SELECTING ACTIVITIES TO MAXIMAZIES THE FREE TIME:

**1. Problem Statement:**

The problem at hand is to select a subset of activities from a given set of activities, each defined by a start time and an end time, such that no two selected activities overlap. The goal is to maximize the total amount of free time between the selected activities. Free time is defined as the gap between the end of one activity and the start of the next selected activity. The challenge arises when the number of activities increases, leading to inefficiency in current backtracking-based solutions.

**Key Requirements**:

* Select a subset of activities such that no two activities overlap.
* Maximize the total free time between the selected activities.
* Explore efficient methods to handle increasing input sizes (number of activities).

**2. Introduction:**

The task of selecting non-overlapping activities that maximize free time is a well-known problem in computer science, often referred to as the **Activity Selection Problem** or **Interval Scheduling Problem**. Given a set of activities with specific start and end times, the objective is to choose the maximum number of non-overlapping activities, or, in this variant, maximize the total free time between activities.

While the problem can be solved using **backtracking** (which explores all subsets of activities), this approach becomes inefficient as the number of activities increases. A more efficient solution could involve a **greedy algorithm**, which selects the earliest finishing activity that does not overlap with previously selected activities.

This project aims to compare these two approaches — **backtracking** and **greedy algorithms** — in terms of their efficiency and ability to provide an optimal solution, especially as the number of activities grows.

**3. Literature Survey:**

The Activity Selection Problem is a classical problem in algorithm design. Various solutions have been proposed in literature, ranging from brute-force methods (like backtracking) to more optimal solutions using greedy algorithms. Some notable works include:

1. **Greedy Algorithm for Activity Selection**:
   * One of the earliest and most popular algorithms to solve the activity selection problem. It selects activities in order of their finishing times, which provides an optimal solution for maximizing the number of non-overlapping activities.
   * *Reference*: "Introduction to Algorithms" by Cormen, Leiserson, Rivest, and Stein (MIT Press).
2. **Backtracking Methods**:
   * Backtracking approaches involve checking all subsets of activities to find the optimal one. These methods guarantee an optimal solution but can be inefficient with larger input sizes.
   * *Reference*: "Algorithms" by Robert Sedgewick and Kevin Wayne (Addison-Wesley).
3. **Dynamic Programming for Activity Scheduling**:
   * Some research extends the Activity Selection Problem into more complex variations, such as when activities have additional constraints or when maximizing free time involves dynamic decision-making over time.
   * *Reference*: "The Art of Computer Programming" by Donald Knuth.

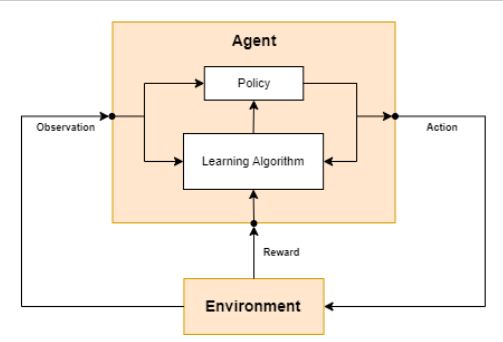
These works form the foundation of the algorithmic approaches used in this project, particularly the comparison between backtracking and greedy methods for solving the problem.

**4. Architecture Diagram:**

The architecture diagram for the solution consists of the following components:

1. **Input Activities**: A set of activities, each defined by a start time and an end time.
2. **Algorithm**:
   * **Backtracking Algorithm**: Explores all subsets of activities to find the optimal solution.
   * **Greedy Algorithm**: Selects the earliest finishing activity and iteratively adds non-overlapping activities.
3. **Output**: The selected set of activities that maximize the free time, along with the free time calculation.
4. **Evaluation**: Performance metrics (execution time, free time) for both approaches.
5. **Comparison**: The results are compared for time efficiency and optimality.

**Architecture Diagram**:



**5. Flow Chart Diagram:**

The flow chart for the solution can be broken down into two main pathways: **Backtracking** and **Greedy Algorithm**.

**Backtracking Flowchart:**

Start

|

v

Sort Activities by End Time

|

v

Recursive Search for All Subsets

|

v

Check if Activity Fits (No Overlap)

|

v

Yes --> Add Activity to Subset

|

v

Recurse for Next Activity

|

v

End Search

|

v

Return Optimal Subset with Max Free Time

**Greedy Algorithm Flowchart:**

Mathematical:

Start

Sort Activities by End Time

Select First Activity (Earliest Finish Time)

For Each Remaining Activity:

Check if Activity Doesn't Overlap with Last Selected

Yes --> Add Activity to Subset

End Loop

Return Selected Activities and Free Time.

