Intelligent Algorithm
(BIA601)

S24

STUDENTS:

- Bayan_165593 C2
- Afaf_151093 C2
- shahd_154257 C1
- marim_156262 C1
- ayah_158516 C1
- yousef_158719 C1
- homam_199967 C1

Submitted to: Eng. Issam Salman

رابط الموقع:

https://baihomework.pythonanywhere.com

الرابط على الموقع github:

https://github.com/yousef193/Employee-Scheduling-System

الفهرس

تحليل المشكلة المقترحة وحلها باستخدام الخوارزميات الجينية:

المشكلة: جدولة ساعات عمل الموظفين

وصف المشكلة:

تواجه إدارة الشركة تحديًا في تنظيم ساعات العمل اليومية أو الأسبوعية للموظفين. الهدف هو إعداد جدول عمل فعال يلبي الأهداف التالية:

- تغطية جميع الساعات المطلوبة: ضمان وجود العدد المناسب من الموظفين في كل فترة زمنية لتلبية احتياجات العمل.
- .تقليل الساعات الإضافية: العمل على تقليل ساعات العمل الإضافية للموظفين خارج أوقاتهم المحددة.
- .العدالة في التوزيع: توزيع ساعات العمل بشكل عادل بحيث يحصل جميع الموظفين على ساعات متقاربة قدر الإمكان.
- .مراعاة القيود الشخصية: أخذ التفضيلات والقيود الفردية لكل موظف في الاعتبار، مثل أوقات الراحة أو عدم القدرة على العمل في فترات محددة.

التحديات:

- حجم البيانات الكبير، خاصة مع زيادة عدد الموظفين وتعدد الساعات.
- وجود قيود متعددة يجب أخذها في الاعتبار.
- الحاجة إلى التحسين المستمر للوصول إلى أفضل توزيع ممكن لساعات العمل.

الحل المقترح باستخدام الخوارزميات الجينية:

ما هي الخوارزميات الجينية؟

الخوارزميات الجينية هي تقنية مستوحاة من عملية الانتقاء الطبيعي في علم الأحياء. تعتمد على مفهوم "البقاء للأصلح" بهدف إيجاد حلول مثالية تقريبية عبر محاكاة عمليات التزاوج، الطفرات، والانتقاء الطبيعي.

البنية المقترحة:

تعتمد الحلول على العناصر التالية:

- الكروموسوم :(Chromosome) يمثل جدول عمل متكامل للموظفين.
- الجينات :(Genes) تمثل عدد ساعات العمل لكل موظف.
- القيم : تحدد الجين القيم الزمنية التي يعمل فيها الموظف

دالة الملاءمة :(Fitness Function) تُقيّم جودة الجدول بناءً على عدة معايير:

- تحقيق تغطية كاملة للساعات المطلوبة.
- تقليل الساعات غير الفعالة.
- تخفيض عدد الساعات الإضافية إلى الحد الأدنى.

إلتزام القيود والتفضيلات الشخصية:

العمليات الجينية:

- التزاوج (Crossover): يتم اختيار جداول عمل من الجيل الحالي ودمجها لإنتاج جداول جديدة، حيث يتم تبادل أجزاء من الجداول بين بعضها البعض.
- الطفرات (Mutation): إدخال تغييرات طفيفة على الجداول لتحسين النتائج وتجنب الحلول المتكررة.
- الانتقاء الطبيعي (Selection): اختيار أفضل الجداول بناءً على دالة الملاءمة، واستبعاد الجداول ذات الأداء الأضعف.

تحسين الحل:

العملية تتكرر على عدة أجيال لضمان الوصول إلى أفضل توزيع لساعات العمل

شرح الخوارزمية الجينية بالتفصيل:

المدخلات:

- تحديد عدد الموظفين وأوقات راحتهم (مثل الحد الأقصى لساعات العمل لكل موظف).
- إدخال البيانات الخاصة بالساعات المطلوبة يوميًا لكل فترة زمنية.
- اختيار عدد الأجيال (Generations) لتحسين الحل.
- تحديد حجم السكان (Population Size) أي عدد الجداول التي يتم توليدها في كل جيل.

المخرجات:

- جدول عمل يُحقق تغطية كاملة للساعات المطلوبة، مع مراعاة القيود الشخصية لكل موظف.

خطوات تنفيذ الخوارزمية:

١. توليد الجيل الأول:

يتم إنشاء مجموعة من جداول العمل عشوائيًا (بحجم السكان المحدد).

٢. حساب دالة الملاءمة:

تقييم جودة كل جدول عمل بناءً على مدى تحقيقه لتغطية الساعات المطلوبة وتقليل الساعات الزائدة لكل موظف.

٣. اختيار الحلول الأفضل:

يتم انتقاء الجداول ذات أعلى قيم ملاءمة للمشاركة في العمليات الجينية المستقبلية.

٤. التزاوج والطفرات:

دمج جداول العمل المختارة لإنتاج جداول جديدة باستخدام آلية التزاوج.

