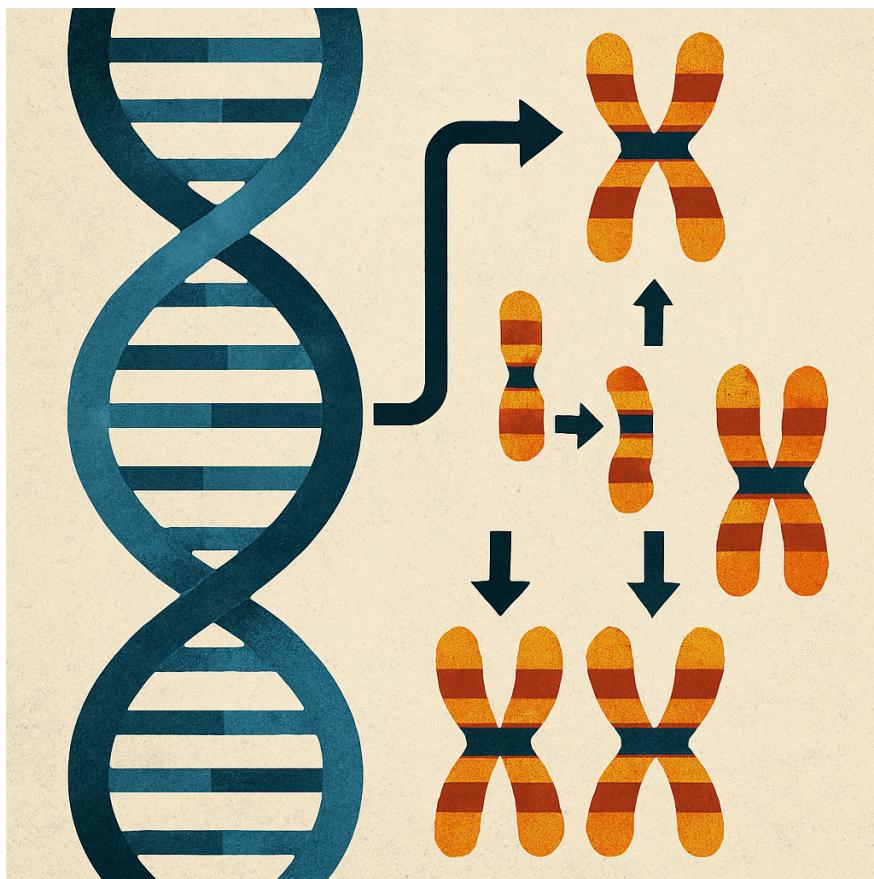


MANUAL TÉCNICO

ALGORITMO GENÉTICO PARA HORARIOS



Introducción

En el presente documento se presenta la documentación técnica del algoritmo genético para la selección de horarios óptimos.

Arquitectura del Proyecto

El proyecto está completamente desarrollado en Python usando las siguientes librerías:

- Pandas: Para la carga de información desde archivos CSV
-

-
- PyQt5: Para el desarrollo de la interfaz de la aplicación
 - ReportLab: Para la generación de reportes PDF
 - PyMuPDF: Para embeber el reporte PDF en la interfaz
 - Psutil: Para monitorear el uso de memoria del proceso
 - Matplotlib: Para la generación y visualización de gráficas

Como se puede asumir por el uso de la librería PyQt5 la aplicación es una de escritorio que conecta con el algoritmo de forma acoplada.

```
└─ src
    └─ interface
        └─ __init__.py
        └─ cursos_layout.py
        └─ docentes_layout.py
        └─ genetic_algorithm_la
        └─ logger.py
        └─ main_layout.py
        └─ pdf_viewer.py
        └─ plot_viewer.py
        └─ relacion_layout.py
        └─ salones_layout.py
    └─ models
        └─ __init__.py
        └─ curso.py
        └─ docente.py
        └─ docente_curso.py
        └─ salon.py
    └─ utils
        └─ __init__.py
        └─ data_handler.py
        └─ genetic_algorithm.py
        └─ pdf_handler.py
        └─ main.py
    └─ .gitignore
    └─ requirements.txt
```

El proyecto está dividido en 3 carpetas principales.

interface

Es la carpeta que contiene todo lo relacionado a la interfaz de usuario con PyQt5, ademas de visualizadores para graficas y PDF.

models

Contiene representaciones de la información usada por el algoritmo, incluyendo cursos, salones, docentes y sus relaciones.

utils

Contiene los archivos con lo necesario para el funcionamiento del algoritmo, incluyendo la carga de archivos a través del data_handler.

También se tiene el pdf_handler que es el encargado de la creación del reporte de PDF con el formato deseado.

Por último se tiene el archivo genetic_algorithm que contiene toda la lógica del algoritmo genético que se explicara en la siguiente sección.

Además hay dos carpetas extra, data y reports que funcionan como almacenamiento de información, ya que ahí se guardan los listados de la información que se usarán para el funcionamiento del algoritmo.



