Critical Path Method

Team members: Yuqi Hu, Hao Li, Hejia Zhang (last name alphabetic order)

1. Introduction

The critical path method (CPM) is an essential project modeling technique employed for analyzing, planning, and scheduling complex projects. This concept was first developed by James E. Kelley and Morgan R. Walker back in the late 1950s. They were attempting to find ways to reduce the costs relating to plant shutdowns and restarts that were being caused by inefficient scheduling.

It is commonly used in conjunction with the program evaluation in order to ensure that the right tasks were performed at the right times and avoid any additional unnecessary costs. A critical path is determined by identifying the longest stretch of dependent activities and measuring the time required to complete them from start to finish.

Personally, we have all worked on complex projects/tasks. These tasks can be broken down into smaller tasks and some tasks may depend on other tasks. Each task takes some time and we only have limited time to complete the whole project. That is why we think it's important to be able to identify the critical path so that we can better organize our time and efforts. Therefore, in this project, we are going to identify and evaluate several solutions to the critical path problem.

Hao Li - The first time I got curious about this problem was when I was learning greedy algorithms. We worked on the *activity selection problem* and we were required to choose a maximum number of non-overlapping intervals. Although an optimal solution could be achieved by choosing the activities earliest finishing time, I wondered if it's possible for each activity to be arranged in a different time slot and still minimize the total completion time, since in real life activities are usually flexibly scheduled. Therefore, I found out the CPM problem, which perfectly provides a solution to the project scheduling in practical project management.

Hejia Zhang - I find this topic fascinating because this concept can be implemented in all real-world applications and it can help people keep track of their project in a high-level perspective and ensure the project process is at an optimal and efficient pace. Understanding and applying CPM will allow me to optimize resource allocation effectively, ensuring that projects are not only completed on time but also within budget. It is a useful tool which can be beneficial for my future career in software development.

Yuqi Hu - CPM is a topic which is so closely related to us. I first did some research on the background of this topic and found that it can be used in various fields and even beneficial to us individuals. This is also why I am interested in diving into this topic. We have learned a lot about greedy algorithms this semester and I feel like this would be the best topic to put what we learn into practice.

2. Algorithms and technical analysis

The Critical Path Method is a project management technique used to identify the sequence of tasks that must be completed on time to ensure the overall project is completed within the specified timeframe. By determining the critical path, project managers can focus their efforts on the most crucial activities, allocate resources efficiently, and mitigate potential bottlenecks that could delay project completion.

This project aims to delve deeper into the concepts, applications, and benefits of the Critical Path Method in project management. We will explore how CPM works and compare it with other types of algorithms and see how CPM facilitates better scheduling, resource allocation and overall project optimization.

2.1 Project management using CPM

Definition:

Assume we have a set of tasks waiting to be scheduled, each task has its specific task duration(t) and dependencies (the task must start after other tasks are completed). We can start establishing the algorithms by assigning attributes to each task: ES, EF, LS and LF:

Earliest start time (ES): The earliest possible date you can start an activity considering the dependencies.

Earliest finish time (EF): The earliest possible date you can complete an activity considering its ES and duration.

Latest start time (LS): The last possible date you can start an activity before causing a significant project delay.

Latest finish time (LF): The latest possible date you can complete a task based on its LF and duration.

Task duration (t): The total amount of time it takes to complete an activity.

While designing an algorithm for the Critical Path Method, it also involves creating a valid diagram to illustrate how we decide on the time arrangement for each task.

We start the construction of a project network diagram by first considering each task as a node, and the dependencies of nodes as edges. For example, if task B can't start before task A is completed, we should add a directed edge from A to B to illustrate the dependency.

After establishing the diagram, the algorithm will start two passes: the forward pass and the backward pass.

The forward pass is determined by using the earliest start for each task known as ES and the earliest finishing time EF. The ES of a task equals the maximum EF of all the dependency tasks before it. If the task doesn't have any dependency task, ES will start at time 0. The earliest finishing time EF is calculated by EF = ES + t, which is the earliest start time plus task duration.

