Task Scheduling and Graph Visualization

By Hsa Moo

Project Objectives

- Get the most efficient paths and schedules
- Create a clear visual representations of task dependencies
- To see which area needs more resource to be allocated
- Find shortest paths in weighted graphs.

Given Tasks

1. Task A: Start project (Duration: 2 hours)
2. Task B: Gather requirements (Duration: 4 hours)
3. Task C: Design system (Duration: 6 hours)
4. Task D: Develop system (Duration: 8 hours)
5. Task E: Test system (Duration: 3 hours)
6. Task F: Deploy system (Duration: 2 hours)

Dependencies:

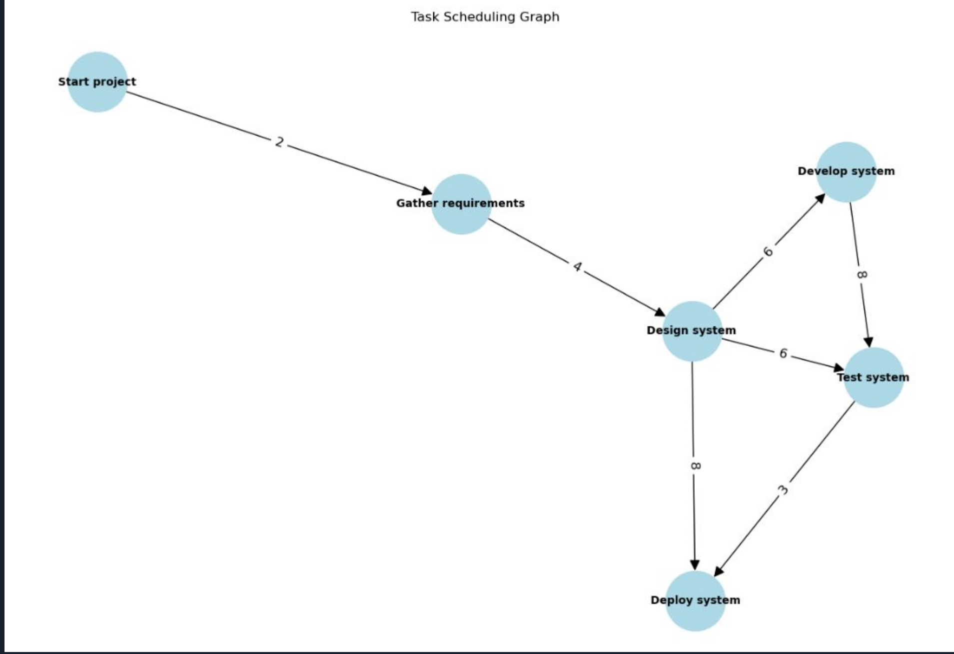
Task B depends on Task A. Task C and Task D depends on Task B. Task D depends on Task C. Task E depends on Task D. Task F depends on Task E.

Implementation in Python

- Using dictionaries for distances and previous nodes, and a priority queue (min-heap) for efficient node selection.
- Time Complexity: O(V log V + E), where V is the number of vertices (tasks) and E is the number of edges (dependencies)
- Space Complexity: O(V): The algorithm requires space proportional to the number of vertices (tasks) for maintaining the distances, previous_nodes, and priority_queue data structures.

```python
import heapq

def dijkstra(graph, start):
    # Initialize distances dictionary with all nodes set to infinity
    distances = {node: float('infinity') for node in graph}
    # Distance to start node is 0
    distances[start] = 0

    # Initialize priority queue with tuple (distance, node)
    priority_queue = [(0, start)]
    # Convert list to heap
    heapq.heapify(priority_queue)

    # Initialize dictionary to store previous nodes in optimal path
    previous_nodes = {node: None for node in graph}

    # Dijkstra's algorithm main loop
    while priority_queue:
        # Pop node with smallest distance from priority queue
        current_distance, current_node = heapq.heappop(priority_queue)

        # If current distance is greater than recorded distance, skip processing
        if current_distance > distances[current_node]:
            continue

        # Iterate over neighbors of current node
        for neighbor, weight in graph[current_node].items():
            # Calculate distance to neighbor via current node
            distance = current_distance + weight

            # If new path to neighbor is shorter, update distance and previous node
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                previous_nodes[neighbor] = current_node
                # Push updated distance and neighbor to priority queue
                heapq.heappush(priority_queue, (distance, neighbor))

    # Return distances and previous nodes for each node
    return distances, previous_nodes

def get_task_order(previous_nodes, start, end):
    path = []
    current_node = end
    # Reconstruct optimal path from end to start node
    while current_node is not None:
        path.append(current_node)
        current_node = previous_nodes[current_node]
    # Reverse path to get correct order from start to end node
    path = path[::-1]
    return path

# Representing tasks and their dependencies
graph = {
    'TaskA': {'TaskB': 2},
    'TaskB': {'TaskC': 4, 'TaskD': 6},
    'TaskC': {'TaskE': 6},
    'TaskD': {'TaskE': 8},
    'TaskE': {'TaskF': 3},
    'TaskF': {}
}

# Apply Dijkstra's algorithm from 'TaskA' to find distances and optimal paths
distances, previous_nodes = dijkstra(graph, 'TaskA')

# Get the optimal task order from 'TaskA' to 'TaskF'
task_order = get_task_order(previous_nodes, 'TaskA', 'TaskF')

# Print results: optimal task order and distances from 'TaskA'
print("Task Order:", task_order)
print("Distances:", distances)
```

Result Interpretation



The task order starts with TaskA(start project) then follow by TaskB(Gather requirements), TaskC(Design System), TaskE(Test System), and TaskF(Deploy System).

This takes a total of 15 hours for it to reach TaskF and since the Task C and Task D depends on Task B there can be a shorter path to Deploy System.

```
Task Order: ['TaskA', 'TaskB', 'TaskC', 'TaskE', 'TaskF']
Distances: {'TaskA': 0, 'TaskB': 2, 'TaskC': 6, 'TaskD': 8, 'TaskE': 12, 'TaskF': 15}
```

Conclusion

- Benefits: Minimizes project duration by skipping over tasks
- Applications: Project management, resource allocation
- Future Work: Enhancements, real-time scheduling, use more specified constraints