Algoritmo Genético

Ambiente del Algoritmo

Como primera parte el algoritmo genético necesita de un ambiente, este ambiente contiene toda la información para su funcionamiento, incluyendo parámetros y resultados.

```
class AmbienteAlgoritmo:
    def __init__(self):
        self.cursos: list[Curso] = []
        self.salones: list[Salon] = []
        self.docentes = []
        self.relaciones = []
        self.horarios = []
        self.docentes_por_curso: dict[str, list[Docente]] = {}

        self.penalizacion_continuidad: float = 0

        self.generacion_actual: int = 0
        self.total_generaciones: int = 0

        self.resultado: Individuo | None = None
        self.conflictos_por_generacion: list = []
        self.continuidad_por_generacion: list = []
        self.conflictos_mejor_individuo: int = 0
        self.iteraciones_optimas: int = 0
        self.tiempo_ejecucion: float = 0
        self.porcentaje_continuidad: float = 0
        self.memoria_consumida: int = 0
        self.reporte_horarios_pdf: str | None = None
```

Genes y Cromosomas

Para el algoritmo genético se hizo uso de la siguiente estructura para representar a los genes y cromosomas.

```
type Individuo = dict[Curso, tuple[Salon, str, Docente | None]]
```

El type Individuo es un diccionario que relaciona todos los cursos con un salon, horario y docente específico, bajo esta lógica los genes son los cursos, salones, horas y docentes y el cromosoma son los objetos de tipo Individuo que colecciona toda esta información de forma codificada.

La razón para esta estructura fue primeramente para evitar la duplicidad de la información, al usar un diccionario cuya llave era un curso específico y el valor era un horario asignado se evitaba que hubiese en un mismo horario un curso con dos horarios diferentes, el usar esta estructura facilita el manejo de la información. Como segunda razón es la eficiencia, para acceder al horario de un curso en específico no es necesario recorrer todo el horario, basta con saber el curso que se quiere y el diccionario lo obtendrá fácilmente en O(1).

Preparación de la información

El ambiente del algoritmo contiene un método que permite la carga de toda la información al inicio para su posterior uso.

```
def preparar_data(self):
    self.cursos = cargar_cursos("data/cursos.csv")
    self.salones = cargar_salones("data/salones.csv")
    self.docentes = cargar_docentes("data/docentes.csv")
    self.relaciones = cargar_relaciones("data/relaciones_docente_curso.csv")
    self.horarios = ["13:40", "14:30", "15:20", "16:10", "17:00", "17:50", "18:40", "19:30", "20:20", "21:10"]

    for curso in self.cursos:
        self.docentes_por_curso[curso.codigo] = []

    for relacion in self.relaciones:
        # Se busca el docente correspondiente según su registro
        for docente in self.docentes:
            if docente.registro == relacion.registro_docente:
                if docente not in self.docentes_por_curso[relacion.codigo_curso]:
                    self.docentes_por_curso[relacion.codigo_curso].append(docente)
```

Aptitud y Función de Costo

La manera por medio de la cual el algoritmo sabe cuando una solución es apta o no es a través de la función de costo, porque en este caso es más fácil encontrar una solución, respondiendo la pregunta no de que tan buena es una solución, sino que tantos errores tiene, tratando de minimizar la cantidad de los mismos.

```
def funcion_costo(self, individuo: Individuo) -> tuple[float, int, float]:
    penalizacion = 0
    conflictos = 0
    cursos = list(individuo.items())
    for i in range(len(cursos)):
        curso_i, asignacion_i = cursos[i]
        salon_i, hora_i, docente_i = asignacion_i

        # Conflicto si hay un docente en un curso en un horario en el que no trabaja
        if (
            docente_i is not None and
            not docente_i.está_disponible(hora_i)
        ):
            pass
            penalizacion += 5
            conflictos += 1

        for j in range(i + 1, len(cursos)):
            curso_j, asignacion_j = cursos[j]
            salon_j, hora_j, docente_j = asignacion_j

            # Conflicto de salón y horario
            if salon_i == salon_j and hora_i == hora_j:
                penalizacion += 5
                conflictos += 1

            # Conflicto si hay mismo docente en el mismo horario
            if (
                docente_j is not None and
                docente_i == docente_j and
                hora_i == hora_j
            ):
                penalizacion += 1
                conflictos += 1

            # Penalización si hay dos cursos del mismo semestre y carrera en el mismo horario
            if (
                curso_i.semestre == curso_j.semestre and
                curso_i.carrera == curso_j.carrera and
                hora_i == hora_j
            ):
                penalizacion += 1

    peso_continuidad = self.penalizacion_continuidad_dinamica(
        self.generación_actual, self.total_generaciones, self.penalizacion_continuidad)

    porcentaje_continuidad_solución = self.calcular_continuidad(individuo)
    puntuo_continuidad = (porcentaje_continuidad_solución * peso_continuidad) / 100

    # Penalización por falta de continuidad
    penalizacion_continuidad = peso_continuidad - puntuo_continuidad
    penalizacion += penalizacion_continuidad

    return (penalizacion, conflictos, porcentaje_continuidad_solución)
```

De ahí surge la función de costo que penaliza a los diferentes errores que una solución pueda tener, estos errores son conflictos, con horas laborales de un docente, conflictos entre horarios y salones, conflictos entre docentes en mismos horarios y por último la penalización por continuidad.

La penalización por continuidad es un tipo de penalización especial cuyo valor es inversamente proporcional al porcentaje de continuidad de los cursos de una solución, por lo que mientras menos porcentaje de continuidad tiene una solución mayor es la penalización. Siendo que un horario 100% continuo no tiene penalización mientras que uno 0% obtiene todo el punteo de penalización calculado.

