Final Project Report: Draft
Pathly Planner

Team: Raul Pineda

Novermber 17, 2024

1 Project Scope

This project aims to design and develop a program that intelligently schedules your activities, classes, deadlines, and the time needed to work on each task on a calendar. The objective is to optimize for maximum productivity, efficiency, and simplicity.

The project's main objectives are:

- To schedule activities, classes, deadlines, and the time needed to work on each task on a calendar.
- To dynamically adapt a schedule based on deadline shifts, missed activities, or a shift in priorities.
- To leave users with satisfied program performance and seamless user input.

The expected outcomes include a **Smart Planner**, proper documentation, and a final project report.

2 Project Plan

2.1 Timeline

The overall timeline for the project is divided into phases:

- **Week 1 (October 14 - October 20)**: Define project scope, think about Frontend (Calendar UI) and Backend (Priority optimization) integration, create a comprehensive list of all the features anyone could ask of this project, and make a project timeline including only the most essential features.
- **Week 2 (October 21 - October 27):** Begin development, set up the project repository. Implement an object structure that is highly modular, with each class handling a specific responsibility.

- **Week 3 (October 28 - November 3)**: Continue coding, work on backend optimization (3 main parts: 1. Add events with no conflicts, 2. Optimize based on priority, 3. Optimize based on deadlines), and start writing technical documentation. Also, draw a wireframe of what the UI will look like—goal is functional simplicity.
- **Week 4 (November 4 - November 10)**: Continue backend coding, look for edge cases and try to finalize any major changes. If not already started, begin the front-end wireframe with backend integration in mind. Properly test exporting calendar (.ICS file) and integration with third-party calendar apps like Apple Calendar, Outlook, and Google Calendar. Begin PowerPoint presentation.
- **Week 5 (November 11 - November 17):** Continue the system (front and backend), conduct testing, and continue working on the report.
- **Week 6 (November 18 - November 24):** Continue testing, revise, and perfect the code. Revise and finalize the technical report and PowerPoint presentation. Begin presentation practice.
- **Week 7 (November 25 - December 4):** Finalize system and complete fine tuning of foundational features. Continue practicing presentations. Final presentation, report submission, and project closure.

2.2 Milestones

Key milestones include:

- Project Scope, List of Features, and Project Timeline (October 20).
- GitHub Repository Setup and Initial Development (October 25).
- Backend Core Feature Completion (November 5).
- UI/Frontend Integration and Testing (November 12).
- Final System Testing, Frontend Completion, and Report Draft (November 18).
- Final Presentation and Report Submission (November 28).

2.3 Team Roles

- **Frontend Developer (Raul):**
  - Responsible for designing and implementing the user interface (UI) of the calendar optimizer app using JSX with React.
  - Focus on building an intuitive and functional interface for creating, viewing, and managing events on the calendar.

  - Ensure the integration of frontend components with the backend and maintain a modular structure for flexibility and scalability.

- **Backend Developer (Raul):**
  - Handle the server-side logic and database management using Python, potentially with Flask.
  - Implement the priority optimization engine to add events without conflicts, prioritize tasks, and manage deadlines.
  - Ensure smooth export of calendar files in .ICS format and compatibility with third-party calendars like Apple Calendar, Google Calendar, and Outlook.

- **Documentation and Report Writer (Raul):**
  - Manage project documentation and version control using GitHub.
  - Write the technical report in Overleaf, detailing the system's architecture, frontend-backend integration, and optimization strategies.

- **Testing and Quality Assurance (Raul):**
  - Conduct tests on both the frontend and backend, ensuring seamless integration and functionality.
  - Test calendar export features and third-party app integrations, along with thorough edge case testing.

3 Team Discussion Summary (October 20)

3.1 Skills Assessment

- **Front End:**
  - Proficiency in React and JSX.
  - UI Design.
  - Backend skills in Python for the server-side logic.
  - Familiarity with State Management for the use of the React framework.

- **Back End:**
  - Proficiency in Python.
  - Reliable data management—moderate.
  - Integration with ICS export.

- **Documentation and Project Management:**
  - Familiarity with GitHub for version control.

