HELWAN NATIONAL UNIVERSITY

Academic Year **2024 – 2025**

Artificial intelligence course.

Under the supervision of Dr. Amr Ghoneim and Dr. Marwa El Sayed.

Job Scheduling Problem Solver Using the Backtracking Search Algorithm and Genetic Algorithm

This project aims to solve the **Job Scheduling Problem** using two algorithms: the **Backtracking Search Algorithm** and the **Genetic Algorithm**. The goal is to efficiently allocate resources to tasks while considering constraints like deadlines and resource limits.

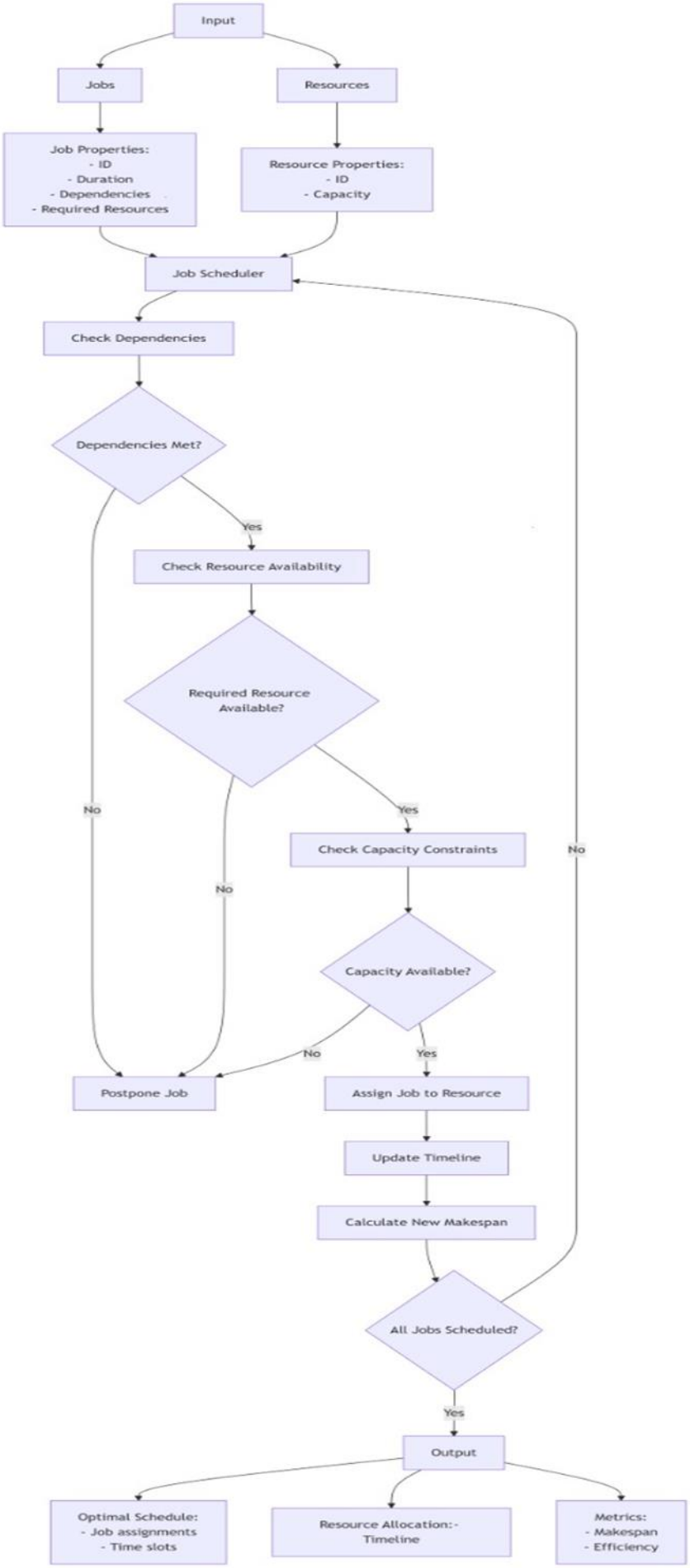
The **Backtracking Algorithm** ensures an optimal solution by exploring all possibilities, while the **Genetic Algorithm** offers a faster, heuristic-based approach, especially for larger problems. This project compares both methods in terms of efficiency and effectiveness, providing insights into their practical applications for job scheduling.

Introduction and Overview

Efficient job scheduling is a cornerstone of productivity and operational effectiveness in many complex systems. The Job Scheduling Problem (JSP) is a well-known optimization challenge that involves assigning a set of jobs to limited resources in a way that meets specific constraints and optimizes performance objectives such as minimizing total completion time or maximizing resource utilization. JSP plays a critical role in various industries, including manufacturing systems, where it improves production efficiency, and in computing, where it ensures optimal resource allocation and task execution. This project aims to design and implement a Job Scheduling Problem Solver using both the Backtracking Search Algorithm and a Genetic Algorithm. By combining the exhaustive yet precise nature of backtracking with the adaptive optimization capabilities of genetic algorithms, the system seeks to provide effective and scalable solutions to complex scheduling scenarios.

To address the complexity of the Job Scheduling Problem, this project explores two distinct AI-based approaches: the Backtracking Search Algorithm and the Genetic Algorithm. Each algorithm offers a unique problem-solving perspective—backtracking provides an exhaustive search strategy, while genetic algorithms utilize evolutionary principles for optimization. In the following sections, each algorithm will be discussed separately to highlight its design, functionality, and contribution to solving the scheduling problem.

The following diagrams provide a comprehensive overview of the Genetic Algorithm and Backtracking code structures. They illustrate the key processes involved in both optimization methods, including population initialization, fitness evaluation, selection, crossover, mutation, and recursive constraint-based search. Together, these diagrams capture the complementary approaches used for efficient problem-solving, reflecting both the evolutionary and exhaustive search paradigms.


Input
Jobs
Resources
Job Properties:
- ID
- Duration
- Dependencies
- Required Resources
Resource Properties:
- ID
- Capacity
Job Scheduler
Check Dependencies
Dependencies Met?
Yes
Check Resource Availability
Required Resource Available?
No
Yes
Check Capacity Constraints
No
No
Capacity Available?
No
Yes
Postpone Job
Assign Job to Resource
Update Timeline
Calculate New Makespan
All Jobs Scheduled?
No
Yes
Output
Optimal Schedule:
- Job assignments
- Time slots
Resource Allocation: -
Timeline
Metrics:
- Makespan
- Efficiency

Backtracking Search Algorithm for Job Scheduling - Overview

The **Backtracking Search Algorithm (BSA)** is a systematic, recursive approach used to solve combinatorial optimization problems like **Job Scheduling**. It explores the entire solution space by constructing possible schedules incrementally, removing those that violate the given constraints, and backtracking whenever a conflict is encountered. This method is particularly effective for problems where finding an optimal or feasible solution is challenging due to complex constraints and dependencies.

Key Features of the BSA for Job Scheduling:

1. **Recursive Exploration:**
    - The algorithm starts by assigning the first job to the first available time slot.
    - It then recursively attempts to schedule the remaining jobs.
2. **Constraint Checking:**
    - At each decision point, the algorithm checks whether the current partial schedule violates any constraints (e.g., resource conflicts, deadlines, or precedence).
    - If a conflict is detected, the algorithm backtracks to the previous step and tries a different path.
3. **Pruning the Search Space:**
    - The algorithm effectively reduces the search space by discarding paths that cannot lead to feasible schedules, significantly improving efficiency.
4. **Backtracking Mechanism:**
    - If no feasible assignment can be found at a certain level, the algorithm backtracks to the previous decision point, adjusting assignments accordingly.
5. **Termination:**
    - The algorithm terminates once all jobs are successfully scheduled or when all possible configurations have been exhausted.

Advantages:

- Guarantees finding an optimal solution if one exists.
- Efficiently handles complex constraints.

- Simple and intuitive implementation.

Limitations:

- Can be computationally expensive for large problem sizes.
- High memory usage due to extensive recursion.

Conclusion:
The Backtracking Search Algorithm is a powerful tool for solving Job Scheduling problems, providing precise solutions where heuristic methods may struggle. Its structured approach to constraint handling and recursive nature make it a critical algorithm for complex scheduling tasks.

Genetic Algorithm for Job Scheduling - Overview

The **Genetic Algorithm (GA)** is an evolutionary optimization technique inspired by the principles of natural selection and genetics. It is widely used for solving complex **Job Scheduling** problems, where the goal is to allocate tasks efficiently while minimizing cost and maximizing resource utilization. Unlike traditional algorithms, GA explores a diverse set of solutions simultaneously, making it highly effective in escaping local optima and finding near-optimal solutions in large, complex search spaces.

Key Features of the GA for Job Scheduling:

1. **Population Initialization:**
    - The algorithm starts by generating a **population** of potential job schedules, typically created randomly or using heuristics to improve initial quality.
    - Each schedule is represented as a **chromosome**, with individual jobs encoded as **genes**.
2. **Fitness Evaluation:**
    - Each chromosome is evaluated based on a **fitness function** that measures how well the schedule meets the desired objectives (e.g., minimizing makespan, reducing delays, or balancing resource utilization).
3. **Selection:**
    - The fittest individuals are selected as parents for the next generation using techniques like **roulette wheel selection, tournament selection,** or **rank-based selection.**
4. **Crossover (Recombination):**

- Pairs of parent chromosomes are combined to produce **offspring**, inheriting characteristics from both parents.
- Common crossover methods include **single-point, multi-point,** or **uniform crossover.**

5. **Mutation:**

    - To maintain diversity and avoid premature convergence, a small, random mutation is applied to some offspring, altering their gene values.
    - This prevents the population from getting trapped in local optima.

