución de las generadas, junto con su valor de *fitness* correspondiente.

2.8 Estructura de la plantilla

La plantilla posee la siguiente estructura:

common Ficheros comunes. Únicamente el fichero solution.h se modificará, en función de la representación de una solución.

inc Ficheros .h de las clases creadas, tanto del problema como de los algoritmos implementados.

src Implementación de las clases cuyas cabeceras se encuentran en **inc**/.

main.cpp Programa de referencia, para modificarlo con el problema concreto e implementaciones de los distintos algoritmos de la práctica.

CMakeLists.txt Fichero para compilar, en principio no sería necesario .

2.9 Cómo compilar

Para compilar es necesario el software CMake. Este programa permite generar la estructura adecuada para compilar. Puede generar un Makefile, o bien proyectos para VS o para Xcode. Se recomienda la generación de Makefile. Aunque cmake genera el makefile, siempre es necesario tenerlo instalado para compilarlo:

```
cmake . && make
```

Por defecto cmake compila para depurar fácilmente, pero eso puede suponer un empeoramiento en eficiencia muy relevante por el uso de la librería STL.

Si se desea mejorar el rendimiento es conveniente compilar en modo *"Release"* en vez de en modo *"Debug"*

```
cmake -DCMAKE_BUILD_TYPE=Release . && make
```

3 Otros Lenguajes de Programación

Aunque la plantilla está claramente pensada para C++, eso es porque es el principal lenguaje usado tradicionalmente en las prácticas. Sin embargo, se puede adaptar el interfaz igualmente a otros lenguajes de programación.

A continuación presentamos cómo se puede adaptar a distintos lenguajes:

3.1 Java

La adaptación a *Java* es muy fácil, ya que es un lenguaje totalmente Orientado a Objeto.

Para definir la clase solución se puede aplicar directamente herencia de una collección, como:

```
class tSolution extends ArrayLists<Float> {}
```

La clase **Problem** puede definirse directamente como lo que es, un *interfaz*:

```
public interface Problem {
  float fitness(tSolution);
  tSolution createSolution();
  int getSize();
}
```

Mientras que la clase MH puede tener métodos externos, así que es más fácil definirlo como clase.

```
public class MH {
   // ...
   public Pair<tSolution, tFitness> optimize(Problem problem,
                                             int maxevals);
  };
```

E implementarlo como:

```
public class Greedy extends MH {
    // ...
    @overide
    public Pair<tSolution, tFitness> optimize(Problem problem,
                                              int maxevals);

}
```

3.2 Python/Julia

Explico en un mismo apartado, porque se puede implementar de forma funcional o mediante OO.

Voy a explicar el modelo funcional, ya que el OO es similar al de otros lenguajes ya comentados.

Es muy sencillo, en vez de trabajar recibiendo a un objeto puede recibir directamente una función como parámetro.