  - Use of Overleaf for writing the final technical report.
  - Technical writing capable.

- **Testing:**
  - Unit testing.
  - Integration testing.
  - Edge case handling.
  - User Acceptance Testing—handles real-world scenarios.

- **Calendar Optimization Algorithms:**
  - Backtracking.
  - Greedy algorithms.
  - Dynamic Programming.
  - Knowledge of Data structures: Dictionaries, queues, priority queues, etc.
  - Time complexity analysis.

- **Communication and Soft Skills:**
  - Problem Solving.
  - Time Management.
  - Presentation and Communication.

3.2 Tools & Technologies

- Programming Languages: Python, JavaScript, JSX, etc.
- Database: SQLite or MySQL.
- Collaboration Tools: GitHub for version control, Overleaf for report writing.

3.3 Team Responsibilities

I will complete all areas because I am the sole developer, but roles may adapt as the project progresses or realizations of ambitious features currently not thought of as ambitious:

- Frontend Development.
- Backend Development.
- Technical Documentation.
- System Testing.

4 Skills & Tools Assessment (October 20)

4.1 Skills Gaps & Resource Plan

I have 80-90% of the skills in the Skills Assessment. However, as the sole developer, I fully acknowledge that I may need to scale back later on when I have a clearer perspective, better knowledge of the system, and more experience. To fill the knowledge gap, I will maintain frequent contact with the professor and reach out to peers for advice in areas that are new to me, while leveraging online resources. I currently do not see any major barriers aside from time commitment and a busy schedule to filling the knowledge gap.

4.2 Tools

We will use the following tools:

- Python (Visual Studio Code) for backend development.
- JSX, React, MUI Library for the user interface.
- SQLite for database management—maybe.
- GitHub for version control and collaboration.
- Overleaf for the technical report.
- Other tools may be added as needed.

5 Initial Setup Evidence (October 24)

5.1 Project Repository

The project repository has been created on GitHub and is accessible by all team members. It can be found at https://github.com/Raul-Pineda/PathlyPlanner.githttps://github.com/Raul-Pineda/PathlyPlanner.git.

5.2 Setup Proof

The development environment has been successfully set up. Screenshots of the setup process are provided in the Appendix.

6 Progress Review (October 25)

6.1 Progress Update

6.2 Issues Encountered

One ongoing challenge is whether the system architecture is the most optimal choice. A flexible and scalable system architecture was chosen. While no changes are anticipated, this will be reassessed throughout the semester.

7 Revised Project Plan (October 27)

7.1 Updated Plan

After reviewing our progress, we updated the timeline to allow additional time to address front-end integration challenges. The milestone for frontend completion has now been moved to October 25.

8 Algorithm Design (October 27)

8.1 System Architecture

My app's system architecture can be mostly simply communicated with a diagram. In the chart below, you can see my intended class and file breakdown. At the top, you have the file entry with main.py and config.py immediately below by the class tree broken down into five central classes: optimization, tests, calendar, utility, and models. The optimization section of my code handles a few key components. One is priority-based optimization, where events and deadlines are scheduled based on a priority level (1-10); time-constraint, where conflict detection happens; and Priority Manager, where user data is preprocessed. You also have the Models that contain the user, task, scheduled_task, deadline, and event subclasses. These are the different kinds of objects that can be scheduled in the calendar. Next is the calendar class, where functionality like adding and updating tasks and events happens. Time constraint is also used here to check for conflicts; lastly, besides calendar, calendar integration also exists, which handles any necessary formatting and exporting of optimized calendar-related data. Finally, the JSX UI is not listed here because it does not pertain to a specific class and primarily handles all user data. The UI is in charge of seamlessly gathering data from the user and passing it along to the back end (python).

