Job Schedule

- ## Job Scheduler Application

- Job scheduling is the process where different tasks get executed at pre-determined time or when the right event happens. A job scheduler is a system that can be integrated with other software systems for the purpose of executing or notifying other software components when a pre-determined, scheduled time arrives.
- The project objective is to design (GUI) application using Tkinter to schedule a set of jobs on a specific number of machines, using the Backtracking algorithm and Genetic algorithm.
- Now I will explain each class separately

---

1. Job Class: The `Job` class is used to represent a job in the scheduling system. It contains the following attributes:

- **Job-id**: An integer identifier for the job.
- **Duration**: The total time required to complete the job.
- **Allowed-resources**: A list of machine indices where the job is allowed to run. Default is an empty list
- **Dependencies**: A list of job IDs that must be completed before this job can be scheduled.

- _____

2. The main class, Job-Scheduler-App, is designed to provide

An interactive user interface (UI) where users can:

- Define the number of jobs and machines
- Add job details (duration, allowed resources, dependencies)
- Choose between scheduling algorithms (Backtracking or Genetic)
- View the job schedule and its corresponding Gantt chart
- Zoom in and out to adjust the Gantt chart's scale.

- **root**: The root Tkinter window.
- **jobs**: A list of Job instances to be scheduled.
- **Max-jobs**: The maximum number of jobs that can be added.
- **Num-machines**: The number of machines available for scheduling.
- **Resource-caps**: A list representing the capacity of each machine.
- **quantum**: The time slice (quantum) used in the scheduling algorithm.
- **init**: Initializes the main window and the user interface components. It sets up different frames for selecting

algorithms, resource settings, job configurations, and job management.

- **Set-job-count**: Sets the maximum number of jobs to be added and resets the job list.
- **Add-job**: Adds a new job to the list based on user input (duration, allowed resources, dependencies).
- **Run-scheduler**: Runs the scheduling algorithm based on the selected algorithm. It gathers resource and job configurations, checks for capacity constraints, and then calls the appropriate scheduler function.
- **Round-robin-scheduler**: A function that implements the **Backtracking scheduler** with a **Quantum-based Round Robin** scheduling approach. It attempts to assign jobs to machines, respecting resource constraints, job durations, and allowed machines.
- **Show-gantt-chart**: Generates a Gantt chart of the job schedule using matplotlib. It visualizes the job allocation across different machines and the time slots when each job is being executed.
- **Add-job**: This method is used to add new jobs to the scheduler, validating inputs like job duration, allowed resources, and dependencies
- _____

3.GUI This part of the interface allows the user to input Number of machines , Capacities , Quantum ,Job , Duration , Allowed Resources , Dependencies .
Makes frames that show the Dependencies , Run Scheduler ,Gantt Chart, Backtracking, Genetic, Number of machines, Capacities , Quantum ,Job , Duration ,Allowed Recourses.

- This part of the interface allows the user to select the scheduling algorithm between **Backtracking** and **Genetic**
- **Radio Buttons**: Users can select either "Backtracking" or "Genetic" from the algorithm options.

Now I will explain what the meaning of them is.

1. The Backtracking algorithm is a class of algorithms for finding solutions to some computational problems, notably constraint satisfaction problems , that incrementally builds candidates to the solutions, and abandons a candidate as soon as it determines that the candidate cannot possibly be completed to a valid solution.

2. The genetic algorithm is a method for solving both constrained and unconstrained optimization problems that is based on natural selection, the process that drives biological evolution. The genetic algorithm repeatedly modifies a population of individual solutions.

3. **Number of Machines**: Specifies how many machines are available for job scheduling.

4. **Capacities**: A list of machine capacities, where each machine has a specific available resource capacity.

5. **Quantum**: The time slice each job is allowed to execute in a Round Robin cycle.

6. **Job Configuration:** The number of jobs to be added is defined here. Users input the number of jobs and press the "Set" button to configure the job list.

7. **Duration**: The time required to complete the job.

8. **Allowed Resources**: A list of comma-separated machine indices where the job can be scheduled.

9. **Dependencies**: A list of job IDs that must be completed before the job can start.

10. **Scheduler Control: After** adding all the jobs, users can press the "Run Scheduler" button to begin the scheduling process. This triggers the scheduling algorithm to be

executed based on the selected algorithm and input configurations.

11. **Gantt Chart :** The **Gantt Chart** button generates a visual representation of the job schedule using matplotlib. The chart shows the start and end times for each job, plotted on the machines.

---

3. add-job()When the user fills the fields and clicks "Add Job", this method:

1. **Reads input values.**
2. **Creates a Job object.**
3. **Adds it to a list of jobs.**

---

4 .run-scheduler: Executes job scheduling based on user input, validates it, and then runs the specified algorithm.

1 .Clear previous results from the text box.
2. Read input from user:
Number of machines.
Resource capacity per machine.
Quantum

3.Validate input:Ensure that the numbers are correct.
The number of resources matches the number of machines.
The presence of added jobs.
4.Validate job schedulability:
Ensure that the duration of each job does not exceed the total capacity of the machines.
5.Run algorithm:
If the selected algorithm is Backtracking, run round_robin_scheduler.
Otherwise, a message appears stating that the algorithm is not supported.

4.round_robin_scheduler:

- **time-slot: Used to track the time slots for each job.**
- **Remaining-time: Stores the remaining execution time for each job.**
- **schedule: A dictionary where each key is a machine and the value is a list of scheduled tasks.**
- **Current-time: Tracks the current time on each machine.**

1.Main Loop (Until All Jobs Are Completed):
- **Iterate through each job:**
Skip the job if its remaining time is zero.

Determine the time that can be executed for the job (based on the quantum).

Try to find a machine to execute the job:

- **Consider only the machines allowed for the job (if any).**
- **Check if the machine has enough resource capacity for the execution.**

2. Execute the Scheduled Portion of the Job on the Selected Machine:

Schedule the job for each time unit one by one.

Update:

- **The remaining time of the job.**
- **The current time of the machine.**

3 .Error Handling:

If the job can't be scheduled on any machine → display an error message and stop the scheduling.

3. Display Results:

- **Print the scheduling timeline for each machine.**
- **Display the Gantt chart using show-gantt-chart.**

5. **Add_jobs: Adds a new job to the job list based on user input.**

Reads values from the input fields:

- **Duration**
- **Allowed Resources**
- **Dependencies**

Converts allowed resources and dependencies into lists of integers (if provided).

- If parsing fails, shows an error message and exits the function.

Create and Store the Job:

- Assigns a unique `job-id` (based on current list length).
- Creates a new `Job` object with the parsed data.
- Appends it to the `jobs` list.

Update UI:

- Adds a summary of the job to the job listbox.
- Clears the input fields.
- Shows a success message.

7. Gantt Chart: Displays a Gantt Chart showing how jobs are scheduled across machines over time.

1. **Check for Existing Schedule:**
   - **If no schedule exists → show error message and stop.**
2. **Validate Machine Count Input:**
   - **Retrieves the number of machines from the entry field.**
   - **If invalid → show error and stop.**
3. **Prepare Gantt Chart:**
   - **Uses matplotlib to create the chart.**
   - **Defines a list of distinct colors to visually differentiate jobs.**
4. **Draw Tasks:**
   - **For each machine and its scheduled tasks:**
     - **Draws a colored rectangle representing the job's time slot.**
     - **Adds text inside each rectangle to label it with the job ID.**
5. **Customize Chart:**
   - **Sets y-axis to label machines.**
   - **Determines the x-axis limit based on the maximum time used.**

- **Labels the axes and adds a title.**
- **Enables the grid and tight layout for clean appearance.**

6. **Show Chart:**
   - **Displays the final Gantt chart in a new window.**

–

Documentation (Job Scheduling) (Genetic):

This project aims to solve a big problem in huge Factories we need to balance between Machines and make the best utilization of all resources with respect to

the processing time and deadline of each job.

As we know Genetic Algorithm should transfer between some stages:

1-Encoding (using in this problem is:

- Hybrid encoding (mixture of permutation and direct way).

2-Create population

3-Design of Fitness Function (our fitness function designed according to

Make-span (completion time) as

Fitness increases when make-span decrease. As we take the makespan as

Standard to determine the position of each one in the population (who is the strongest and who is the weakness).

4-Selection (we use Tournament Selection)

We choose some chromosomes randomly then make a tournament between them and get the winner as the strongest.

5-Crossover (using JOX) = (Job ordered Based)

6-Mutation

7-Stopping generations (as we make the constraint of stop generating population is when fitness be constant or decreasing) Then, in this case we stopped.

Explanation of code:

1-we created 3 classes :(Job,

Job-Scheduler, Resource):

We track some data (attributes) about each class:

About (Job):

- Job id
- Duration(processing time)
- Dependence(if it has prerequisite)

About Resource:

- Resource id
- Resource capacity
- Time-line (To know if resource is idle or not).

About Job-Scheduler:

- Job id
- Resource id
- Quantum (scanned from user.. with respect that q should be less or equal to capacity).

- Allow-chunking (whether it can be split or No).
- Best schedule (Will hold the best schedule found).
- Min-make span (to put in it the minimum make span after calculating it during many generations).

We create a function that called (Fitness Function):
It accepts the (chromosome=list of genes).

- Evaluates the **correctness and efficiency** of a chromosome (schedule).
- Adds **penalties** for violations (e.g., incorrect chunk size).
- Prepares to calculate **makespan** and **resource utilization** (although the image cuts off before we see the return statement).

And track some data such as:

1. Completion time: To know when each job finishes.
2. Resource utilization: how busy each resource is busy.
3. Penalty=time exceeded the duration

Then, Sort jobs by start time.
And calculate penalties.

To continue in checking cases and constraints:
We firstly check that there is not any overlapping.
Check and calculate penalty time and make the difference between the exact time and the penalty time as cost of this job.