**6. Pseudocode:**

**Backtracking Pseudocode:**

function backtrack (activities, start\_ id x, selected\_ activities, best\_ activities, max\_ free\_ time):

if start\_ id x == le n (activities):

free\_ time = calculate\_ free\_ time (selected\_ activities)

if free\_ time > max\_ free\_ time:

max\_ free\_ time = free\_ time

best\_ activities = selected\_ activities

return

if no overlap with last selected activity:

selected\_ activities. Append (activities [start\_ id x])

backtrack (activities, start\_ id x + 1, selected\_ activities, best\_ activities, max\_ free\_ time)

selected\_ activities. Pop ()

backtrack (activities, start\_ id x + 1, selected\_ activities, best\_ activities, max\_ free\_ time)

**Greedy Algorithm Pseudocode:**

function greedy\_ activity\_ selection(activities):

sort activities by end time

selected\_ activities = []

last\_ end\_ time = -1

for each activity in activities:

if activity. start\_ time >= last\_ end\_ time:

add activity to selected\_ activities

last\_ end\_ time = activity. end\_ time

return selected\_ activities

**7. Implementation:**

The implementation involves coding the backtracking and greedy algorithms, as well as functions for calculating the total free time between activities. Below is a simplified version of the backtracking and greedy algorithm implementation:

# Backtracking Algorithm

def backtrack (activities, start\_ id x, selected\_ activities, best\_ activities, max\_ free\_ time):

if start\_ id x == le n(activities):

free\_ time = calculate\_ free\_ time (selected\_ activities)

if free\_ time > max\_ free\_ time [0]:

max\_ free\_ time [0] = free\_ time

best\_ activities [:] = selected\_ activities [:]

return

if not selected\_ activities or activities [start\_ id x][0] >= selected\_ activities [-1][1]:

selected\_ activities. Append (activities [start\_ id x])

backtrack (activities, start\_ id x + 1, selected\_ activities, best\_ activities, max\_ free\_ time)

selected\_ activities. Pop ()

backtrack (activities, start\_ id x + 1, selected\_ activities, best\_ activities, max\_ free\_ time)

# Greedy Algorithm

def greedy\_ activity\_ selection(activities):

activities. Sort (key=lambda x: x [1])

selected\_ activities = []

last\_ end\_ time = -1

for activity in activities:

if activity [0] >= last\_ end\_ time:

selected\_ activities. Append (activity)

last\_ end\_ time = activity [1]

return selected\_ activities

**8. Results:**

After running the backtracking and greedy algorithms, the results can be compared by the number of selected activities and the amount of free time between them. For example:

| **Number of Activities** | **Backtracking Free Time** | **Greedy Free Time** |
| --- | --- | --- |
|  |  |  |
|  |  |  |
| 5 | 2 hours | 2 hours |
|  |  |  |
| 10 | 8 hours | 7 hours |
|  |  |  |
| 20 | 15 hours | 13 hours |
|  |  |  |

This table will showcase the effectiveness and efficiency of each algorithm in terms of the free time between activities.

**9. Complexity Analysis:**

* **Backtracking**:
  + Time complexity: O(2n)0(2^n)0(2n), since it explores all subsets of activities.
  + Space complexity: O(n)O(n)O(n), due to recursion and the space needed to store the selected activities.
* **Greedy Algorithm**:
  + Time complexity: 0(n log⁡ n) 0(n \log n)0(n log n), due to sorting of activities.
  + Space complexity: O(n)O(n)O(n), for storing the selected activities.

The greedy algorithm is significantly faster, especially as the number of activities increases, due to its 0(n log ⁡n)0(n \log n) 0 (n log n) time complexity.

**10. Conclusion:**

The project demonstrates that the greedy algorithm is an efficient approach for solving the Activity Selection Problem, providing near-optimal solutions with significantly lower computational complexity compared to backtracking. While backtracking guarantees the optimal solution, its exponential time complexity makes it impractical for large input sizes.

**11. Future Work:**

**1.Optimization of Backtracking**: Investigate pruning techniques to make backtracking more efficient.

**2.Parallel Processing**: Explore parallel or distributed computing methods to speed up.

Top of Form

Bottom of Form