La continuidad se calcula del siguiente modo

```
# Se calcula el porcentaje de continuidad que tienen los cursos de un horario
def calcular_continuidad(self, individuo: Individuo) → float:
    grupos = {}

    # Agrupa los cursos según carrera y semestre.
    for curso in individuo:
        key = (curso.carrera, curso.semestre)
        grupos.setdefault(key, []).append(individuo[curso][1])

    suma_continuidad = 0.0
    grupos_validos = 0

    for horas in grupos.values():
        # Convertir cada horario en su índice según self.horarios
        indices = sorted([self.horarios.index(h) for h in horas if h in self.horarios])
        # Solo consideramos grupos con al menos dos cursos
        if len(indices) < 2:
            continue
        total_pares = len(indices) - 1
        consecutivos = 0
        for i in range(1, len(indices)):
            if indices[i] - indices[i - 1] == 1:
                consecutivos += 1
        continuidad = (consecutivos / total_pares) * 100
        suma_continuidad += continuidad
        grupos_validos += 1

    if grupos_validos > 0:
        return suma_continuidad / grupos_validos
    else:
        return 100
```

Se crea un diccionario que agrupa todos los cursos por carrera y semestre, para facilitar el recorrido.

Luego se recorren todos los cursos agrupados y para cada uno se indexan las horas, para saber más fácilmente que horas son adyacentes a otras.

Posteriormente se recorren todos los índices de los cursos y si los índices son adyacentes se suma un punto a la continuidad, si la agrupación es válida, entonces ese semestre-carrera es válido. Luego se saca un promedio de la continuidad por grupo para todos los grupos y esa es la continuidad de la solución.

Un punto importante a tomar en cuenta es que la penalización por continuidad es dinámica, aumenta conforme aumentan las generaciones de este modo.

```
# La penalizacion por la continuidad aumenta dinamicamente conforme pasan las generaciones
def penalizacion_continuidad_dinamica(self, generacion, total_generaciones, peso_inicial, peso_final=50):
    ratio = generacion / total_generaciones
    return peso_inicial + (peso_final - peso_inicial) * ratio
```

La penalización crece desde un valor inicial hasta uno final linealmente, siendo que la primera generación tiene la penalización inicial y la última generación tiene la penalización final. Se dinamizó esta penalización debido a que en generaciones avanzadas ya no había conflictos, pero la continuidad del horario era un problema ya que se estancaba en ciertos valores, dinamizarlo ayuda a mejorar las soluciones.

Creacion de Individuos

La creación de Individuos con sus distintos genes es bastante directa ya que simplemente para cada uno de los cursos que hay se elige al azar un salón, hora y docente dentro de una lista de docentes que se les permite dar el curso (según la relación dada en el CSV).

```
# Creacion de un individuo
def crear_individuo(self) -> Individuo:
    horario_ind: Individuo = {}
    for curso in self.cursos:
        salon = random.choice(self.salones)
        hora = random.choice(self.horarios)
        docentes_permitidos = self.docentes_por_curso.get(curso.codigo, [])
        if docentes_permitidos:
            profesor = random.choice(docentes_permitidos)
        else:
            profesor = None # En caso de que no haya docentes permitidos
        horario_ind[curso] = (salon, hora, profesor)
    return horario_ind
```

Ejecución del algoritmo

La función ejecutar engloba todo lo necesario para el funcionamiento del algoritmo y es el punto de entrada para cualquier inicio del mismo.

Este tiene varios parámetros que se explican más adelante.

```
# se ejecuta el algoritmo
def ejecutar(self, poblacion_inicial, generaciones: int, tasa_mutacion, penalizacion_continuidad,
            conflicto Esperado, evaluar_conflicto,
            continuidad Esperada, evaluar_continuidad,
            penalizacion Esperada, evaluar_penalizacion,
            intervalo_reinsersion = 10, porcentaje_reinsersion = 0.6, fraccion_elite_min = 0.3, fraccion_elite_max = 0.7):
```

Primero se inicia el tracking de tiempo y memoria del algoritmo, esto se hace a través de time de Python y la librería psutil para medir el proceso.

Luego se inicializan las variables que servirán para dar los resultados, esto incluye a la mejor solución, la penalización por continuidad elegida, la generación actual y el total de generaciones que se quieren hacer.

Por último se crea una población inicial completamente aleatoria usando el método de crear individuo cuantas veces se haya indicado.

Se inicializan las variables de evaluación como la cantidad de conflictos, los conflictos por generación y la convergencia que indica la generación en la cual se obtuvo la solución requerida (se asume el peor caso para empezar).

```

start_time = time.time()
process = psutil.Process(os.getpid())

mejor_individuo: Individuo = dict()
self.penalizacion_continuidad = penalizacion_continuidad
self.generacion_actual = 0
self.total_generaciones = generaciones
#print(self.penalizacion_continuidad)
#print(self.generacion_actual)
#print(self.total_generaciones)

# Creación de la población inicial
poblacion = [self.crear_individuo() for _ in range(poblacion_inicial)]

conflictos: int = 0
self.conflictos_por_generacion = []
self.continuidad_por_generacion = []
convergencia = generaciones # Si no converge, asumimos que se realizaron todas las iteraciones