After we iterate through the forward pass, the backward pass starts, which assigns the last task's latest finishing time LF as the finishing time for the whole project. The latest start time LS, which is calculated by LF – t. We iterate through the digraph in a reversed order, and assign the minimum latest start time LS of all the next tasks as the LF of their former task (dependencies).

Finally, we determined all of ES, EF, LS, LF for each task. Activities that are not on the critical path will have slack or float, which represents the amount of time an activity can be delayed without affecting the project's overall completion time. The slack for an activity is calculated as LS - ES or LF - EF. The critical path for a project will be tasks that have slack = 0.

Pseudocode:

```
Define a Task structure with fields for:
   - Task ID (task_id)
   - Duration (duration)
   - Earliest Start Time (earliest_start)
   - Earliest Finish Time (earliest_finish)
   - Latest Start Time (latest_start)
   - Latest Finish Time (latest_finish)
   - Slack Time (slack)

Define a Graph structure to represent the project, with nodes and edges:
   - Nodes represent tasks (Task objects)
   - Edges represent dependencies between tasks

Initialize an empty graph G to represent the project.

Create a function ConstructGraph(tasks):
  a. Create an empty graph G.
  b. For each task in tasks:
     i. Add a node to G representing the task with task_id as node ID and task object as node
attributes.
     ii. For each dependency in task.get('dependencies', []):
         - Add a directed edge from dependency task to the current task in G.

Create a function CalculateEarliestTimes(tasks, G):
  a. For each task in tasks:
     i. Add a node to G representing the task.
     ii. Set the node's duration based on task.duration.
     iii. If task has no dependencies:
         - Set task's earliest_start to 0 and earliest_finish to duration.
     iv. Else:
         - Set task's earliest_start to maximum earliest_finish of dependencies.
         - Set task's earliest_finish to earliest_start + duration.

Create a function CalculateLatestTimes(tasks, G, project_duration):
  a. Set project_duration to the maximum earliest_finish of all tasks.
  b. For each task in tasks in reverse topological order of G:
     i. If task has no successors:
         - Set task's latest_finish to project_duration.
     ii. Else:
         - Set task's latest_finish to minimum latest_start of successors.
         - Set task's latest_start to latest_finish - duration.

Create a function CalculateSlack(tasks, G):
  a. For each task in tasks:
     i. Calculate task's slack as latest_start - earliest_start.

Create a function IdentifyCriticalPath(tasks):
  a. Initialize an empty list critical_path.
  b. For each task in tasks:
     i. If task's slack is 0, add task to critical_path.

Example:
  - Define tasks as a list of Task objects with task details.
  - Call CalculateEarliestTimes(tasks, G) to calculate earliest times.
  - Call CalculateLatestTimes(tasks, G, project_duration) to calculate latest times.
  - Call CalculateSlack(tasks, G) to calculate slack times.
```

- Call IdentifyCriticalPath(tasks) to identify the critical path.

Output the critical_path list to display the critical path in the project.

Explanation for pseudocode:

1. Define a Task structure with fields for name, duration, earliest start, earliest finish, latest start, latest finish, and slack.
2. Define a Graph structure to represent the project, with nodes and edges.Nodes represent individual tasks, while edges denote dependencies where one task must be completed before another can start.
3. Initialize an empty graph to represent the project.
4. For each task:
    1. Create a node in the graph to represent the task.
    2. Add the task node to the graph.
    3. Define dependencies between tasks as directed edges in the graph.
5. For each task in the graph:
    1. Calculate earliest start and finish times based on dependencies and task durations.
    2. Store calculated values in task objects.
6. For each task in the graph (in reverse order):
    1. Calculate latest start and finish times based on dependencies and task durations.
    2. Store calculated values in task objects.
7. For each task in the graph:
    1. Calculate slack by subtracting the earliest start from the latest start.
    2. Store calculated slack value in task objects.
8. Identify critical paths by finding tasks with zero slack. Output the critical path.

