Cover

Created →
Dec 18, 2024 8:00 PM GMT+3

Last update →
Dec 22, 2024 10:00 PM GMT+3

PROJECT ROADMAP

Created By:

Hakan İSPİR
Deniz Yağmur Adaş
Ege Aslan

No	Student Name-Surname	What he/she did?	Contribution(%)
150322052	Hakan İSPİR	2.2, 2.3, 3.1.4, 3.2, 4.2, Creating codes and visualizations	40
150319053	Deniz Yağmur Adaş	Part 4, Part 2, Check-out procedure	30
150322053	Ege Aslan	Part 1, Part 3, 2.2, Helping in creating codes	30

Part 1: Introducti on

GreenPower Batteries Inc.

01

Introduction

1.1 Industry

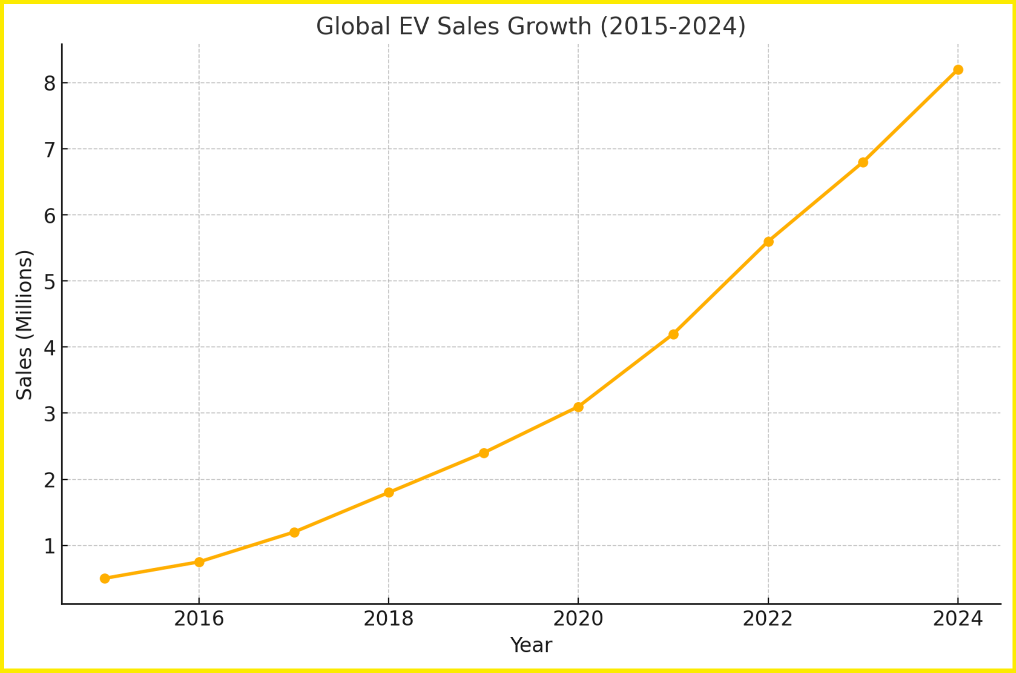
Automotive Industry

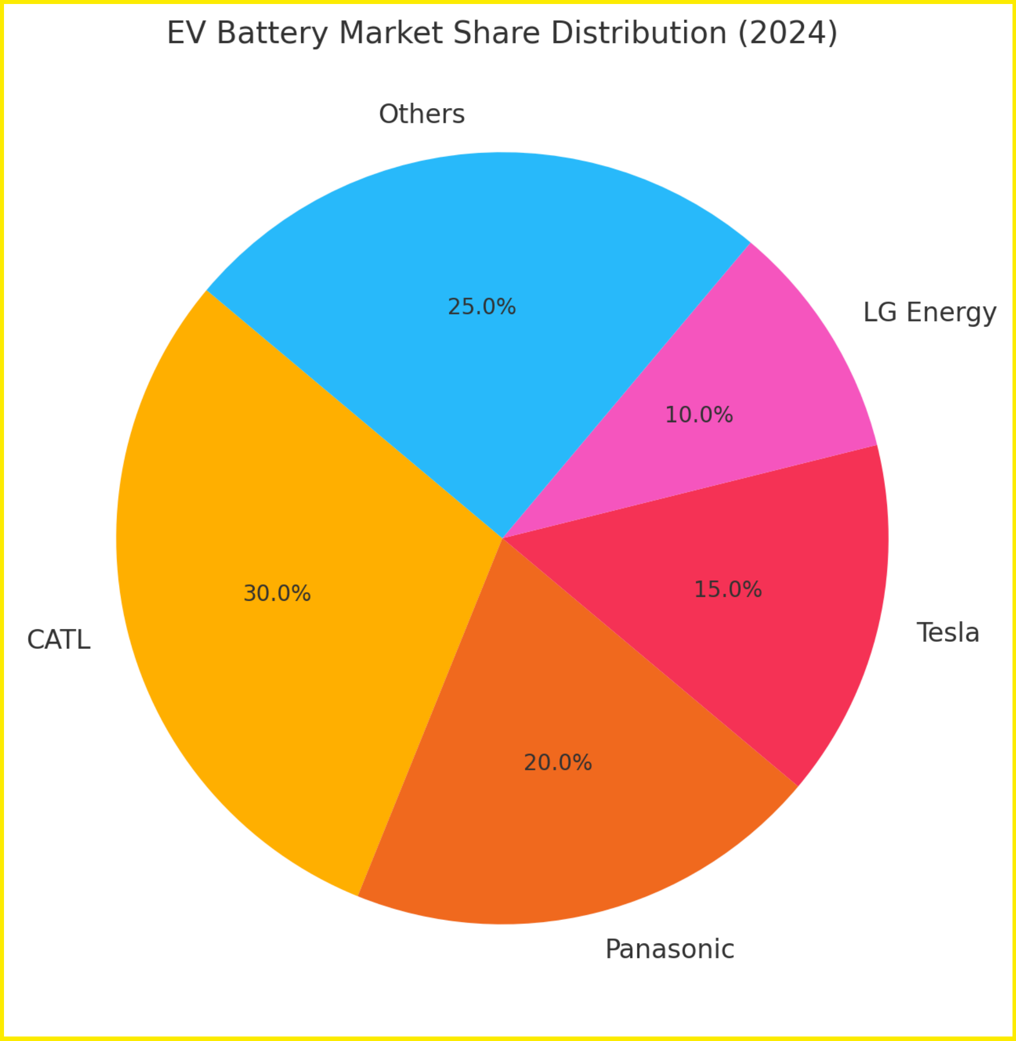
1.2 Product

Electric Vehicle (EV) Batteries

1.3 Description of EVs

- EV batteries are rechargeable energy sources that power the electric motors of electric vehicles (EVs).
- Typically made with lithium-ion technology, they are prized for their high energy density and long cycle life.
- These batteries are composed of individual cells organized into modules, which are then combined to form the complete battery pack.
- As critical components of EVs, they store and deliver electrical energy through carefully designed and precisely assembled systems, ensuring efficient and reliable performance.


Global EV Sales Growth (2015-2024)
Sales (Millions)
Year
1
2
3
4
5
6
7
8
2016
2018
2020
2022
2024

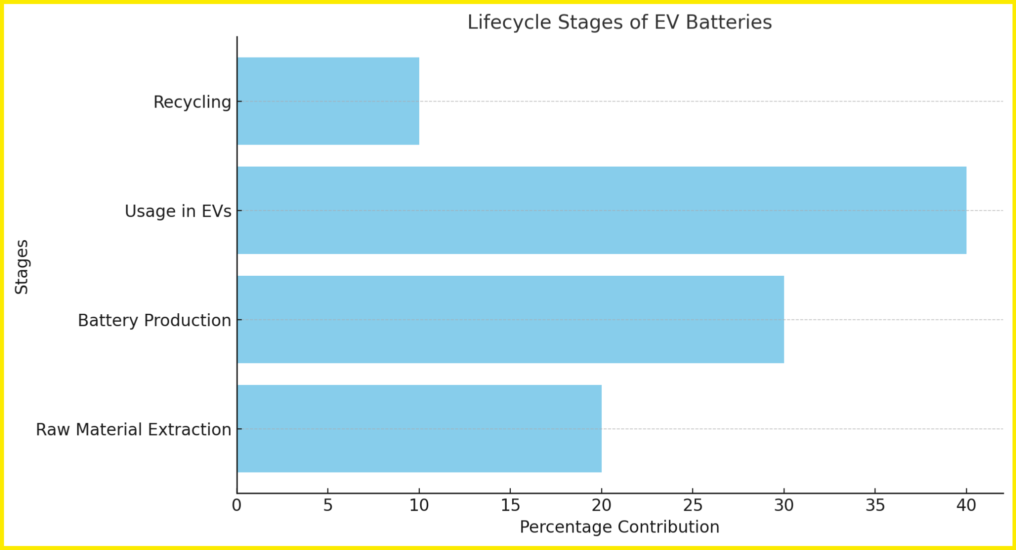


1.4 Company Name

GreenPower Batteries Inc.