6. **Replacement:**

    - The newly generated offspring replace the least fit individuals in the population, creating a new generation.

7. **Termination:**

    - The algorithm continues evolving the population through multiple generations until a stopping criterion is met, such as a fixed number of generations or convergence to a high-quality solution.

Advantages:

- Capable of finding near-optimal solutions in large, complex search spaces.
- Inherently parallel, allowing for efficient exploration of multiple solutions simultaneously.
- Robust to noisy or incomplete data.

Limitations:

- Can be computationally intensive.
- Requires careful tuning of parameters (e.g., population size, mutation rate).
- No guarantee of finding the global optimal solution.

Conclusion:
The Genetic Algorithm is a versatile and powerful method for solving Job Scheduling problems, providing high-quality solutions where traditional algorithms may struggle. Its population-based approach and adaptive search capabilities make it an essential tool for complex scheduling tasks.

Applications
Task scheduling solutions are widely implemented across desktop, web, and mobile platforms to enhance productivity, resource management, and automation. For example, project management tools like **Microsoft Project** (desktop) and **Trello** (web/mobile) allow users to assign tasks, set deadlines, and track progress through

visual interfaces and automated notifications. In the mobile domain, apps like **Google Calendar** and **Asana** provide real-time scheduling, reminders, and resource allocation capabilities. These applications typically use internal scheduling algorithms to resolve task conflicts, prioritize deadlines, and optimize task distribution among users or systems. While many consumer-facing applications simplify the user experience, the underlying scheduling logic often addresses complex constraints similar to those in the Job Scheduling Problem (JSP). This project draws inspiration from such systems and aims to build a core scheduling engine capable of solving JSP instances with higher computational intelligence.

Literature Review for Job Scheduling Algorithms

One prominent approach is the use of **Genetic Algorithms (GAs)**, which are well-suited for exploring large solution spaces. In their foundational work, **Holland (1975)** introduced the concept of genetic algorithms, laying the groundwork for evolutionary computation. This was further developed by **Goldberg (1989)**, who presented a comprehensive overview of selection, crossover, and mutation techniques essential for solving scheduling problems. Additionally, **Davis (1991)** provided practical implementations of GAs, emphasizing their adaptability to complex optimization challenges like job scheduling.

- **Holland, J. H. (1975). Adaptation in Natural and Artificial Systems. University of Michigan Press.**
- **Goldberg, D. E. (1989). Genetic Algorithms in Search, Optimization, and Machine Learning. Addison-Wesley.**
- **Davis, L. (1991). Handbook of Genetic Algorithms. Van Nostrand Reinhold.**

In the context of job scheduling and resource allocation, **Pinedo (2016)** offered a detailed exploration of scheduling theory and algorithms, while **Blazewicz et al. (2007)** discussed the complexities of resource-constrained scheduling, providing critical insights for implementing GAs in real-world scenarios.

- **Pinedo, M. (2016). Scheduling: Theory, Algorithms, and Systems. Springer.**
- **Blazewicz, J., Ecker, K., Pesch, E., Schmidt, G., & Weglarz, J. (2007). Handbook on Scheduling: From Theory to Applications. Springer.**

For a broader perspective on evolutionary approaches, **Bäck, Fogel, and Michalewicz (1997)** compiled a comprehensive guide on evolutionary computation techniques, including scheduling, while **Gen and Cheng (1997)** provided specific guidance on applying GAs to engineering design and scheduling.

- **Bäck, T., Fogel, D. B., & Michalewicz, Z. (1997). Handbook of Evolutionary Computation. Oxford University Press.**
- **Gen, M., & Cheng, R. (1997). Genetic Algorithms and Engineering Design. Wiley.**

Handling constraints effectively in genetic algorithms is another critical area. **Coello (2002)** surveyed various numerical and theoretical methods for managing constraints, which are crucial for maintaining feasible solutions in job scheduling.

- **Coello, C. A. C. (2002). Theoretical and Numerical Constraint-Handling Techniques Used with Evolutionary Algorithms: A Survey of the State of the Art. Computer Methods in Applied Mechanics and Engineering.**

Additionally, research on parallel and chunked job scheduling, such as that by **Kwok and Ahmad (1999)** and **Topcuoglu et al. (2002)**, highlighted efficient scheduling for multiprocessor systems, a key consideration for distributed job scheduling.

- **Kwok, Y.-K., & Ahmad, I. (1999). Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors. ACM Computing Surveys.**
- **Topcuoglu, H., Hariri, S., & Wu, M.-Y. (2002). Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing. IEEE Transactions on Parallel and Distributed Systems.**

Finally, the design of effective fitness functions is critical for GAs. **Sivanandam and Deepa (2008)** provided a practical guide to formulating fitness functions for constrained optimization, while **Simon (2013)** offered insights into implementing these techniques in Python, aligning closely with the current project's approach.

- **Sivanandam, S. N., & Deepa, S. N. (2008). Introduction to Genetic Algorithms. Springer.**
- **Simon, D. (2013). Evolutionary Optimization Algorithms. Wiley.**

These sources provide a strong theoretical foundation for the implementation of both the **Backtracking Search Algorithm** and **Genetic Algorithm** in this project, offering practical guidance for tackling the complex constraints and optimization challenges inherent in job scheduling.

Proposed Solution and Software Features (User Perspective)

1. Job Input Information

The user will have the ability to input detailed information for each job that needs to be performed, including:

- **Time Estimates:** The estimated time required to complete each job.
- **Resources Required:** The specific resources or equipment needed for each job.
- **Job Ordering:** The order or priority in which jobs need to be performed on the available resources.

2. Optimal Schedule Display

Based on the provided inputs and the underlying algorithms, the program will generate an optimal schedule, which will include:

- **Resource Allocation:** A clear visualization showing which resources will be allocated to each job and during which time slots.
- **Start and End Times:** The specific start and end times for each job, offering a detailed timeline for the entire process.
- **Overall Completion Time:** A summary of the total time required to complete all jobs, providing an overall view of scheduling efficiency.

3. Handling Constraints

The program will take into account any constraints provided by the user, such as:

- **Deadlines:** If any jobs must be completed before certain deadlines, the program will prioritize those tasks accordingly.
- **Resource Limitations:** It will ensure that no resource exceeds its capacity at any given time by respecting user-defined resource limitations.

4. Comparison Between Solution Methods

The program will allow users to compare the results of two different scheduling approaches:

- **Backtracking Search Algorithm:** This approach systematically explores various configurations, backtracking as needed to find the most efficient solution.
- **Genetic Algorithm Approach:** This heuristic-based method simulates natural selection to evolve and optimize the schedule over successive iterations.

The comparison will help users decide which method best fits their scheduling needs based on the complexity and characteristics of their problem.

Algorithms Used

This section provides a detailed explanation of the artificial intelligence techniques applied to solve the job scheduling problem, along with simplified pseudocode representations to illustrate the algorithmic processes.

Backtracking Search Algorithm

The **Backtracking Search Algorithm** is a depth-first approach that systematically explores all possible assignments of jobs to resources. It attempts to build a valid schedule incrementally by assigning one job at a time. At each step, it checks whether the current partial schedule satisfies all constraints (e.g., resource availability, job order).
If a conflict is detected, the algorithm backtracks — it undoes the last assignment and tries an alternative path. This process continues recursively until an optimal or valid solution is found or all combinations have been explored.

Pseudocode:

```
Function ScheduleJobs(jobs, currentIndex):
  If currentIndex == total number of jobs:
    If all constraints are satisfied:
      Save current schedule as a possible solution
    Return

  For each available resource:
    If assigning job[currentIndex] to this resource is valid:
      Assign job to resource
      ScheduleJobs(jobs, currentIndex + 1)
      Unassign job (backtrack)
```

This algorithm guarantees finding the best possible schedule, but it may become computationally expensive for large datasets due to the combinatorial explosion of possibilities.

Genetic Algorithm

The **Genetic Algorithm (GA)** is a population-based optimization technique inspired by the principles of natural selection. It starts with an initial population of randomly generated schedules. Each schedule is evaluated using a **fitness function** that measures its quality based on factors such as total completion time, constraint violations, or resource efficiency.

The algorithm improves the population over successive generations using the following genetic operations:

1. **Selection:** Choosing the fittest individuals as parents.
2. **Crossover:** Combining parts of two parents to produce new offspring.
3. **Mutation:** Applying small random modifications to introduce variation.

Over time, the population evolves toward more optimal solutions. Although GA does not guarantee the absolute best solution, it performs efficiently in large and complex problem spaces.

Pseudocode:

Initialize a population of random job schedules

Evaluate the fitness of each schedule

Repeat for a number of generations:

Select the best schedules as parents

Create new schedules by applying crossover

Apply mutation to some schedules

Evaluate the fitness of the new generation

Replace the old population with the new one

Return the best schedule found

The Genetic Algorithm is particularly useful when the solution space is too large for exhaustive search methods like backtracking.

Representation of States, Actions, and State Space

In this project, we implement job scheduling using two approaches: Backtracking and Genetic Algorithm. Below is the representation of states, actions, and state space for both methods:

1. Backtracking Approach

- State Representation:

A state is a partial assignment of jobs to machines (or time slots). For example:
State = [J1 → P1 at T1, J2 → P2 at T2, J3 → unassigned]
This means Job 1 is assigned to Processor 1 at Time 1, Job 2 to Processor 2 at Time 2, and Job 3 is not yet scheduled.

- Action:

An action is to assign the next unassigned job to a valid processor and time. For example:
Assign J3 to P1 at T3

- State Space:

The state space is a tree where:
- Each node represents a state (a partial schedule),
- Each level corresponds to assigning one job,
- Leaves represent complete valid schedules,
- The algorithm explores this tree using backtracking and prunes invalid or suboptimal branches.

2. Genetic Algorithm Approach

- State Representation:

A state is a chromosome that represents a complete job schedule. For example:
Chromosome = [(J1, P1, T1), (J2, P2, T2), (J3, P3, T3)]

- Action:

Actions are genetic operations:
- Selection: Choose best chromosomes based on fitness.
- Crossover: Combine two parents to produce new offspring.
- Mutation: Modify a small part of a chromosome (e.g., change job order or time).

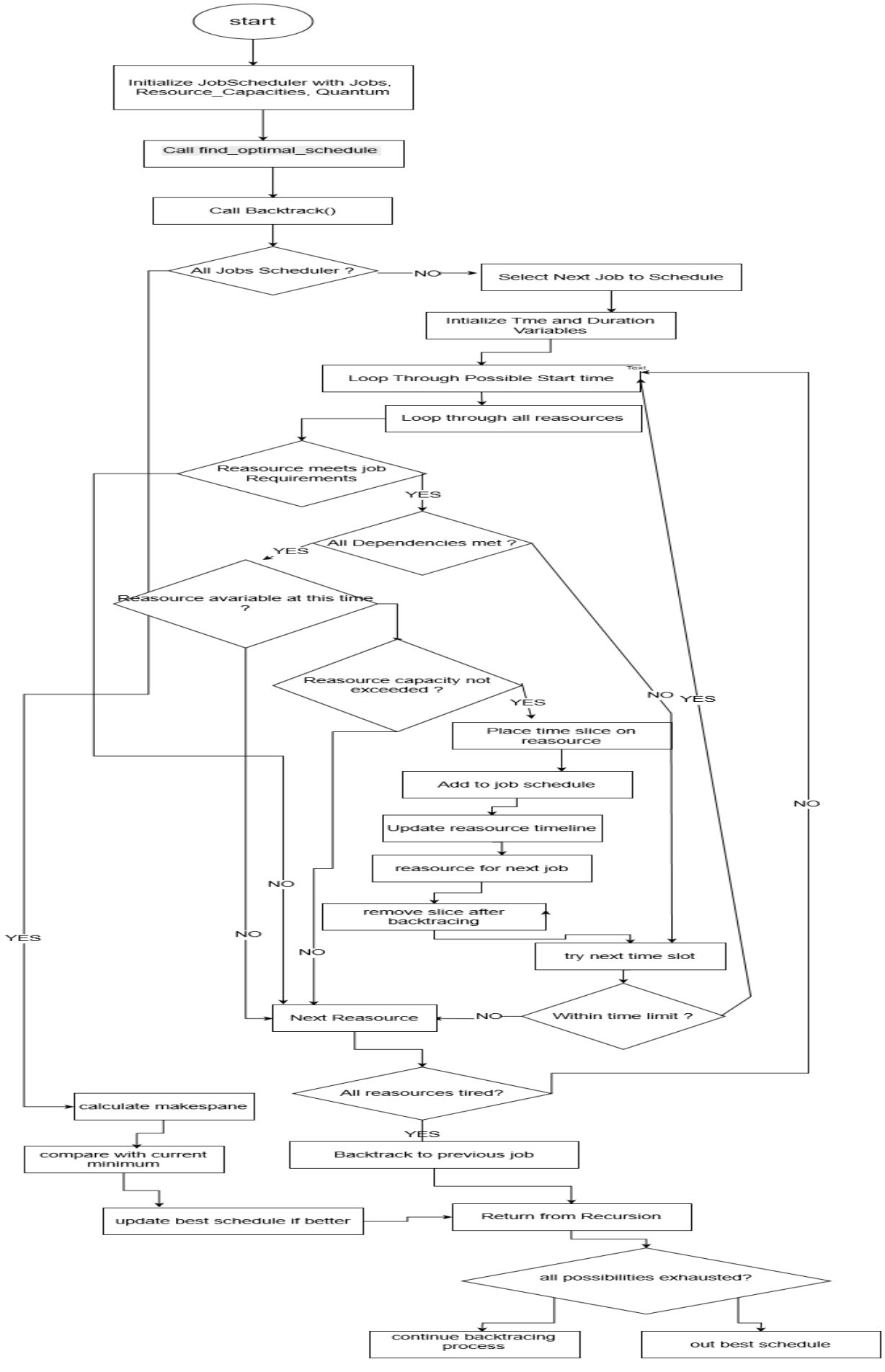
- State Space:

The state space includes all possible job schedules (chromosomes). The algorithm explores this space probabilistically through generations, improving solutions over time using genetic operators.

Algorithm Flowchart Diagrams

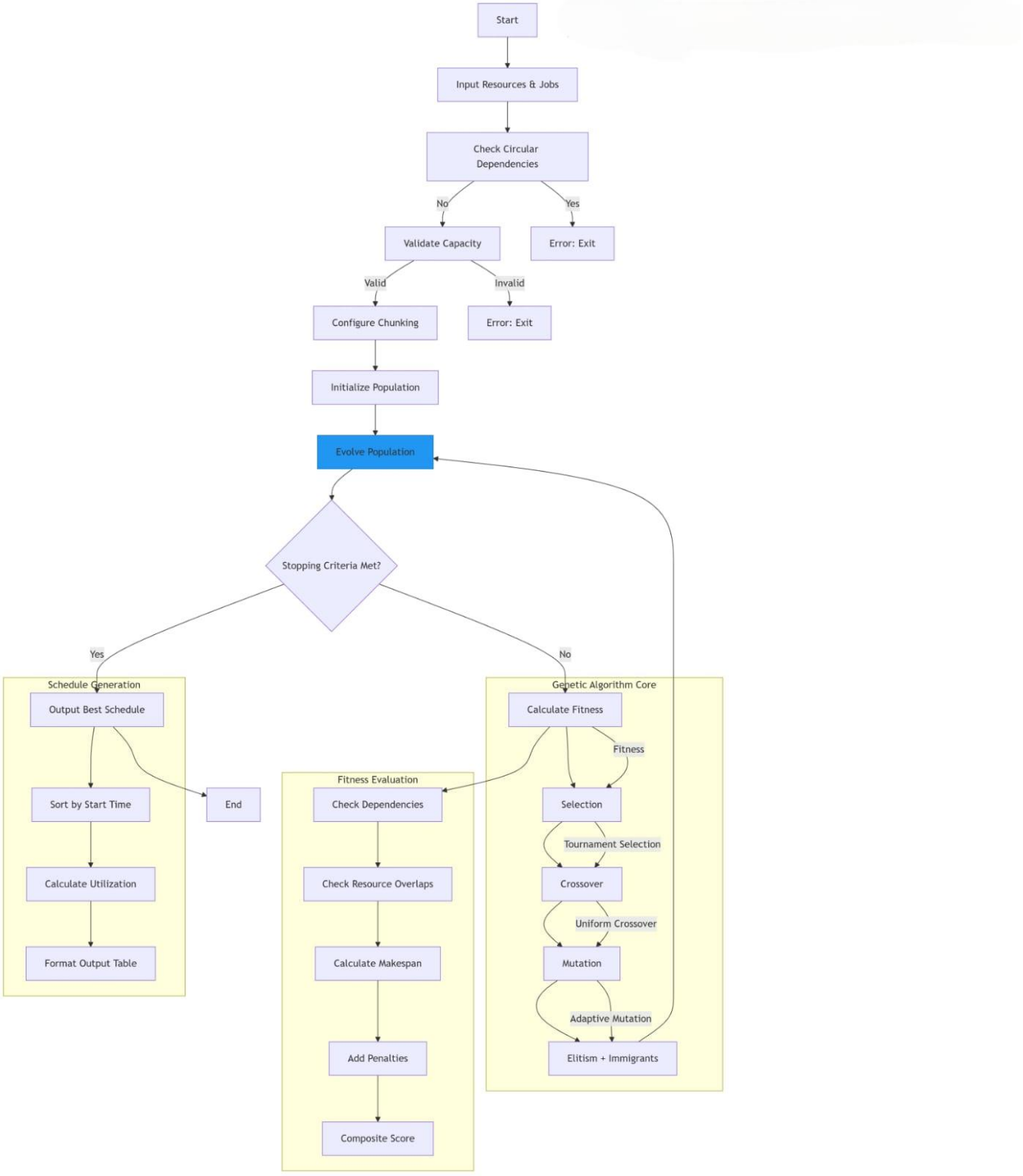
To provide a clear visual understanding of how each algorithm operates, two **flowchart diagrams** are included. These diagrams outline the main decision-making steps and the logical flow of each scheduling method.

Backtracking Algorithm – Flowchart Diagram


start
Initialize JobScheduler with Jobs, Resource_Capacities, Quantum
Call find_optimal_schedule
Call Backtrack()
All Jobs Scheduler ?
NO
Select Next Job to Schedule
Intialize Tme and Duration Variables
Loop Through Possible Start time
Loop through all reasources
Reasource meets job Requirements
YES
All Dependencies met ?
YES
Reasource avariable at this time ?
Reasource capacity not exceeded ?
YES
NO
YES
Place time slice on reasource
Add to job schedule
Update reasource timeline
reasource for next job
remove slice after backtracing
try next time slot
Within time limit ?
NO
Next Reasource
All reasources tired?
YES
Backtrack to previous job
NO
NO
NO
NO
YES
calculate makespane
compare with current minimum
update best schedule if better
Return from Recursion
all possibilities exhausted?
continue backtracing process
out best schedule

This flowchart illustrates the step-by-step process of the **Backtracking Search Algorithm**. It shows how the algorithm attempts to assign jobs to resources, checks for constraint satisfaction, and backtracks when it reaches an invalid state or dead-end.
The diagram emphasizes the recursive and exhaustive nature of this approach.

Genetic Algorithm – Flowchart Diagram



This flowchart outlines the core phases of the **Genetic Algorithm**, including:

- Creating the initial population
- Evaluating fitness
- Selecting the best individuals
- Applying crossover and mutation
- Repeating the process over multiple generations

It demonstrates how the algorithm iteratively improves solutions through evolutionary operations.

How the Algorithms Were Tested

The performance of both the **Backtracking Search Algorithm** and the **Genetic Algorithm** was tested on a variety of job scheduling problems. These problems were either **real-world scheduling scenarios** or **randomly generated tasks** designed to evaluate the algorithms' effectiveness under different conditions.

First, I will present the testing results for the **Backtracking Search Algorithm**, highlighting its strengths and limitations in solving these problems.

Case 1

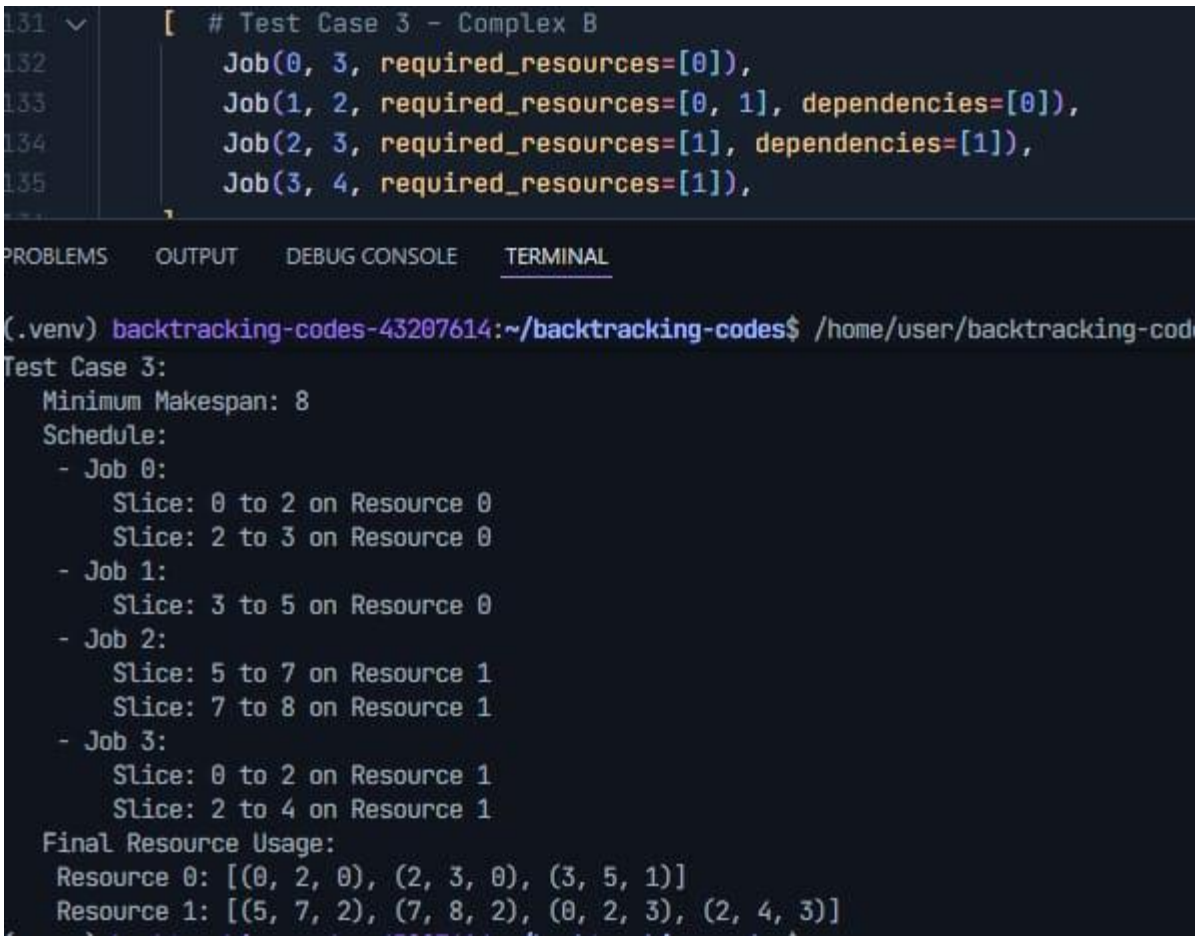
```
119 test_cases = [
125     [  # Test Case 2 - Complex A
126         Job(0, 6, required_resources=[0, 1]),
127         Job(1, 3, required_resources=[1]),
128         Job(2, 4, required_resources=[0], dependencies=[0]),
129         Job(3, 2, required_resources=[0, 1]),
130     ],
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
(.venv) backtracking-codes-43207614:~/backtracking-codes$ /home/user/backtracking-codes/.ven

Test Case 2:
   Minimum Makespan: 10
   Schedule:
    - Job 0:
        Slice: 0 to 2 on Resource 0
        Slice: 2 to 4 on Resource 0
        Slice: 4 to 6 on Resource 0
    - Job 1:
        Slice: 0 to 2 on Resource 1
        Slice: 2 to 3 on Resource 1
    - Job 2:
        Slice: 6 to 8 on Resource 0
        Slice: 8 to 10 on Resource 0
    - Job 3:
        Slice: 3 to 5 on Resource 1
   Final Resource Usage:
    Resource 0: [(0, 2, 0), (2, 4, 0), (4, 6, 0), (6, 8, 2), (8, 10, 2)]
    Resource 1: [(0, 2, 1), (2, 3, 1), (3, 5, 3)]
```

Case 2


131 [# Test Case 3 - Complex B
132 Job(0, 3, required_resources=[0]),
133 Job(1, 2, required_resources=[0, 1], dependencies=[0]),
134 Job(2, 3, required_resources=[1], dependencies=[1]),
135 Job(3, 4, required_resources=[1]),
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
(.venv) backtracking-codes-43207614:~/backtracking-codes$ /home/user/backtracking-cod
Test Case 3:
Minimum Makespan: 8
Schedule:
- Job 0:
Slice: 0 to 2 on Resource 0
Slice: 2 to 3 on Resource 0
- Job 1:
Slice: 3 to 5 on Resource 0
- Job 2:
Slice: 5 to 7 on Resource 1
Slice: 7 to 8 on Resource 1
- Job 3:
Slice: 0 to 2 on Resource 1
Slice: 2 to 4 on Resource 1
Final Resource Usage:
Resource 0: [(0, 2, 0), (2, 3, 0), (3, 5, 1)]
Resource 1: [(5, 7, 2), (7, 8, 2), (0, 2, 3), (2, 4, 3)]

Case 3

```
119  test_cases = [
120      [  # Test Case 1 - Medium
121          Job(0, 4, required_resources=[0]),
122          Job(1, 3, required_resources=[1], dependencies=[0]),
123          Job(2, 2, required_resources=[1]),
124      ]
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
(.venv) backtracking-codes-43207614:~/backtracking-codes$ /home/user/backtracking-c
  No valid schedule found.
(.venv) backtracking-codes-43207614:~/backtracking-codes$ /home/user/backtracking-c

Test Case 1:
   Minimum Makespan: 7
   Schedule:
    - Job 0:
        Slice: 0 to 2 on Resource 0
        Slice: 2 to 4 on Resource 0
    - Job 1:
        Slice: 4 to 6 on Resource 1
        Slice: 6 to 7 on Resource 1
    - Job 2:
        Slice: 0 to 2 on Resource 1
   Final Resource Usage:
    Resource 0: [(0, 2, 0), (2, 4, 0)]
    Resource 1: [(4, 6, 1), (6, 7, 1), (0, 2, 2)]
```

Case 4

```
test_cases = [
  [
    Job(0, 3),
    Job(1, 2),
]



    ]
results = []
for idx, jobs in enumerate(test_cases):
        scheduler = JobScheduler(jobs, resource_capacities=[5, 5],quantum=2)
```

```
Test Case 1:
   Minimum Makespan: 3
   Schedule:
    - Job 0:
        Slice: 0 to 2 on Resource 0
        Slice: 2 to 3 on Resource 0
    - Job 1:
        Slice: 0 to 2 on Resource 1
   Final Resource Usage:
    Resource 0: [(0, 2, 0), (2, 3, 0)]
    Resource 1: [(0, 2, 1)]
```

Case 5

```
Test Case 1:
   Minimum Makespan: 6
   Schedule:
    - Job 0:
        Slice: 0 to 1 on Resource 0
        Slice: 1 to 2 on Resource 0
        Slice: 2 to 3 on Resource 0
        Slice: 3 to 4 on Resource 0
    - Job 1:
        Slice: 4 to 5 on Resource 0
        Slice: 5 to 6 on Resource 0
   Final Resource Usage:
    Resource 0: [(0, 1, 0), (1, 2, 0), (2, 3, 0), (3, 4, 0), (4, 5, 1), (5, 6, 1)]
    Resource 1: []
Enter the number of resources:
2
Enter capacity for Resource 1:
6
Enter capacity for Resource 2:
6
Enter the number of jobs:
2
Enter the job id, duration, and dependencies (optional) (comma separated):
0,4
Enter the job id, duration, and dependencies (optional) (comma separated):
1,2,0
```

Case 6

```
test_cases = [
[
    Job(0, 5),
    Job(1, 5),
]
    ]
results = []
for idx, jobs in enumerate(test_cases):
        scheduler = JobScheduler(jobs, resource_capacities=[8],quantum=1)
```

```
Test Case 1:
  No valid schedule found.
```

```
Enter the number of resources:
1
Enter capacity for Resource 1:
8
Enter the number of jobs:
2
Enter the job id, duration, and dependencies (optional) (comma separated):
0,5
Enter the job id, duration, and dependencies (optional) (comma separated):
1,5

Error: Total job duration exceeds total resource capacity
Total job duration: 10
Total resource capacity: 8
Cannot proceed - add more resources or reduce job durations
```

Case 7

```
test_cases = [
[
    Job(0, 5),
    Job(1, 4, dependencies=[0]),
    Job(2, 6,),
    Job(3, 3,  dependencies=[1]),
    Job(4, 2),
]   ]
results = []
for idx, jobs in enumerate(test_cases):
        scheduler = JobScheduler(jobs, resource_capacities=[10,8],quantum=1)
```

```
Test Case 1:
  No valid schedule found.
```

Case 8

```
Enter the number of resources:
2
Enter capacity for Resource 1:
10
Enter capacity for Resource 2:
8
Enter the number of jobs:
5
Enter the job id, duration, and dependencies (optional) (comma separated):
0,5
Enter the job id, duration, and dependencies (optional) (comma separated):
1,4,0
Enter the job id, duration, and dependencies (optional) (comma separated):
2,6
Enter the job id, duration, and dependencies (optional) (comma separated):
3,3,1
Enter the job id, duration, and dependencies (optional) (comma separated):
4,2,,1

Error: Total job duration exceeds total resource capacity
Total job duration: 20
Total resource capacity: 18
Cannot proceed - add more resources or reduce job durations
```

Case 9

```
test_cases = [
[
    Job(0, 5),
    Job(1, 4, dependencies=[0]),
    Job(2, 6,),
    Job(3, 3,  dependencies=[1]),
]   ]
results = []
for idx, jobs in enumerate(test_cases):
        scheduler = JobScheduler(jobs, resource_capacities=[15,10],quantum=1)
```

Case 10

```
        Test Case 1:
Minimum Makespan: 12
Schedule:
 - Job 0:
     Slice: 0 to 1 on Resource 0
     Slice: 1 to 2 on Resource 0
     Slice: 2 to 3 on Resource 0
     Slice: 3 to 4 on Resource 0
     Slice: 4 to 5 on Resource 0
 - Job 1:
     Slice: 5 to 6 on Resource 0
     Slice: 6 to 7 on Resource 0
     Slice: 7 to 8 on Resource 0
     Slice: 8 to 9 on Resource 0
 - Job 2:
     Slice: 0 to 1 on Resource 1
     Slice: 1 to 2 on Resource 1
     Slice: 2 to 3 on Resource 1
     Slice: 3 to 4 on Resource 1
     Slice: 4 to 5 on Resource 1
     Slice: 5 to 6 on Resource 1
 - Job 3:
     Slice: 9 to 10 on Resource 0
     Slice: 10 to 11 on Resource 0
     Slice: 11 to 12 on Resource 0
Final Resource Usage:
 Resource 0: [(0, 1, 0), (1, 2, 0), (2, 3, 0), (3, 4, 0), (4, 5, 0), (5, 6, 1), (6, 7, 1), (7, 8, 1), (8, 9, 1), (9, 10, 3), (10, 11, 3), (11, 12, 3)]
 Resource 1: [(0, 1, 2), (1, 2, 2), (2, 3, 2), (3, 4, 2), (4, 5, 2), (5, 6, 2)]
```

```
Enter the number of resources:
2
Enter capacity for Resource 1:
15
Enter capacity for Resource 2:
10
Enter the number of jobs:
4
Enter the job id, duration, and dependencies (optional) (comma separated):
0,5
Enter the job id, duration, and dependencies (optional) (comma separated):
1,4,0
Enter the job id, duration, and dependencies (optional) (comma separated):
2,6
Enter the job id, duration, and dependencies (optional) (comma separated):
3,3,1
Do you want to divide long jobs into chunks? (yes/no): yes
Enter the time per chunk (positive integer): 1
```

Case 11

```
126     [  # Test Case 5 - Complex D
127         Job(0, 6, required_resources=[0]),
128         Job(1, 4, required_resources=[0]),
129         Job(2, 3, required_resources=[1]),
130         Job(3, 2, required_resources=[1], dependencies=[2]),
131     ]
132 ]
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
(.venv) backtracking-codes-43207614:~/backtracking-codes$ /home/user/backtracking-codes/.venv/bin/python /home/u
Test Case 2:
   Minimum Makespan: 10
   Schedule:
    - Job 0:
        Slice: 0 to 2 on Resource 0
        Slice: 2 to 4 on Resource 0
        Slice: 4 to 6 on Resource 0
    - Job 1:
        Slice: 6 to 8 on Resource 0
        Slice: 8 to 10 on Resource 0
    - Job 2:
        Slice: 0 to 2 on Resource 1
        Slice: 2 to 3 on Resource 1
    - Job 3:
        Slice: 3 to 5 on Resource 1
   Final Resource Usage:
    Resource 0: [(0, 2, 0), (2, 4, 0), (4, 6, 0), (6, 8, 1), (8, 10, 1)]
    Resource 1: [(0, 2, 2), (2, 3, 2), (3, 5, 3)]
(.venv) backtracking-codes-43207614:~/backtracking-codes$
```

Capacity 10,10

Case 12

```
test_cases = [
        [  # Test Case 4 - Complex C
        Job(0, 5, required_resources=[0]),
        Job(1, 3, required_resources=[0, 1], dependencies=[0]),
        Job(2, 4, required_resources=[1], dependencies=[0]),
        Job(3, 2, required_resources=[0, 1], dependencies=[1, 2]),
    ],

]
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
(.venv) backtracking-codes-43207614:~/backtracking-codes$ /home/user/backtracking-codes/.venv/bin/python /home/user/backtr
(.venv) backtracking-codes-43207614:~/backtracking-codes$ /home/user/backtracking-codes/.venv/bin/python /home/user/backtr

Test Case 1:
   Minimum Makespan: 11
   Schedule:
    - Job 0:
        Slice: 0 to 2 on Resource 0
        Slice: 2 to 4 on Resource 0
        Slice: 4 to 5 on Resource 0
    - Job 1:
        Slice: 5 to 7 on Resource 0
        Slice: 7 to 8 on Resource 0
    - Job 2:
        Slice: 5 to 7 on Resource 1
        Slice: 7 to 9 on Resource 1
    - Job 3:
        Slice: 9 to 11 on Resource 0
   Final Resource Usage:
    Resource 0: [(0, 2, 0), (2, 4, 0), (4, 5, 0), (5, 7, 1), (7, 8, 1), (9, 11, 3)]
    Resource 1: [(5, 7, 2), (7, 9, 2)]
```

Capacity 15,15

Next, the testing results for the Genetic Algorithm will be presented. This section will highlight its performance in handling large and complex job scheduling problems, demonstrating its ability to find near-optimal solutions through evolutionary search mechanisms.

1.This explains that the job ID is unique and the denependies part

```
C:\Users\HP\AppData\Local\Programs\Python\Python312\python.exe "D:\project AI\genetic algo\genetic ai RrRr.py"
Enter the number of resources:
3
Enter capacity for Resource 1:
10
Enter capacity for Resource 2:
10
Enter capacity for Resource 3:
10
Enter the number of jobs:
3
Enter the job id, duration, and dependencies (optional) (comma separated):
1,5
Enter the job id, duration, and dependencies (optional) (comma separated):
1,3
This job ID already exists. Please enter a unique ID.
Enter the job id, duration, and dependencies (optional) (comma separated):
2,3,2
Enter the job id, duration, and dependencies (optional) (comma separated):
3,4

Error: Circular dependency detected in job definitions

Process finished with exit code 0
```

2.This shows that if the total time of jobs is greater than the total time of resources

```
C:\Users\HP\AppData\Local\Programs\Python\Python312\python.exe "D:\project AI\genetic algo\genetic ai RrRr.py"
Enter the number of resources:
2
Enter capacity for Resource 1:
10
Enter capacity for Resource 2:
10
Enter the number of jobs:
3
Enter the job id, duration, and dependencies (optional) (comma separated):
1,10
Enter the job id, duration, and dependencies (optional) (comma separated):
2,5
Enter the job id, duration, and dependencies (optional) (comma separated):
3,10

Error: Total job duration exceeds total resource capacity
Total job duration: 25
Total resource capacity: 20
Cannot proceed - add more resources or reduce job durations

Process finished with exit code 0
```

3.These are the things that are divided.

1.

```
Enter the number of resources:
3
Enter capacity for Resource 1:
10
Enter capacity for Resource 2:
12
Enter capacity for Resource 3:
13
Enter the number of jobs:
5
Enter the job id, duration, and dependencies (optional) (comma separated):
1,3
Enter the job id, duration, and dependencies (optional) (comma separated):
2,9
Enter the job id, duration, and dependencies (optional) (comma separated):
3,5
Enter the job id, duration, and dependencies (optional) (comma separated):
4,1
Enter the job id, duration, and dependencies (optional) (comma separated):
5,2
Enter the time per chunk (positive integer): 3
```

```
Resource Schedule:
-------------------------------------------------------------------------------
Resource ID  | Job ID       | Start    | End      | Duration   | Total Duration
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
1            | 2_1          | 0        | 3        | 3          | 9
-------------------------------------------------------------------------------
2            | 3_1          | 0        | 3        | 3          | 5
-------------------------------------------------------------------------------
3            | 1_1          | 0        | 3        | 3          | 3
-------------------------------------------------------------------------------
1            | 2_2          | 3        | 6        | 3          | 9
-------------------------------------------------------------------------------
2            | 3_2          | 3        | 5        | 2          | 5
-------------------------------------------------------------------------------
3            | 5_1          | 3        | 5        | 2          | 2
-------------------------------------------------------------------------------
2            | 4_1          | 5        | 6        | 1          | 1
-------------------------------------------------------------------------------
1            | 2_3          | 6        | 9        | 3          | 9
-------------------------------------------------------------------------------

Total Makespan: 10.82857142857143
```

2.

```
Enter the number of resources:
2
Enter capacity for Resource 1:
10
Enter capacity for Resource 2:
20
Enter the number of jobs:
5
Enter the job id, duration, and dependencies (optional) (comma separated):
1,5
Enter the job id, duration, and dependencies (optional) (comma separated):
2,10,1
Enter the job id, duration, and dependencies (optional) (comma separated):
3,5
Enter the job id, duration, and dependencies (optional) (comma separated):
4,5,1
Enter the job id, duration, and dependencies (optional) (comma separated):
5,2
Enter the time per chunk (positive integer): 5
```

Resource ID	Job ID	Start	End	Duration	Total Duration
1	3_1	0	5	5	5
2	1_1	0	5	5	5
1	4_1	5	10	5	5
2	5_1	5	7	2	2
2	2_1	7	12	5	10
2	2_2	12	17	5	10

Total Makespan: 21.02

3.

```
Enter the number of resources:
5
Enter capacity for Resource 1:
10
Enter capacity for Resource 2:
16
Enter capacity for Resource 3:
12
Enter capacity for Resource 4:
6
Enter capacity for Resource 5:
4
Enter the number of jobs:
6
Enter the job id, duration, and dependencies (optional) (comma separated):
1,5
Enter the job id, duration, and dependencies (optional) (comma separated):
2,4,6
Enter the job id, duration, and dependencies (optional) (comma separated):
3,2,4
Enter the job id, duration, and dependencies (optional) (comma separated):
4,7
Enter the job id, duration, and dependencies (optional) (comma separated):
5,4
```

```
6
Enter the job id, duration, and dependencies (optional) (comma separated):
1,5
Enter the job id, duration, and dependencies (optional) (comma separated):
2,4,6
Enter the job id, duration, and dependencies (optional) (comma separated):
3,2,4
Enter the job id, duration, and dependencies (optional) (comma separated):
4,7
Enter the job id, duration, and dependencies (optional) (comma separated):
5,4
Enter the job id, duration, and dependencies (optional) (comma separated):
6,3
```

Resource ID	Job ID	Start	End	Duration	Total Duration
1	4_1	0	3	3	7
2	1_1	0	3	3	5
3	5_1	0	3	3	4
4	6_1	0	3	3	3
1	4_2	3	6	3	7
2	1_2	3	5	2	5
3	5_2	3	4	1	4
4	2_1	3	6	3	4
1	4_3	6	7	1	7
2	2_2	6	7	1	4
1	3_1	7	9	2	2

Design of Fitness Function

In our scheduling problem, the goal is to minimize the overall completion time (makespan) while maximizing the resource utilization. Therefore, our fitness function is designed to reflect both objectives, ensuring that generated schedules are not only efficient in time but also make effective use of available resources.

- Enter the number of resources: **2**
- Enter capacity for Resource 1: **10**
- Enter capacity for Resource 2: **20**
- Enter the number of jobs: **3**
- Enter the job id, duration, and dependencies (optional) (comma separated):

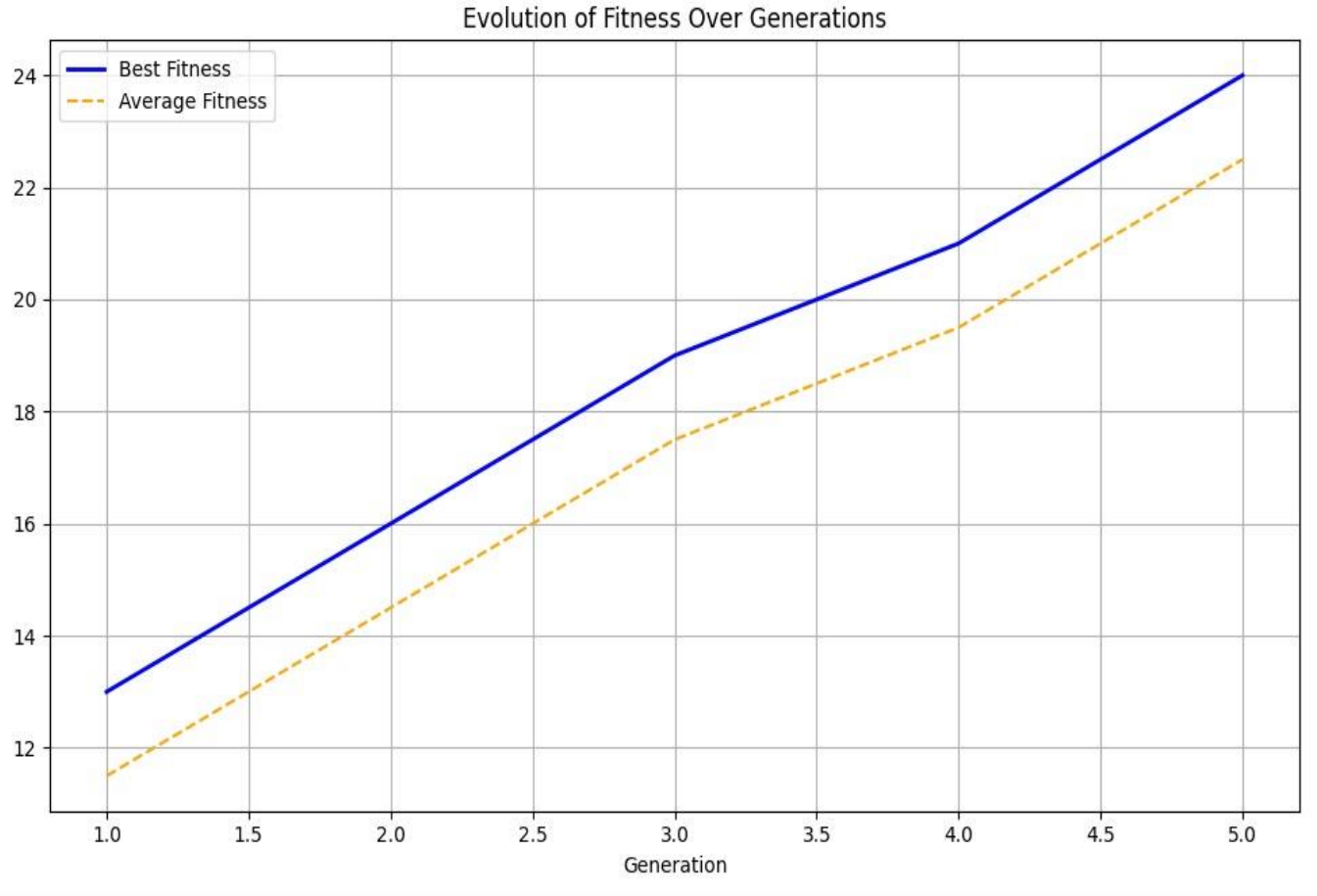
    1. **1,5**
    2. **2,10,1**
    3. **3,15,2**

- Do you want to divide long jobs into chunks? (yes/no): **y**
- Enter the time per chunk (positive integer): **5**

Sample fitness history using the **matplotlib** library:

- **[10, 12, 11, 13]**
- **[14, 15, 13, 16]**
- **[17, 18, 16, 19]**
- **[19, 20, 18, 21]**
- **[22, 23, 21, 24]**