```

Luego se realiza el ciclo principal del algoritmo, se explicara parte por parte.

```

# Ciclo del algoritmo
for generacion in range(generaciones):
    self.generacion_actual = generacion
    Logger.instance().log(f"=====Generacion {generacion}====")
    #tasa_actual = self.tasa_mutacion_dinamica(tasa_mutacion, generacion, generaciones)
    diversidad = self.calcular_diversidad(poblacion)
    umbral_diversidad = 0.01
    tasa_actual = self.tasa_mutacion_adaptativa(tasa_mutacion, generacion, generaciones, diversidad, umbral_diversidad)

    poblacion_evaluada = self.evaluar_poblacion(poblacion)
    menor_penalizacion, conflictos, mejor_individuo, continuidad_actual = poblacion_evaluada[0]
    self.porcentaje_continuidad = continuidad_actual

    self.conflictos_por_generacion.append(conflictos)
    self.continuidad_por_generacion.append(continuidad_actual)

    porcentaje_aptitud = (1 / (1 + menor_penalizacion)) * 100
    Logger.instance().log(f"Aptitud: {porcentaje_aptitud:.5f}% Penalizacion: {menor_penalizacion:.5f} Mutacion: {tasa_actual:.5f} Diversidad: {diversidad:.5f}")

    converge = True

    if evaluar_conflicto and not (conflictos <= conflicto Esperado):
        converge = False

    if evaluar_continuidad and not (continuidad_actual >= continuidad Esperada):
        converge = False

    if evaluar_penalizacion and not (menor_penalizacion <= penalizacion Esperada):
        converge = False

    if converge:
        convergencia = generacion
        break

    nueva_poblacion = self.gerar_poblacion(poblacion_inicial, poblacion, poblacion_evaluada,
                                             fraccion_elite_min, fraccion_elite_max, tasa_actual,
                                             intervalo_reinsersion, porcentaje_reinsersion, diversidad, umbral_diversidad)

    poblacion = nueva_poblacion

```

Primero se actualiza la generación actual en la que nos encontramos.

Luego se calcula la diversidad de la población para ese momento.

- Cálculo de Diversidad y Distancia

Para calcular la diversidad de una población en ese momento se utiliza el siguiente método.

```
# Se calcula la diversidad de una poblacion haciendo un promedio de las distancias entre individuos
def calcular_diversidad(self, poblacion: list[Individuo]) → float:
    if not poblacion:
        return 0

    n = len(poblacion)
    suma_distancias = 0.0
    contador = 0
    for i in range(n):
        for j in range(i + 1, n):
            suma_distancias += self.distancia(poblacion[i], poblacion[j])
            contador += 1
    if contador == 0:
        return 0
    return suma_distancias / contador
```

Para una población se comparan todos los valores unos con otros usando la distancia calculada.

```
# Se calcula la distancia entre dos individuos, esto en base a la cantidad de genes que son iguales
def distancia(self, ind1: Individuo, ind2: Individuo) → float:
    diferencias = 0
    total = len(self.cursos)
    for curso in self.cursos:
        # Compara la asignación de cada curso en ambos individuos
        if ind1.get(curso) ≠ ind2.get(curso):
            diferencias += 1
    return diferencias / total # 0 iguales, 1 distintos
```

La distancia entre dos individuos se calcula revisando cada uno de los cursos y comparando cuales tienen iguales asignaciones, mientras más cursos tengan la misma asignación más parecidos son, por lo que la diversidad indica la distancia promedio entre todos los individuos de una población.

Posteriormente se calcula la tasa de mutación que se quiere para esa población en ese momento.

- Tasa de Mutación

La tasa de mutación es calculada de forma dinámica tomando en cuenta la diversidad y la generación en que se encuentra actualmente.

La tasa de mutación en una situación con suficiente diversidad disminuye linealmente igual que otros parámetros, la primera generación tiene la tasa inicial, y la última generación tiene la tasa final que se definió. Esto para evitar que haya demasiados valores aleatorios por una mutación demasiado alta.

```
# La tasa de mutacion se reduce conforme pasan las generaciones (de forma lineal)
# Si hay poca diversidad se aumenta
def tasa_mutacion_adaptativa(self, tasa_inicial: float, generacion: int, total_generaciones: int,
                               diversidad: float, umbral_diversidad, min_tasa: float = 0.1,
                               potenciador_min = 1, potenciador_max = 8) → float:
    tasa_base = tasa_inicial - (tasa_inicial - min_tasa) * (generacion / total_generaciones)
    potenciador = potenciador_min + (potenciador_max - potenciador_min) * (generacion / total_generaciones)
    # Si la diversidad es muy baja, incrementa la tasa de mutación hasta max_tasa
    #Logger.instance().log(f"Diversidad: {diversidad} Umbral: {umbral_diversidad}")
    if diversidad < umbral_diversidad:
        tasa_base = tasa_base * potenciador
    return tasa_base
```

En casos especiales donde la diversidad es baja se usa un “potenciador” esto es un valor que multiplica la tasa de mutación para aumentarla, este potenciador también es dinámico, para que mientras más avanza más fuerte es este potenciador.

La intención de esta tasa de mutación dinámica es que en las últimas generaciones en donde la continuidad es un problema se pueda mantener una población que busque nuevas opciones para buscar opciones con mejor continuidad, esto se logra manteniendo una diversidad alta mutando más en las últimas generaciones donde la diversidad y (por defecto) la tasa de mutación es baja.

Luego se evalúa toda la población que se tiene en el momento para conocer la puntuación de la misma.