1.5 Background of the Project

- The assembly line balancing problem is a critical aspect of production planning and control, particularly in the manufacturing of complex products such as Electric Vehicle (EV) batteries.
- Efficient assembly line balancing can lead to significant improvements in productivity, cost reduction, and product quality.
- The shift toward electric mobility driven by environmental concerns and government regulations.
- Statistics on EV adoption rates, growth projections, and market share of EV batteries.



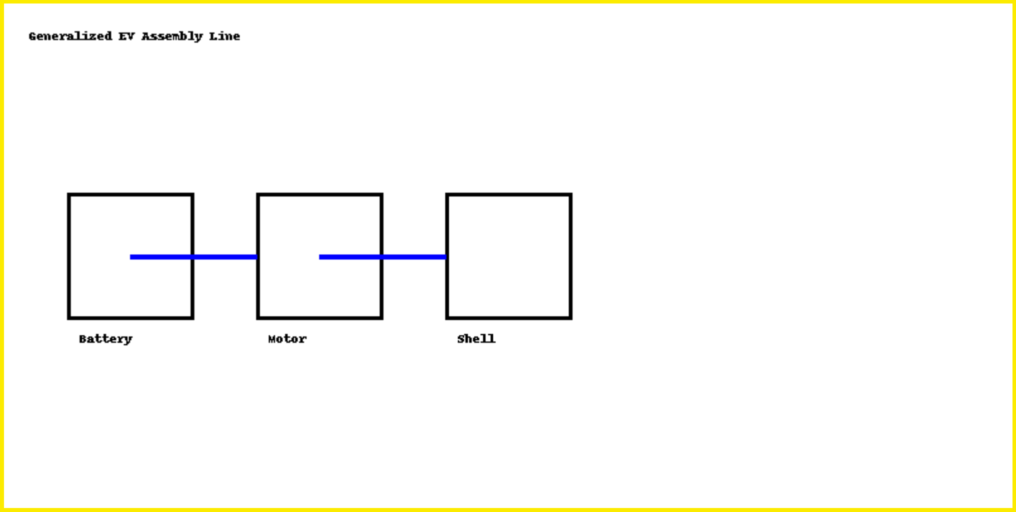
1.6 Objective

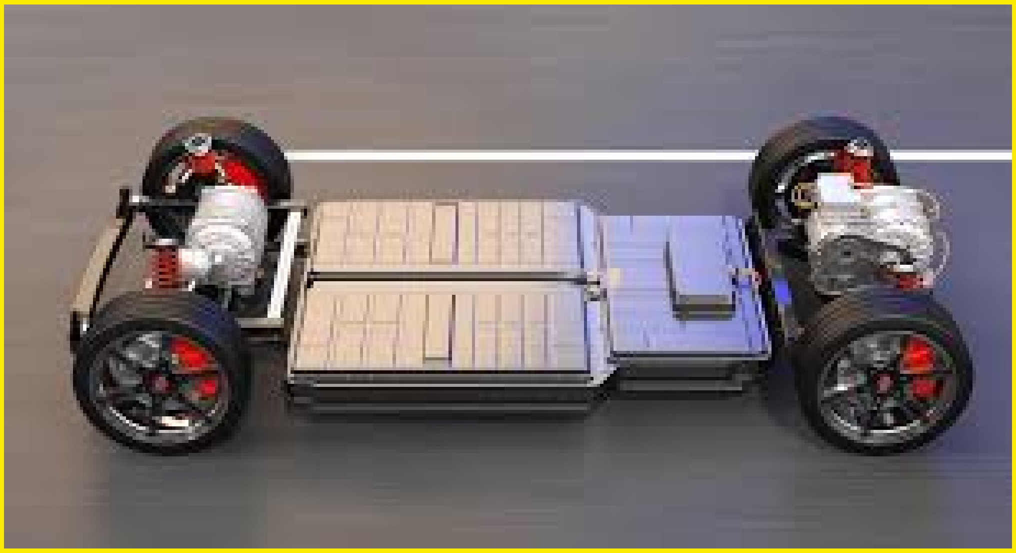
The objective of this project is to optimize the assembly process of EV batteries using Python. The project aims to:
- Generate synthetic data for task times and precedence relationships.

- Implement the COMSOAL (Computer Method of Sequencing Operations for Assembly Lines) method.
- Calculate performance metrics such as balance delay, line efficiency, and smoothness index.
- Visualize the results to provide insights into the assembly line performance.

1.7 Business Model of the Company

GreenPower Batteries Inc. focuses on the production and supply of high-quality lithium-ion batteries for electric vehicles.
The company aims to support the transition to sustainable transportation by providing reliable and efficient energy storage solutions.
The business model includes manufacturing, quality testing, and distribution of EV batteries to automotive manufacturers.
The company also invests in R&D to innovate and improve battery technology.


Generalized EV Assembly Line
Battery
Motor
Shell



Part 2: Methods Analysis

GreenPower Batteries Inc.

02

Methods Analysis

2.1 Tasks Constituting the Job

The tasks involved in the production of an EV battery include:

1. Cell Inspection

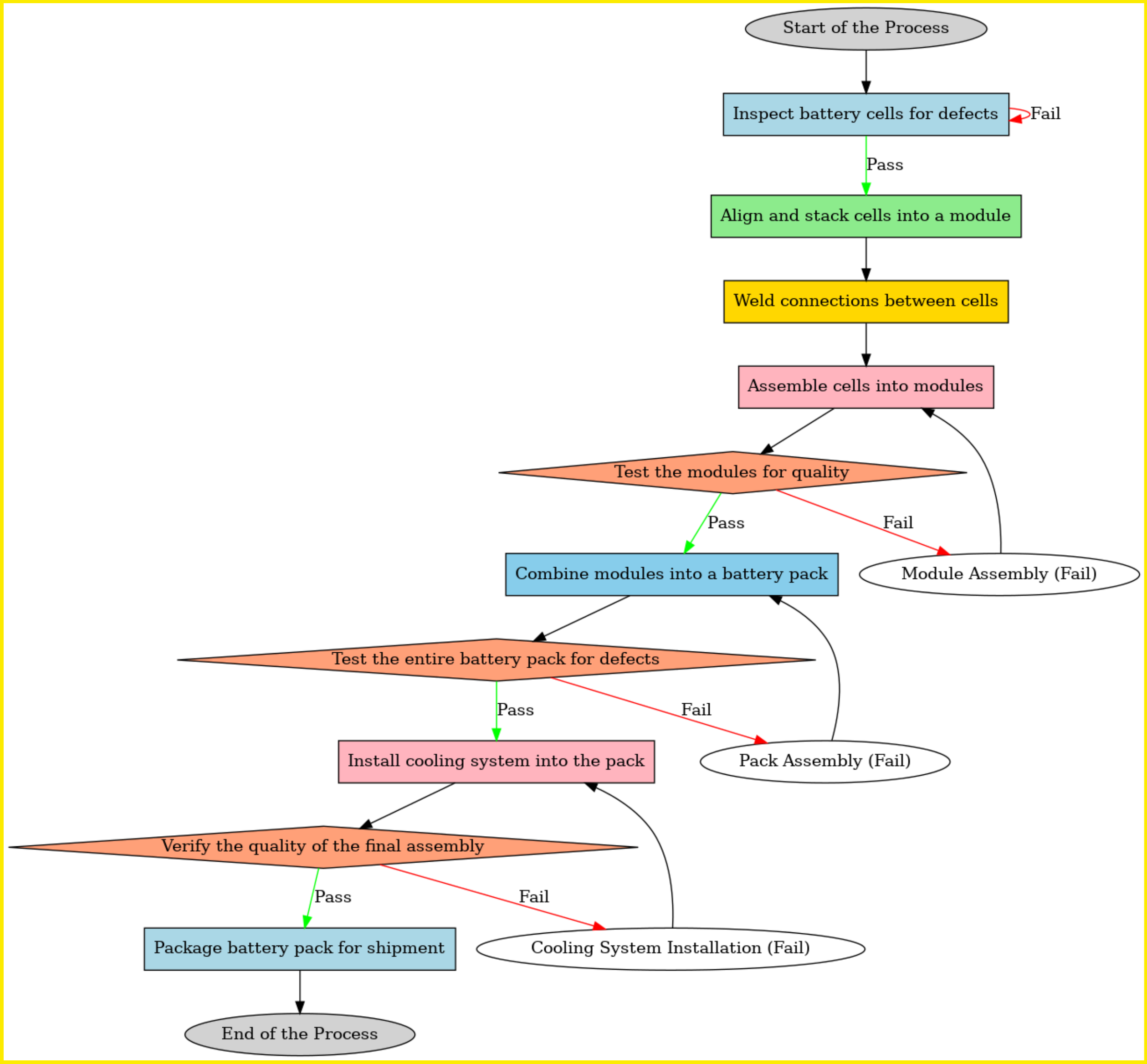
2. Cell Stacking
3. Welding
4. Module Assembly
5. Module Testing
6. Pack Assembly
7. Pack Testing
8. Cooling System Installation
9. Final Inspection
10. Packaging

2.2 Flow Diagram

A flow diagram is used to investigate the movement of people or materials during the assembly process.