إدخال تغييرات طفيفة على بعض الجداول لتحسين التنوع وضمان استكشاف حلول جديدة.

٥. تكرار العملية:

يتم توليد جيل جديد ، ثم تُعاد العمليات من الخطوات (٢) إلى (٤) لعدد معين من الأجيال، أو حتى يتم الوصول إلى حل يحقق معايير القبول المحددة مسبقاً.

٦. إخراج النتيجة:

يتم تحديد الجيل الذي يمتلك أعلى قيمة ملاءمة كحل نهائي للمشكلة.

تقييم الخوارزمية:

المزايا:

١. القدرة على التعامل مع المشكلات المعقدة التي تحتوي على قيود متعددة.

٢. تحسين الحلول بشكل مستمر.

٣. سهولة التعديل لتلبية احتياجات متنوعة.

العيوب:

١. قد تستغرق وقتًا طويلًا في حال كان حجم السكان كبيرًا أو عدد الأجيال مرتفعًا.

٢. في بعض الحالات، قد لا تحقق الحل الأمثل.

الخلاصة:

المشكلة التي تم طرحها هي كيفية جدولة ساعات العمل للموظفين بطريقة تحقق توازنًا بين تغطية الساعات، العدالة بين الموظفين، وتقليل الساعات الإضافية. باستخدام الخوارزميات الجينية، يمكن تصميم حل فعال يتم تحسينه بشكل تدريجي للوصول إلى جدول عمل مثالي يلبي كافة الشروط والقيود. تتمتع الخوارزميات الجينية بمرونة كبيرة وقدرة على التكيف مع مختلف التحديات، مما يجعلها خيارًا مثاليًا لحل هذه المشكلة.

الكود التالي يعرض لنا التصميم الأولي لحل المشكلة التي تم اقتراحها سابقا :

```
import random

# Define the problem constraints
EMPLOYEES = ["Emp1", "Emp2", "Emp3", "Emp4", "Emp5"]
HOURS = 24  # Total hours to cover in a day
MAX_HOURS_PER_EMPLOYEE = 8  # Maximum hours an employee can work in a day
MIN_HOURS_PER_EMPLOYEE = 4  # Minimum hours an employee should work in a day
POPULATION_SIZE = 100  # Number of schedules in a generation
GENERATIONS = 500  # Number of generations to evolve
MUTATION_RATE = 0.1  # Probability of mutation

# Generate initial population
def generate_schedule():
    schedule = {}
    for employee in EMPLOYEES:
        schedule[employee] = random.randint(MIN_HOURS_PER_EMPLOYEE,
MAX_HOURS_PER_EMPLOYEE)
    return schedule

def generate_population():
    return [generate_schedule() for _ in range(POPULATION_SIZE)]

# Fitness function
def fitness(schedule):
    total_hours = sum(schedule.values())
    fairness = len(set(schedule.values()))  # Higher is better
    if total_hours < HOURS:
        return 0  # Penalize schedules that don't cover all hours
    return 1 / (1 + abs(total_hours - HOURS)) + fairness

# Selection function
def select_population(population):
    weighted_population = [(schedule, fitness(schedule)) for schedule in population]
    weighted_population.sort(key=lambda x: x[1], reverse=True)
    return [schedule for schedule, _ in weighted_population[:POPULATION_SIZE // 2]]

# Crossover function
def crossover(parent1, parent2):
    child = {}
    for employee in EMPLOYEES:
        child[employee] = random.choice([parent1[employee], parent2[employee]])
    return child









# Mutation function
```

```
def mutate(schedule):
    if random.random() < MUTATION_RATE:

      employee = random.choice(EMPLOYEES)
        schedule[employee] = random.randint(MIN_HOURS_PER_EMPLOYEE,
MAX_HOURS_PER_EMPLOYEE)


# Genetic algorithm implementation
def genetic_algorithm():
    population = generate_population()

    for generation in range(GENERATIONS):
        selected_population = select_population(population)
        new_population = []

        # Crossover
        while len(new_population) < POPULATION_SIZE:
            parent1 = random.choice(selected_population)
            parent2 = random.choice(selected_population)
            child = crossover(parent1, parent2)
            mutate(child)
            new_population.append(child)

        population = new_population

        # Check for the best solution in the current generation
        best_schedule = max(population, key=fitness)
        print(f"Generation {generation + 1}: Best Fitness = {fitness(best_schedule):.2f}")

        if fitness(best_schedule) > 0.99:  # Early stopping condition
            break

    return best_schedule


# Run the genetic algorithm
best_schedule = genetic_algorithm()
print("\nBest Schedule Found:")
for employee, hours in best_schedule.items():
    print(f"{employee}: {hours} hours")
```

شرح الكود السابق :

تعريف قيود المشكلة:

-يتم تعريف قائمة بالموظفين الذين سيتم توزيع ساعات العمل عليهم.

-يتم تحديد العدد الإجمالي للساعات المطلوب تغطيته خلال اليوم (٢٤ ساعة).

-يتم نحديد حدود ساعات العمل لكل موظف، بما في ذلك الحد الأقصى والحد الأدنى للساعات اليومية المسموح بها.

إنشاء الجيل الأول من الحلول (Population) :

-يتم إنشاء جدول عمل عشوائي لكل موظف، حيث يتم تخصيص عدد ساعات عشوائي لكل موظف ضمن الحدود المحددة (الحد الأدنى والأقصى).

-تتكرر هذه العملية لإنشاء مجموعة من الجداول (Population)، والتي تمثل الجيل الأول من الحلول التي سيتم تحسينها لاحقًا.

حساب جودة الحلول (Fitness Function):

-يتم تقييم جودة كل جدول عمل باستخدام دالة الملاءمة.

-تعتمد دالة الملاءمة على عدة معايير أساسية:

١.تغطية جميع الساعات المطلوبة: إذا كان مجموع ساعات العمل في الجدول أقل من ٢٤ ساعة، يتم منح الدرجة الأدنى.

٢.العدالة في التوزيع: تُمنح درجات أعلى للجداول التي تتمتع بتوزيع متساوٍ نسبيًا لساعات العمل بين الموظفين.

٣.الاقتراب من عدد الساعات المطلوب :الحلول الأقرب إلى العدد المطلوب للساعات تحصل على درجات أعلى.

اختيار أفضل الحلول Selection:

١. يتم تقييم كافة الجداول باستخدام دالة التناسب.
٢. يتم ترتيب الجداول من الأفضل الى الأسوأ استناداً الى قيم دالة التناسب
٣. يتم اختيار أفضل نصف من الجداول ليكونوا أساساً لتكوين الجيل الثالي

انشاء الجيل الجديد:

١. يتم إنشاء جداول جديدة باستخدام عمليتي التزاوج (Crossover) والطفرات :(Mutation)
   ١. **التزاوج**: يتم اختيار جدولين عشوائيًا من أفضل الجداول، ثم دمج ساعات العمل الخاصة بهما لتكوين جدول عمل جديد.
   ٢. **الطفرات**: يتم تعديل جدول العمل الناتج بشكل عشوائي بين الحين والآخر لتجربة حلول جديدة، مثل تعديل ساعات العمل لموظف واحد.
٢. يتم تكرار هذه العمليات حتى يتم إنشاء الجيل الجديد بالكامل.

تكرار العملية عبر الأجيال:

١. يتم تقييم الجيل الجديد بنفس الأسلوب، وتكرار الخطوات السابقة لتحسين الحلول بشكل تدريجي.
٢. في كل جيل، يتم اختيار الجدول الذي يمتلك أفضل قيمة ملاءمة ليكون الحل الأمثل لذلك الجيل.

الانـهاء:

١. تنتهي العملية عند الوصول إلى عدد محدد من الأجيال، أو عند العثور على جدول عمل مثالي يفي بجميع القيود مع تحقيق درجات ملاءمة مرتفعة للغاية.

عرض الجدول الأفضل:

١. في النهاية، يتم اختيار الجدول الذي يمتلك أفضل قيمة ملاءمة من بين جميع الأجيال ويُعرض كالحل النهائي.
٢. يتم عرض توزيع ساعات العمل لكل موظف بشكل مفصل وواضح.

ملخص عمل الخوارزمية:

١. يتم استخدام الأجيال لتحسين الحلول بشكل تدريجي، حيث يتم استبعاد الجداول الأقل كفاءة واستبدالها بجداول جديدة تعتمد على أفضل الجداول السابقة.
٢. تساعد العمليات العشوائية مثل التزاوج والطفرات في استكشاف حلول جديدة، مما يقلل من احتمالية الوصول إلى حلول غير مثالية.

البدء بعمل تطبيق flask من أجل الخوارزمية السابقة:

تم تصميم تطبيق Flask متكامل يتضمن الكود الأساسي لحل مشكلة الجدولة باستخدام الخوارزمية الجينية.

مزايا التطبيق:

١. يتيح للمستخدم إدخال تفاصيل مثل أسماء الموظفين، عدد الساعات الإجمالية، والحد الأدنى والأقصى لساعات العمل لكل موظف.
٢. يقوم التطبيق بتشغيل الخوارزمية الجينية لحساب أفضل جدول عمل استنادًا إلى المدخلات.
٣. يعرض الجدول الناتج على واجهة ويب تفاعلية لسهولة التفاعل مع المستخدم.