- Evaluacion de Población

La evaluación de la población es bastante simple, se recorren todos los individuos de la misma y se pasa por la función de coste para saber los errores de los

individuos, luego se ordenan de menor a mayor penalización, ya que menor penalización es una mejor solución.

```
# Se evalua la poblacion en base a la funcion costo
def evaluar_poblacion(self, poblacion) -> list[tuple[float, int, Individuo, float]]:
    poblacion_evaluada = []
    for ind in poblacion:
        costo, conflictos, porcentaje_continuidad_solucion = self.funcion_costo(ind)
        poblacion_evaluada.append((costo, conflictos, ind, porcentaje_continuidad_solucion))
    # se ordenan de menor a mayor penalizacion
    poblacion_evaluada.sort(key=lambda tup: tup[0])
    return poblacion_evaluada
```

Luego se obtiene la información de la mejor solución encontrada y se registran los conflictos y la continuidad de la generación actual.

```
poblacion_evaluada = self.evaluar_poblacion(poblacion)
menor_penalizacion, conflictos, mejor_individuo, continuidad_actual = poblacion_evaluada[0]
self.porcentaje_continuidad = continuidad_actual

self.conflictos_por_generacion.append(conflictos)
self.continuidad_por_generacion.append(continuidad_actual)
```

Luego se muestra la información en logs, junto con un porcentaje de aptitud que es sólo **para referencia**.

```
porcentaje_aptitud = (1 / (1 + menor_penalizacion)) * 100
Logger.instance().log(f"Aptitud: {porcentaje_aptitud:.5f}% Penalizacion: {menor_penalizacion:.5f} Mutacion: {tasa_actual:.5f} Continuidad: {continuidad_actual:.5f} Diversidad: {diversidad:.5f}")
```

Luego se verifica si la población actual y el mejor individuo cumplen con los parámetros de convergencia.

- Convergencia

La convergencia no es más que el parámetro que indica si una solución encontrada cumple con los parámetros de evaluación configurados, se dice que converge si los cumple, y eso hace que se acabe el algoritmo.

```
converge = True

if evaluar_conflicto and not (conflictos <= conflicto Esperado):
    converge = False

if evaluar_continuidad and not (continuidad_actual >= continuidad Esperada):
    converge = False

if evaluar_penalizacion and not (menor_penalizacion <= penalizacion Esperada):
    converge = False

if converge:
    convergencia = generacion
    break
```

En caso de que la solución anterior no converja se debe de generar una nueva población para la siguiente generación, esta parte será explicada en la siguiente sección.

```
nueva_poblacion = self.generar_poblacion(poblacion_inicial, poblacion, poblacion_evaluada,
                                             fraccion_elite_min, fraccion_elite_max, tasa_actual,
                                             intervalo_reinsercion, porcentaje_reinsercion, diversidad, umbral_diversidad)

poblacion = nueva_poblacion
```

Luego de que converja o que se acaben las generaciones se termina el benchmarking del algoritmo y se crean los resultados incluidos el mejor horario en PDF.

```
end_time = time.time()

self.resultado = mejor_individuo
self.conflictos_mejor_individuo = conflictos
self.tiempo_ejecucion = end_time - start_time
self.iteraciones_optimas = convergencia
self.porcentaje_continuidad = self.calcular_continuidad(mejor_individuo)
self.memoria_consumida = process.memory_info().rss / (1024 * 1024)

crear_horarios_pdf(self.resultado)
self.reporte_horarios_pdf = os.path.join(os.getcwd(), "reports", "reporte_horarios.pdf")
```

Generación de Población

```
# Se genera una población
def generar_poblacion(
    self, poblacion_inicial, poblacion, poblacion_evaluada,
    fraccion_elite_min, fraccion_elite_max, tasa_mutacion,
    intervalo_reinsercion, porcentaje_reinsercion, diversidad, umbral_diversidad):
    elites = self.obtener_elites()
    poblacion_evaluada, self.generacion_actual, self.total_generaciones, fraccion_elite_min, fraccion_elite_max,
    diversidad, umbral_diversidad)

    nuevos_hijos = []
    while len(nuevos_hijos) < (poblacion_inicial - len(elites)):
        nuevos_hijos.append(self.generar_hijo(poblacion, tasa_mutacion, self.generacion_actual, self.total_generaciones))
    nueva_poblacion = nuevos_hijos

    nuevos_hijos = self.reinsertar_poblacion_adaptativo(intervalo_reinsercion,
                                                       nueva_poblacion, (poblacion_inicial - len(elites)), porcentaje_reinsercion)
    nueva_poblacion = elites + nuevos_hijos

    return nueva_poblacion
```

La generación de una población es una parte compleja del algoritmo para mantener el dinamismo entre todos sus parámetros.

Se usa una combinación entre el elitismo y la reinserción de población, esto para equilibrar el mantener soluciones que son buenas a través de las generaciones y evitar que se caigan en mínimos locales (en especial en cuanto a la continuidad). Cada una de estas partes se explican en las siguientes secciones.

