Inteligencia Computacional

Práctica 1 - Memoria

Xavier Femenias Seguí
David Carretero García

Introducción.

Esta práctica consiste en diseñar un algoritmo que cree horarios para un centro educativo, donde cada clase y cada profesor tiene una disponibilidad horaria diferente, y es necesario impartir un determinado número de asignaturas unas respectivas horas durante la semana.

El algoritmo deberá devolver el mejor horario posible ajustándose a esas disponibilidades, junto con otras restricciones para dar mayor calidad a los resultados, las denominadas restricciones HARD (restricciones físicas) y SOFT (las impuestas para generar mejores horarios), en nuestra práctica son las siguientes:

- H1: Un profesor no puede estar en dos clases al mismo tiempo
- H2: Un profesor no puede impartir una asignatura si no está en el centro.
- S1: No conviene que una clase haga demasiadas horas en un mismo día.
- S2: Similar a la anterior pero para el horario de los profesores.
- S3: Una asignatura no puede tener más de un número determinado de horas al día, estableciendo este número como parámetro.
- S4: Si una asignatura tiene varias horas en un día, estas deberían ser seguidas.
- S5: Se considerarán mejores los horarios que no tengan huecos.
- S6: Lo mismo que la anterior para el horario de los profesores.

2. Los datos de entrada.

El algoritmo tendrá como entrada un archivo JSON, en donde tendremos una lista de clases, donde cada una especifica, su nombre, su disponibilidad horaria sencillamente con un ‘from’ y un ‘to’ y dentro de ella una lista de las asignaturas que se imparten en ella:

```
{
  "classes": [
    {
      "name": "Ciencia de Datos",
      "availability": {
        "from": 8,
        "to": 15
      },
      "subjects": [
        {
          "name": "Inteligencia Computacional",
          "weekHours": 4,
          "teacher": 1
```

…

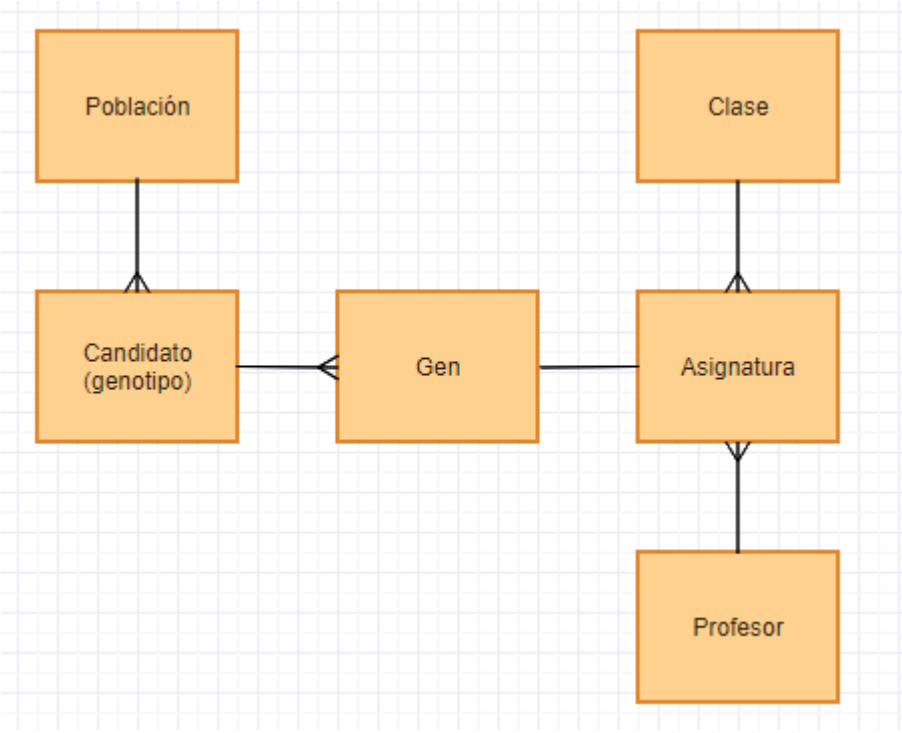
Para los profesores seguimos una estructura similar.