Time and Space Complexity:

1. ConstructGraph(tasks):
   - Time Complexity: O(V + E), where V is the number of tasks and E is the number of dependencies.
   - Space Complexity: O(V + E) for storing the graph.

2. CalculateEarliestTimes(G):
   - Time Complexity: O(V + E), for topological sorting and calculating earliest times.
   - Space Complexity: O(V) for storing earliest start and finish times.

3. CalculateLatestTimes(G, project_duration):

- Time Complexity: O(V + E)for reverse topological sorting and calculating latest times.
- Space Complexity: O(V) for storing the latest start and finish times.

4. CalculateSlack(G):
   - Time Complexity: O(V)for calculating slack times.
   - Space Complexity: O(V) for storing slack values.

5. IdentifyCriticalPath(G):
   - Time Complexity: O(V) for identifying critical path tasks.
   - Space Complexity: O(V) for storing the critical path list.

Overall Complexity:
   - Time Complexity: O(V + E) for constructing the graph and performing critical path calculations.
   - Space Complexity: O(V + E) for storing the graph and intermediate calculations.

In summary, the time complexity of the critical path method is primarily determined by the number of tasks V and dependencies E in the project. The space complexity also depends on the size of the input data and the number of tasks and dependencies. Overall, the critical path method has a linear time complexity and linear space complexity in terms of the number of tasks and dependencies.

Python library used in source code:

Networkx
The goal of using *networkx* is to leverage its graph manipulation and algorithmic capabilities to implement the Critical Path Method (CPM) efficiently. It provides convenient functions for graph creation, manipulation, and analysis, making it well-suited for tasks involving project scheduling and critical path analysis.

The code utilizes the `networkx` library in Python to handle graph-related operations efficiently. Here's the breakdown of the library functions used, along with explanations and references:

Function used
nx.DiGraph(): creates a directed graph object in the *networkx* library.

G.add_node(task['id'], duration=task['duration']): adds a node (task) to the graph G with attributes like duration.

G.add_edge(dependency, task['id']): adds a directed edge from a dependency task to the current task node in the graph.

nx.topological_sort(G): performs a topological sort on the directed graph G.

G.predecessors(node) and G.successors(node): return predecessors (tasks that must be completed before the current task) and successors (tasks dependent on the current task) of a node, respectively.

The goal of using *networkx* is to leverage its graph manipulation and algorithmic capabilities to implement the Critical Path Method (CPM) efficiently. It provides convenient functions for graph creation, manipulation, and analysis, making it well-suited for tasks involving project scheduling and critical path analysis.

2.2 Other approaches (include some reference to the coursework)

2.2.1 Brute Force (Backtracking)

Summary:
This method works by first representing the tasks and their dependencies as a graph. Then, it systematically explores all possible paths starting from each task to find the longest path, which represents the critical path. It does this recursively, considering each task and its dependencies, until it reaches the end of the project. By comparing the durations of these paths, it identifies the one with the maximum duration as the critical path. While this approach ensures accuracy in finding the critical path, it may be computationally expensive for larger projects due to its exponential time complexity, as it considers every possible combination of tasks. Please see Appendix 5.1 for its implementation (pseudocode) and explanation.

Time Complexity:
For each starting node, the algorithm explores all possible paths to reach the finish node. Since there are N nodes in the graph, and each node can potentially be a starting node, the total number of paths explored is 2^N. Each path exploration involves traversing through the graph and checking each neighbor, resulting in a time complexity of O(N) for each path, where N is the number of vertices in the graph. Therefore, the overall time complexity is $O(2^N \cdot N)$.

2.2.2 Dijkstra's Method

Summary:
This method finds the critical method by applying Dijkstra's algorithm to find the critical path in a directed acyclic graph by reversing all edge weights. First**,** reverse the weights of all edges in the graph. Instead of representing the duration of tasks, they will now represent the time at which a task can be started after its dependencies are completed. Then, apply Dijkstra's algorithm to find the shortest paths from each node to all other nodes in the reversed-weight graph. This will

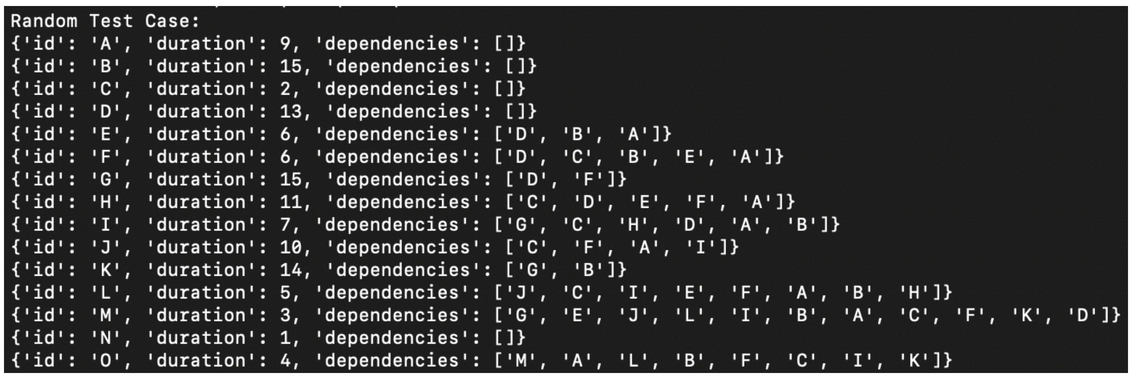
give you the earliest possible start times for each task. After obtaining the earliest start times for each task, traverse the graph and select the path that maximizes the sum of durations while minimizing the start times. This path will represent the critical path. Please see Appendix 5.2 for its implementation (pseudocode) and explanation.

Time Complexity:
For the dijkstra function, the use of a heap data structure ensures that the node with the shortest distance is always selected next, leading to an overall time complexity of O((V + E) log V), where V is the number of vertices and E is the number of edges in the graph. The time complexity of other steps are ignorable. Since we will execute it for each of the nodes in the graph, the overall time complexity of this method is O((V + E) Vlog V), which is significantly better than the runtime of the brute force method.