- Elitismo

El elitismo implica que se conserven los mejores individuos para la siguiente generación, esto se hace a través del método `obtener_elites`.

```
# Basado en generaciones, elites y diversidad se calcula la cantidad de individuos conservados como elites
def obtener_elites(self, poblacion_evaluada, generacion, total_generaciones, elite_fraction_min, elite_fraction_max,
                   diversidad, umbral_diversidad):
    ratio = generacion / total_generaciones
    elite_fraction_actual = elite_fraction_min + (elite_fraction_max - elite_fraction_min) * ratio

    # si la diversidad cae bajo el umbral se suaviza la cantidad de elites
    if diversidad < umbral_diversidad:
        factor = diversidad / umbral_diversidad
        elite_fraction_actual = elite_fraction_min + (elite_fraction_actual - elite_fraction_min) * factor

    elite_count = max(1, int(len(poblacion_evaluada) * elite_fraction_actual))
    # Extraer los 'elite_count' mejores individuos (ya ordenados)
    elites = [tup[2] for tup in poblacion_evaluada[:elite_count]]
    Logger.instance().log(f"Generación {generacion}: Se conservan {elite_count} élites (fractions: {elite_fraction_actual:.2f}).")
    return elites
```

Esta función calcula la cantidad de individuos elites que se quieren por generación basado en la generación que se encuentra actualmente, la diversidad de la

población y dos parámetros que indican cual es la fracción mínima y máxima de la población que puede ser elite.

El enfoque quiere que mientras más avancen las generaciones cada vez se conserve una fracción más grande de la población como élites, para mantener la velocidad del algoritmo y la conservación de buenos resultados, al igual que otros parámetros, crece linealmente, la fracción mínima ocurre en la primera generación y la fracción máxima en la última generación.

Este método debe tomar en cuenta también la diversidad de la población ya que existe el problema que en generaciones más avanzadas el algoritmo tiende a caer en mínimos locales, por lo que es necesario aumentar la diversidad, esto implica reducir la cantidad de individuos que no se modifican (elites) ya que el conservar muchos elites al final de las generaciones causa disminución en la diversidad, por lo que si la diversidad cae más bajo que el umbral definido la cantidad de elites conservados se reduce en un factor que es la diferencia entre la diversidad y el umbral de diversidad, esto causa que si la diversidad es baja la cantidad de elites baja también fomentando la diversidad y la mutación de individuos para intentar salir de mínimos locales buscando nuevas opciones.

Luego de obtener los individuos elites se generan hijos para suplir el resto de la población no conservada.

```
nuevos_hijos = []
while len(nuevos_hijos) < (poblacion_inicial - len(elites)):
    nuevos_hijos.append(self.generar_hijo(poblacion, tasa_mutacion, self.generacion_actual, self.total_generaciones))
nueva_poblacion = nuevos_hijos
```

- Generacion de Hijos

La generación de hijos del algoritmo es una parte que utiliza 3 fases, la selección por torneos, cruce adaptativa y luego la mutación del individuo.

```
# Se genera un hijo
def generar_hijo(self, poblacion, tasa_mutacion, generacion, total_generaciones):
    padre1 = self.seleccion_torneo(poblacion)
    padre2 = self.seleccion_torneo(poblacion)
    hijo = self.cruza_adaptativa(padre1, padre2, generacion, total_generaciones)
    hijo = self.mutacion_adaptativa(hijo, tasa_mutacion)
    return hijo
```

Cada parte será explicada en la siguiente sección debido a su complejidad.

Luego de la generación de todos los hijos estos se hace una reinserción de población para mantener la diversidad.

```
nuevos_hijos = self.reinsertar_poblacion_adaptativo(intervalo_reinsercion,
                                                     nueva_poblacion, (poblacion_inicial - len(elites)), porcentaje_reinsercion)
nueva_poblacion = elites + nuevos_hijos
```

- Re inserción de Población

La reinserción de la población se realiza para mantener la diversidad de la población ya que implica el reemplazo de una parte de la población por individuos aleatorios.

```
# Alterna entre la reinserción por diversidad y la reinserción normal
def reinsertar_poblacion_adaptativo(self, intervalo_reinsercion, poblacion, size_poblacion,
                                      porcentaje_reinsercion, umbral_diversidad=0.001):
    diversidad_actual = self.calcular_diversidad(poblacion)
    #Logger.instance().log(f'Reinserción adaptativa: Diversidad {diversidad_actual:.5f} Umbral {umbral_diversidad}.')
    # Si la diversidad cae por debajo del umbral y se cumple el intervalo, se realiza la reinserción
    if diversidad_actual < umbral_diversidad:
        num_reinsertar = int(size_poblacion * porcentaje_reinsercion)
        #Logger.instance().log(f'Reinserción adaptativa: Diversidad {diversidad_actual:.5f} menor a {umbral_diversidad}. Se reintroducen individuos.')
        for _ in range(num_reinsertar):
            nuevo_individuo = self.crear_individuo()
            # Reemplazar al peor individuo
            poblacion[-1] = nuevo_individuo
            poblacion.sort(key=lambda ind: self.funcion_costo(ind)[0])
    elif self.generacion_actual > 0 and self.generacion_actual % intervalo_reinsercion == 0:
        self.reinsertar_poblacion(self.generacion_actual, intervalo_reinsercion, poblacion, size_poblacion, porcentaje_reinsercion)

    return poblacion
```