3. Diagrama de clases (modelo de datos)

Dada la entrada citada en el apartado anterior, la clase JsonReader, mediante el método read_input_json() transforma los datos contenidos en input.json en instancias de la clase College_Class (Clase), Subject (Asignatura), Teacher (Profesor) y en sus respectivos horarios.

Más adelante, mediante Evolutive.get_initial_population() se crea la primera población (un conjunto de horarios aleatorios) con un número de genotipos (en adelante candidatos) añadiendo a cada uno un gen por cada asignatura de cada clase. A cada gen se le asignará de manera aleatoria una hora y un día dentro del calendario de la clase a la que pertenece.

De esta manera nos queda el siguiente diagrama de clases:



3.1 Codificación de la solución

Una solución está representada por un **candidato** (un genotipo).

```
# Genotype
class Candidate:

    def __init__(self):
        # Fitness score obtained by the fitness function
        self.fitness = 0
        # Set of genes
        self.calendar = []
```

Un candidato tiene un fitness y un calendario, que es un array de genes, donde cada **gen** representa una asignatura en un slot horario.

```
class Gene:
    def __init__(self, day, hour, subject):
        self.day = day
        self.hour = hour
        self.subject = subject
```

Mediante la asignatura del gen podemos obtener qué profesor la imparte y la clase a la que pertenece, que son datos que utilizaremos a lo largo del algoritmo.

La solución a este problema sería un candidato, que no es más que una lista de genes, con el mayor fitness encontrado. Lo que hace el algoritmo es ir cambiando la hora y el día de cada gen (la asignatura no se toca nunca) hasta llegar a una combinación que dé un fitness óptimo.

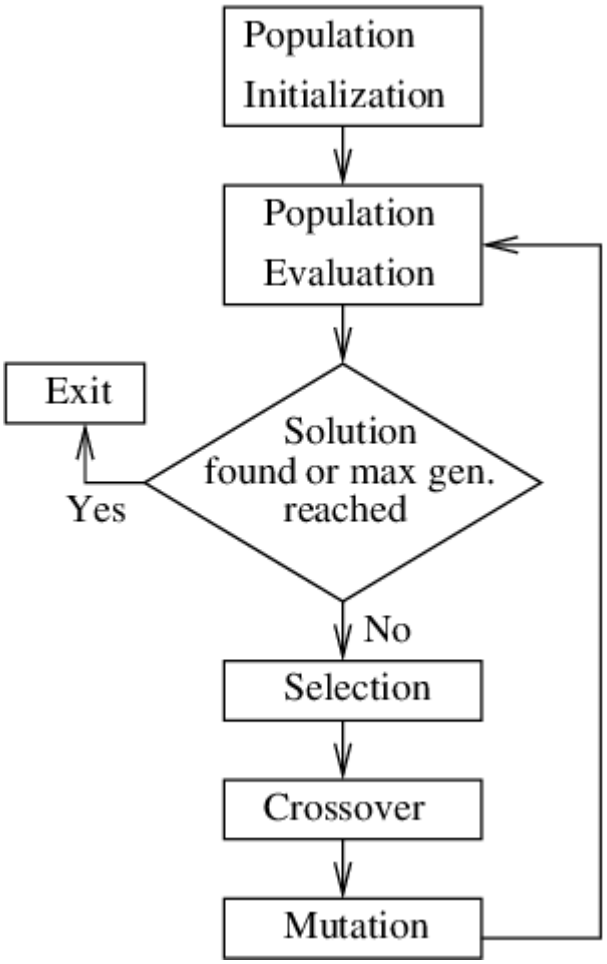
Para trabajar con varias soluciones a la vez se usa una **población**, que no es más que un array de candidatos:

```
class Population:

    def __init__(self):
        self.candidates = []
```

4. Algoritmo evolutivo

El algoritmo utilizado es el que se encuentra en los apuntes. El siguiente gráfico muestra cada uno de los pasos seguidos:



4.1 Inicialización

Se crea una población y se le asignan N candidatos, donde N = *population_size*, que es una variable definida en main.py. A cada candidato se le asigna un gen por cada slot horario de cada clase (de 1 hora) y a cada gen se le asigna un día y hora aleatorios. A la hora de asignar slots horarios a las asignaturas nos aseguramos de que no haya colisiones desde un inicio para mejorar la eficiencia del algoritmo, ya que si no tendríamos casi siempre horarios malos con colisiones, y para ahorrarnos una hard condition que tenga que evaluar si hay asignaturas que colisionan entre sí.

```python
# Initializes the population with candidates with random values
def get_initial_population(self):
    population = Population()

    for _ in range(self.population_size):
        candidate = Candidate()
        for college_class in self.college_classes:
            for subject in college_class.subjects:
                for _ in range(subject.week_hours):
                    day, hour = candidate.get_free_slot(college_class)
                    candidate.calendar.append(Gene(day, hour, subject))

        population.candidates.append(candidate)

    return population
```

4.2 Evaluación

En este apartado se define la función fitness, encargada de dar una puntuación a cada candidato. Esta se divide en varias funciones, una para cada restricción.

Partiremos de una puntuación de cero para el candidato a evaluar. Las **restricciones soft** irán sumando fitness al candidato según la calidad de la solución. La cantidad de puntuación que otorga cada restricción es proporcional a cuánto se cumple la restricción; por ejemplo, si se están evaluando los gaps en el calendario, un candidato que no tenga gaps puntuará mucho, mientras que uno que tenga gaps puntuará menos (1/gaps+1^2). El hecho de que se puntúe a pesar de no ser una solución buena del todo se ha hecho para que el fitness no se haga demasiado pequeño o 0 y así dificulte el proceso de selección como veremos más adelante.

Las **restricciones hard** en lugar de sumar, restan. Son restricciones que indican cuándo un calendario es inviable (p.ej. profesor en 2 aulas a la vez). La manera de restar se ha hecho dividiendo el fitness total del candidato entre 2 (se aplicará la división una vez por cada inconsistencia encontrada). La operación de dividir en lugar de restar se ha escogido porque al restar, las restricciones hard siempre dejan a 0 de fitness la mayoría de candidatos en el inicio del algoritmo, dificultando el distinguir entre candidatos malos y menos malos, mientras que la división no (p.ej. nos permite distinguir entre candidatos con fitness 10 de candidatos con fitness 0).

Cada restricción tiene asociado un hiper parámetro, H/*Sn_score*, que es la cantidad que va a sumar o restar según el caso, el valor de estos parámetros se ha establecido para que cada restricción hard o soft tenga el mismo peso que las otras del mismo tipo. Si una restricción tiene que iterar más veces que otra y por tanto aplicar la suma más veces, esta tendría una puntuación asociada menor.

Todas las funciones tienen un parámetro debug que sirve para que se devuelva el máximo fitness posible (esto se usa para saber la condición de parada del algoritmo).

Empezamos con la restricción H1:

- Un profesor no puede estar en dos clases al mismo tiempo

```
def h1(self, candidate, teacher_list, current_score):
    already_punished_subjects = []

    for teacher in teacher_list:
        calendar_teacher_subjects = [gene for gene in candidate.calendar if gene.subject.teacher == teacher]

        for teacher_subject in calendar_teacher_subjects:
            if all(teacher_subject != aps for aps in already_punished_subjects):
                conflicting_subjects = [ts for ts in calendar_teacher_subjects if ts != teacher_subject
                                                                                  and ts.day == teacher_subject.day
                                                                                  and ts.hour == teacher_subject.hour]
                for _ in range(len(conflicting_subjects)):
                    current_score = math.floor(current_score / 2)

                already_punished_subjects += conflicting_subjects

    return current_score
```

Primero ponemos en una lista los genes (slots horarios, recordemos) asociados a un mismo profesor, para más adelante hacer otro bucle para esos valores. Si hay alguna de ellas que colisione en dia y horario, lo ponemos en la lista "*conflicting_subjects*". Cuando terminamos con un profesor volvemos al inicio y hacemos lo mismo con otro profesor.

A continuación por cada colisión la puntuación se verá penalizada por la mitad.

H2:

- Un profesor no puede impartir una asignatura si no está en el centro.

```
def h2(self, candidate, teacher_list, current_score):
    for teacher in teacher_list:
        calendar_teacher_subjects = [gene for gene in candidate.calendar if gene.subject.teacher == teacher]

        for teacher_subject in calendar_teacher_subjects:
            teacher_day_availability = teacher.get_day_availability(teacher_subject.day)
            if not any(availability_interval.time_from <= teacher_subject.hour and
                       availability_interval.time_to >= teacher_subject.hour
                       for availability_interval in teacher_day_availability):
                current_score = math.floor(current_score / 2)

    return current_score
```

Empezamos como en la restricción anterior, guardando los genes de un mismo profesor durante la semana. Si el horario que encontramos dentro de ese gen está fuera de la disponibilidad horaria del profesor, dividimos entre 2 por tantas veces esto se produzca.

S1:

- No conviene que una clase haga demasiadas horas en un mismo día.

```python
# Soft condition - Penalize class schedules with many hours on the same day.
def s1(self, candidate, college_classes, debug = False):
    score = 0

    for college_class in college_classes:
        for day in range(5):
            day_subjects = [gene for gene in candidate.calendar if gene.subject.college_class == college_class and gene.day == day]
            if debug or len(day_subjects) <= self.max_number_of_subjects_per_day:
                score += self.s1_score

    return score
```

Contamos los genes con la misma clase y el mismo día y los comparamos con el hiper parametro '*max_number_of_subjects_per_day*'.

S2:

- Similar a la anterior pero para el horario de los profesores.

```python
# Soft condition - Penalize teachers schedules with many hours on the same day.
def s2(self, candidate, teacher_list, debug = False):
    score = 0

    for teacher in teacher_list:
        for day in range(5):
            day_subjects = [gene for gene in candidate.calendar if gene.day == day and gene.subject.teacher == teacher]
            if debug or len(day_subjects) <= self.max_teacher_number_of_subjects_per_day:
                score += self.s2_score

    return score
```

Procedemos igual que con la anterior pero haciendo los genes que contienen el mismo profesor y el mismo día.

S3:

- Una asignatura no puede tener más de un número determinado de horas al día, estableciendo este número como parámetro.

```python
# Soft condition - Penalize schedules with days with more than self.max_hours_per_subject_per_day hours of the same subject for students.
def s3(self, candidate, college_classes, debug = False):
    score = 0
    already_checked_genes = []

    for college_class in college_classes:
        for day in range(5):
            class_day_genes = [gene for gene in candidate.calendar if gene.subject.college_class == college_class and gene.day == day]
            day_is_ok = True

            for gene in class_day_genes:
                # Check if the gene has been already checked
                if all(gene != apg for apg in already_checked_genes):
                    same_subject_genes = [ts for ts in class_day_genes if ts.subject == gene.subject]
                    already_checked_genes += same_subject_genes

                    if len(same_subject_genes) > self.max_hours_per_subject_per_day:
                        day_is_ok = False
                        break

            if debug or day_is_ok:
                score += self.s3_score

    return score
```

Ponemos en una lista los genes con el mismo día y la misma clase. A continuación ponemos en '*same_subject_genes*' todos aquellos que contengan el mismo subject, y lo añadimos en '*already_checked_genes*' para no contarlo de nuevo, en la siguiente iteración se comprueba que el gen no esté en dicha lista. Si la longitud de esta lista no supera el hiperparametro '*max_hours_per_subject_day*' sumamos *s3_score* a la puntuación.

S4:

- Si una asignatura tiene varias horas en un día, estas deberían ser seguidas.

```python
# Soft condition - If a subject has hours in a day, they should be followed.
def s4(self, candidate, college_classes, debug = False):
    score = 0
    already_scored_genes = []

    for college_class in college_classes:
        for day in range(5):
            class_day_genes = [gene for gene in candidate.calendar if gene.subject.college_class == college_class and gene.day == day]
            day_gaps = 0

            for gene in class_day_genes:
                # Check if the gene has been already scored
                if all(gene != apg for apg in already_scored_genes):
                    same_subject_genes = [ts for ts in class_day_genes if ts.subject == gene.subject]

                    already_scored_genes += same_subject_genes
                    day_gaps += self.get_number_of_gaps(same_subject_genes)

            if debug:
                day_gaps = 0

            # More gaps, less score
            score += math.floor(self.s4_score / pow(day_gaps + 1, 2))

    return score
```

Ponemos en la lista *class_day_genes* todos los genes con la misma clase y el mismo dia. Y se lo pasamos a la función "*get_number_of_gaps*", que nos dara el numero de separaciones '*day_gaps*'. La puntuación a sumar será inversamente proporcional a (*day_gaps+1)*.

S5 y S6:

- S5: Se considerarán mejores los horarios que no tengan huecos.
- S6: Lo mismo que la anterior para el horario de los profesores.

```python
def s5(self, candidate, college_classes, debug = False):
    score = 0

    for college_class in college_classes:
        for day in range(5):
            class_day_subjects = [gene for gene in candidate.calendar if gene.day == day and gene.subject.college_class == college_class]
            gaps = self.get_number_of_gaps(class_day_subjects)

            if debug:
                gaps = 0

            # More gaps, less score
            score += math.floor(self.s5_score / pow(gaps + 1, 2))

    return score
```

```python
# Soft condition - There are no gaps in the teacher's class schedule.
def s6(self, candidate, teacher_list, debug = False):
    score = 0

    for teacher in teacher_list:
        for day in range(5):
            teacher_day_subjects = [gene for gene in candidate.calendar if gene.day == day and gene.subject.teacher == teacher]
            gaps = self.get_number_of_gaps(teacher_day_subjects)

            if debug:
                gaps = 0

            # More gaps, less score
            score += math.floor(self.s5_score / pow(gaps + 1, 2))

    return score
```

El procedimiento de estas dos es similar a la S4.

4.3 Selección

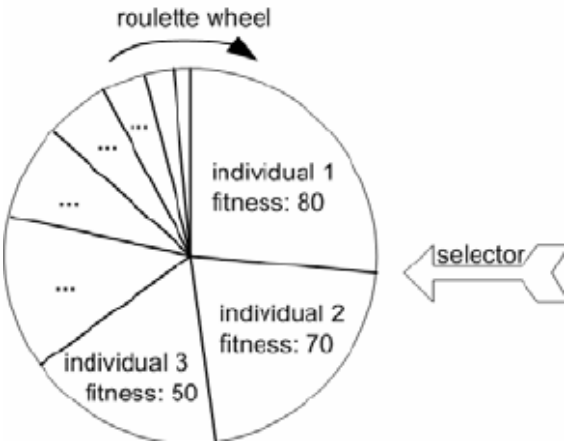
En cada iteración del algoritmo se seleccionan los N mejores candidatos, donde N = selection_size, que es una variable definida en main.py.

Se usa una combinación de 2 métodos para la selección de candidatos:

- Se escogen 1/4 de los candidatos con mejor fitness
- El resto (3/4) de los candidatos se escogen mediante el método de la ruleta

Hemos optado por este enfoque mixto ya que veíamos que usando únicamente la ruleta la aleatoriedad se iba cargando continuamente candidatos con fitness muy bueno, dejando otros candidatos mucho peores. A pesar de que los candidatos con buen fitness tengan más posibilidad de ser escogidos, siempre queda la posibilidad de que se escojan peores y se descarten los buenos, por lo que para proteger a los mejores candidatos los sacamos antes de escoger al resto. Para evitar caer en máximos locales por coger solo los de mejor fitness, únicamente se cogen la cuarta parte de candidatos de esta manera.

El resto de candidatos se escogen siguiendo el algoritmo de la ruleta:



Se suman todos los fitness y se escoge un número aleatorio entre 0 y max_fitness (suma de todos los fitness de los candidatos de la población). Se va iterando sobre los candidatos acumulando su fitness y el candidato cuyo fitness acumulado sobrepase el número es escogido.

El resultado del enfoque mixto mejoró notoriamente los resultados obtenidos.

```python
# Selects #selection_size candidates randomly with a probability equals to candidate_fitness_score/population_total_score
# This function will take the #selection_size/4 candidates with most fitness in order
# to protect the best candidates against randomness. The rest of the elements are chosen
# using the roulette strategy
def select_candidates(self, population):
    selection = []

    # Get the best candidates
    for _ in range(math.ceil(self.selection_size / 4)):
        selection.append(population.pop_best_candidate())

    # Get the rest of the candidates using the roulette method
    for _ in range(self.selection_size - (math.ceil(self.selection_size / 4))):
        fitness_accumulated = 0

        if population.get_total_fitness() > 0:
            rand_number = randrange(population.get_total_fitness())

            for candidate in population.candidates:
                fitness_accumulated += candidate.fitness
                if fitness_accumulated >= rand_number:
                    selection.append(candidate)
                    population.candidates.remove(candidate)
                    break

    selection_size = len(selection)

    # If selection lenght is less than selection_size is because more than #selection_size
    # elements has fitness = 0. Select randomly the rest of the candidates
    if selection_size < self.selection_size:
        selection += random.sample(population.candidates, self.selection_size - selection_size)

    return selection
```

Nota: Si todos los candidatos tienen 0 fitness o no se llega a la selección total, los que faltan por escoger se hacen de manera aleatoria (parte inferior de la función).

4.4 Combinación

En este paso se crean K hijos donde K = *child_creation_rate*, que siempre es equivalente a la mitad del tamaño de la población. Por cada dos padres se crea un hijo.

El procedimiento para crear un hijo es el siguiente:

- Se itera por cada asignatura de cada clase
- Se obtienen todos los genes que contengan la asignatura por la que estamos iterando
- Se asignan la mitad de los genes de un padre al hijo y la mitad del otro padre
- Si hay colisiones se coge el gen conflictivo y se le asigna una hora y día aleatorios que no colisionan con ningún otro gen. Esto se hace así porque si no el algoritmo creaba continuamente candidatos malos con genes que colisionaban y le costaba muchísimo avanzar, quedándose con fitness 0 la mayoría de veces.

```python
# We will create a number of childs equal to the child_creation_rate (it is population_size / 2)
# For every subject at the input Json, we will assign the half of the subjects schedule of one
# parent and the half of the other to the child. If there is a conflict, we will assign a random free slot instead
# in order to create a valid child
def combination(self, population):
    children = []

    for _ in range(self.child_creation_rate):
        parent1, parent2 = random.sample(population.candidates, 2)
        child = Candidate()

        for college_class in self.college_classes:
            for subject in college_class.subjects:
                parent1_subject_genes = [gene for gene in parent1.calendar if gene.subject == subject]
                parent2_subject_genes = [gene for gene in parent2.calendar if gene.subject == subject]

                number_of_hours_of_subject_to_get_from_parent_1 = math.ceil(subject.week_hours / 2)

                parent_genes = random.sample(parent1_subject_genes,
                                             number_of_hours_of_subject_to_get_from_parent_1)
                parent_genes += random.sample(parent2_subject_genes,
                                              subject.week_hours - number_of_hours_of_subject_to_get_from_parent_1)

                for parent_gene in parent_genes:
                    child_gene = parent_gene.clone()

                    # Check if there is a conflict
                    conflicting_gene = child.get_gene(parent_gene.day, parent_gene.hour, parent_gene.subject.college_class)

                    if conflicting_gene:
                        # There is a conflict. Let's assign a random free slot to the parent gene
                        child_gene.day, child_gene.hour = child.get_free_slot(parent_gene.subject.college_class)

                    child.calendar.append(child_gene)

        children.append(child)

    population.candidates += children

    return population.candidates
```

4.5 Mutación

Se escoge un gen al azar y se le asigna una hora y día aleatorios. Si colisiona con otro gen, se hace un swap entre este gen y el afectado.

Se realiza una mutación con una probabilidad de *mutation_rate*, que es una variable definida en main.py.

```python
# Mutates a random gene in every candidate with a probability equals to mutation_rate
def mutation(self, population):
    rand_number = random.random()

    if rand_number <= self.mutation_rate:
        candidate = random.choice(population.candidates)
        random_gene = random.choice(candidate.calendar)
        day = random.choice(range(5))
        hour = random.choice(range(random_gene.subject.college_class.availability.time_from,
                                   random_gene.subject.college_class.availability.time_to))

        # Check if there is a conflict. If so, assign a random free slot to the subject
        conflicting_gene = candidate.get_gene(day, hour, random_gene.subject.college_class)

        if not conflicting_gene:
            random_gene.day = day
            random_gene.hour = hour
        else:
            random_gene.day, random_gene.hour = candidate.get_free_slot(random_gene.subject.college_class)

    return population.candidates
```

4.6 Condición de parada

El algoritmo parará cuando cumpla una de las siguientes condiciones:

- Se haya conseguido un candidato que tenga al menos el 95% del fitness posible total
- Se llegue al máximo número de iteraciones establecido

```python
def end_condition(self, generation, population):
    # 95% of the maximum fitness value is enaugh to consider the solution as optimal
    if any(candidate.fitness >= (self.fitness.get_max_possible_fitness(self.college_classes, self.teacher_list, candidate) * 0.95)
           for candidate in population.candidates):
        return True

    # Print the % of the generations completed
    if round(self.max_generations * self.percentage_completed) == generation:
        print(f"{round(self.percentage_completed * 100)}% completed...")
        self.percentage_completed = self.percentage_completed + 0.05

    return generation == self.max_generations
```

5. Ejecución del programa

Para ejecutar el programa será necesario abrir una consola en la carpeta que se encuentre el archivo main.py y ejecutar el comando: py .\main.py

El programa empezará a ejecutar el algoritmo evolutivo mostrando en primer lugar el máximo fitness posible al que puede llegar un candidato y mostrando por pantalla cada vez que llega a un máximo fitness en uno de los candidatos. También mostrará el porcentaje de iteraciones que lleva ejecutadas para saber cuánto le queda para acabar (en caso de no encontrar una función con fitness >95% del máximo.

Hemos creado 2 ficheros JSON con los inputs del programa. Uno más sencillo: “data/input_easy_mode.json” y otro más complicado “data/input.json”

Por defecto está puesto que lea el más complicado. Para leer el otro archivo hay que cambiar el valor de la variable “data_file” en el archivo “json_reader.py”.

6. Hiperparámetros

Se definen en main.py.

- **population_size = 24**

Se ha escogido un número relativamente alto para que el programa pueda mantener un número considerable de soluciones “buenas” (recordemos que siempre cogemos la 4ª parte de las mejores soluciones). De esta manera protegemos un resultado bueno de la aleatoriedad de la mutación.

- **selection_size = 12**

La selección se hace de la mitad de los candidatos. Una selección de un porcentaje mayor hacía que hubiese poca diversidad ya que se generaban pocos hijos nuevos con posibles soluciones mejores. Un porcentaje más bajo consideramos que descarta demasiados candidatos y se pierden un mayor número de soluciones buenas.

- **mutation_rate = 0.1**

El mutation rate lo hemos establecido empíricamente también, de manera que no destruya demasiadas soluciones buenas pero que favorezca lo suficiente la diversidad de manera que se creen nuevas posibles soluciones.

- **max_generations = 20000**

Hemos visto que la mayoría de soluciones buenas se llegan a generar antes de este valor por lo que poner más iteraciones prácticamente no aporta demasiado al algoritmo.

7. Resultados

En el algoritmo vamos guardando el valor máximo de fitness encontrado hasta el momento. Cada vez que el algoritmo encuentra un candidato con el fitness mayor al último máximo encontrado se crearán 2 archivos .CSV para poder visualizar el resultado:

- teacher_schedule.csv: Es un CSV con el calendario de cada uno de los profesores. Muestra las horas a las que el profesor tiene clase, la asignatura y la clase a la que dará la asignatura. Cada columna representa un día de la semana.
- class_schedule.csv: Es un CSV con el calendario de cada clase. Muestra las horas a las que hay clase, la asignatura que toca y el profesor que la imparte. Cada columna representa un día de la semana.

Las ventajas de ir guardando el calendario cada vez que se llega a un máximo son:

- No se pierden los mejores candidatos encontrados por el camino
- Si el proceso se interrumpe nos quedará siempre el último mejor calendario obtenido (antiguamente se imprimía al final del proceso y corría el riesgo de perderse)

Como comentamos anteriormente, hay 2 ficheros .json con inputs de diferente dificultad. Mientras que el de dificultad alta tarda mucho en conseguir un resultado óptimo y hay que ejecutarlo varias veces para asegurarnos de que es realmente bueno, el fácil tarda muy pocas iteraciones en conseguir un resultado con el 95% de fitness. En la carpeta results hemos dejado un ejemplo de cada uno de los calendarios (el que tiene el sufijo _ez es el calendario fácil y el otro el difícil).