2.3 Performance Evaluation of the Algorithms

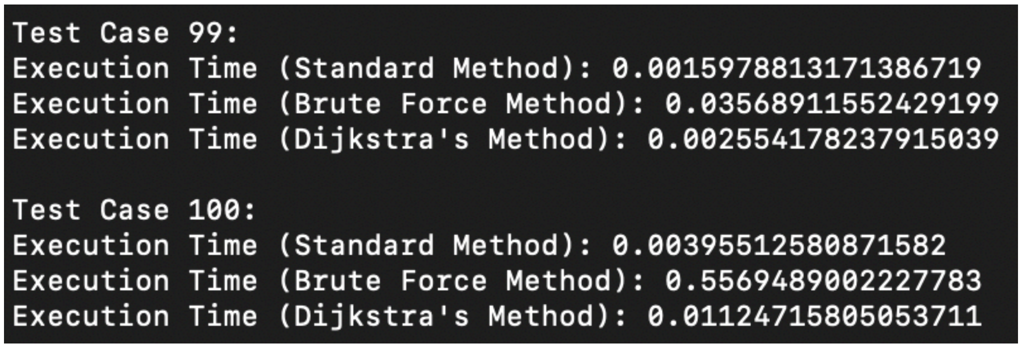
2.3.1 Test Cases and Methods

To evaluate the performance of the 3 methods above, we created a program that generates 100 random test cases, each test having between 1 and 50 tasks (nodes) and another 100 random test cases with between 50 and 100 tasks. Below is an example test case:



```
Random Test Case:
{'id': 'A', 'duration': 9, 'dependencies': []}
{'id': 'B', 'duration': 15, 'dependencies': []}
{'id': 'C', 'duration': 2, 'dependencies': []}
{'id': 'D', 'duration': 13, 'dependencies': []}
{'id': 'E', 'duration': 6, 'dependencies': ['D', 'B', 'A']}
{'id': 'F', 'duration': 6, 'dependencies': ['D', 'C', 'B', 'E', 'A']}
{'id': 'G', 'duration': 15, 'dependencies': ['D', 'F']}
{'id': 'H', 'duration': 11, 'dependencies': ['C', 'D', 'E', 'F', 'A']}
{'id': 'I', 'duration': 7, 'dependencies': ['G', 'C', 'H', 'D', 'A', 'B']}
{'id': 'J', 'duration': 10, 'dependencies': ['C', 'F', 'A', 'I']}
{'id': 'K', 'duration': 14, 'dependencies': ['G', 'B']}
{'id': 'L', 'duration': 5, 'dependencies': ['J', 'C', 'I', 'E', 'F', 'A', 'B', 'H']}
{'id': 'M', 'duration': 3, 'dependencies': ['G', 'E', 'J', 'L', 'I', 'B', 'A', 'C', 'F', 'K', 'D']}
{'id': 'N', 'duration': 1, 'dependencies': []}
{'id': 'O', 'duration': 4, 'dependencies': ['M', 'A', 'L', 'B', 'F', 'C', 'I', 'K']}
```

The program uses each of the 3 methods to find the critical path for the given test case and output the time taken for execution. Below is an example output:



```
Test Case 99:
Execution Time (Standard Method): 0.0015978813171386719
Execution Time (Brute Force Method): 0.03568911552429199
Execution Time (Dijkstra's Method): 0.002554178237915039

Test Case 100:
Execution Time (Standard Method): 0.00395512580871582
Execution Time (Brute Force Method): 0.5569489002227783
Execution Time (Dijkstra's Method): 0.01124715805053711
```

To evaluate the performance of the methods, we will take the average time taken for the 100 random test cases. (All code and instructions to use are included with the report).

2.3.2 Results

The performance evaluation results are presented in the table below. The unit is second. As we can see, when the input size is small (1-50), the Critical Path Method is already significantly faster than the Brute Force Method and slightly better than Dijkstra’s method. With a greater input size (50-100), the Brute Force method is way too slow than the other two methods and the Critical Path Method has a greater advantage (over 4 times faster now) than Dijkstra’s..

We can see that even with small (we would say even 50-100 is still pretty small) input size, the Critical Path Method is more efficient than Dijkstra’s, not to mention the Brute Force Method. And we can reasonably infer that the performance gap will only widen as the input size further increases.

	Test Case Size 1-50	Test Case Size 50-100
Critical Path Method	0.000476460	0.00157213
Brute Force	14.8626	Too slow, probably takes days
Dijkstra’s	0.000765674	0.00728725

Table 1: The performance of different methods with input of different sizes.

3. Conclusion:

<u>Answering the question</u>

Overall, we introduce two ways of implementing the Critical Path Method. The first method - brute force method - by its nature, ensures our accuracy of finding the critical path within the project network. However, the limitation of this method is very obvious - the time complexity is huge. As the projects get bigger and the number of tasks increases, it will lead to exponential growth in computational time and be very inefficient for memory usage. This method can only be used for smaller projects so it is more of a theoretical tool for understanding the CPM theory in our algorithm class setting.

That is why we introduce Dijkstra’s which provides a significant improvement in efficiency. As we learned from our class, starting with a node and exploring all paths originating from it can allow us to handle real-world applications with a larger number of tasks.

As we can see, CPM is very useful for scheduling, monitoring and controlling the project and it can be used for fields such as operation management e.g. manufacturing industries, agriculture, academic research and technology and also for personal budget planning, everyday schedule planning etc.

Limitations and weaknesses

While CPM can help you prioritize your tasks and deliver the project on time, it has several limitations too; such that designing a CPM chart is very time consuming, especially for larger projects, you need to go through hundreds/thousands of tasks and dependency relationships. Additionally, people will need to predetermine the task durations, but in a real world setting, human errors and delays always happen, so the timeline is not always reliable.

While our project successfully implemented and compared different methods for finding the critical path, there are some weaknesses and limitations to acknowledge. Firstly, our evaluation focused primarily on the efficiency and performance of the algorithms in terms of execution time. While this is important, it doesn't provide a comprehensive analysis of other factors such as memory usage or scalability. Furthermore, our evaluation was based on synthetic test cases generated randomly. While this allows for controlled experimentation, real-world project networks may exhibit different characteristics and complexities.

Suggestions for future research

To address the weakness and limitations of this project, future research can dig deeper into the memory and scalability aspects to provide a more holistic understanding of the algorithms' usefulness for real-world applications. Future research could also involve testing the algorithms on real project data to validate their performance in practical scenarios instead of random tests. Lastly, our project primarily addressed the technical aspects of implementing and comparing algorithms for project management. Future research could explore the integration of algorithmic approaches with human-centered methodologies for improved project outcomes, since the human and organizational factors in project management are equally important.

What we have learned

Li Hao - This topic has provided me with an intuitive insight of combining algorithms we learned in class into useful tools we have in everyday life. Algorithms is a very important class for computer science students and by doing this project, I gained a much deeper understanding of the benefit different algorithms can provide us.

Hejia Zhang - CPM is a topic closely related to project management and it is also similar to the concept we saw in the textbook - activity selection problem. Instead of learning the concept of CPM from readings/online resources, I've had an opportunity to get hands-on experience of writing and creating the critical path method.

Yuqi Hu - By implementing and comparing brute force method and Dijkstra's approach to CPM, I find the importance of selecting algorithms for doing computer science projects. It is crucial, especially after we graduate, to take the advantage of different algorithms we learned in class in order to improve the effectiveness and accuracy of our model.

4.Reference

https://cloudfresh.com/en/blog/critical-path-method-how-it-influences-the-project-management-process/
https://blog.hubspot.com/marketing/critical-path-method
https://networkx.org/documentation/stable/reference/introduction.html

5.Appendix

5.1 Brute Force Method Pseudocode and Explanation

Pseudocode:

```
build_graph(tasks):
    graph = {}
    for each task in tasks:
    graph[task['id']] = {'duration': task['duration'], 'neighbors': [], 'incoming': 0}
    for each task in tasks:
    if task has dependencies:
    for each dependency in task['dependencies']:
        add task['id'] to neighbors list of dependency in graph
        increment incoming count of task['id'] in graph
    return graph

find_longest_path(graph, current_node, visited):
    add current_node to visited set
    max_duration = 0
    critical_path = [current_node]
    for each neighbor in neighbors list of current_node in graph:
    if neighbor not in visited:
    duration, path = recursively call find_longest_path with neighbor, visited
    if duration + duration of current_node in graph > max_duration:
        update max_duration to duration + duration of current_node in graph
        update critical_path to [current_node] concatenated with path
    return max_duration, critical_path

calculate_critical_path_brute(tasks):
    graph = build_graph(tasks)
    initialize longest_duration to 0
    initialize critical_path to empty list
    for each node in graph:
    duration, path = call find_longest_path(graph, node, empty set)
    if duration > longest_duration:
    update longest_duration to duration
    update critical_path to path
    return critical_path
```

<u>Pseudocode Explanations:</u>

1. **Building the Graph:**
    a. The **build_graph** function constructs a graph representation of the tasks.
    b. Each task is represented as a node in the graph, and dependencies are represented as directed edges between nodes.
2. **Finding the Longest Path:**
    a. The **find_longest_path** function recursively explores all possible paths starting from a given node in the graph.
    b. It maintains a set of visited nodes to avoid revisiting the same nodes, ensuring that we don't get stuck in cycles.
    c. For each neighbor of the current node, it recursively explores the path, calculating the total duration of the path.
    d. It returns the maximum duration and the corresponding critical path found starting from the current node.
3. **Calculating the Critical Path:**
    a. The **calculate_critical_path** function iterates over each node in the graph and finds the longest path starting from that node using the find_longest_path function.
    b. It keeps track of the longest duration and the corresponding critical path found so far.
    c. After exploring paths starting from all nodes, it returns the critical path with the maximum duration as the overall critical path for the project.