En una población suficientemente diversa un porcentaje definido de la población se reemplaza cada cierta cantidad de generaciones, la reinserción no es más que generar un número definido de individuos aleatorios, se insertan en la población seleccionada y se reevalúa.

```
# Se reemplaza un porcentaje de la población cada ciertas generaciones por individuos aleatorios para mantener la diversidad
def reinsertar_poblacion(self, generacion, intervalo_reinsercion, poblacion, size_poblacion, porcentaje_reinsercion):
    if generacion > 0 and generacion % intervalo_reinsercion == 0:
        num_reinsertar = int(size_poblacion * porcentaje_reinsercion)
        #Logger.instance().log(f'Reinserción: Se reintroducen {num_reinsertar} individuos aleatorios en la generación {generacion}.')
        for _ in range(num_reinsertar):
            # Generamos un individuo aleatorio
            individuo_random = self.crear_individuo()
            # Reemplazamos el peor individuo (el último en la lista ordenada)
            poblacion[-1] = individuo_random
            # Vuelve a ordenar la población nueva para mantener la mejor solución al inicio
            poblacion.sort(key=lambda ind: self.funcion_costo(ind)[0])

    return poblacion
```

El detalle es que en una población con una diversidad baja se hace este proceso de una manera mucho más compulsiva, ya que si la diversidad se reduce bajo el

umbral el porcentaje de la población es reemplazado sin importar si es una generación en la que se indicó.

Esto debido a que en generaciones más avanzadas se tiende a caer en mínimos locales en especial en la continuidad de los horarios, la única forma de salir de estos de una manera más segura es crear nuevas opciones alejadas de las soluciones indicadas, osea individuos aleatorios, esto en el esfuerzo por que algún individuo contenga algunas soluciones aptas para conseguir una mejor continuidad.

Proceso de Generacion de Hijos

Como se mencionó anteriormente la generación de un hijo comprende 3 fases, la selección por torneo, la cruza adaptativa y la mutación.

```
# Se genera un hijo
def generar_hijo(self, poblacion, tasa_mutacion, generacion, total_generaciones):
    padre1 = self.seleccion_torneo(poblacion)
    padre2 = self.seleccion_torneo(poblacion)
    hijo = self.cruza_adaptativa(padre1, padre2, generacion, total_generaciones)
    hijo = self.mutacion_adaptativa(hijo, tasa_mutacion)
    return hijo
```

El primer paso es seleccionar dos partes a través de una selección por torneo.

- Selección por Torneo

Este tipo de selección es simple ya que se eligen 3 individuos aleatorios de una población y se obtiene al mejor de ellos evaluándolos.

```
# Selección: Se utiliza torneo simple
def seleccion_torneo(self, poblacion, tamano=3):
    #Logger.instance().log(f"Torneo con {tamano} individuos")
    candidatos = random.sample(poblacion, tamano)
    candidatos.sort(key=lambda ind: self.funcion_costo(ind)[0])
    return candidatos[0]
```

Se optó por este torneo para buscar un equilibrio entre mantener buenas opciones al mismo tiempo que se intenta mantener la diversidad dando la opción de que individuos menos aptos puedan transmitir sus genes a la siguiente generación.

- Cruza Adaptativa

La cruza adaptativa es un tipo de cruza que alterna entre la cruza uniforme y la cruza de punto medio a lo largo de las generaciones, para que mientras más avancen las generaciones se beneficie más la cruza uniforme aleatoria a la cruza de punto medio.

```
# Alterna entre la cruza normal y uniforme para mejorar la diversidad
def cruza_adaptativa(self, padre1: Individuo, padre2: Individuo, generacion: int, total_generaciones: int) → Individuo:
    # se calcula el ratio basado en que tan avanzado va el proceso de generación
    ratio = generacion / total_generaciones
    # mientras mas avance el proceso de generacion mas se favorecera la cruza uniforme
    if random.random() < (1 - ratio):
        return self.cruza(padre1, padre2)
    else:
        return self.cruza_uniforme(padre1, padre2)
```

La cruza uniforme favorece la aleatoriedad transmitiendo genes aleatorios de los padres a los hijos mientras que la cruza de punto medio transmite igual cantidad de genes de ambos padres a los hijos.

```
# Cruce: Se realiza un cruce de punto medio para mezclar asignaciones
def cruza(self, padre1, padre2):
    hijo = {}
    punto_cruce = len(self.cursos) // 2
    lista_cursos = list(self.cursos)
    for i, curso in enumerate(lista_cursos):
        if i < punto_cruce:
            hijo[curso] = padre1[curso]
        else:
            hijo[curso] = padre2[curso]
    return hijo

# Se usa un cruce uniforme para mezclar parejo a las asignaciones
def cruza_uniforme(self, padre1: Individuo, padre2: Individuo) → Individuo:
    hijo = {}
    for curso in self.cursos:
        if random.random() < 0.5:
            hijo[curso] = padre1[curso]
        else:
            hijo[curso] = padre2[curso]
    return hijo
```

La idea de favorecer una mayor aleatoriedad conforme avanzan las generaciones es intentar mantener una mayor diversidad y buscar combinaciones nuevas de soluciones para intentar salir de los mínimos locales.

- Mutación Adaptativa

La forma en que el algoritmo muta a los individuos es un proceso que alterna entre la mutación reparadora y la mutación normal aleatoria beneficiando a la mutación normal mientras las generaciones avanzan.

```
def mutacion_adaptativa(self, individuo, tasa_mutacion):
    ratio = self.generacion_actual / self.total_generaciones
    randomNum = random.random()
    if randomNum < (1 - ratio):
        return self.mutacion_reparadora(individuo, tasa_mutacion)
    else:
        return self.mutacion(individuo)
```

Esto con la intención de que en generaciones avanzadas haya mayor diversidad en las opciones al permitir alternativas que quizá no sean tan óptimas, la posible reducción de la calidad y la aparición de conflictos es compensada por el método elitista.