After checking the constraints, It is the time to calculate the makespan and calculate the fitness .

The `generate schedule table` function processes a chromosome to compute and correct the timing of jobs, especially handling final chunks with remaining duration. It builds a

schedule table with accurate durations based on quantum and job definitions.

This function to calculate utilization of each resource and know if they are in balance or not.

This Function is to (encode)…Parameters are assigned to it:

1. Jobs: a list of jobs we need to encode

2. divide-job: it is Boolean (True or False)

Which determine if this job can be divided or not.

4. time- per -chunk: it is like "Quantum"…as the maximum time. The chunk can be loaded.

4. valid-resource-id: a list of resources that is valid or allowed to take jobs.

Then we make a list called chromosome to store final encoded chromosomes schedule in it.

Added-job: a set we created to store each job encoded. To check job dependencies.

Make a shallow copy of the list job. Called

"Job –to- schedule". We'll modify this copy while keeping the original `-jobs` list safe.

Keep looping **as long as** there are still jobs left to schedule.

5. At the start of each loop, assume **no job has been scheduled yet**.
6. We will set this to `true` if we successfully schedule any job during this iteration..
7. Iterate over a **copy** of `jobs-to-schedule` (`[:]` creates a shallow copy).
8. This allows removing jobs during iteration without messing up the loop.

This handle if there is a suitable resource or not.

1. Function: create-population:

Generates a list of encoded job schedules (a population) for a genetic algorithm.

Each individual in the population is created using the encode-jobs function, with optional job chunking and specified valid resources.

2.Function 2: Selection:

Selects the fittest individuals for the next generation using elitism and tournament selection.

It keeps the best individual and then selects the rest by running mini-tournaments among random individuals.

The number of selected individuals is based on the selection-ratio (e.g., 80% of the population).

Selects the fittest individuals for the next generation using elitism and tournament selection.

It keeps the best individual and then selects the rest by running mini-tournaments among random individuals.

The number of selected individuals is based on the selection-ratio (e.g., 80% of the population)

3: Crossover:

Creates two new children by exchanging segments between two parent solutions.

It selects a random segment from each parent and ensures that each job appears only once in each child.

The rest of the genes are filled from the other parent while avoiding duplicates using job IDs.

Crossover only occurs if a random value is less than or equal to the crossover -rate (default 0.8).

4. Mutation:

Applies random changes to a chromosome to introduce variation.
With a certain probability (mutation rate), it either changes the resource or shifts the job's start and end times.
Chunked jobs (with _ in their ID) only allow resource changes, not time modifications.
This helps explore new solutions while preserving the job structure and constraints.

5. Function: Validate dependence:

Applies random changes to a chromosome to introduce variation.

With a certain probability (mutation-rate), it either changes the resource or shifts the job's start and end times.

Chunked jobs (with _ in their ID) only allow resource changes, not time modifications.

This helps explore new solutions while preserving the job structure and constraints.

5. Function (input jobs):

Prompts the user to input job details, including job ID, duration, and optional dependencies.
It ensures each job ID is unique and correctly formatted, with dependencies properly parsed.
If the input is invalid, it keeps asking for correct job information.
The function returns a list of Job objects created from the input data.

6.Main Function:

Prompts the user to decide whether to divide long jobs into chunks.
If the user chooses "yes," it asks for the time per chunk and ensures it's a positive integer.
It returns True with the chunk time if dividing, or False with none if not dividing.
Generates offspring by applying crossover and mutation to pairs of selected parents.

- It uses the crossover function to combine two parents and then applies mutation to introduce variation.
- Afterward, it validates the offspring chromosomes to ensure they are feasible.
- Valid offspring are added to the new generation, and the function returns the list of valid offspring.
- is the main loop of the genetic algorithm for job scheduling.
- It starts by generating an initial population and validating job feasibility

based on resource capacity and chunking.

- It evaluates fitness for each chromosome and selects the best solutions using tournament selection.
- The loop continues to create offspring through crossover and mutation, keeping only valid solutions.

It tracks the best solution found so far and stops after a number of generations without improvement (patience).
It prints helpful progress information like fitness values and chromosome structure for each generation.
Finally, it returns the best chromosome (job schedule) and its fitness score.

Dependency:

Checks if any job dependency graph contains a cycle.
It uses DFS with a recursion stack to detect cycles by exploring each job's dependencies.
If any cycle is found during traversal, it returns true; otherwise, it returns False.

Is the entry point of the job scheduling program.
It collects input from the user for resources and jobs, then checks for circular dependencies.
If total job durations exceed available resource capacity and chunking is disabled, it warns the user.
It runs the genetic algorithm (evolve) to find the best schedule based on fitness.
The final schedule is printed in a formatted table with job timings and resource usage.

If errors occur (e.g. due to schedulable jobs), the user is informed and guided on possible fixes.

That is all for now. This is
Our Documentation for Job Scheduling
Using
Genetic Algorithm
And backtracking algorithm.