5.2 Dijkstra's Method Pseudocode and Explanation

<u>Pseudocode:</u>

```
Function: calculate_critical_path_dijkstra(tasks)
        Input: tasks - list of task dictionaries, each containing id, duration, and dependencies
(optional)

        Function: build_graph(tasks)
        Initialize an empty graph dictionary.
```

```
    For each task in tasks:
        Add the task to the graph with its id as the key and a dictionary containing
duration and an empty list of neighbors as the value.
        If the task has dependencies:
            For each dependency in the task's dependencies:
            Add the task's id to the list of neighbors for the dependency in the graph.
    Return the built graph.

    Function: dijkstra(graph, start)
    Initialize a distances dictionary with all nodes in the graph set to infinity.
    Set the distance of the start node to 0.
    Initialize a priority queue with the start node and its distance.
    Initialize a previous dictionary to keep track of the previous node in the shortest path.
    While the priority queue is not empty:
        Pop the node with the smallest distance from the priority queue.
        For each neighbor of the current node:
            Calculate the distance to the neighbor via the current node.
            If this distance is smaller than the current distance to the neighbor:
            Update the distance to the neighbor.
            Update the previous node for the neighbor.
            Push the neighbor and its updated distance to the priority queue.
    Adjust the distances by adding the durations of each task to them.
    Return the distances and previous dictionaries.

    Function: get_critical_path(distances, previous)
    Initialize max_distance to -1 and critical_node to None.
    For each node and its distance in distances:
        If the distance is not infinity and it's greater than max_distance:
            Update max_distance and critical_node.
    Initialize a path list with the critical_node.
    While there is a previous node:
        Append the previous node to the path list.
        Update the current node to the previous node.
    Return the max_distance and the reversed path list.

    Initialize an empty list paths to store critical paths.
    Build the graph from the tasks.
    For each task in tasks:
        Find the critical path starting from the task using Dijkstra's algorithm.
        Append the critical path to the paths list.
    Sort the paths list.
    Return the last item in the sorted paths list, which contains the longest critical path.
```

Pseudocode Explanation:

1. **build_graph(tasks)**:
   - This function constructs a graph representation of the tasks.
   - It iterates over each task in the input list of tasks.
   - For each task, it adds a node to the graph with the task's ID as the key and a dictionary containing its duration and an empty list of neighbors as the value.
   - If the task has dependencies, it iterates over them and adds the task's ID to the list of neighbors for each dependency.
   - Finally, it returns the built graph.
2. **dijkstra(graph, start)**:
   - This function implements Dijkstra's algorithm to find the shortest paths from a given start node to all other nodes in the graph.
   - It initializes a distance dictionary with all nodes in the graph set to infinity, except for the start node, which is set to 0.
   - It maintains a priority queue of nodes ordered by their distances from the start node.
   - It iterates over nodes in the priority queue, updating their distances and previous nodes as necessary until all reachable nodes have been visited.
   - After updating distances, it adjusts them by adding the duration of each task to account for the time taken to complete each task.
   - Finally, it returns the distance dictionary and the previous node dictionary.
3. **get_critical_path(distances, previous)**:
   - This function extracts the critical path from the distances and previous node dictionaries obtained from Dijkstra's algorithm.
   - It initializes max_distance to -1 and critical_node to None.
   - It iterates over nodes in the distances dictionary, finding the node with the maximum distance that is not infinity.
   - It then constructs the critical path by traversing the previous nodes backward from the critical node.
   - Finally, it returns the maximum distance and the reversed critical path.
4. **Main Algorithm**:
   - It initializes an empty list paths to store critical paths.
   - It builds the graph from the input list of tasks using the build_graph function.
   - It iterates over each task in the input list of tasks.
   - For each task, it finds the critical path starting from that task using Dijkstra's algorithm.
   - It appends each critical path to the paths list.

- After finding all critical paths, it sorts the paths list to identify the longest critical path.
- Finally, it returns the longest critical path found.