الكود:

```
from flask import Flask, render_template, request, jsonify
import random

# Flask app setup
app = Flask(__name__)


# Genetic Algorithm Implementation
EMPLOYEES = []
HOURS = 24
MAX_HOURS_PER_EMPLOYEE = 8
MIN_HOURS_PER_EMPLOYEE = 4
POPULATION_SIZE = 100
GENERATIONS = 500
MUTATION_RATE = 0.1


# Generate initial population
def generate_schedule():
    schedule = {}
    for employee in EMPLOYEES:
        schedule[employee] = random.randint(MIN_HOURS_PER_EMPLOYEE,
MAX_HOURS_PER_EMPLOYEE)
    return schedule

def generate_population():
    return [generate_schedule() for _ in range(POPULATION_SIZE)]




# Fitness function
def fitness(schedule):
    total_hours = sum(schedule.values())
```

```
    fairness = len(set(schedule.values()))
    if total_hours < HOURS:
        return 0
    return 1 / (1 + abs(total_hours - HOURS)) + fairness

# Selection function
def select_population(population):
    weighted_population = [(schedule, fitness(schedule)) for schedule in population]
    weighted_population.sort(key=lambda x: x[1], reverse=True)
    return [schedule for schedule, _ in weighted_population[:POPULATION_SIZE // 2]]

# Crossover function
def crossover(parent1, parent2):
    child = {}
    for employee in EMPLOYEES:
        child[employee] = random.choice([parent1[employee], parent2[employee]])
    return child

# Mutation function
def mutate(schedule):
    if random.random() < MUTATION_RATE:
        employee = random.choice(EMPLOYEES)
        schedule[employee] = random.randint(MIN_HOURS_PER_EMPLOYEE,
MAX_HOURS_PER_EMPLOYEE)

# Genetic algorithm implementation
def genetic_algorithm():
    population = generate_population()

    for generation in range(GENERATIONS):
        selected_population = select_population(population)
        new_population = []

        while len(new_population) < POPULATION_SIZE:
            parent1 = random.choice(selected_population)
            parent2 = random.choice(selected_population)
            child = crossover(parent1, parent2)
            mutate(child)
            new_population.append(child)

        population = new_population

        best_schedule = max(population, key=fitness)
        if fitness(best_schedule) > 0.99:
            break

    return best_schedule


# Flask Routes
@app.route('/')
def index():
```

```
    return render_template('index.html')

@app.route('/schedule', methods=['POST'])
def schedule():
    global EMPLOYEES, HOURS, MAX_HOURS_PER_EMPLOYEE, MIN_HOURS_PER_EMPLOYEE

    # Get inputs from the form
    EMPLOYEES = request.json['employees']
    HOURS = int(request.json['hours'])
    MAX_HOURS_PER_EMPLOYEE = int(request.json['max_hours'])
    MIN_HOURS_PER_EMPLOYEE = int(request.json['min_hours'])

    # Run genetic algorithm
    best_schedule = genetic_algorithm()

    # Return the best schedule as JSON
    return jsonify(best_schedule)

if __name__ == '__main__':
    app.run(debug=True)
```

التطوير والتحسين والتقرير النهائي عن المشكلة:

التطبيق المحسن:

```
from flask import Flask, render_template, request, jsonify
import random

# Flask app setup
app = Flask(__name__)


# Genetic Algorithm Implementation
EMPLOYEES = []
HOURS = 24
MAX_HOURS_PER_EMPLOYEE = 8
MIN_HOURS_PER_EMPLOYEE = 4
POPULATION_SIZE = 100
GENERATIONS = 500
MUTATION_RATE = 0.1




# Generate initial population
def generate_schedule():
    schedule = {}
```

```
    for employee in EMPLOYEES:
        schedule[employee] = random.randint(MIN_HOURS_PER_EMPLOYEE,
MAX_HOURS_PER_EMPLOYEE)
    return schedule


def generate_population():
    return [generate_schedule() for _ in range(POPULATION_SIZE)]


# Fitness function
def fitness(schedule):
    total_hours = sum(schedule.values())
    fairness = len(set(schedule.values()))
    if total_hours < HOURS:
        return 0
    return 1 / (1 + abs(total_hours - HOURS)) + fairness


# Selection function
def select_population(population):
    weighted_population = [(schedule, fitness(schedule)) for schedule in population]
    weighted_population.sort(key=lambda x: x[1], reverse=True)
    return [schedule for schedule, _ in weighted_population[:POPULATION_SIZE // 2]]


# Crossover function
def crossover(parent1, parent2):
    child = {}
    for employee in EMPLOYEES:
        child[employee] = random.choice([parent1[employee], parent2[employee]])
    return child


# Mutation function
def mutate(schedule):
    if random.random() < MUTATION_RATE:
        employee = random.choice(EMPLOYEES)
        schedule[employee] = random.randint(MIN_HOURS_PER_EMPLOYEE,
MAX_HOURS_PER_EMPLOYEE)









# Genetic algorithm implementation
def genetic_algorithm():
```

```
    population = generate_population()

    for generation in range(GENERATIONS):
        selected_population = select_population(population)
        new_population = []

        while len(new_population) < POPULATION_SIZE:
            parent1 = random.choice(selected_population)
            parent2 = random.choice(selected_population)
            child = crossover(parent1, parent2)
            mutate(child)
            new_population.append(child)

        population = new_population

        best_schedule = max(population, key=fitness)
        if fitness(best_schedule) > 0.99:
            break

    return best_schedule

# Flask Routes
@app.route('/')
def index():
    return render_template('index.html')

@app.route('/schedule', methods=['POST'])
def schedule():
    global EMPLOYEES, HOURS, MAX_HOURS_PER_EMPLOYEE, MIN_HOURS_PER_EMPLOYEE

    # Get inputs from the form
    EMPLOYEES = request.json['employees']
    HOURS = int(request.json['hours'])
    MAX_HOURS_PER_EMPLOYEE = int(request.json['max_hours'])
    MIN_HOURS_PER_EMPLOYEE = int(request.json['min_hours'])

    # Run genetic algorithm
    best_schedule = genetic_algorithm()

    # Return the best schedule as JSON
    return jsonify(best_schedule)

if __name__ == '__main__':
    app.run(debug=True)
```