My app's system architecture is illustrated with a class and file breakdown, as shown below:

- **Main Files:**
  - `main.py`: Entry file
  - `config.py`: Configuration file
- **Classes:**
  - **Optimization:**
    - Priority-based optimization: Schedules events based on priority (1-10)
    - Time-constraint: Detects conflicts

    * Priority Manager: Preprocesses user data
  - **Models:**
    * User, Task, Scheduled_Task, Deadline, Event
  - **Calendar:**
    * Adding and updating tasks/events
    * Conflict checking (time constraints)
    * Calendar integration: Formatting/exporting optimized calendar data
  - **Utility:**
    * Handles support functions across modules
  - **UI (JSX):**
    * Gathers user data, interfaces with the backend

8.2 Algorithm choices for each class

Due to the scale in several classes and the possibility that a complete 100% ideal project is not feasible due to a lack of partners, I have primarily focused on deciding my algorithm choices for crucial functionality like optimizing user calendar data. The first place user data is passed to is called the priority manager. The priority manager takes in user inputs, such as event details and deadlines, and organizes them by priority. To achieve this, I chose QuickSort, an efficient sorting algorithm, to arrange tasks in descending order of priority. This makes it easy to always have the highest priority tasks at the top and ready for the next steps.

In addition to simple priority sorting, the manager includes dependency-aware sorting. If a task depends on other tasks, those dependencies are automatically "boosted" in priority, ensuring they're completed first. This approach guarantees that even lower-priority tasks are completed in the correct sequence if needed for higher-priority ones. By handling these dependencies carefully, my system maintains an up-to-date order of tasks and prevents conflicts in scheduling.

The sorting process is broken down into two parts:

1. **QuickSort Process:** I first partition the list of tasks around a pivot, separating them into those with higher and lower priorities. This sorting recursively arranges tasks in descending priority.

2. **Dependency Sorting**: For tasks with dependencies, I rank them by priority and dependency count, ensuring high-priority nodes without dependencies are sorted first, followed by tasks with the fewest dependencies, increasing as necessary. This dependency-aware sorting keeps the timeline efficient, factoring in priority and task dependencies.

The following are two examples of how each sorting algorithm works, thus validating its effectiveness:

1. QuickSort Process

This is a step-by-step example of how QuickSort is used to sort tasks by priority:

1. **Choose a Pivot**: Select a pivot point (this could be the middle element, first element, last element, or a random one). In this example, let's start with the list [9, 3, 7, 5, 6, 4, 8, 2] and choose 2 as our pivot.

2. **Partition the Array**:

• Rearrange all elements so that elements less than the pivot are on the left side, and elements greater than the pivot are on the right.

• Keep an index i to track the elements that are less than the pivot. If an element is less than the pivot, it's swapped with the element at i, and i is incremented.

• After partitioning around 2, the list becomes [2, 3, 7, 5, 6, 4, 8, 9], with 2 in its correct position.

3. **Recursive Sorting**:

• Apply QuickSort recursively to the left and right subarrays.

• In this example, the left partition has no elements less than 2, so it's done.

• The right partition [3, 7, 5, 6, 4, 8, 9] goes through the same steps:

• Select a new pivot (e.g., 3), partition around it, and continue until each subarray has only one element.

By the end of this process, the tasks are sorted in descending priority order.

2. Dependency-Aware Sorting

This process considers both task priority and dependencies between tasks. Here's how the example unfolds:

1. **Define Task Priorities and Dependencies**:

• Suppose we have the following tasks with their priorities (p) and dependencies (dep):

• **Task A**: p = 1, dep = { (no dependencies)

• **Task B**: p = 2, dep = {Task D}

• **Task C**: p = 2, dep = { (no dependencies)

• **Task D**: p = 2, dep = {Task B, Task C}

2. **Sorting Criteria**:

• **High-priority nodes without dependencies** come first.

• **High-priority nodes with the fewest dependencies** are sorted next.

• **High-priority nodes with increasing numbers of dependencies** are next.

• **Lower-priority nodes** come last.

3. **Sorting Steps**:

• **Task A** is high priority with no dependencies, so it's first.

• **Task B** has high priority with only one dependency (Task D), so it's sorted next.

• **Task C** has high priority with no dependencies, but since it shares the same priority level as Task B, it follows after.

• **Task D** has high priority but the most dependencies, so it's sorted last among the high-priority tasks.

4. **Result**:

• Following these steps, the sorted order is [A, D, B, C].

These two sorting algorithms are efficient at O(n log(n)) time complexity. After the data is sorted for dependencies and priority, it is returned as a queue.