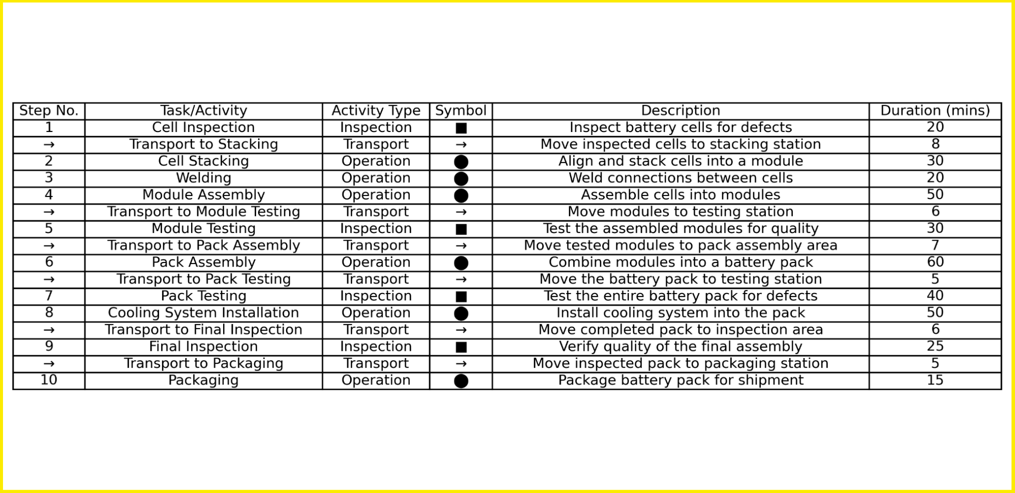
Below is a simplified flow diagram for the EV battery assembly:
Cell Inspection -> Cell Stacking -> Welding -> Module Assembly -> Module Testing -> Pack Assembly -> Pack Testing -> Cooling System Installation -> Final Inspection -> Packaging

Below is a more complex flow diagram for the EV battery assembly, created with python:

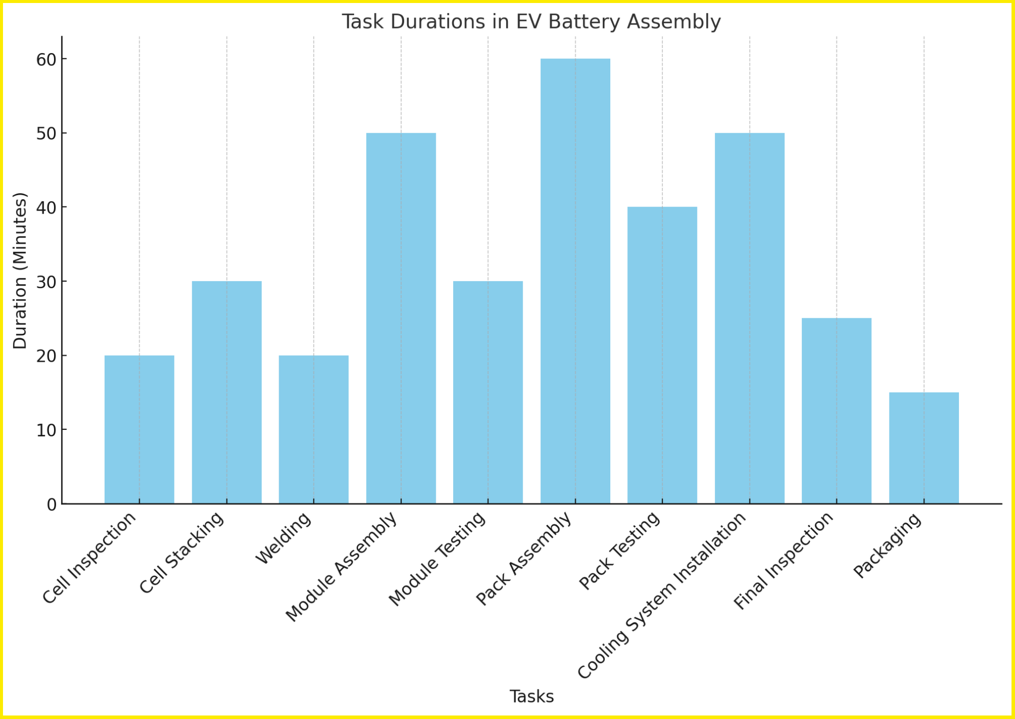

Start of the Process
Inspect battery cells for defects
Fail
Pass
Align and stack cells into a module
Weld connections between cells
Assemble cells into modules
Test the modules for quality
Pass
Fail
Combine modules into a battery pack
Module Assembly (Fail)
Test the entire battery pack for defects
Pass
Fail
Install cooling system into the pack
Pack Assembly (Fail)
Verify the quality of the final assembly
Pass
Fail
Package battery pack for shipment
Cooling System Installation (Fail)
End of the Process

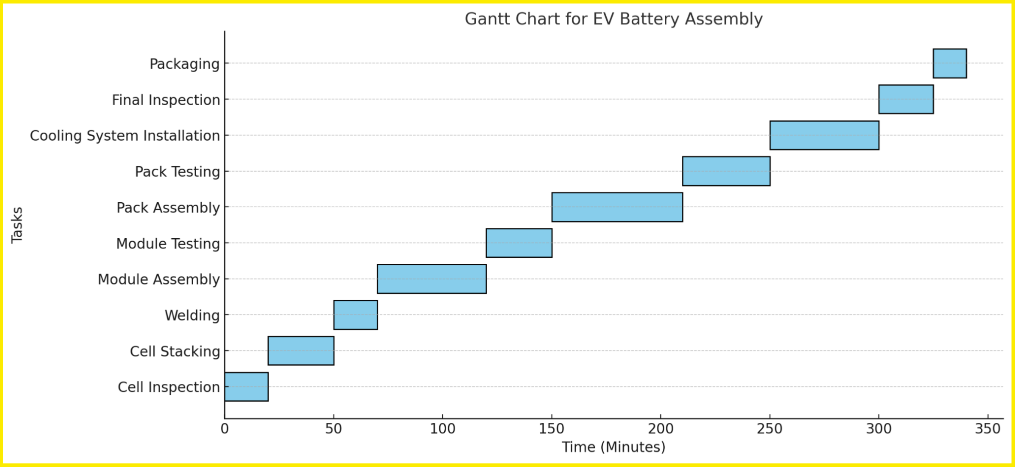
2.3 Process Chart

A process chart supplements the flow diagram by detailing each step in the assembly process, including the sequence and duration of tasks.



