

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  | | Analysis of AlgorithmsMostapha 227824. |  | | |
|  |  | |  | |  | |
|  |  | | Optimizing Resource Allocation in Manufacturing environment. | |  | |
|  | **Table of Contents**  Problem Statement--------------------------------------------------------------------------2  1.1 Description------------------------------------------------------------------------------------2  Paradigms--------------------------------------------------------------------------------------2  2.1 Greedy------------------------------------------------------------------------------------------2  2.2 Dynamic Programming-----------------------------------------------------------------------3  2.3 Evolutionary Algorithms----------------------------------------------------------------------3  2.4 Why These Paradigms-------------------------------------------------------------------------3  Pseudo Code for Paradigms-----------------------------------------------------------------4  3.1 Greedy Algorithm----------------------------------------------------------------------------4  3.2 Dynamic Programming Algorithm-----------------------------------------------------------5  3.3 Evolutionary Algorithm-----------------------------------------------------------------------5  Analysis of the Implemented Algorithms------------------------------------------------6  4.1 Greedy Algorithm----------------------------------------------------------------------------7  4.1.1 Time Complexity Analysis--------------------------------------------------------------7  4.1.2 Memory Consumption Analysis----------------------------------------------------------7  4.2 Dynamic Programming Algorithm-----------------------------------------------------------8  4.2.1 Time Complexity Analysis--------------------------------------------------------------8  4.2.2 Memory Consumption Analysis----------------------------------------------------------8  4.3 Evolutionary Algorithm-----------------------------------------------------------------------8  4.3.1 Time Complexity Analysis--------------------------------------------------------------8  4.3.2 Memory Consumption Analysis----------------------------------------------------------8  Comparison Between the Implemented Algorithms------------------------------------9  5.1 Time and Memory Efficiency----------------------------------------------------------------9  5.2 Sorting the Algorithms-----------------------------------------------------------------------9  Recommendations------------------------------------------------------------------------------10 | | | | |  |
|  | **1.Problem Statement:**  To reduce manufacturing time and increase efficiency, a corporation that manufactures goods must strategically assign its equipment to different production activities. The organization has a list of manufacturing jobs with precise criteria and dates, as well as a collection of equipment with varying capacities. The goal is to give machines assignments in a way that maximizes production time and guarantees that every work is finished ahead of schedule.  **1.1.Description:**  The firm has many manufacturing lines, each of which has a number of equipment that can carry out various functions including welding, cutting, and assembling. To fulfil client deadlines, every manufacturing task has to be accomplished in a precise order and within a set amount of time. Furthermore, it's possible that certain jobs depend on others, necessitating their completion in order.  **2.Paradigms:**   * + 1. **Greedy:**   Under the greedy method, machines are given tasks according to a heuristic that chooses the most convenient or locally optimum option at each stage. The machine with the lowest processing time or the fewest outstanding tasks might be given priority when tasks are assigned by the algorithm. Although this method is computationally efficient, it might not always produce the best answer since it might sacrifice long-term profits for better short-term assignments.   * + 1. **Dynamic Programming:**   Dynamic programming breaks down the production scheduling problem into smaller overlapping subproblems and solves each subproblem only once, storing the solutions to avoid redundant calculations. To store the ideal timetable for finishing each production work by the deadline, the algorithm may, for instance, generate a dynamic programming table. The method makes effective use of machine capabilities and task dependencies to determine the best schedule for every machine.   * + 1. **Evolutionary Algorithms:**   Evolutionary algorithms use principles of natural selection to iteratively evolve a population of candidate solutions towards the optimal solution. Each potential answer might be represented by the algorithm, for instance, as a schedule of machine-assigned production jobs. The system creates new candidate solutions through processes like crossover and mutation, then assesses their fitness using metrics like task deadline adherence and overall production time. The algorithm gradually improves the population through several generations until the ideal production schedule is achieved.   * 1. **Why these paradigms:**   **2.2.1 Greedy:**   * + 1. The greedy approach provides a straightforward and computationally effective way to make judgements at each stage, which makes it appropriate for this task. Grateful algorithms' local optimization in manufacturing resource allocation may frequently provide in workable solutions, particularly when there are specific and urgent goals to optimize for, including reducing processing time or the quantity of open tasks on each machine. Greedy algorithms don't have to go through the whole solution space to swiftly distribute work to machines based on certain criteria. It's crucial to remember that the greedy strategy could not always result in the globally optimal solution since it might make choices that, while seemingly ideal at the time, could ultimately lead to less-than-ideal results.     2. **Dynamic programming:**   Dynamic programming is well-suited for problems with overlapping subproblems and optimal substructure, both of which are characteristics of the manufacturing resource allocation problem. Dynamic programming can efficiently compute the ideal timetable for completing each production job within its deadline while taking interdependence between tasks and machine capabilities into consideration. This is achieved by splitting the issue down into smaller subproblems and solving each subproblem only once. By eliminating duplicate calculations, this methodology may handle bigger issue instances more effectively than brute force methods.   * + 1. **Evolutionary Algorithms:**   Evolutionary algorithms offer a flexible and adaptive approach to solving optimization problems, making them suitable for complex and dynamic environments such as manufacturing resource allocation. Evolutionary algorithms can explore a diverse range of solutions and adaptively refine them over multiple iterations, potentially discovering novel and unexpected solutions that may not be apparent through other approaches. Additionally, evolutionary algorithms can handle constraints and objectives that are not easily captured by traditional optimization techniques. In the context of manufacturing resource allocation, evolutionary algorithms can evolve schedules of production tasks assigned to machines, optimizing for criteria such as total production time and adherence to task deadlines. They can also handle variations and uncertainties in production environments, making them robust solutions for real-world manufacturing scenarios.  **3.pesudo code for paradigms:**   * 1. **Greedy Algorithm:**   **Input:**  **-**set of machines 𝑀 = { 𝑚 1 , 𝑚 2 , . . . , 𝑚 𝑚 } M={m 1 ​ ,m 2 ​ ,...,m m ​ }.  -set of production tasks 𝑇 = { 𝑡 1 , 𝑡 2 , . . . , 𝑡 𝑛 } T={t 1 ​ ,t 2 ​ ,...,t n ​ },  with processing times 𝑝 𝑗 p j ​ and deadlines 𝑑 𝑗 d j .  **Output:** Assignment of tasks to machines that minimizes total production time.   * + 1. **Pseudo Code:**   **Note: text files is also provided.**  Greedy (M, T):  Sort tasks in non-decreasing order of processing time  Initialize an empty assignment list for each machine  for each task t in sorted(T):  Find the machine with the earliest available completion time for task t  Assign task t to the selected machine  return assignment list for each machine  **3.1.2 Explanation:**  The jobs are sorted in a non-decreasing order by the greedy task assignment method, which is based on processing times. The job is then assigned to the machine with the earliest possible completion time after iteratively going through each task. This heuristic prioritises shorter jobs and makes optimal use of machinery in an effort to reduce the overall production time. But because it ignores superior long-term assignments in favour of short-term benefits and doesn't account for future job assignments, this technique might not always produce the best option.  **3.1.3 Tracing:**  By considering a set of 3 machines (M1, M2, M3) and 5 tasks (T1, T2, T3, T4, T5) with their processing times and deadlines: T1: (2, 5) T2: (4, 8) T3: (1, 4) T4: (3, 7) T5: (2, 6) After sorting tasks based on processing times: [T3, T1, T5, T4, T2] The algorithm assigns tasks as follows: M1: T3 (Completion time: 1), T1 (Completion time: 3), T5 (Completion time: 5) M2: T4 (Completion time: 3) M3: T2 (Completion time: 7) Total production time: 7.  **3.1.4 How the algorithm implement the paradigm:**  This algorithm implements the greedy paradigm by making locally optimal choices at each step without considering future consequences. It prioritizes tasks with shorter processing times, aiming to minimize the total production time. However, it may not always result in the globally optimal solution due to its myopic nature.   * 1. **Dynamic programming algorithm:**   **Input:** Set of machines 𝑀 = { 𝑚 1 , 𝑚 2 , . . . , 𝑚 𝑚 } M={m 1 ​ ,m 2 ​ ,...,m m ​ }, set of production tasks 𝑇 = { 𝑡 1 , 𝑡 2 , . . . , 𝑡 𝑛 } T={t 1 ​ ,t 2 ​ ,...,t n ​ } with processing times 𝑝 𝑗 p j ​ and deadlines 𝑑 𝑗 d j ​ .  **Output:** Assignment of tasks to machines that minimizes total production time.  **3.2.1 Pseudo Code:**  Dynamic(M, T):  Initialize a dynamic programming table DP with dimensions (number of tasks + 1) x (total time + 1) x (number of machines + 1)  Set initial values in DP table for base cases  for i from 1 to number of tasks:  for j from 0 to total time:  for k from 1 to number of machines:  Calculate DP[i][j][k] based on previous values in DP table  Backtrack to find optimal assignment of tasks to machines  return assignment list for each machine  **3.2.2 Explanation:**  The job scheduling issue is solved using the dynamic programming algorithm by decomposing it into smaller, overlapping subproblems. Each subproblem is only solved once, and the answers are stored in a dynamic programming table to prevent duplicate computations. Iteratively, taking into account job dependencies and machine capabilities, it fills the table with ideal values. The algorithm finds the best machine work assignment by going backwards from the final result.  **3.2.3 Tracing:**  Tracing dynamic programming requires a more complex example due to the nature of the algorithm. Let's consider a larger set of machines and tasks with specific processing times and deadlines. We'll walk through the dynamic programming table filling process and backtracking to find the optimal assignment of tasks to machines.  **3.2.4 How the algorithm implement the paradigm:**  The task scheduling problem is approached using the dynamic programming algorithm, which divides the problem into smaller subproblems and solves each subproblem only once. In order to calculate the best solution to the overall issue, it builds a dynamic programming table to hold the best answers to the subproblems. The algorithm determines the best way to assign tasks to machines by taking into account the interdependence between tasks and machine capabilities. By reconstructing the final solution from the DP table, the backtracking phase makes sure that the right tasks are allocated to the right machines. Through a thorough consideration of every potential combination of job assignments and machine utilization, this technique ensures the determination of the globally best solution.   * 1. **Evolutionary algorithm:**   **Input:**  Set of machines 𝑀 = { 𝑚 1 , 𝑚 2 , . . . , 𝑚 𝑚 } M={m 1 ​ ,m 2 ​ ,...,m m ​ }, set of production tasks 𝑇 = { 𝑡 1 , 𝑡 2 , . . . , 𝑡 𝑛 } T={t 1 ​ ,t 2 ​ ,...,t n ​ } with processing times 𝑝 𝑗 p j ​ and deadlines 𝑑 𝑗 d j ​ .  **Output:**  Assignment of tasks to machines that minimizes total production time.  **3.3.1 Pseudo Code:**  Evolutionary(M, T):  Initialize a population of candidate solutions  repeat until convergence criterion is met:  Apply genetic operators (mutation, crossover) to generate new candidate solutions  Evaluate the fitness of each candidate solution based on production time and adherence to deadlines  Select individuals for the next generation based on fitness  return the best solution found  **3.3.2 Explanation:**  The evolutionary task assignment algorithm iteratively evolves a population of candidate solutions using principles of natural selection. It applies genetic operators such as mutation and crossover to generate new solutions, evaluates their fitness based on criteria such as production time and adherence to deadlines, and selects the most promising individuals for the next generation. Over multiple iterations, the algorithm converges towards an optimal assignment of tasks to machines.  **3.3.3 Tracing:**  Tracing evolutionary algorithms involves simulating the evolution process over multiple generations. We'll initialize a population of candidate solutions, apply genetic operators to generate new solutions, evaluate their fitness, and select individuals for the next generation until the convergence criterion is met. We'll track the best solution found in each generation and observe how the algorithm evolves towards an optimal assignment of tasks to machines.  **3.3.4 How the algorithm implement the paradigm:**  The evolutionary task assignment algorithm mimics the process of natural selection to iteratively evolve a population of candidate solutions towards the optimal solution. It begins by initializing a population of random candidate solutions, each representing a possible assignment of tasks to machines. Through successive generations, the algorithm applies genetic operators such as mutation and crossover to generate new candidate solutions. These solutions are evaluated based on their fitness, which measures how well they minimize the total production time and adhere to task deadlines. The selection process favors solutions with higher fitness for reproduction, ensuring that promising individuals have a higher chance of contributing to the next generation. This iterative process continues until a convergence criterion is met, producing the best solution found. While evolutionary algorithms do not guarantee finding the globally optimal solution, they provide a robust and adaptive approach to solving complex optimization problems by exploring a diverse range of solutions and adapting them over multiple iterations.  **4.** **Analysis of the implemented algorithms:**   * 1. **Greedy Algorithm:**   **1 -Sorting Step:**  Suppose we have 𝑛 = 10 n=10 tasks to sort based on their processing times. Using an efficient sorting algorithm like merge sort or quicksort, sorting 𝑛 = 10 n=10 tasks typically takes  𝑂 ( 𝑛 log 𝑛 ) time.  For example, if we use merge sort, the time complexity would be approximately 𝑂 ( 10 × log 10 ) = 𝑂 ( 10 × 3.32 ) = O(33.2).  **2- Assignment Step:**  After sorting the tasks, let's assume we have 𝑚 = 3 machines available for assignment. In the worstcase scenario, each of the 𝑛 = 10 tasks needs to be compared with all 𝑚 = 3 machines to determine the best assignment. Therefore the assignment step has a time complexity of 𝑂 ( 𝑛 × 𝑚 ) = 𝑂 ( 10 × 3 ) = 𝑂 ( 30 ).  **3-Overall Time Complexity:**  By considering the dominant factor, which is the sorting step, the overall time complexity of the Greedy algorithm is 𝑂 ( 𝑛 log 𝑛 ).the total time complexity would be approximately 𝑂 ( 33.2 ).  **Analysis Type:**  This analysis utilizes both mathematical calculations and back substitution to determine the time complexity.We calculate the time complexity for each step based on the given problem data and then combine them to find the overall time complexity.   * 1. **Dynamic programming Algorithm:**   **1-Initialization Step:**  In this step, we initialize a dynamic programming table with dimensions ( 𝑛 + 1 ) × ( 𝑡 𝑜 𝑡 𝑎 𝑙 \_ 𝑡 𝑖 𝑚 𝑒 + 1 ) × ( 𝑚 + 1 ), where 𝑛 is the number of tasks, total time is the sum of processing times of all tasks, and 𝑚 is the number of machines. Initializing the table requires 𝑂 ( 𝑛 × 𝑡 𝑜 𝑡 𝑎 𝑙 \_ 𝑡 𝑖 𝑚 𝑒 × 𝑚 ) time.  **2-Filling DP Table:**  Here we fill in the dynamic programming table by considering all possible combinations of tasks, times, and machines. Each cell in the table requires constant time to compute, resulting in 𝑂 ( 𝑛 × 𝑡 𝑜 𝑡 𝑎 𝑙 \_ 𝑡 𝑖 𝑚 𝑒 × 𝑚 ) time complexity. Backtracking Step: After filling the DP table, we backtrack to find the optimal assignment of tasks to machines. Backtracking involves tracing back from the final cell to the initial cell, determining which tasks were assigned to which machines. Backtracking typically takes 𝑂 ( 𝑛 × 𝑚 ) time.  **3-Overall Time Complexity:**  Combining the time complexities of all steps, the overall time complexity of the Dynamic Programming algorithm is 𝑂 ( 𝑛 × 𝑡 𝑜 𝑡 𝑎 𝑙 \_ 𝑡 𝑖 𝑚 𝑒 × 𝑚 ). In terms of 𝑛 , can be simplified to 𝑂 ( 𝑛 × 𝑛 × 𝑚 ), as 𝑡 𝑜 𝑡 𝑎 𝑙 \_ 𝑡 𝑖 𝑚 𝑒 is dependent on the processing times of tasks.  This analysis primarily utilizes back substitution to determine the time complexity.  **Analysis Type:**  Back substetution involves breaking down the algorithm into its individual steps, determining the time complexity of each step, and then combining them to find the overall time complexity.  We don't use the Master Theorem here because the algorithm's time complexity is not defined by a recurrence relation that fits the theorem's framework. Instead, the time complexity is directly calculated based on the number of tasks, total time, and number of machines.   1. **Evoloutnary Algorithm:**   **1-Initialization Step:**  we initialize a population of candidate solutions. By assuming the population size is 𝑃. Initializing the population requires 𝑂 ( 𝑃 × 𝑛 )time to generate random solutions for each candidate.  **2-** **Iterative Evolution Step:**  It iteratively evolves the population over 𝐺 generations. Each generation involves the following steps:  a. Evaluating the fitness of each candidate solution. This step requires evaluating the fitness function for each of the 𝑃 solutions. By assuming the fitness function takes 𝑂 ( 𝑓 ( 𝑛 ) ) time.  b. Applying genetic operators such as mutation and crossover to generate new candidate solutions. This step requires modifying a portion of the population, which can be done in 𝑂 ( 𝑃 × 𝑛 )time.  c. Selecting individuals for the next generation based on fitness. This step involves selecting a subset of the population, which typically takes 𝑂 ( 𝑃 ) time. Therefore, each generation takes 𝑂 ( 𝑃 × ( 𝑓 ( 𝑛 ) + 𝑛 ) ) time.  **3-Overall Time Complexity:**  By combining the time complexities of all generations, the overall time complexity of the Evolutionary Algorithm is 𝑂 ( 𝐺 × 𝑃 × ( 𝑓 ( 𝑛 ) + 𝑛 ) ). In terms of 𝑛 , 𝑂 ( 𝐺 × 𝑃 × ( 𝑓 ( 𝑛 ) + 𝑛 ) ) complexity as a function of 𝑛 .  **Analysis Type:**  This analysis primarily utilizes back substitution to determine the time complexity. Back substitution involves breaking down the algorithm into its individual steps, determining the time complexity of each step, and then combining them to find the overall time complexity. We don't use the Master Theorem here because the algorithm's time complexity is not defined by a recurrence relation that fits the theorem’s framework. Instead, it's based on the number of generations, population size, and the computational cost of evaluating fitness and applying genetic operators.  **4.** **Comparison between the implemented algorithms:**   * + 1. **Greedy Algorithm:**   **Time efficiency:**  The Greedy algorithm typically has a time complexity of 𝑂 ( 𝑛 log 𝑛), where 𝑛 is the number of tasks. It is relatively fast compared to algorithms with higher time complexities.  **Memory efficiency:**  The memory consumption of the Greedy algorithm is relatively low since it doesn't require storing a large amount of data. It mainly operates on the input data and doesn't create additional data structures that consume significant memory.   * + 1. **Dynamic Programming Algorithm:**   **Time efficiency:**  The Dynamic Programming algorithm usually has a time complexity of 𝑂 ( 𝑛 2 × 𝑚 ),where 𝑛 is the number of tasks and 𝑚 is the number of machines. It can be efficient for moderate problem sizes but may become slower for larger instances due to its polynomial time complexity.  **Memory efficiency:**  The memory consumption of the Dynamic Programming algorithm depends on the size of the dynamic programming table, which can be significant for larger problem sizes. However, it generally requires less memory than brute force for the same problem size.   * + 1. **Evolutionary Algorithm:**   **Time efficiency:**  The time efficiency of Evolutionary Algorithms can vary widely depending on factors like the number of generations, population size, and computational cost of fitness evaluation. It can be efficient for finding near-optimal solutions, especially for complex problems, but may require many iterations to converge.  **Memory efficiency:**  Evolutionary Algorithms typically require storing a population of candidate solutions, which can consume significant memory resources, especially for large populations and complex problem instances.   * 1. **Sorting the algorithms:**   In terms of time efficiency, the Greedy algorithm is the fastest, followed by Dynamic Programming and then Evolutionary Algorithms.  For memory efficiency, Greedy is usually the most memory-efficient, followed by Dynamic Programming and then Evolutionary Algorithms.   * 1. **Recommendations:**   Based on the efficiency analysis, the choice of the best algorithm depends on the specific requirements of the problem, including the size of the problem data, time constraints, and available memory resources.   * For small problems Greedy algorithm may be the best choice due to its fast execution time. * For larger problem sizes where both time and memory efficiency are important, the Dynamic Programming algorithm is recommended as it strikes a balance between time and memory consumption. * Evolutionary Algorithms are suitable for complex optimization problems where finding near-optimal solutions is more important than execution time, and memory resources are not a constraint.   So, in our case the Greedy algorithm for most scenarios due to its simplicity and relatively fast execution time is the most suitable. However, if we supposed a larger problem size with a significant number of tasks and machines, where scalability and efficiency are paramount, the Dynamic Programming algorithm should be considered. | | | | |  |
|  |
|  |
|  |  |