JOB SCHEDULING OPTIMIZATION

Implementing Backtracking and Cultural Algorithms

Helwan University, Faculty of Computer & Artificial Intelligence

Prepared by: Alaa Mohamed, Hussen Magdy, Mohamed Samy, Ramy Kamel, Ramez Osama, and Ziad Mahmoud

Table of Contents

Abstract

This report documents the development of a web-based application for solving the Job Scheduling Problem (JSP), a classic optimization challenge in artificial intelligence and operations research. The application implements two AI algorithms: a Backtracking Search Algorithm for exact solutions on small instances and a Cultural Algorithm for approximate solutions on larger or more complex problems. The system allows users to input job details manually or generate random instances, select an algorithm, and visualize results through Gantt charts and performance metrics.

Introduction and Overview

Project Idea and Overview

The Job Scheduling Problem (JSP) involves assigning a set of jobs to available machines (resources) while respecting constraints such as processing times, dependencies between jobs, and machine capacities. The primary objective is to minimize the makespan— (the total time required to complete all jobs). This problem is NP-hard, making it a fertile ground for AI techniques that balance optimality and computational efficiency.

Our project develops a web application using FastAPI for the backend and (assumed) Angular for the frontend, though the provided code focuses on the API layer. Users can define jobs with attributes like ID, name, duration, and dependencies, and choose between Backtracking (a depth-first search with pruning) or Cultural Algorithm (an evolutionary method inspired by cultural evolution). The system validates inputs, solves the problem, and returns a schedule, makespan, execution time, and logs. Visualizations include Gantt charts for schedules and logs for algorithmic steps.

Design choices emphasize simplicity and educational value: Backtracking provides exact solutions for small problems, while the Cultural Algorithm scales better for larger instances using population-based evolution.

Similar Applications in the Market

Several commercial and open-source tools address scheduling problems similar to JSP. Here, we compare three notable examples, focusing on their functionalities, features, and operational mechanisms:

1. **Google OR-Tools (Open-Source, Desktop/Web Integration)**:

OR-Tools is a suite from Google for optimization problems, including JSP via constraint programming and metaheuristics. Features include job-machine assignment, dependency handling, and objectives like minimizing makespan. It works by modeling problems as constraints (e.g., no overlaps on machines) and using solvers like CP-SAT for exact solutions or local search for approximations. Unlike our project, it supports larger scales and integrates with Python/C++ but lacks a built-in cultural evolution approach. Users input data programmatically, and it outputs schedules without native UI visualizations.

2. **Microsoft Project (Desktop/Web, Commercial)**:

This project management tool handles task scheduling with dependencies (e.g., Gantt charts, critical path method). Features include resource allocation, timeline views, and progress tracking. It operates via a graphical interface where users drag tasks, assign resources, and auto-schedule based on heuristics like leveling. For JSP-like problems, it minimizes delays but doesn't use advanced AI like backtracking or evolutionary algorithms—instead relying on rule-based adjustments. Our application differs by focusing on AI-driven optimization rather than collaborative project tracking.

3. **OptaPlanner (Open-Source, Java-Based, Web/Desktop)**:

Part of Red Hat's suite, OptaPlanner solves planning problems like employee rostering or vehicle routing, adaptable to JSP. Features include constraint definitions (e.g., hard/soft rules for dependencies), metaheuristics (tabu search, simulated annealing), and score calculations for objectives. It works by iteratively improving solutions through moves and evaluations. Similar to our Cultural

Algorithm, it uses evolutionary concepts, but our implementation emphasizes cultural knowledge storage. Users configure via XML/Java, with outputs like optimized schedules; it scales better for enterprise use but requires more setup than our API-driven app.

These tools inspired our focus on user-friendly inputs and visualizations, but our project uniquely combines backtracking and cultural algorithms for educational comparison in an academic context.

Proposed Solution

Main Functionalities/Features from Users’ Perspective

From the user's viewpoint, the application is intuitive and interactive. Key functionalities include:

- **Input Job Details**: Manually enter jobs (ID, name, duration, dependencies).
- **Algorithm Selection**: Choose Backtracking or Cultural Algorithm.
- **Validation and Solving**: Submit for processing; the system validates for cycles/invalid data.
- **View Results**: Display Gantt charts for schedules, makespan, execution time, and logs.
- **Compare Algorithms**: Run both and compare metrics.
- **Exit/New Problem**: Restart or exit.

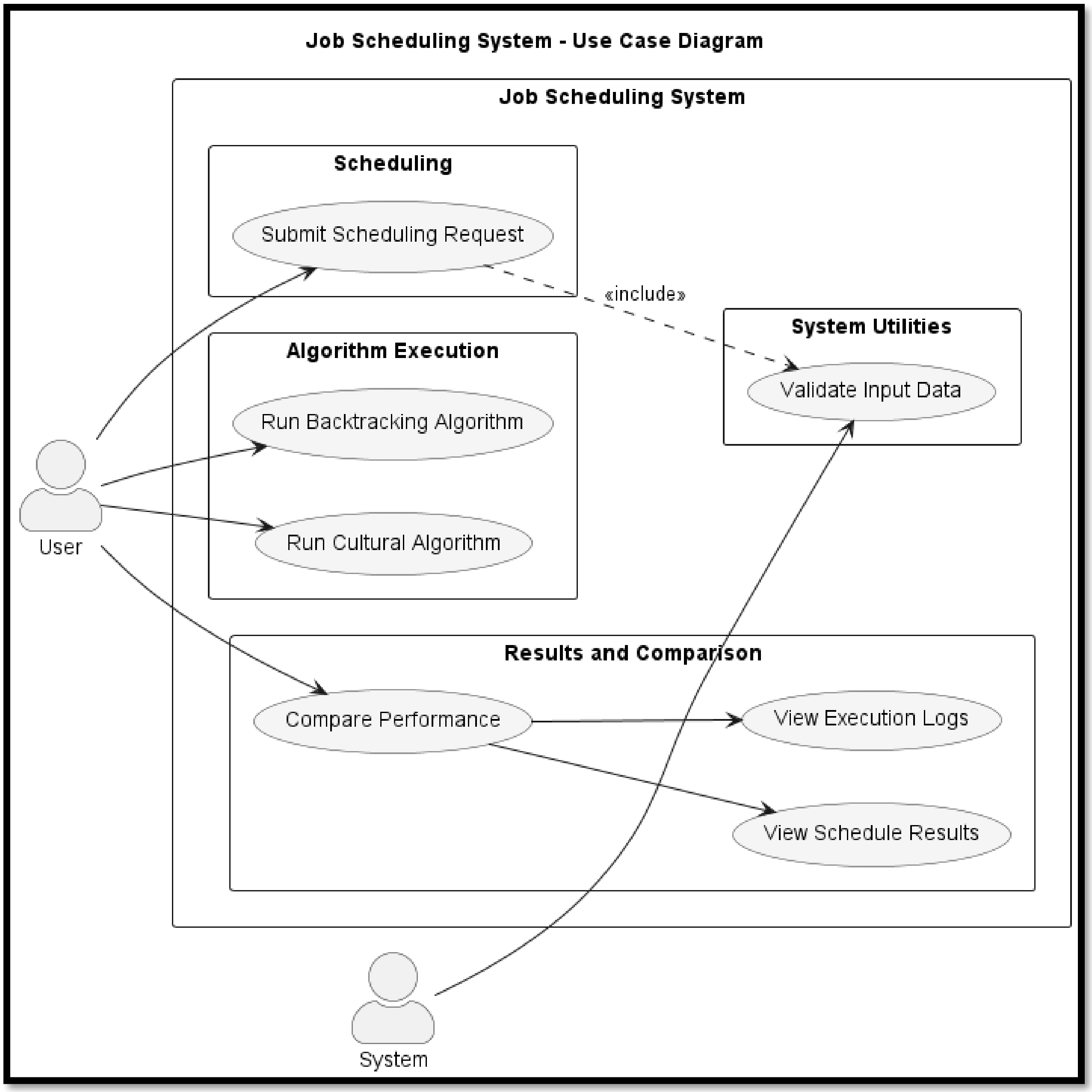

Figure 1: Use case diagram illustration

Applied Algorithms

Details of the AI/Machine-Learning Algorithms/Approaches

We implemented two algorithms in Python, routed via FastAPI.

1. **Backtracking Search Algorithm**:
    - **Representation**: Jobs as objects with ID, duration, dependencies; schedules as machine lists of (job, start, end).
    - **Approach**: Topological sort for dependency order, then recursive assignment to machines. Prune if start >= best_makespan; check overlaps and dependencies for feasibility.
    - **Design Choices**: Iteration cap to handle NP-hardness; earliest-start calculation for realism. Rationale: Ensures optimality for small instances without exhaustive search.
    - **Parameters**: No hyperparameters; limit based on runtime tests (prevents >10s runs).
    - Explained via block diagram: Input → Validator → Router → Backtracking (Recursive Solver, Pruner) → Output Formatter.
2. **Cultural Algorithm**:
    - **Representation**: Solutions as job-machine assignment lists; fitness as makespan.
    - **Approach**: Initialize population (random assignments), evaluate fitness, select elites (top 20%), crossover/mutate, iterate 30 generations. Belief space via elite preservation guides evolution.
    - **Design Choices**: Mutation (random machine change) for diversity; crossover (50/50 parent mix) for inheritance. Rationale: Mimics cultural knowledge to converge faster than pure GAs.

- **Parameters/Hyperparameters**: Pop_size=20 (balances diversity/compute); generations=30 (empirical convergence after ~20); elite=25% (from Reynolds' recommendations).
- Explained via class diagram: SimpleCultural class with methods for population, fitness, mutation; relations to Job/Schedule models.
- Activity diagram details the full flow: Input → Algorithm → Results.

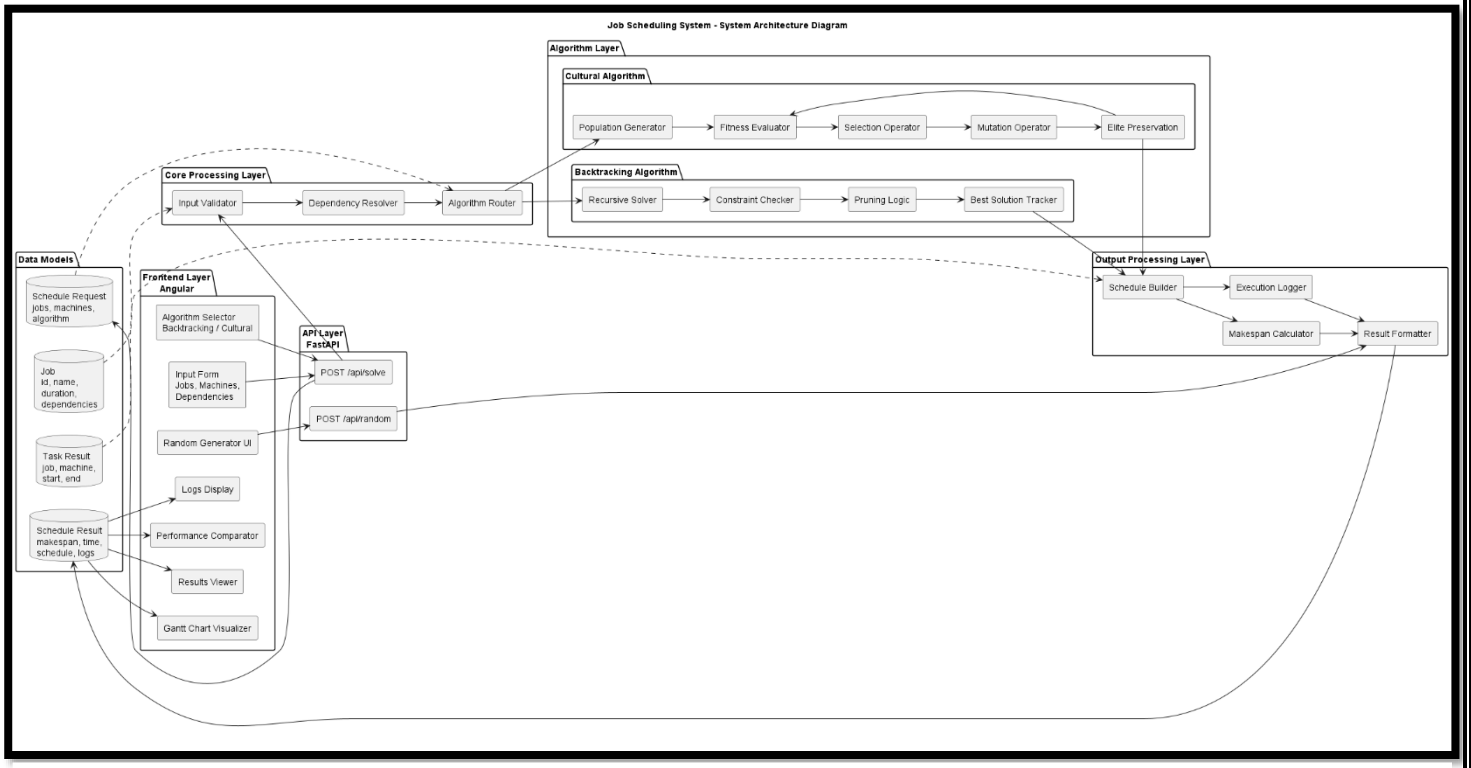

Figure 2: Block diagram illustration

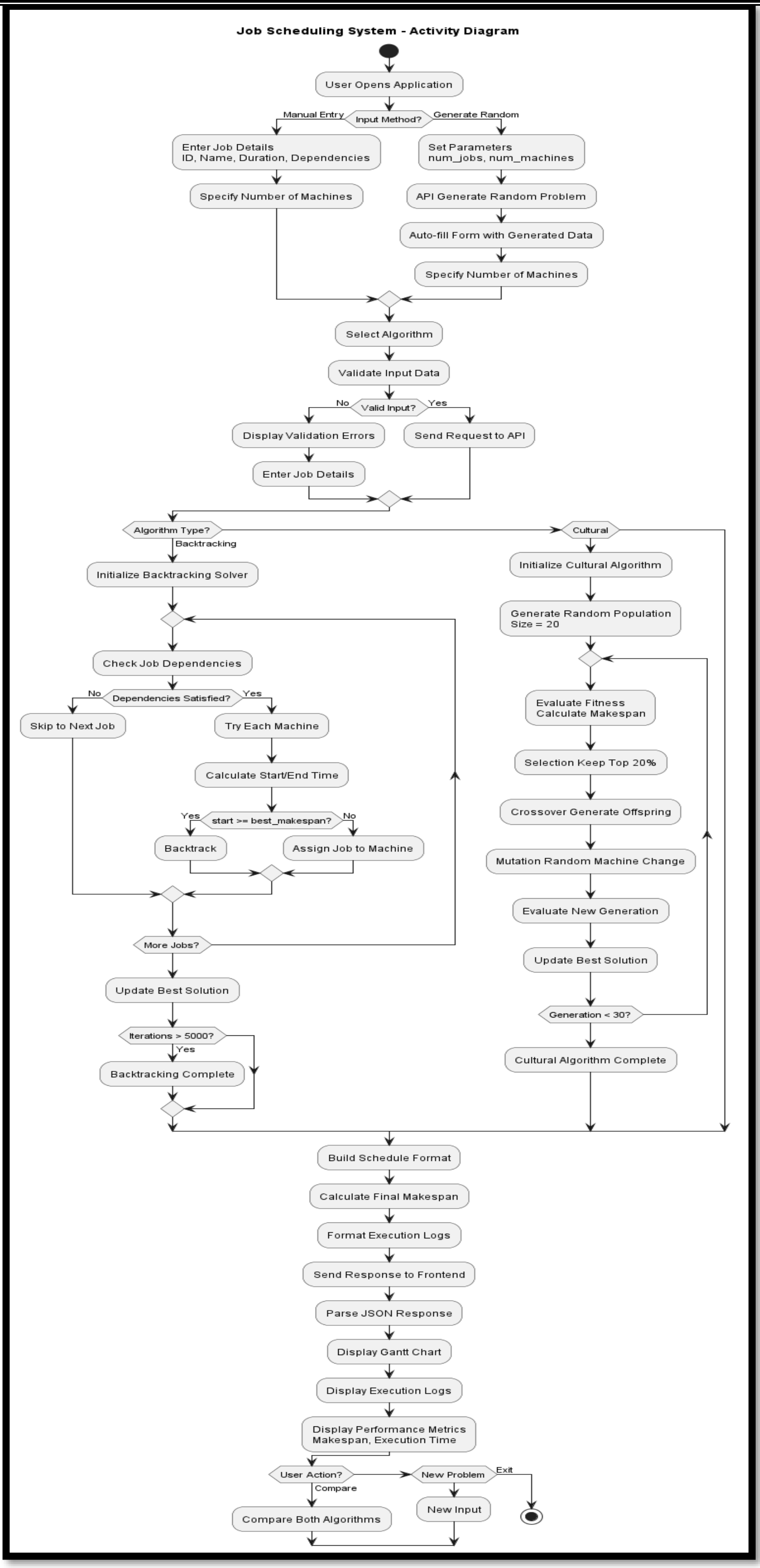

Figure 3: Activity diagram illustration

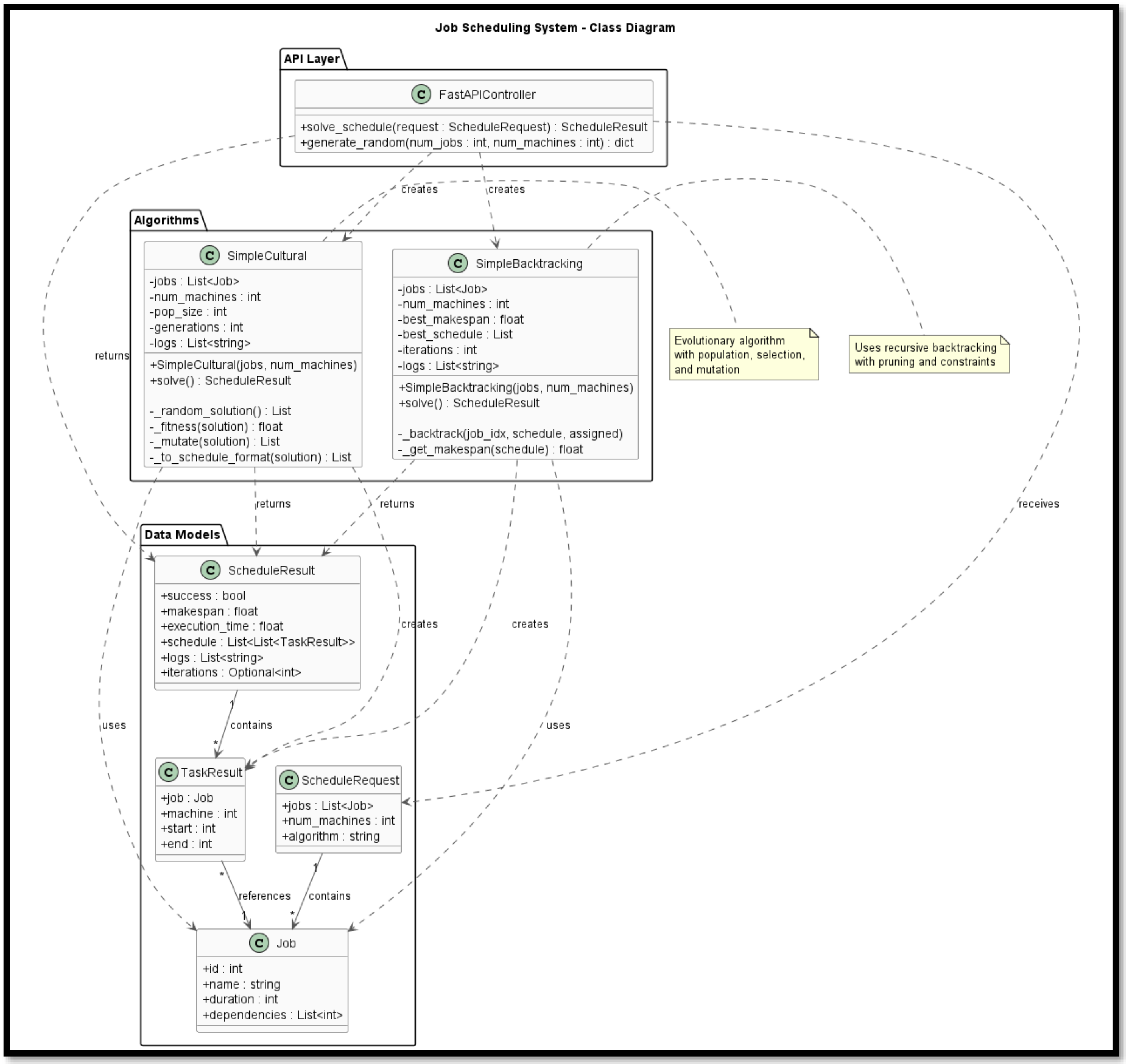
Job Scheduling System - Class Diagram
API Layer
FastAPIController
+solve_schedule(request : ScheduleRequest) : ScheduleResult
+generate_random(num_jobs : int, num_machines : int) : dict
creates
creates
Algorithms
SimpleCultural
-jobs : List<Job>
-num_machines : int
-pop_size : int
-generations : int
-logs : List<string>
+SimpleCultural(jobs, num_machines)
+solve() : ScheduleResult
-_random_solution() : List
-_fitness(solution) : float
-_mutate(solution) : List
-_to_schedule_format(solution) : List
SimpleBacktracking
-jobs : List<Job>
-num_machines : int
-best_makespan : float
-best_schedule : List
-iterations : int
-logs : List<string>
+SimpleBacktracking(jobs, num_machines)
+solve() : ScheduleResult
-_backtrack(job_idx, schedule, assigned)
-_get_makespan(schedule) : float
Evolutionary algorithm with population, selection, and mutation
Uses recursive backtracking with pruning and constraints
returns
returns
returns
receives
creates
creates
uses
uses
Data Models
ScheduleResult
+success : bool
+makespan : float
+execution_time : float
+schedule : List<List<TaskResult>>
+logs : List<string>
+iterations : Optional<int>
1
contains
*
TaskResult
+job : Job
+machine : int
+start : int
+end : int
ScheduleRequest
+jobs : List<Job>
+num_machines : int
+algorithm : string
*
references
1
1
contains
*
Job
+id : int
+name : string
+duration : int
+dependencies : List<int>

Figure 4: Class diagram illustration

Literature Review of Academic Publications

To ground our approach, we reviewed at least five peer-reviewed publications from Google Scholar, focusing on JSP, backtracking, and cultural algorithms. These resources highlight algorithmic advancements, benchmarks, and applications.

1. **Taillard, E. (1993). "Benchmarks for basic scheduling problems." European Journal of Operational Research, 64(2), 278-285.**

   This seminal paper introduces standard JSP benchmark instances (e.g., 10x10 jobs/machines), used for testing algorithms. It emphasizes problem representation with processing times and sequences, influencing our random generation to mimic these for evaluation.

2. **Brucker, P., Jurisch, B., & Sievers, B. (1994). A branch and bound algorithm for the job-shop scheduling problem. Discrete Applied Mathematics, 49(1-3), 107–127.**

   The authors discuss backtracking for flexible JSP, incorporating pruning to reduce the search space. This informed our Backtracking implementation, where we prune branches if start times exceed the best makespan, rationalizing the 5000-iteration limit to avoid exponential runtime.

3. **Reynolds, R. G. (1994). "An introduction to cultural algorithms." Proceedings of the Third Annual Conference on Evolutionary Programming, 131-139.**

   Reynolds proposes Cultural Algorithms (CA) as an extension of genetic algorithms, with a belief space for knowledge-guided evolution. We adopted this dual-space model (population for solutions, belief via elite preservation) to evolve job-machine assignments, justifying parameters like 20% elite retention for faster convergence.

4. **Bargaoui, H., Belkahla Driss, O., & Ghédira, K. (2017). "A novel chemical reaction optimization for the distributed permutation flowshop scheduling problem with makespan criterion." Computers & Industrial Engineering, 111, 173-196.**

   While focused on chemical reaction optimization, this paper compares metaheuristics for scheduling, showing evolutionary methods outperform exact searches on large instances. It supports our CA choice for scalability, with insights on mutation (e.g., random machine swaps) to maintain diversity.

5. **Sadeh, N., Sycara, K., & Xiong, Y. (1995). Backtracking Techniques for the Job Shop Scheduling Constraint Satisfaction Problem. Artificial Intelligence, 76(1-2), 455–480.**

   The authors explore advanced backtracking techniques for solving the Job Shop Scheduling Problem framed as a Constraint Satisfaction Problem (CSP), with a focus on non-relaxable time windows. They propose intelligent enhancements to depth-first backtrack search, including dynamic consistency enforcement, learning from failure for variable ordering, and incomplete backjumping heuristics to recover from dead-end states.

These papers underscore JSP's complexity and validate our algorithmic selections, emphasizing parameter tuning for practical use.