Step No.	Task/Activity	Activity Type	Symbol	Description	Duration (mins)
1	Cell Inspection	Inspection	■	Inspect battery cells for defects	20
→	Transport to Stacking	Transport	→	Move inspected cells to stacking station	8
2	Cell Stacking	Operation	●	Align and stack cells into a module	30
3	Welding	Operation	●	Weld connections between cells	20
4	Module Assembly	Operation	●	Assemble cells into modules	50
→	Transport to Module Testing	Transport	→	Move modules to testing station	6
5	Module Testing	Inspection	■	Test the assembled modules for quality	30
→	Transport to Pack Assembly	Transport	→	Move tested modules to pack assembly area	7
6	Pack Assembly	Operation	●	Combine modules into a battery pack	60
→	Transport to Pack Testing	Transport	→	Move the battery pack to testing station	5
7	Pack Testing	Inspection	■	Test the entire battery pack for defects	40
8	Cooling System Installation	Operation	●	Install cooling system into the pack	50
→	Transport to Final Inspection	Transport	→	Move completed pack to inspection area	6
9	Final Inspection	Inspection	■	Verify quality of the final assembly	25
→	Transport to Packaging	Transport	→	Move inspected pack to packaging station	5
10	Packaging	Operation	●	Package battery pack for shipment	15





2.4 Appendix

Creating a flow chart using Digraph class from graphviz module

```python
from graphviz import Digraph

# Create a new directed graph
dot = Digraph(comment="Manufacturing Process Flow Chart")

# Define the tasks and decision points
tasks = {
    "Start": "ellipse",
    "Material Inspection": "box",
    "Inspection Passed?": "diamond",
    "Cutting": "box",
    "Welding": "box",
    "Welding Quality OK?": "diamond",
    "Assembly": "box",
    "Painting": "box",
    "Painting Quality OK?": "diamond",
    "Final Inspection": "box",
    "Final Quality OK?": "diamond",
    "Packaging": "box",
    "End": "ellipse",
}

# Add nodes for each task with specified shapes
for task, shape in tasks.items():
    dot.node(task, task, shape=shape)

# Define the edges (flow) between tasks with decision points
edges = [
    ("Start", "Material Inspection"),
    ("Material Inspection", "Inspection Passed?"),
    ("Inspection Passed?", "Cutting", "yes"),
    ("Inspection Passed?", "Material Inspection", "no"),
    ("Cutting", "Welding"),
    ("Welding", "Welding Quality OK?"),
    ("Welding Quality OK?", "Assembly", "yes"),
    ("Welding Quality OK?", "Cutting", "no"),
    ("Assembly", "Painting"),
    ("Painting", "Painting Quality OK?"),
    ("Painting Quality OK?", "Final Inspection", "yes"),
    ("Painting Quality OK?", "Assembly", "no"),
    ("Final Inspection", "Final Quality OK?"),
    ("Final Quality OK?", "Packaging", "yes"),
    ("Final Quality OK?", "Painting", "no"),
    ("Packaging", "End"),
]

# Add edges to the graph with labels for decision points
for edge in edges:
    if len(edge) == 3:
        dot.edge(edge[0], edge[1], label=edge[2])
    else:
        dot.edge(edge[0], edge[1])

# Render the graph to a file
dot.render("manufacturing_process_flow_chart", format="png",
view=True)
```

Creating a process chart using pandas and matplotlib packages

```
import pandas as pd
import matplotlib.pyplot as plt

# Data for the process chart
```

```python
data = {
    "Step No.": [
        "1",
        "→",
        "2",
        "3",
        "4",
        "→",
        "5",
        "→",
        "6",
        "→",
        "7",
        "8",
        "→",
        "9",
        "→",
        "10",
    ],
    "Task/Activity": [
        "Cell Inspection",
        "Transport to Stacking",
        "Cell Stacking",
        "Welding",
        "Module Assembly",
        "Transport to Module Testing",
        "Module Testing",
        "Transport to Pack Assembly",
        "Pack Assembly",
        "Transport to Pack Testing",
        "Pack Testing",
        "Cooling System Installation",
        "Transport to Final Inspection",
        "Final Inspection",
        "Transport to Packaging",
        "Packaging",
    ],
    "Activity Type": [
        "Inspection",
        "Transport",
        "Operation",
        "Operation",
        "Operation",
        "Transport",
        "Inspection",
        "Transport",
        "Operation",
        "Transport",
        "Inspection",
        "Operation",
        "Transport",
        "Inspection",
        "Transport",
        "Operation",
    ],
    "Symbol": [
        "■",
        "→",
        "●",
        "●",
        "●",
        "→",
        "■",
        "→",
        "●",
        "→",
        "■",
        "●",
        "→",
        "■",
        "→",
        "●",
    ],
    "Description": [
        "Inspect battery cells for defects",
        "Move inspected cells to stacking station",
        "Align and stack cells into a module",
        "Weld connections between cells",
        "Assemble cells into modules",
        "Move modules to testing station",
        "Test the assembled modules for quality",
        "Move tested modules to pack assembly area",
        "Combine modules into a battery pack",
        "Move the battery pack to testing station",
        "Test the entire battery pack for defects",
        "Install cooling system into the pack",
        "Move completed pack to inspection area",
        "Verify quality of the final assembly",
        "Move inspected pack to packaging station",
        "Package battery pack for shipment",
    ],
    "Duration (mins)": [20, 8, 30, 20, 50, 6, 30, 7, 60, 5,
40, 50, 6, 25, 5, 15],
}

# Create a dataframe
df = pd.DataFrame(data)

# Plot the table as an image
fig, ax = plt.subplots(figsize=(12, 6))
ax.axis("tight")
ax.axis("off")
table = ax.table(
    cellText=df.values, colLabels=df.columns,
cellLoc="center", loc="center"
)
table.auto_set_font_size(False)
table.set_fontsize(10)
table.auto_set_column_width(col=list(range(len(df.columns)))
)
```

```python
# Save the table as an image or display
plt.savefig("ev_battery_process_chart.png",
bbox_inches="tight", dpi=300)
plt.show()
```

Part 3: Labour Standards

GreenPower Batteries Inc.

03

Labour Standards

3.1 Identifying Labour Standards

The time study method is used to establish labour standards for the EV battery assembly process.

3.1.1 Task to be Studied

The task to be studied is the assembly of an EV battery, which includes the 10 tasks listed in section 2.1.

3.1.2 Precise Elements of the Task

Each task is divided into precise elements, such as inspection, stacking, welding, etc.

3.1.3 Measuring the Task

In this project, the task times are not measured through direct observation but are instead **simulated using data analysis and programming tools**. A total of **30 epochs** (iterations of the task) are generated for each task to ensure a statistically significant dataset.

The simulated measurements are designed to replicate real-world scenarios by incorporating variability into the data. For example:

- Variations in task time are modeled using statistical distributions, such as the normal distribution, to account for realistic performance differences.

- Tools such as Python’s `numpy` library are used to create synthetic task times, ensuring that the data closely mirrors typical industrial processes.

This method provides a practical and efficient way to estimate task durations when direct observations are not feasible or available. By using simulated data, the project maintains statistical rigor and establishes a reliable foundation for further analysis, such as calculating **normal time (NT)** and **standard time (ST)** for each task.

3.1.4 Elemental Times and Ratings of Performance

To simulate the assembly line, synthetic data will be generated using Python's NumPy library. The data will include task times and variations, which are essential for realistic modeling of the assembly process.

Assumptions:
- The assembly line consists of 10 tasks.
- Task times are normally distributed with a mean of 5 minutes and a standard deviation of 1 minute.

```
import numpy as np
```

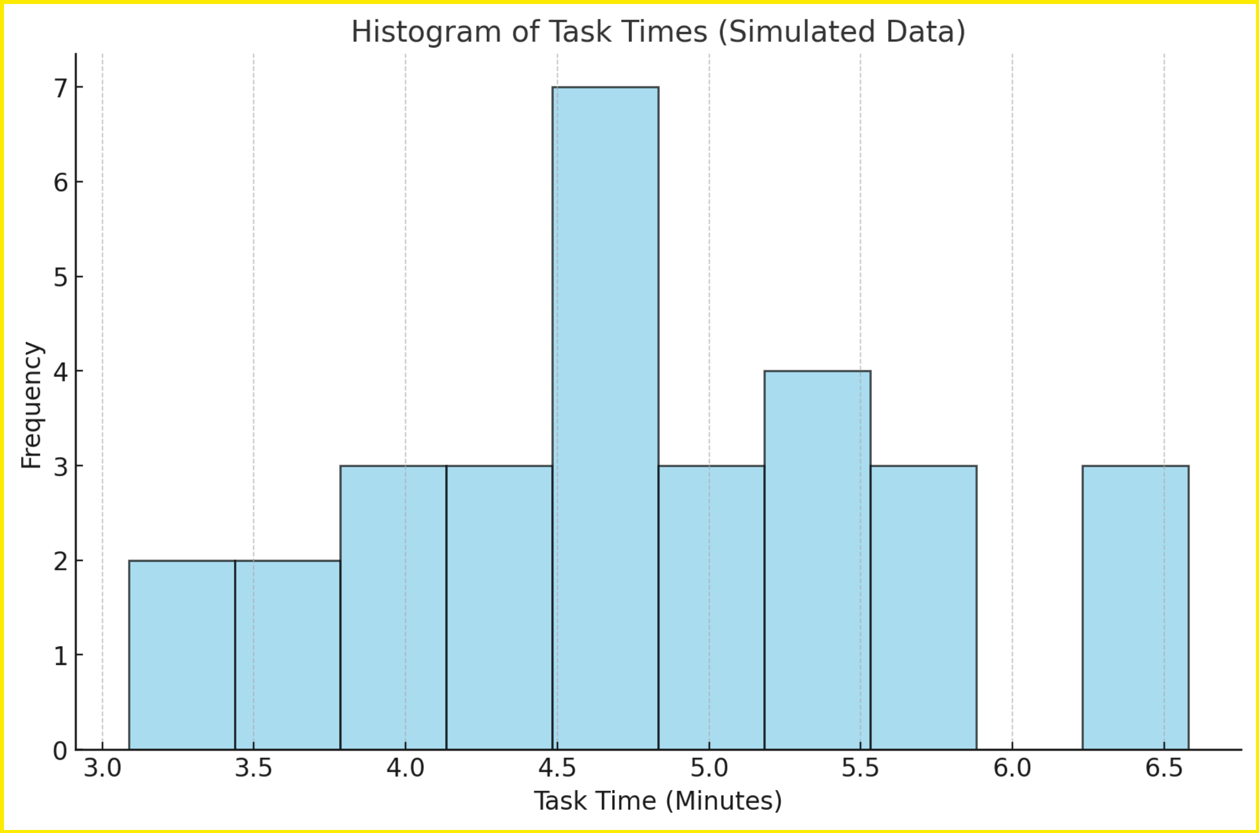
```python
# Seed for reproducibility
np.random.seed(42)

# Number of tasks
num_tasks = 10

# Generate random task times (mean=5, std=1)
task_times = np.random.normal(5, 1, num_tasks)
print("Task Times:", task_times)
```

Result:

```
Task Times:
[5.49671415 4.8617357  5.64768854 6.52302986 4.76584663
4.76586304 6.57921282 5.76743473 4.53052561 5.54256004]
```



3.1.5 Performance Rating and Normal Time

The performance rating is assumed to be 100%, and the normal time is the average of the recorded times.

3.1.6 Allowances

Allowances for fatigue, personal needs, and delays are assumed to be 15%. Allowances account for factors such as:

- **Fatigue (7%)**
    - **e.g.** welding generates heat, leading to faster fatigue.
- **Personal Time (5%)**
- **Delays (3%)**

The standard and normal time is computed as follows:

NT = Mean of task times

ST = NT * (1 + Allowance)

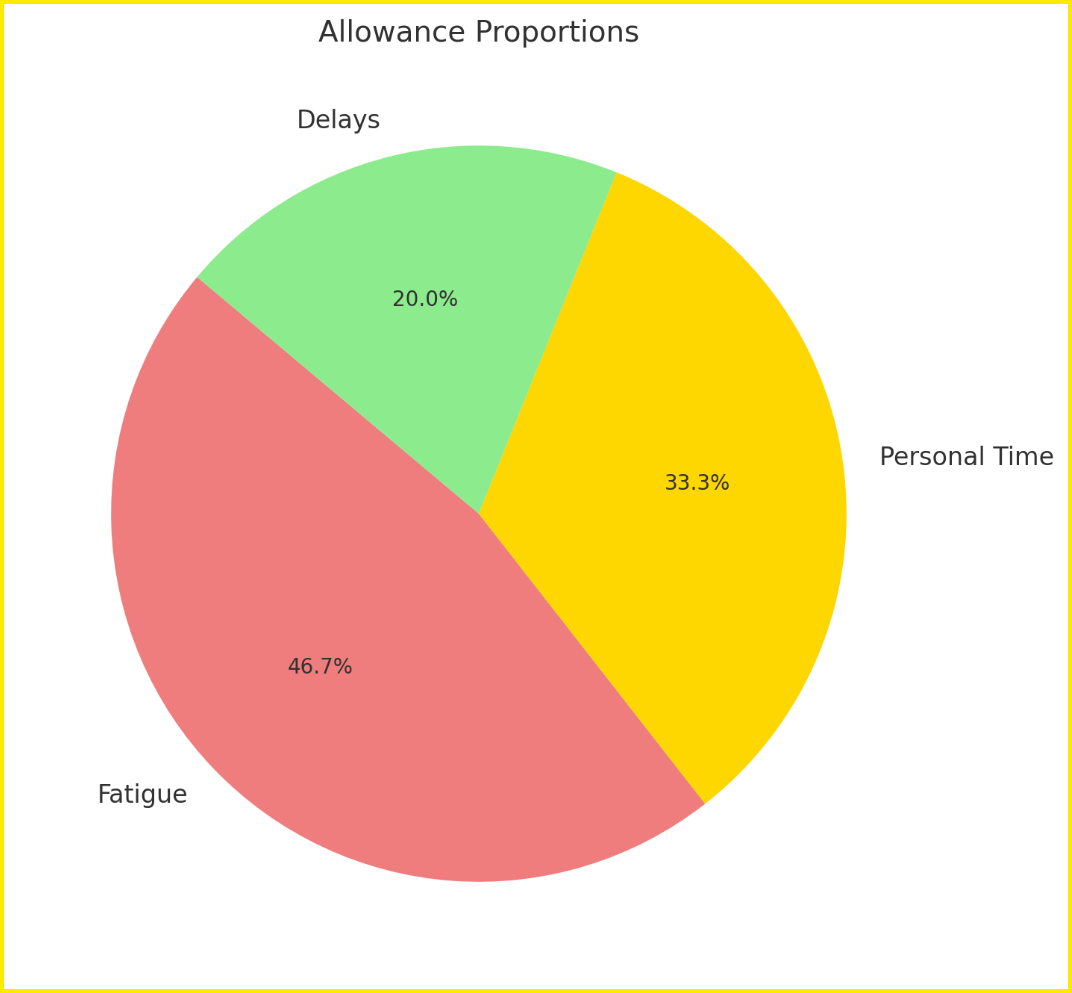
Result:

Normal Time: 5.448061

Standard Time: 6.265270

Code:

```python
allowance = 0.15
normal_time = np.mean(task_times)
standard_time = normal_time * (1 + allowance)
print(f"Normal Time: {normal_time:f}")
print(f"Standard Time: {standard_time:f}")
```



3.2 Percentage of Error and a Significance Level

A 5% error margin and a 95% confidence level are used to calculate the sample size.

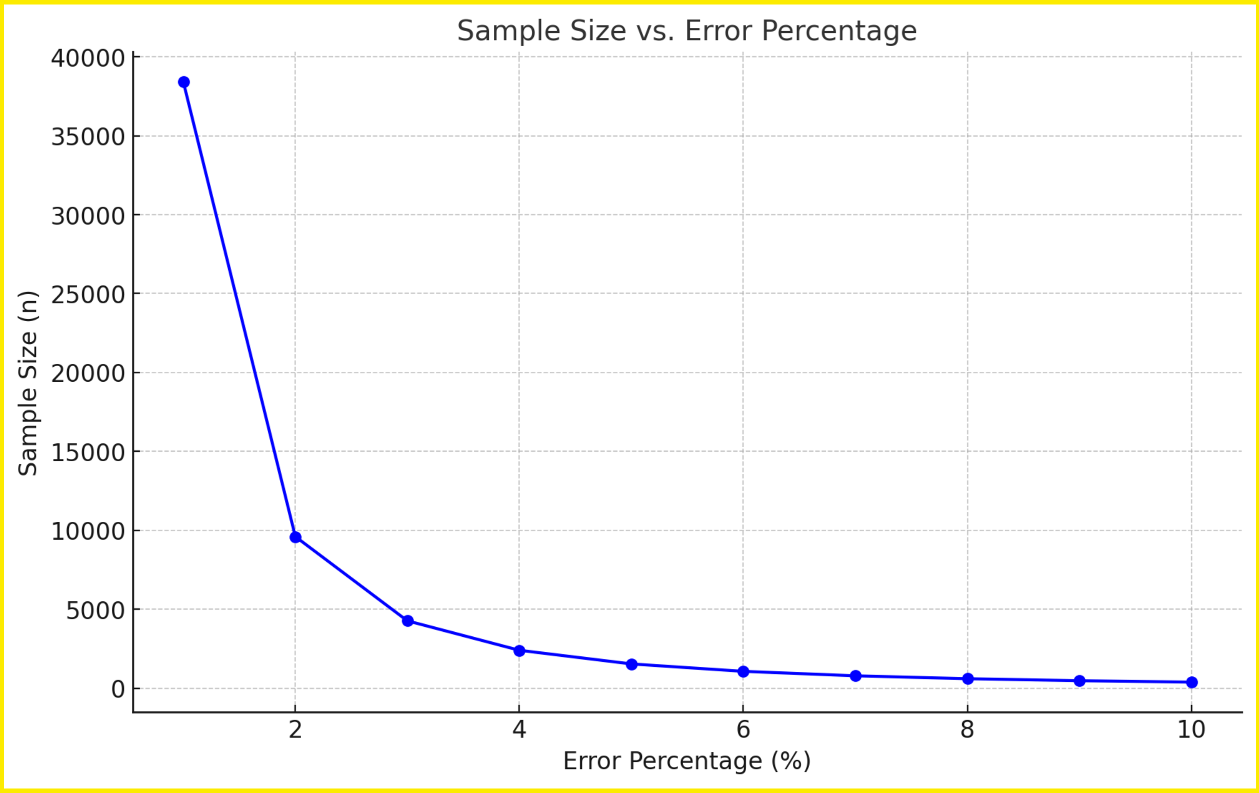
Result:
Sample Size: 25

Code:

```python
from scipy.stats import norm

# Error margin and confidence level
error_margin = 0.05
confidence_level = 0.95

# Calculate sample size
z_value = norm.ppf(1 - (1 - confidence_level) / 2)
sample_size = (z_value * np.std(task_times) / (error_margin *
np.mean(task_times))) ** 2
print("Sample Size:", int(np.ceil(sample_size)))
```