La mutación normal simplemente basada en una probabilidad altera los valores del curso.

```
def mutacion(self, individuo: Individuo, tasa_mutacion=0.1):
    for curso in individuo:
        if random.random() < tasa_mutacion:
            nuevo_salon = random.choice(self.salones)
            nuevo_horario = random.choice(self.horarios)
            docentes_permitidos = self.docentes_por_curso.get(curso.codigo, [])
            if docentes_permitidos:
                nuevo_profesor = random.choice(docentes_permitidos)
            else:
                nuevo_profesor = None
            individuo[curso] = (nuevo_salon, nuevo_horario, nuevo_profesor)
    return individuo
```

- Mutacion Reparadora

Luego de realizar el cruce entre padres se muta el hijo obtenido.

Esta mutación se hace a través de un método llamado mutación reparadora, la mutación reparadora es probablemente la parte más importante del algoritmo ya que aumenta enormemente la calidad de las generaciones porque su objetivo es buscar soluciones que mejoren a los individuos en lugar de realizar mutaciones puramente aleatorias.

```

# Mutación Reparadora
# para cada curso, con cierta probabilidad se prueban varias alternativas y se escoge la que minimice la función de costo.
def mutacion_reparadora(self, individuo: Individuo, tasa_mutacion=0.1, n_alternativas=3) → dict:
    for curso in individuo:
        if random.random() < tasa_mutacion:
            gen_original = individuo[curso]
            mejor_gen = gen_original
            menor_penalizacion,... = self.funcion_costo(individuo)
            # Probar n alternativas
            for _ in range(n_alternativas):
                nuevo_salon = random.choice(self.salones)
                nuevo_horario = random.choice(self.horarios)
                docentes_permitidos = self.docentes_por_curso.get(curso.codigo, [])
                nuevo_profesor = random.choice(docentes_permitidos) if docentes_permitidos else None

                # Asignar temporalmente la nueva opción
                individuo[curso] = (nuevo_salon, nuevo_horario, nuevo_profesor)
                nueva_penalizacion,... = self.funcion_costo(individuo)
                # Se elige la opción que minimiza la penalización de conflictos
                if nueva_penalizacion < menor_penalizacion:
                    menor_penalizacion = nueva_penalizacion
                    mejor_gen = (nuevo_salon, nuevo_horario, nuevo_profesor)

            # Reasignar el mejor gen encontrado
            individuo[curso] = mejor_gen
    return individuo

```

El concepto es simple: se evalúa la aptitud del individuo al que se quiere mutar, en base a esto se genera un número de alternativas para ese individuo habiendo sufrido alguna mutación, todas estas opciones son evaluadas, y si alguna de estas opciones es mejor que el individuo original se devuelve el individuo mutado, pero si no se encuentra una alternativa mejor se devuelve el individuo sin mutar ya que entre las opciones es la mejor, esto debido a que la idea de la mutación reparadora es siempre obtener mutaciones que reducen la penalización o que al menos la mantengan en lugar de aumentarla.

Todo lo anterior compone el funcionamiento del algoritmo genético.

Problemas y Soluciones del Algoritmo

El algoritmo genético que se realizó busca una dinamicidad basada en generaciones y diversidad debido a un problema que surgió al hacer pruebas, el estancamiento en la continuidad de los horarios.

El algoritmo genético a través de la mutación reparadora es capaz de encontrar de forma consistente horarios sin ningún conflicto, es capaz de hacerlo siempre, el problema es que el algoritmo al encontrar un horario sin conflictos muchas veces se estanca en una solución que es casi completamente continua pero el hacerla completamente continua implicaría cambiar muchos cursos a la vez, este es un proceso que el algoritmo no puede resolver sin introducir conflictos a los horarios, algo que impide en gran medida la mutación reparadora y el elitismo, no se eliminaron debido a que completamente aceptable tener

horarios sin ningún conflicto aunque no sean completamente continuos a diferencia tener horarios que sean completamente continuos pero que tengan muchos conflictos.

Aun así es ideal tener ambas, por lo que la solución se basa en la idea de, crear horarios sin conflictos lo más rápido posible, y guardarlos a través del elitismo, mientras que la población no elitista es cada vez más mutada y reemplazada por opciones aleatorias ya que esto beneficia opciones cada vez más diversas que permitan que en algún momento se encuentre una solución con mejor continuidad.

Se puede decir que el algoritmo pasa de una fase acomodada donde su objetivo principal es resolver conflictos, y gradualmente pasa a una fase exploratoria donde una vez se alcanza el objetivo de no tener conflictos el algoritmo empieza a buscar siempre opciones más diversas para encontrar alguna más continua con la idea de que debe estar en otro mínimo local lejano al que se encuentra.

Es necesario aclarar dos puntos, el primero es que debido a la mutación reparadora el algoritmo no es muy rápido, y toma alrededor de 2 minutos en recorrer 100 generaciones y eso lleva al siguiente punto.

El algoritmo **siempre** encuentra una solución sin conflictos, en todas las ejecuciones de este, pero no siempre encuentra soluciones 100% continuas pero sí soluciones con una altísima continuidad (95%+).