صفحة index.html:

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Employee Scheduling</title>
    <!-- Bootstrap CSS -->
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-
alpha3/dist/css/bootstrap.min.css" rel="stylesheet">
    <script src="https://code.jquery.com/jquery-3.6.0.min.js"></script>
</head>

<body>
    <div class="container mt-5">
        <div class="text-center mb-4">
            <h1>Employee Scheduling System</h1>
            <p class="lead">This application uses a genetic algorithm to solve the
scheduling problem by optimizing
                employee working hours to cover a specified total. Enter the details below
to get started.</p>
        </div>

        <div class="card p-4 shadow">
            <form id="scheduleForm">
                <div class="mb-3">
                    <label for="employees" class="form-label">Employee Names (comma
separated):</label>
                    <input type="text" class="form-control" id="employees"
name="employees"
                        placeholder="e.g., Alice, Bob, Charlie" required>
                </div>

                <div class="row">
                    <div class="col-md-6 mb-3">
                        <label for="hours" class="form-label">Total Hours to
Cover:</label>
                        <input type="number" class="form-control" id="hours" name="hours"
placeholder="e.g., 24"
                            required>
                    </div>
                    <div class="col-md-6 mb-3">
                        <label for="max_hours" class="form-label">Max Hours Per
Employee:</label>
                        <input type="number" class="form-control" id="max_hours"
name="max_hours" placeholder="e.g., 8"
                            required>
                    </div>
                </div>

                <div class="mb-3">
```

```
                    <label for="min_hours" class="form-label">Min Hours Per
Employee:</label>
                    <input type="number" class="form-control" id="min_hours"
name="min_hours" placeholder="e.g., 4"
                           required>
                </div>

                <button type="submit" class="btn btn-primary w-100">Generate
Schedule</button>
            </form>
        </div>

        <div class="mt-5">
            <h2>Generated Schedule</h2>
            <div id="output" class="alert alert-info" role="alert">The schedule will
appear here after generation.</div>
        </div>
    </div>

    <!-- Bootstrap JS -->
    <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-
alpha3/dist/js/bootstrap.bundle.min.js"></script>

    <script>
        $("#scheduleForm").on("submit", function (event) {
            event.preventDefault();

            const data = {
                employees: $("#employees").val().split(",").map(e => e.trim()),
                hours: $("#hours").val(),
                max_hours: $("#max_hours").val(),
                min_hours: $("#min_hours").val()
            };

            $.ajax({
                url: '/schedule',
                method: 'POST',
                contentType: 'application/json',
                data: JSON.stringify(data),
                success: function (response) {
                    $("#output").removeClass('alert-info').addClass('alert-
success').text(JSON.stringify(response, null, 2));
                },
                error: function () {
                    $("#output").removeClass('alert-success').addClass('alert-
danger').text('An error occurred while generating the schedule.');
                }
            });
        });
    </script>
</body>
```

```
</html>
```

كيفية عمل التطبيق:

أ. المدخلات:

- أسماء الموظفين (تفصل الأسماء باستخدام الفاصلة).
- إجمالي ساعات العمل المطلوبة.
- الحد الأقصى لساعات العمل المسموح بها لكل موظف.
- الحد الأدنى لساعات العمل المسموح بها لكل موظف.

ب. الخوارزمية الجينية:

الخوارزمية الجينية تُستخدم لحل مشكلة جدولة الساعات باستخدام الخطوات التالية:

١. **توليد السكان الأوليين** :يتم إنشاء مجموعة من الجداول العشوائية استنادًا إلى المدخلات التي يقدمها المستخدم.
٢. **حساب اللياقة** :يتم تقييم كل جدول بناءً على مدى قربه من تحقيق الهدف (إجمالي الساعات) ومدى عدالة توزيع الساعات بين الموظفين.
٣. **الاختيار** :يتم اختيار الجداول ذات اللياقة العالية لتشكيل الجيل التالي من الجداول.
٤. **التزاوج والتكاثر** :يتم إنشاء جداول جديدة من خلال دمج جداول مختارة باستخدام عملية التزاوج.(Crossover)
٥. **التغيير العشوائي** :**(Mutation)** يتم تعديل بعض الجداول عشوائيًا لتعزيز التنوع في الأجيال القادمة، بهدف تجنب الوقوع في حلول غير مثلى.
٦. **التكرار** :تتكرر العمليات السابقة حتى يتم الوصول إلى جدول مثالي يفي بجميع القيود أو حتى يتم بلوغ العدد المحدد من الأجيال.

ج. الإخراج:

- **جدول يوضح توزيع الساعات المخصصة لكل موظف** بحيث يحقق المتطلبات المحددة (إجمالي الساعات، العدالة في التوزيع، وأي قيود أخرى).

تفاصيل التطبيق:

أ. الجزء الخلفي:(Backend)

- يعتمد التطبيق على إطار عمل **Flask** لبناء واجهة برمجة التطبيقات.(API)
- يتم تنفيذ الخوارزمية الجينية داخل كود **Python** لمعالجة المدخلات وحساب النتائج.
- يتم استخدام وظيفة **genetic_algorithm()** لتشغيل العملية الرئيسية للخوارزمية الجينية، التي تُرجع الجدول الأمثل بناءً على المدخلات.

ب. الجزء الأمامي:(Frontend)

- تم تصميم واجهة المستخدم باستخدام مكتبة **Bootstrap** لتوفير تجربة استخدام حديثة وسهلة.
- يتيح النموذج **(Form)** للمستخدم إدخال البيانات المطلوبة، ويتم إرسال هذه البيانات إلى الخادم باستخدام **AJAX** لضمان تجربة تفاعلية وسريعة دون الحاجة لإعادة تحميل الصفحة.
- يتم عرض الجدول الناتج في واجهة المستخدم بشكل واضح وسهل القراءة.

كيفية التفاعل مع التطبيق:

أ. إدخال البيانات:

١. أدخل **أسماء الموظفين** في الحقل المخصص، مفصولة بفواصل.
٢. أدخل **العدد الإجمالي لساعات العمل** المطلوبة في الحقل المخصص.
٣. حدد **الحد الأقصى** و **الحد الأدنى** لساعات العمل لكل موظف.
٤. اضغط على زر **"Generate Schedule"** لتوليد الجدول.

ب. معالجة البيانات:

- يتم إرسال البيانات المدخلة إلى الخادم عبر **طلب POST** .
- يتم تشغيل الخوارزمية الجينية على البيانات المدخلة للحصول على **الجدول الأمثل** بناءً على المعايير المحددة.

ج. عرض النتائج:

- يتم عرض النتائج في قسم خاص بالواجهة كجدول يعرض **الساعات المخصصة لكل موظف**.
- في حال حدوث خطأ أثناء عملية المعالجة، يتم عرض **رسالة خطأ** توضح المشكلة للمستخدم.

مميزات التطبيق:

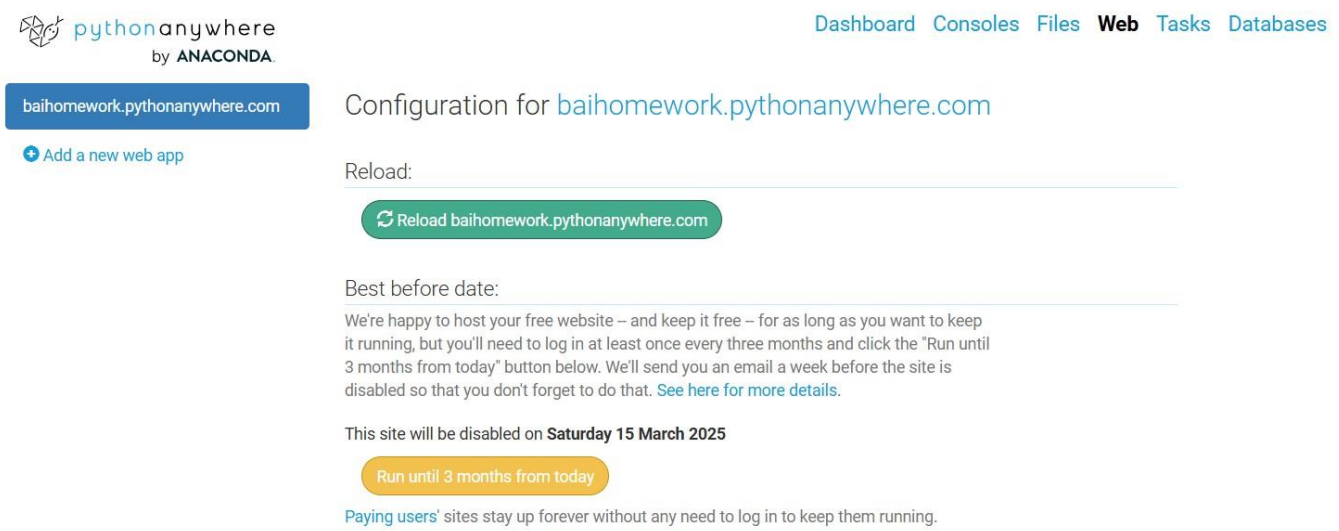
- **مرونة عالية:** يتيح إدخال البيانات بسهولة مع إمكانية تخصيص القيود حسب الحاجة.
- **أداء سريع:** يعالج البيانات بكفاءة ويجد الحلول المثلى في وقت قصير.
- **واجهة مستخدم عصرية:** تصميم حديث وبسيط يجعل التفاعل مع التطبيق مريحًا وسهلًا.
- **تقنيات متطورة:** يعتمد على تقنيات حديثة مثل AJAX لتحسين استجابة التطبيق وتجربة المستخدم.

الاستخدامات المحتملة:

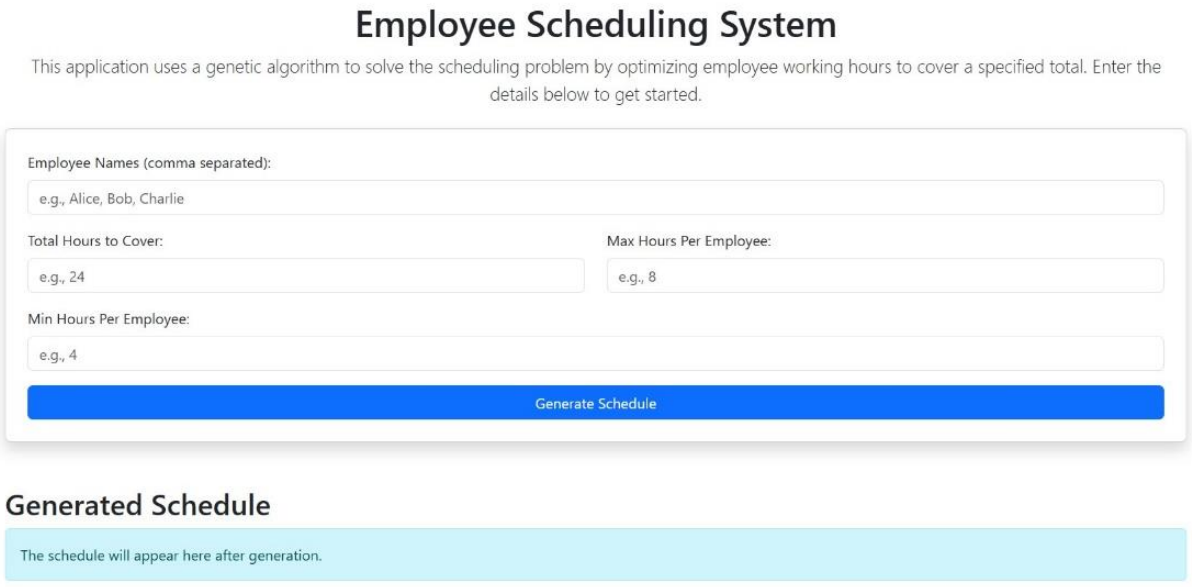
- **إدارة الجداول الزمنية:** تنظيم جداول العمل اليومية أو الأسبوعية بفعالية.
- **تخصيص المهام:** توزيع المهام في بيئات العمل المختلفة بما يناسب الاحتياجات.
- **تحسين الأداء:** تعزيز كفاءة توزيع العمل وضمان العدالة بين العاملين .

المراحل العملية لنشر التطبيق على استضافة مجانية:

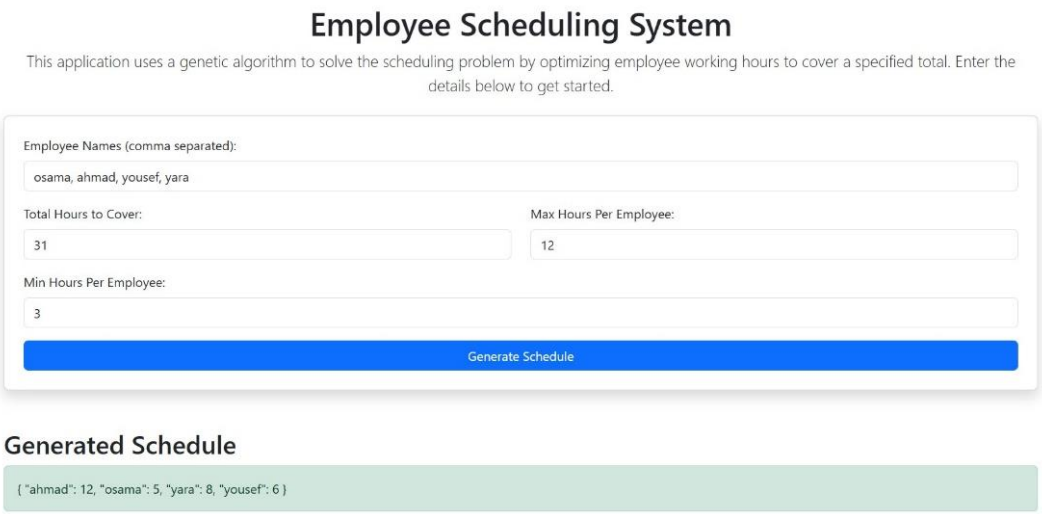
أولا: انشاء السيرفر على python any where:



الخطوة الثانية:رفع اكواد الموقع على الاستضافة في الأماكن المخصصة لها وتفعيل الموقع ومن ثم تحديث السيرفر من اجل تشغيل الموقع وتجربته :



تجربة ادخال الى الموقع:



تحليل النتائج من الموقع:

تحليل النتائج من الموقع للتأكد من أن الخوارزمية الجينية قد أنتجت حلاً صحيحًا وفعالاً، سيتم محاكاة خطوات إنشاء الحل يدويًا كما لو أن الخوارزمية تُطوَّر من البداية. الهدف هو التحقق من دقة توزيع الساعات بين الموظفين وفقاً للقيود المعطاة.

القيود الأساسية

١. عدد الموظفين: ٤ (Osama، Ahmad، Yousef، Yara).

٢. إجمالي الساعات المطلوب تغطيتها : ٣١ ساعة.

٣. الحد الأقصى لساعات العمل لكل موظف : ١٢ ساعة.

٤. الحد الأدنى لساعات العمل لكل موظف : ٣ ساعات.

تحليل يدوياً لإنشاء الحل

الخطوة ١: تحديد الإطار الأولي

- **المطلوب :توزيع ٣١ ساعة بين ٤ موظفين.**
- **يجب أن يحصل كل موظف على عدد ساعات ضمن الحدود الدنيا والقصوى:**
  - **3≤ساعات الموظف≤123 \leq ساعات الموظف \leq 123≤ساعات الموظف12.**

الخطوة ٢: توزيع مبدئي (احترام القيود)

- **لنبدأ بتخصيص الساعات الدنيا لجميع الموظفين:**

  - **osama: 3**
  - **ahmad: 3**
  - **yousef: 3**
  - **yara: 3**

- **المجموع الحالي للساعات: ٣+٣+٣+٣=١٢٣ + ٣ + ٣ + ٣ = ١٢٣+٣+٣+٣=١٢.**
- **الساعات المتبقية للتوزيع: ٣١−١٢=١٩٣١ - ١٢ = ١٩٣١−١٢=١٩.**

الخطوة ٣: التوزيع العادل للساعات المتبقية

- سنبدأ بتوزيع الساعات المتبقية على الموظفين، مع مراعاة الحدود القصوى لكل موظف:
  - نبدأ بـ **ahmad** يمكنه الحصول على الحد الأقصى (١٢ ساعة) :
    - ahmad: 3+9=123 + 9 = 123+9=12 (لم يتجاوز الحد الأقصى).
    - الساعات المتبقية: 19−9=1019 - 9 = 1019−9=10.
  - ننتقل إلى **yara** يمكنها الحصول على ٥ ساعات إضافي.:
    - yara: 3+5=83 + 5 = 83+5=8.
    - الساعات المتبقية: 10−5=510 - 5 = 510−5=5.
  - ننتقل إلى **yousef** يحصل على ٣ ساعات إضافية:
    - yousef: 3+3=63 + 3 = 63+3=6.
    - الساعات المتبقية: 5−3=25 - 3 = 25−3=2.
  - وأخيرًا **osama** يحصل على الساعات المتبقية(2) :
    - osama: 3+2=53 + 2 = 53+2=5.

النتيجة النهائية

- Osama : 5 ساعات.
- Ahmad : 12 ساعة.
- Yousef : 6 ساعات.
- Yara : 8 ساعات.

مقارنة بالحل الناتج عن الخوارزمية

- الخوارزمية قدمت نفس التوزيع:
  - { "ahmad": 12, "osama": 5, "yara": 8, "yousef": 6 }.
- الحل الناتج يحقق جميع القيود:
  - مجموع الساعات: 5+12+6+8=315 + 12 + 6 + 8 = 315+12+6+8=31 (يتطابق مع المطلوب).
    - كل موظف حصل على عدد ساعات ضمن الحدود الدنيا والقصوى:
      - osama: 5∈[3,12]5 \in [3, 12]5∈[3,12].
      - ahmad: 12∈[3,12]12 \in [3, 12]12∈[3,12].
      - yara: 8∈[3,12]8 \in [3, 12]8∈[3,12].
      - yousef: 6∈[3,12]6 \in [3, 12]6∈[3,12].

التحليل النهائي:

-الخوارزمينه الجينية أثبتت فعاليتها ،حيث نجحت في الوصول الى الحل بطريقه تتوافق مع التحليل اليدوي .

- الحل الامثل يتميز بتحقيق التوازن بين :

١. تلبيه الساعات المطلوبة

٢. الالتزام بالحدود الدنيا والقصوى المحددة

٣. توزيع الساعات بشكل عادل بين العناصر المختلفة

إثبات صحة الحل:

١. احترام القيود : جميع الساعات موزعه ضمن الحدود المحدده .

٢. تحقيق الهدف : مجموع الساعات الموزعه يساوي القيمه المطلوبه يساوي القيمه المطلوبه (٣١) .

٣. منطقيه الحل : توزيع الساعات يعتمد على تحسينات تدريجية تضمن كفاءة وفعاليه الحل .