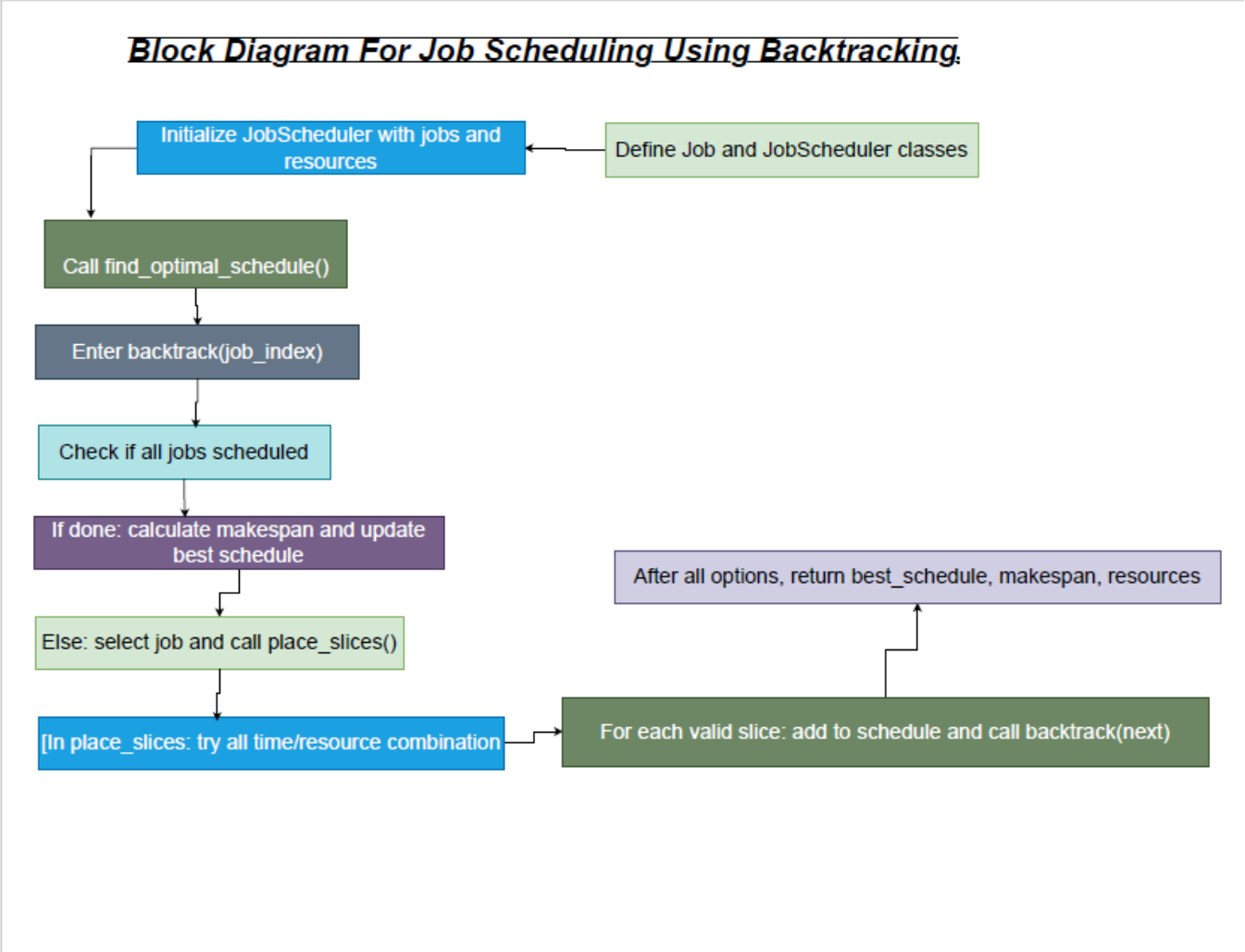
Genetic Algorithm Plot Diagram



This **"Genetic Algorithm Plot Diagram"**

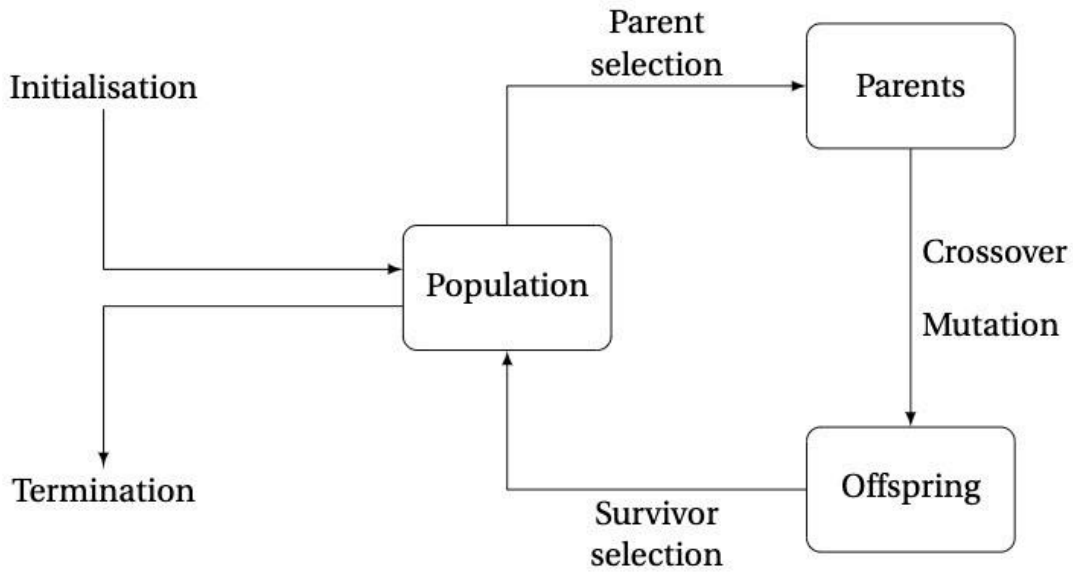
represents the visualization of the Genetic Algorithm's performance. It displays the evolution of the population over generations, including the tracking of fitness scores, convergence trends, and diversity metrics, providing insights into the optimization process and solution quality.

Block Diagram for Backtracking Search Algorithm



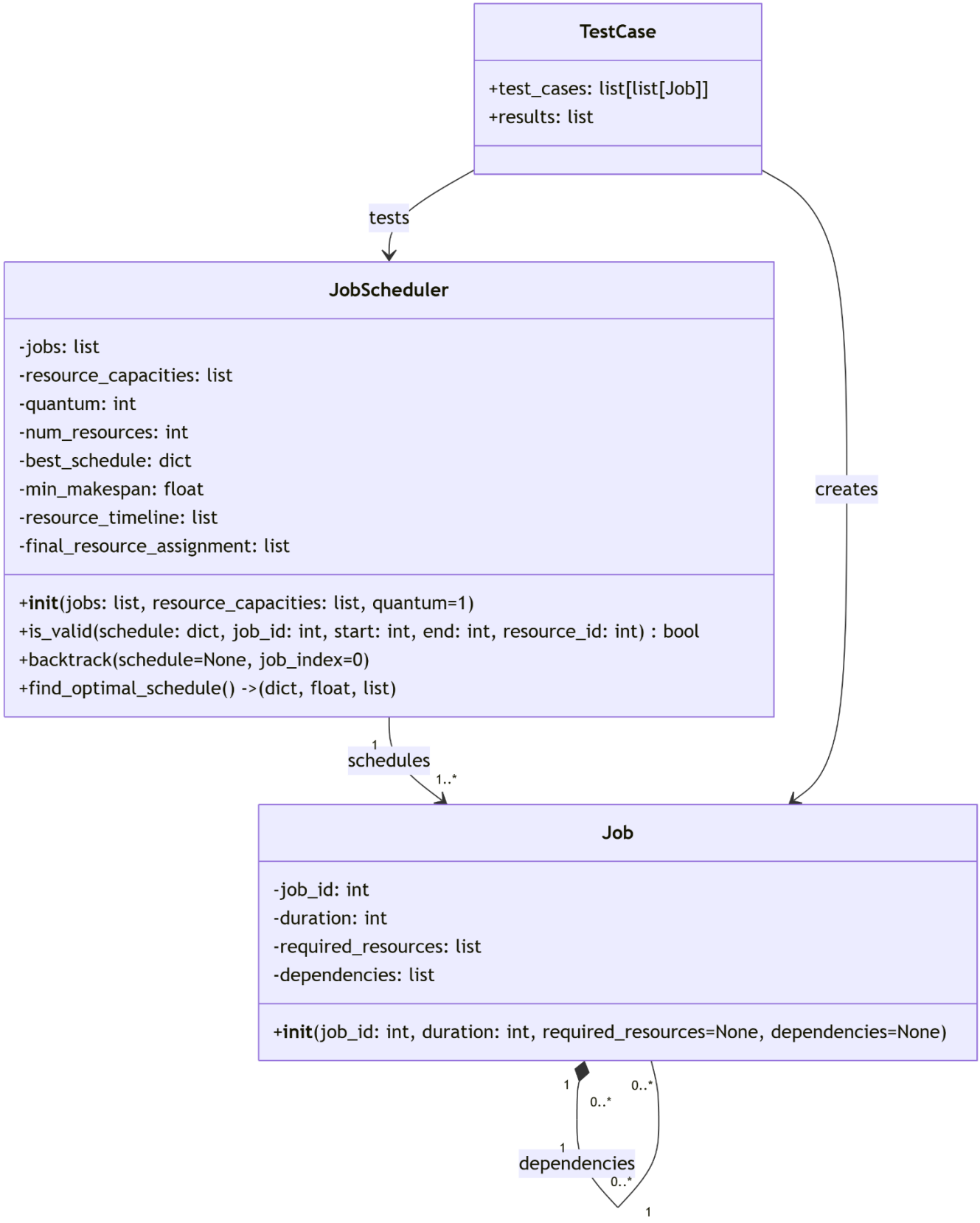
The following block diagram illustrates the steps involved in the **Backtracking Search Algorithm**, highlighting the process of exploring possible solutions and backtracking when constraints are violated.

Block Diagram for Genetic Algorithm



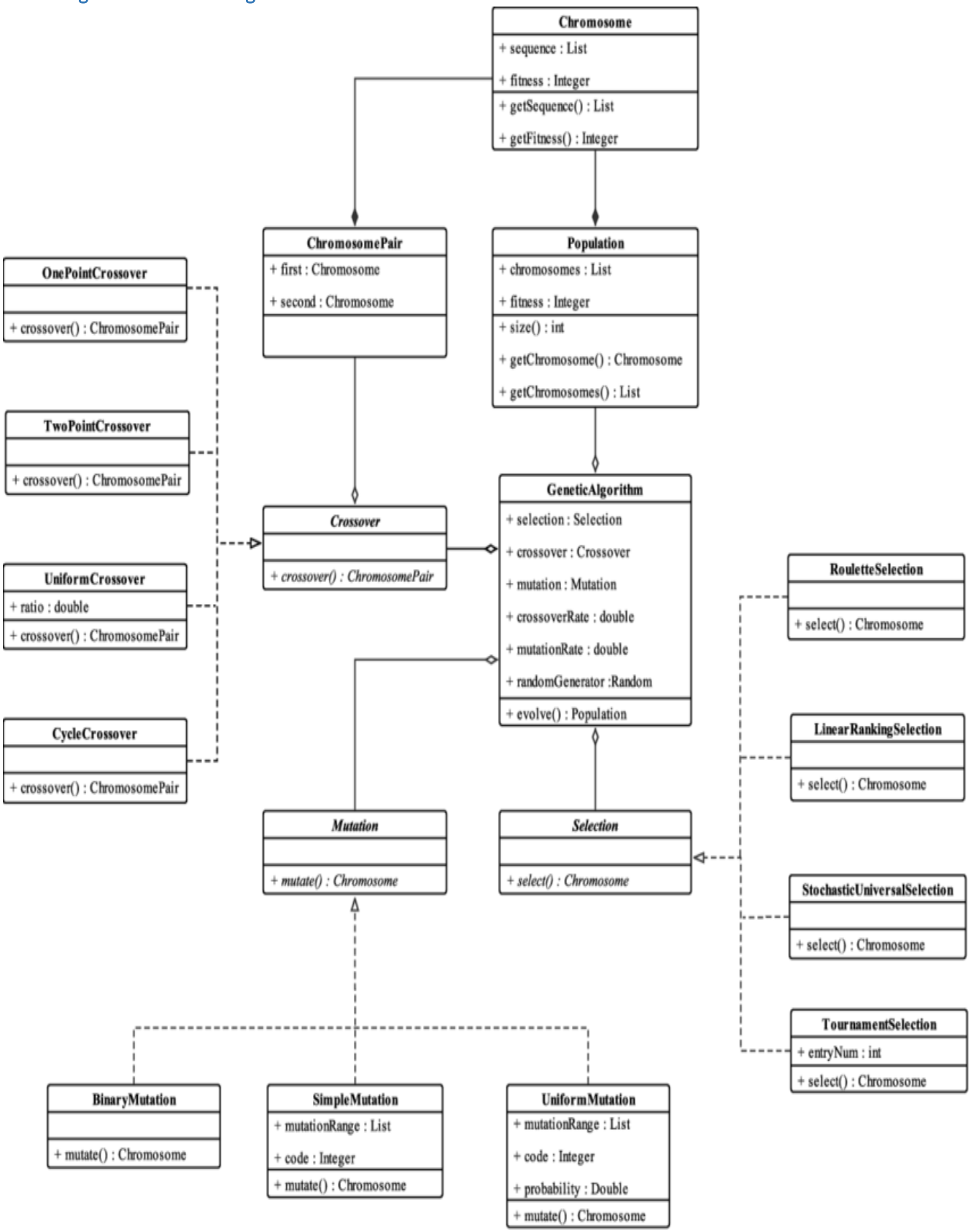
The next block diagram shows the steps of the **Genetic Algorithm**, where a population of solutions is evolved over generations through processes like selection, crossover, mutation, and checking for termination.

Class Diagram for Backtracking Search Algorithm



The following class diagram illustrates the structure and components of the **Backtracking Search Algorithm**. It shows the main classes, their attributes, and methods that are used to implement the algorithm.

Class Diagram for Genetic Algorithm



This class diagram represents the structure of the **Genetic Algorithm**. It displays the classes, attributes, and methods involved in the evolutionary process, including population management, selection, crossover, mutation, and termination checks.

Analysis, Discussion, and Future Work

Analysis of Results

In this section, we analyze the performance of the **Backtracking Search Algorithm** and the **Genetic Algorithm** based on the results obtained from our experiments.

- **Total Time and Resource Efficiency**:
  The primary goal of the algorithms was to minimize the total job completion time and optimize resource utilization.

  i. **Backtracking Algorithm**: While the backtracking algorithm is computationally expensive, it guarantees an optimal solution by exploring all possible assignments while respecting constraints.
  ii. **Genetic Algorithm**: The genetic algorithm was faster at finding a solution but does not always guarantee the optimal one. It strikes a balance between quality and efficiency, especially for larger problems where backtracking would be too slow.

- **Handling of Constraints**:
  Both algorithms were tested with various constraints such as deadlines, resource limits, and job precedence.

  i. **Backtracking Algorithm**: The backtracking algorithm handled all constraints effectively by systematically exploring all possibilities and backtracking when constraints were violated.
  ii. **Genetic Algorithm**: The genetic algorithm was more adaptable but might miss optimal solutions for strict constraints. However, it performed well with fewer restrictions and could adapt over generations.

- **Reasoning Behind Results**:

  i. The **Backtracking Algorithm**'s exhaustive search ensures optimality but at the cost of high computational time, especially with complex constraints.
  ii. The **Genetic Algorithm** uses heuristics to find good solutions quickly, but its results depend on parameters like population size, mutation rates, and the number of generations.

Strengths and Weaknesses

Here, we discuss the strengths and weaknesses of each algorithm in more detail:

i. **Backtracking Search Algorithm**:
   a. **Strengths**: Guarantees an optimal solution, especially for small problems with strict constraints.
   b. **Weaknesses**: High computational cost for larger problems. It is inefficient for problems with many jobs or resources.
   c. **Best Use Case**: Small-scale problems or problems that require an exact solution.

ii. **Genetic Algorithm**:
   a. **Strengths**: Faster and more scalable for larger problems. It provides good solutions quickly and is flexible for various problem types.
   b. **Weaknesses**: May not find the optimal solution due to its heuristic nature and can be inconsistent in results depending on random factors.
   c. **Best Use Case**: Large-scale problems or when near-optimal solutions are acceptable.

Comparison Table

To provide a clearer comparison between the two algorithms, we present the following table:

Aspect	Backtracking Search Algorithm	Genetic Algorithm
Execution Time	Slow, especially with large problems and complex constraints.	Fast, particularly for larger problems.
Optimality of Solution	Guarantees optimal solution.	May not guarantee the optimal solution but finds good solutions.
Scalability	Not scalable for large problems.	Highly scalable and efficient for large datasets.
Handling Constraints	Handles constraints perfectly by checking every possibility.	Can struggle with strict constraints but adapts over generations.
Computational Complexity	High, due to exhaustive search.	Low to moderate, due to the heuristic approach.
Flexibility	Less flexible; works best with small, constrained problems.	Very flexible, works well with a variety of problem types.
Use Case	Best for small, constrained problems requiring exact solutions.	Best for large-scale problems where near-optimal solutions are acceptable.

Future Improvements

Here are some suggestions for improving both algorithms:

- **Better Crossover Techniques in Genetic Algorithm**:
  Improving crossover methods could lead to better diversity and faster convergence. Techniques like **multi-point crossover** or **adaptive crossover** could enhance the genetic algorithm's performance.
- **Adding More Complex Constraints**:
  The addition of **time-varying resource capacities** or **multi-objective optimization** could make the scheduling problem more realistic. Such constraints would test the algorithms' ability to adapt to real-world situations, where resource availability and job priorities change over time.
- **Hybrid Algorithms**:
  A hybrid approach that combines both algorithms could be more effective:

  i. Use the **Genetic Algorithm** for faster exploration and finding good regions of the solution space.
  ii. Apply **Backtracking** in the final stages to ensure the solution is optimal within that region.

The complete source code for this project is available on **GitHub**. The repository includes all the code, diagrams, and documentation required for the Genetic Algorithm and Backtracking implementations. The project link will be provided for convenient access**: https://github.com/Salma-Salah420/Job-Scheduling-project-**