Part 4: Layout Organizati

on and

Discussio

n

GreenPower Batteries Inc.

04

Layout Organization and Discussion

4.1 The Appropriate Layout Type

The appropriate layout type for EV battery manufacturing is a **product layout,** as it allows for a streamlined and sequential flow of tasks.

- Alternatives:
  - **Process Layout:** Suitable for low-volume, high-variety products but not efficient here due to frequent material handling and increased travel time.
  - **Cellular Layout:** Could be considered if modularity is emphasized in the future, but it's less efficient for long production lines like batteries.

4.2 Layout Problem

Consider a manufacturing facility with the following departments that need to be laid out efficiently:

1. A: Assembly
2. B: Painting
3. C: Welding
4. D: Quality Control
5. E: Packaging
6. F: Storage

The goal is to minimize the total distance between departments using the Minimum Spanning Tree (MST) approach. The distances between departments are given in the following From-To Matrix table:

From-To Matrix:

	A	B	C	D	E	F
A	0	10	15	20	25	30
B	10	0	35	25	30	40
C	15	35	0	10	20	25
D	20	25	10	0	15	30
E	25	30	20	15	0	10
F	30	40	25	30	10	0

We will use Kruskal's algorithm to find the Minimum Spanning Tree (MST) for this layout problem.

Solving Using Kruskal's Algorithm

1. Sort all the edges in non-decreasing order of their weight.
2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If a cycle is not formed, include this edge. Else, discard it.
3. Repeat step 2 until there are (V-1) edges in the spanning tree.

Results:

Edges in the Minimum Spanning Tree:

A - B: 10 meters

C - D: 10 meters

E - F: 10 meters

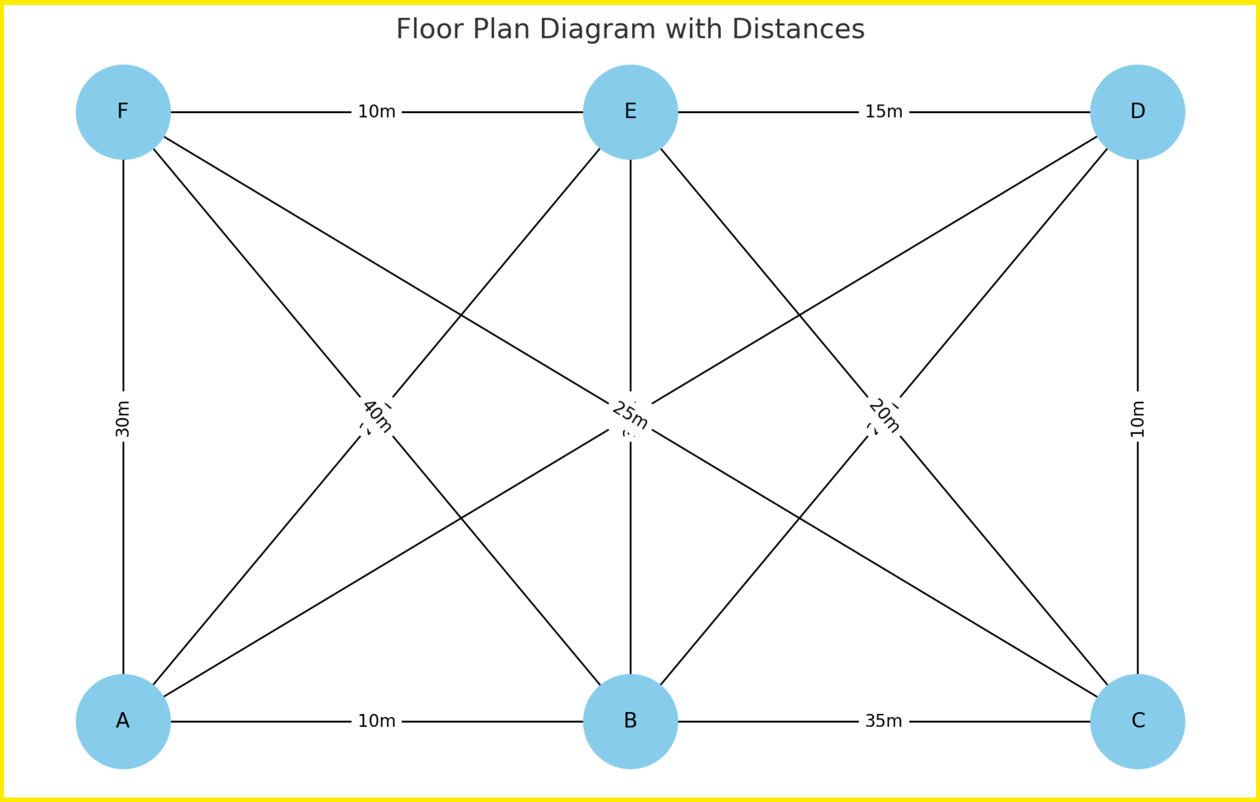
A - C: 15 meters

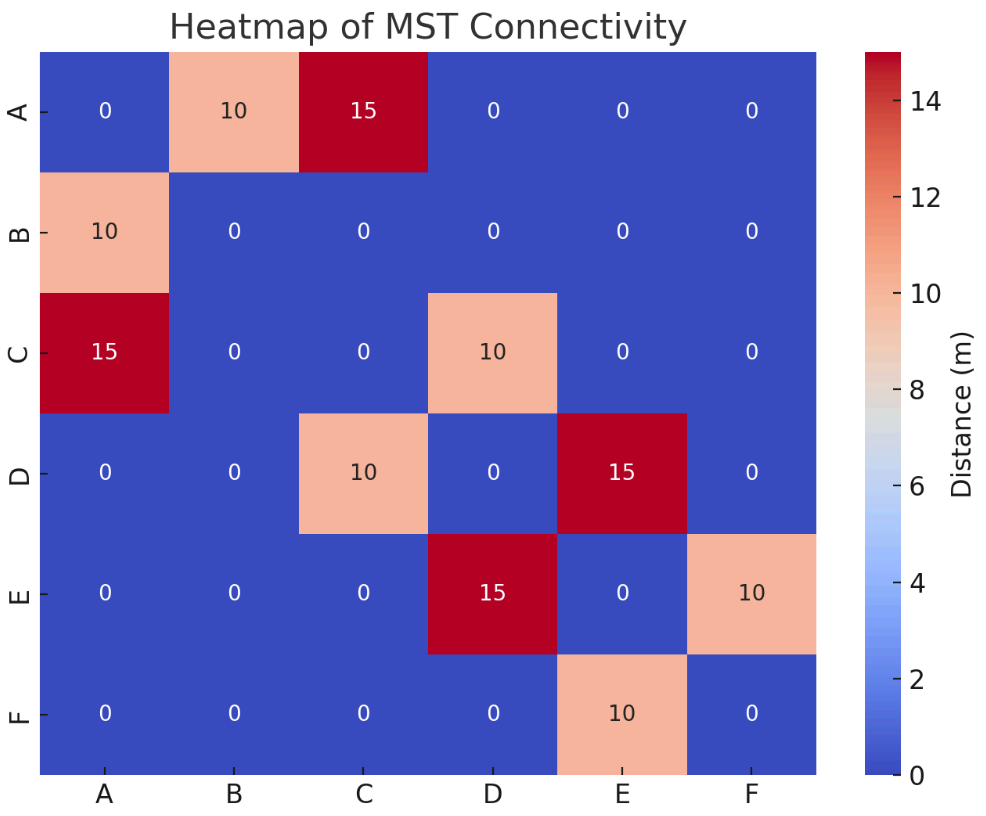
D - E: 15 meters

Therefore, in order to minimize distances, sequence should be:

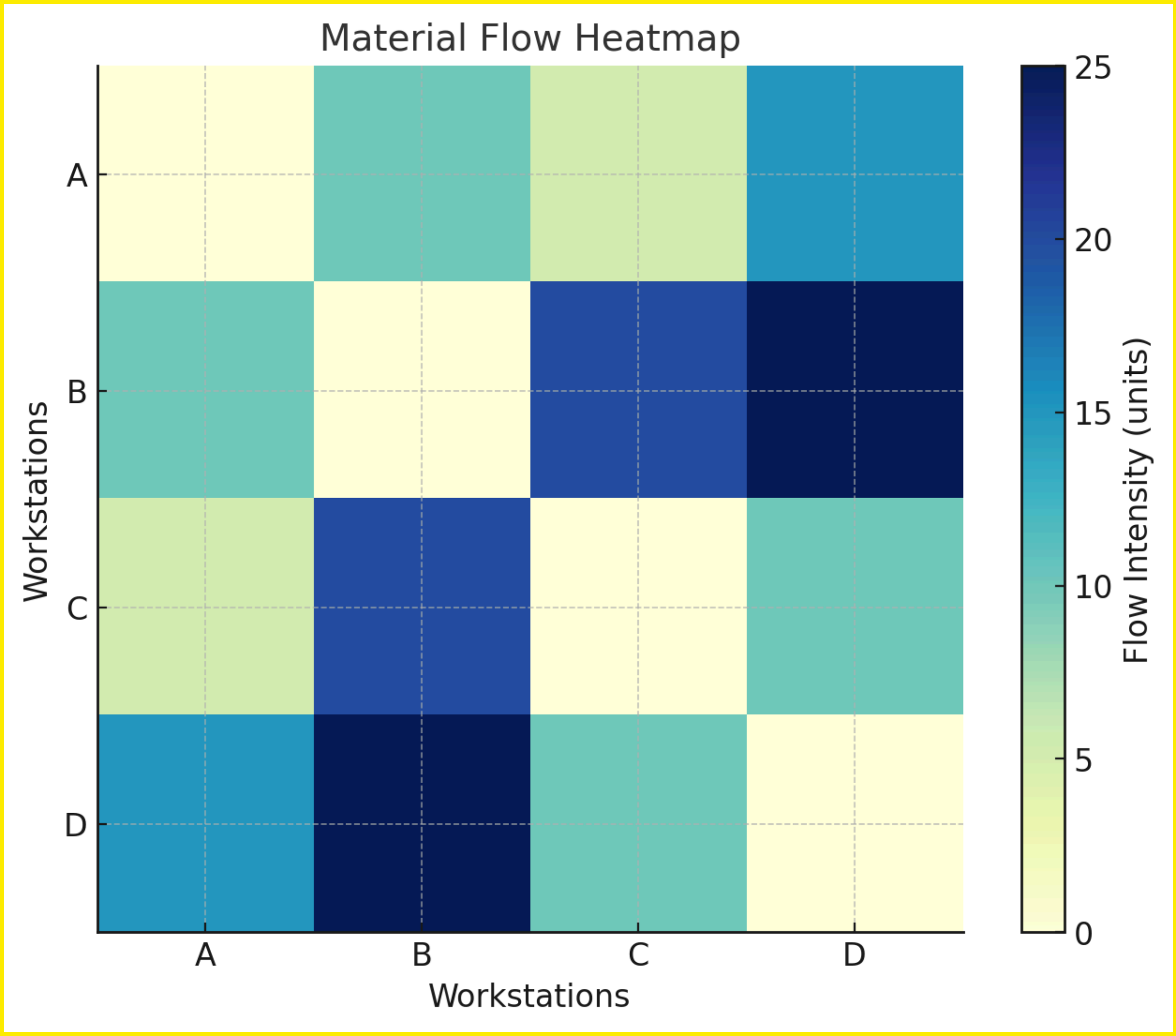
A - B - C - D - E - F

Diagram of the proposed sequence:





	A	B	C	D	E	F
A	0	10	15	0	0	0
B	10	0	0	0	0	0
C	15	0	0	10	0	0
D	0	0	10	0	15	0
E	0	0	0	15	0	10
F	0	0	0	0	10	0



4.3 Measuring the Efficiency of the Layout

To measure the efficiency of the layout, we can calculate the total distance of the MST and compare it to the total possible distance if all departments were directly connected.

Calculating Efficiency:

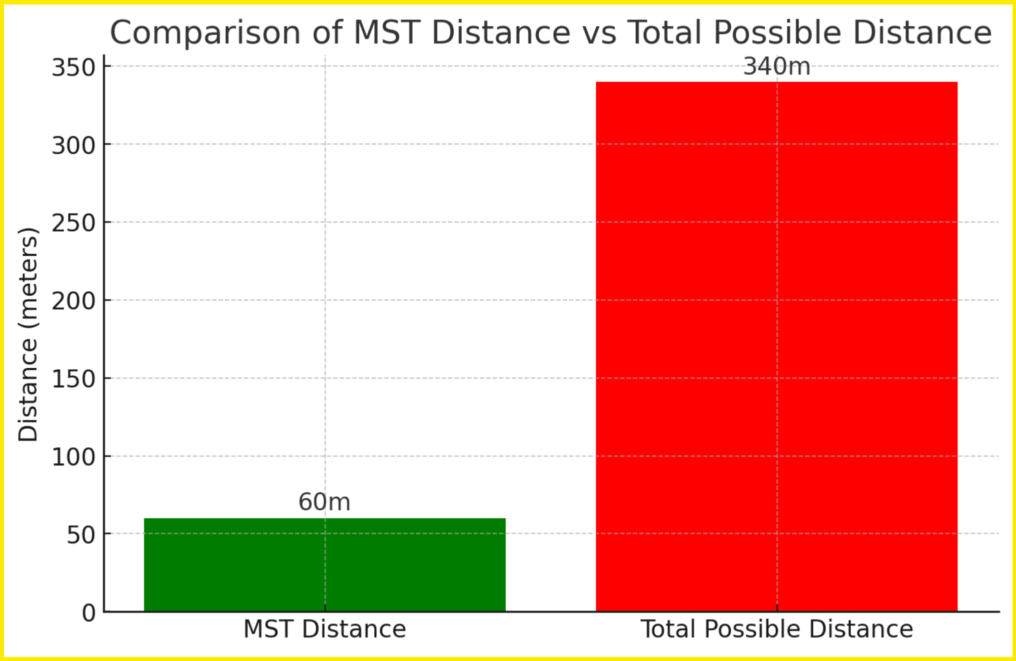
1. Total Distance of MST: Sum of the distances of the edges in the MST.
2. Total Possible Distance: Sum of all the distances in the original distance table.

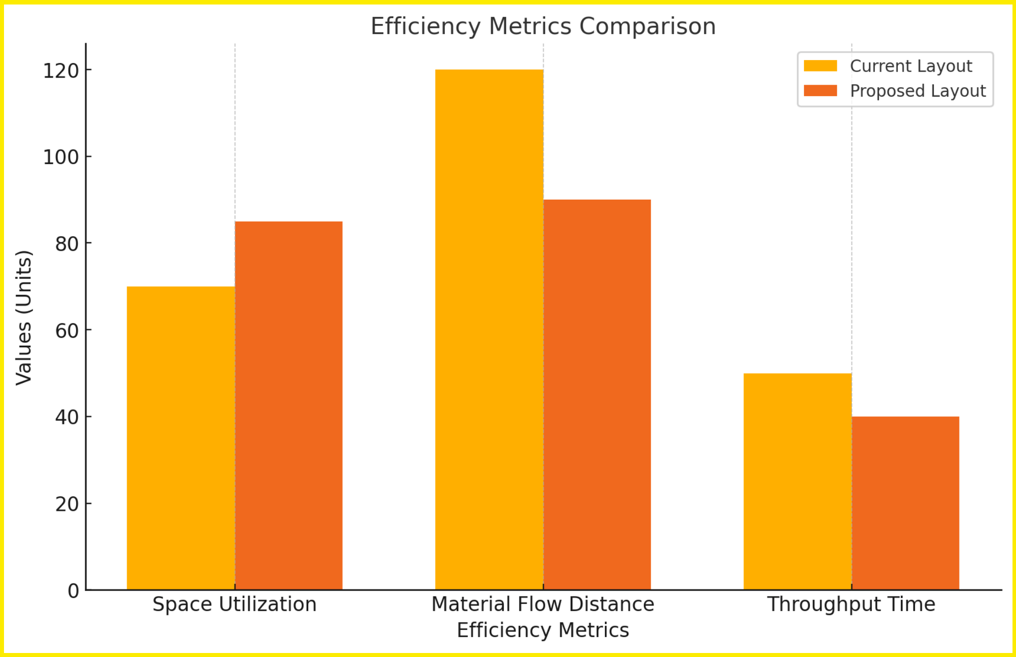
The efficiency can be calculated as:

Efficiency= (1 - Total Possible Distance / Total Distance of MST)×100%

Results:
Total Distance of MST: 60 meters
Total Possible Distance: 340 meters
Efficiency of the Layout: 82.35%

A graph highlighting the results:





4.3.1 Discussion and Suggestions

- Change Shift Time:
Adjusting shift times to match production demand can improve efficiency.

- Update Current Layout:
Regularly reviewing and updating the layout can address bottlenecks.

- Buy a New Machine:
Investing in new technology can enhance productivity.

- Invest in Technology:
Automation and advanced manufacturing techniques can further optimize the assembly process.

- Add automation:

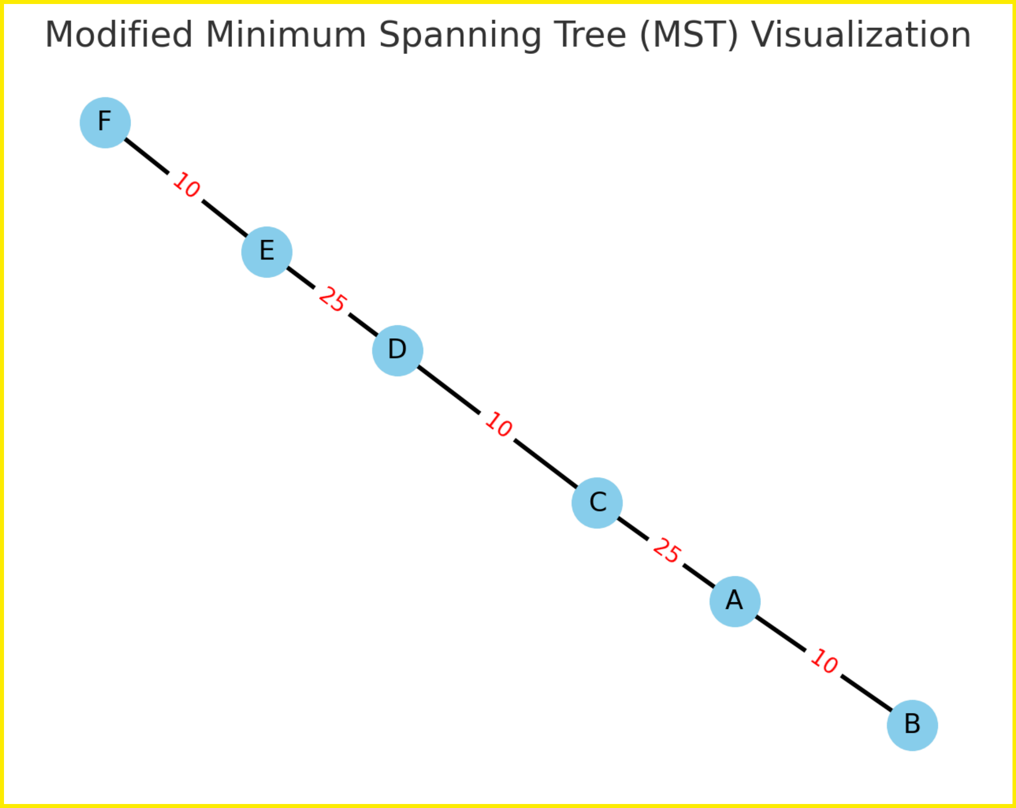
Consider automated transport (e.g., AGVs) between **E (Packaging)** and **F (Storage)**.

- Parallel machines:

Add a secondary welding or painting station to handle peak loads.

Why Kruskal's algorithm?

- Our goal is to minimize total distance, and adjacency requirements are less critical in our EV battery assembly layout.
- Kruskal's algorithm was chosen for its simplicity in solving the distance minimization problem. Modified MST or PAG can be applied if adjacency constraints become critical in future iterations.
- We can quickly switch to the Modified Spanning Tree algorithm by re-defining the weights in our primary code. For example, if "Welding" must be near "Painting," we increase the edge weight of non-adjacent connections as a penalty.
- When penalty is applied, distances become:



4.4 Appendix

Creating a from-to matrix:

```python
import pandas as pd

# Define the departments
departments = ["A", "B", "C", "D", "E", "F"]

# Define the distances between departments
distances = {
    ("A", "B"): 10,
    ("A", "C"): 15,
    ("A", "D"): 20,
    ("A", "E"): 25,
    ("A", "F"): 30,
    ("B", "C"): 35,
    ("B", "D"): 25,
    ("B", "E"): 30,
    ("B", "F"): 40,
    ("C", "D"): 10,
    ("C", "E"): 20,
    ("C", "F"): 25,
    ("D", "E"): 15,
    ("D", "F"): 30,
    ("E", "F"): 10,
}

# Create an empty DataFrame for the from-to matrix
from_to_matrix = pd.DataFrame(index=departments, columns=departments)

# Fill the from-to matrix with distances
for (from_dept, to_dept), distance in distances.items():
    from_to_matrix.at[from_dept, to_dept] = distance
    from_to_matrix.at[to_dept, from_dept] = distance

# Fill diagonal with zeros (distance from a department to itself)
for dept in departments:
    from_to_matrix.at[dept, dept] = 0

# Display the from-to matrix
print("From-To Matrix:")
print(from_to_matrix)
```

Here's the Python code to solve this using Kruskal's algorithm:

```python
class Graph:
    def __init__(self, vertices):
        self.V = vertices
        self.graph = []

    def add_edge(self, u, v, w):
        self.graph.append([u, v, w])

    def find(self, parent, i):
        if parent[i] == i:
            return i
        return self.find(parent, parent[i])

    def union(self, parent, rank, x, y):
        root_x = self.find(parent, x)
        root_y = self.find(parent, y)
        if rank[root_x] < rank[root_y]:
            parent[root_x] = root_y
        elif rank[root_x] > rank[root_y]:
            parent[root_y] = root_x
        else:
            parent[root_y] = root_x
            rank[root_x] += 1

    def kruskal_mst(self):
        result = []
        i = 0
        e = 0
        self.graph = sorted(self.graph, key=lambda item: item[2])
        parent = []
        rank = []
        for node in range(self.V):
            parent.append(node)
            rank.append(0)
        while e < self.V - 1:
            u, v, w = self.graph[i]
            i = i + 1
            x = self.find(parent, u)
            y = self.find(parent, v)
            if x != y:
                e = e + 1
                result.append([u, v, w])
                self.union(parent, rank, x, y)
        return result


# Create a graph with 6 vertices
g = Graph(6)
g.add_edge(0, 1, 10)
g.add_edge(0, 2, 15)
g.add_edge(0, 3, 20)
g.add_edge(0, 4, 25)
g.add_edge(0, 5, 30)
g.add_edge(1, 2, 35)
g.add_edge(1, 3, 25)
g.add_edge(1, 4, 30)
g.add_edge(1, 5, 40)
g.add_edge(2, 3, 10)
g.add_edge(2, 4, 20)
g.add_edge(2, 5, 25)
g.add_edge(3, 4, 15)
g.add_edge(3, 5, 30)
g.add_edge(4, 5, 10)

# Find the MST
mst = g.kruskal_mst()

# Print the MST
print("Edges in the Minimum Spanning Tree:")
for u, v, weight in mst:
    print(f"{chr(u + 65)} - {chr(v + 65)}: {weight} meters")
```

Here's the Python code to calculate the efficiency:

```
# Calculate the total distance of the MST
```

```
total_mst_distance = sum(weight for _, _, weight in mst)

# Calculate the total possible distance
total_possible_distance = sum(
    [10, 15, 20, 25, 30, 35, 25, 30, 40, 10, 20, 25, 15, 30,
10]
)

# Calculate the efficiency
efficiency = (1 - (total_mst_distance /
total_possible_distance)) * 100

print(f"Total Distance of MST: {total_mst_distance} meters")
print(f"Total Possible Distance: {total_possible_distance}
meters")
print(f"Efficiency of the Layout: {efficiency:.2f}%")
```