Another psuedoCode algorithmic design I have developed is for conflict detection. My process for detecting and logging conflicts can be broken down into 4 key parts.

1. **Sort by End Times**:

• The first step is to sort all events by their end times. Sorting ensures that we can quickly check for overlaps in a single pass through the list, since each event will have either a start time that comes after the previous event's end time (no conflict) or overlaps with it (conflict). This step simplifies the process of detecting conflicts.

2. **Compare End Time and Start Time**:

• Starting with the first event, compare the end time of the current event (let's call it prev_end) with the start time of the next event (curr_start).

• If prev_end is greater than curr_start, a conflict is detected because the current event overlaps with the next one.

• When a conflict is found, record the conflicting pair (the two objects or events causing the conflict) in a conflict list. This conflict list will keep track of all overlapping events.

3. **Increment Through the List**:

• Continue this process by incrementing through the list, always checking the end time of the previous event with the start time of the next.

• If there's no conflict (i.e., prev_end is not greater than curr_start), simply move to the next event without adding anything to the conflict list.

4. **Return the Conflict List**:

• After iterating through the list, return the conflict list, which contains overlapping pairs of events. If the list is empty, there are no conflicts.

The second central piece of the time constraint class is buffer time management. For simplicity and unnecessary complication, I have decided to implement buffer time management as a part that handles the spacing of events by comparing events overlaps with a 15-minute shift before and after each task. This function does not implement much logic but rather leverages other existing classes neatly. The exact implementation has not been determined as it depends on the precise implementation of different classes.

The last vital component of the optimization class handles the actual priority scheduling. Still, due to its complexity, I have yet to map out its algorithmic design fully. This class will be one of the most comprehensive integrations of all other functionality. First, data is input into the system, then conflict detection is run on the inputs, and then the data will be passed to the priority manager through all of its sorting, etc. Finally, entering a priority strategy, I would implement Dynamic Programming with memoization for optimization task selection when multiple constraints are present. This approach builds up solutions incrementally by calculating the optimal schedule for each task and storing these results for reuse. Each task can be evaluated based on its priority, duration, and dependencies to find the optimal set. I am also exploring the possibility of using recursive backtracking with pruning to explore multiple task arrangements and find the one that maximizes priority while respecting constraints. The algorithm recursively tries different task orders, backtracking when it encounters conflicts or constraints (e.g., dependencies or overlapping time requirements). Priority-based pruning can eliminate paths with low-priority tasks if higher-priority alternatives exist. The final implementation of the priority strategy class will likely be a combination of the algorithms previously mentioned. If this was a lot of text, I understand. I have included a diagram of my class hierarchy above to clarify matters.

Summary:

Optimization:

• **Priority Manager**:

• Sorting Algorithms:

• **QuickSort**:

• Purpose: Sort tasks by descending priority

• Steps:

1. Choose a pivot
2. Partition list around pivot
3. Recursively sort subarrays

• **Dependency-Aware Sorting**:

• Purpose: Sorts tasks by priority while managing dependencies

• Steps:

1. Sort high-priority tasks without dependencies first
2. Place high-priority tasks with dependencies next, ordered by dependency count

• **Conflict Detection Algorithm**:

• Process:

1. Sort events by end time
2. Compare each event's end time with the next event's start time
3. Log conflicts in a conflict list

• **Buffer Time Management**:

Purpose: Adds a 15-minute buffer before/after each task Dependency: Other existing classes for overlapping management

• **Priority Scheduling**:

• **Dynamic Programming with Memoization**:

• Purpose: Select optimal task scheduling based on constraints

• Role: Stores partial solutions for reuse

• **Recursive Backtracking with Pruning**:

• Purpose: Find optimal task arrangement by exploring possible paths

• Role: Eliminates paths with lower-priority tasks if higher-priority options exist

Results (November 17th):

Although extensive testing still remains much of the functionality discussed above, has been implemented. The two most significant components that require the most fine-tuning are the scheduling strategy class and the bridge between the front end and back end. Current progress shows that the foundation for each side of this metaphorical bridge linking the front and back end is complete but the kind of bridge and how it will be constructed has yet to be implemented successfully.

Discussion:

References:

Appendix: