İhsan Doğramacı Bilkent University

IE 400: Principles of Engineering Management

Term Project Report

Group 11

Ali Aral Takak, EEE, aral.takak@ug.bilkent.edu.tr

Jan Duman, EEE, jan.duman@ug.bilkent.edu.tr

Perit Dinçer, CS, perit.dincer@ug.bilkent.edu.tr

Introduction

The term project of this course addresses two optimization problems, each reflecting decision-making scenarios in logistics and resource allocation. The first problem involves designing a mathematical model in order to determine the optimal promotional tour path for a technology startup, namely Tan-Tech, who seeks to maximize its brand exposure across various cities, given its budget and logistical constraints. The problem is a modified version of the classic optimization problem, The Travelling Salesman. Our solution used the Mixed Integer Programming (MIP) formulation, where we have incorporated binary decision variables to represent city visits and travel routes with constraints on the budget, the number of cities visited, and the continuity of the tour. The further explanations can be observed in the part **Question I.**

The second problem tackles developing a cost-effective nutritional plan for two hikers, Ece and Arda, who require distinct caloric intakes and have limited backpack capacities. We have developed a binary integer programming problem, where each decision variable represents the inclusion of a particular food item, as provided in the question manual. The objective is to minimize the total cost of selected items while meeting individual calorie and weight constraints. The further details can be observed in part **Question II.**

In our implementation of the solutions, we have used our skills in mathematical modeling, and then used computer-aided solution tools such as Python as Gurobi in order to determine the optimal results.

Question I

A tech startup, Tan-Tech, is planning a promotional tour to launch their latest product across multiple cities. The goal is to maximize exposure and attract both potential investors and customers, but their budget and time are limited. Each city has a different level of market potential, represented as benefit points. However, visiting every city is not feasible due to budget and time constraints. To ensure the tour is impactful, Tan-Tech's objectives and constraints are as follows:

- The total travel cost for the tour cannot exceed $1,500.
- Due to logistical and team limitations, Tan-Tech can visit a maximum of 7 cities in total.
- The tour must start and end in city A.

The travel costs from city-to-city can be observed in the figure provided below:

	A	B	C	D	E	F	G	H	I	J
A	0	300	450	150	225	330	495	600	315	405
B	300	0	180	330	315	375	510	195	420	240
C	450	180	0	210	495	135	240	435	180	345
D	150	330	210	0	360	195	300	405	270	285
E	225	315	495	360	0	240	465	180	285	225
F	330	375	135	195	240	0	270	300	345	390
G	495	510	240	300	465	270	0	135	255	210
H	600	195	435	405	180	300	135	0	285	315
I	315	420	180	270	285	345	255	285	0	150
J	405	240	345	285	225	390	210	315	150	0

Figure 1: *The travel costs across cities.*

The benefit points for each city can be observed in the figure provided below:

City	Benefit Points
A	350
B	420
C	270
D	300
E	380
F	410
G	320
H	450
I	330
J	400

Figure 2: *The benefit points for potential cities.*

Hence, we can construct our model as follows:

Decision Variables:

- A binary variable indicating whether the route from city i to city j is selected can be defined as:

$$x[i,j] = \begin{cases} 1, & if\ the\ route\ is\ selected \\ 0, & otherwise \end{cases}$$

- A binary variable indicating whether city i is visited can be defined as:

$$y[i] = \begin{cases} 1, & if\ the\ city\ is\ visited, \\ 0, & otherwise \end{cases}$$

Objective Function: The objective is to maximize the total benefit points obtained from visiting selected cities, and can be mathematically formulated as:

$$maximize\ Z = \sum_{i=1}^{n} (Benefit\ Points[i] \times y[i])$$

Constraints: There exist few constraints, which can be stated as:

1. **Budget Constraint:** The total travel cost must not exceed $1,500.

$$\sum_{i=1}^{n} \sum_{j=1}^{n} (Travel\ Cost[i][j] \times x[i,j]) \leq 1500$$

2. **City Visit Limit:** The maximum number of cities visited, including city A, is 7.

$$\sum_{i=1}^{n} y[i] \leq 7$$

3. **Start and End in City A:** The trip must start and end in city A.

$$\sum_{j=1}^{n} x[0,j] = 1\,, \sum_{i=1}^{n} x[i,0] = 1$$

4. **Flow Constraints:** We must ensure that each city visited is entered and exited exactly once.

$$\sum_{j=1, j\neq i}^{n} x[i,j] = y[i], \qquad \sum_{j=1, j\neq i}^{n} x[i,j] = y[i]\ \forall\, i,j\, \epsilon\, \{1,\dots,n\}, i \neq j$$

5. **Subtour Elimination:** In order to eliminate subtours, we have used the Miller-Tucker-Zemlin (MTZ) formulation.

$$u[i] - u[j] + n \times x[i,j] \leq n-1, \forall\, i,j\, \epsilon\, \{1,\dots,n\}, i \neq j$$

Results

The optimization model successfully determined the optimal tour for Tan-Tech's promotional campaign, achieving a maximum benefit of **2650 points** by strategically selecting high-value cities while adhering to logistical and financial constraints. The optimal route begins and ends in **City A**, as required, and traverses the cities in the following order: **A → E → I → J → G → H → B → A**, encompassing a total of seven cities, the maximum allowed under the constraints. The model ensures that the total travel cost remains within the budget limit of **$1,500**, confirming the feasibility of the solution. Computationally, the model demonstrated efficiency, exploring **51 nodes** and performing **365 simplex iterations** within **0.05 seconds**. The optimality gap of **0.0000%** guarantees that the solution is globally optimal. Furthermore, the presolve phase effectively simplified the problem by reducing redundant rows and columns, enhancing solution accuracy and speed. The figures given below display the results.

```
Optimize a model with 94 rows, 120 columns and 514 nonzeros
Model fingerprint: 0x920d22f0
Variable types: 0 continuous, 120 integer (110 binary)
Coefficient statistics:
  Matrix range     [1e+00, 6e+02]
  Objective range  [3e+02, 4e+02]
  Bounds range     [1e+00, 1e+00]
  RHS range        [1e+00, 2e+03]
Found heuristic solution: objective 1070.0000000
Presolve removed 0 rows and 11 columns
Presolve time: 0.00s
Presolved: 94 rows, 109 columns, 512 nonzeros
Variable types: 0 continuous, 109 integer (100 binary)

Root relaxation: objective 2.739333e+03, 41 iterations, 0.00 seconds (0.00 work units)
```

Figure 3: *Model range and presolve statistics.*

```
    Nodes    |    Current Node    |     Objective Bounds      |     Work
 Expl Unexpl |  Obj  Depth IntInf | Incumbent    BestBd   Gap | It/Node Time

     0     0 2739.33333    0   20 1070.00000 2739.33333   156%     -    0s
H    0     0                      1140.0000000 2739.33333   140%     -    0s
H    0     0                      1840.0000000 2739.33333  48.9%     -    0s
H    0     0                      1960.0000000 2710.90909  38.3%     -    0s
     0     0 2710.87500    0   22 1960.00000 2710.87500  38.3%     -    0s
H    0     0                      2170.0000000 2710.00000  24.9%     -    0s
H    0     0                      2280.0000000 2710.00000  18.9%     -    0s
     0     0 2710.00000    0   18 2280.00000 2710.00000  18.9%     -    0s
H    0     0                      2530.0000000 2710.00000  7.11%     -    0s
     0     0 2703.33333    0   10 2530.00000 2703.33333  6.85%     -    0s
     0     0 2694.04082    0   11 2530.00000 2694.04082  6.48%     -    0s
     0     0 2694.04082    0   10 2530.00000 2694.04082  6.48%     -    0s
     0     2 2694.04082    0   10 2530.00000 2694.04082  6.48%     -    0s
H   15    21                      2650.0000000 2694.04082  1.66%  10.6    0s

Cutting planes:
  Learned: 3
  Gomory: 9
  Implied bound: 11
  Clique: 2
  MIR: 2
  Zero half: 3

Explored 51 nodes (365 simplex iterations) in 0.05 seconds (0.03 work units)
Thread count was 8 (of 8 available processors)
```

Figure 4: *Node statistics.*

```
Solution count 8: 2650 2530 2280 ... 1070

Optimal solution found (tolerance 1.00e-04)
Best objective 2.650000000000e+03, best bound 2.650000000000e+03, gap 0.0000%
Optimal route found:
Travel from A to E
Travel from B to A
Travel from E to I
Travel from G to H
Travel from H to B
Travel from I to J
Travel from J to G
Total Benefit Points: 2650.0
```

Figure 5: *Optimal solution results.*

The figure given below displays the followed route for visits.

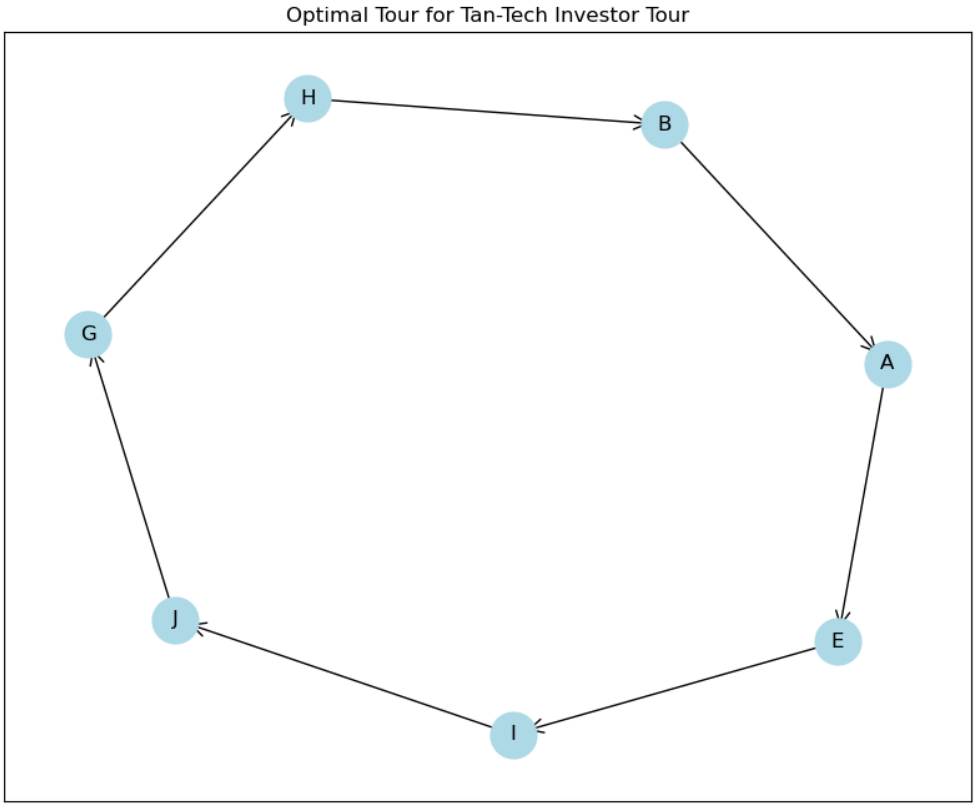

Figure 6: *Optimal visit path.*

The figure given below displays the benefit points obtained at each city.

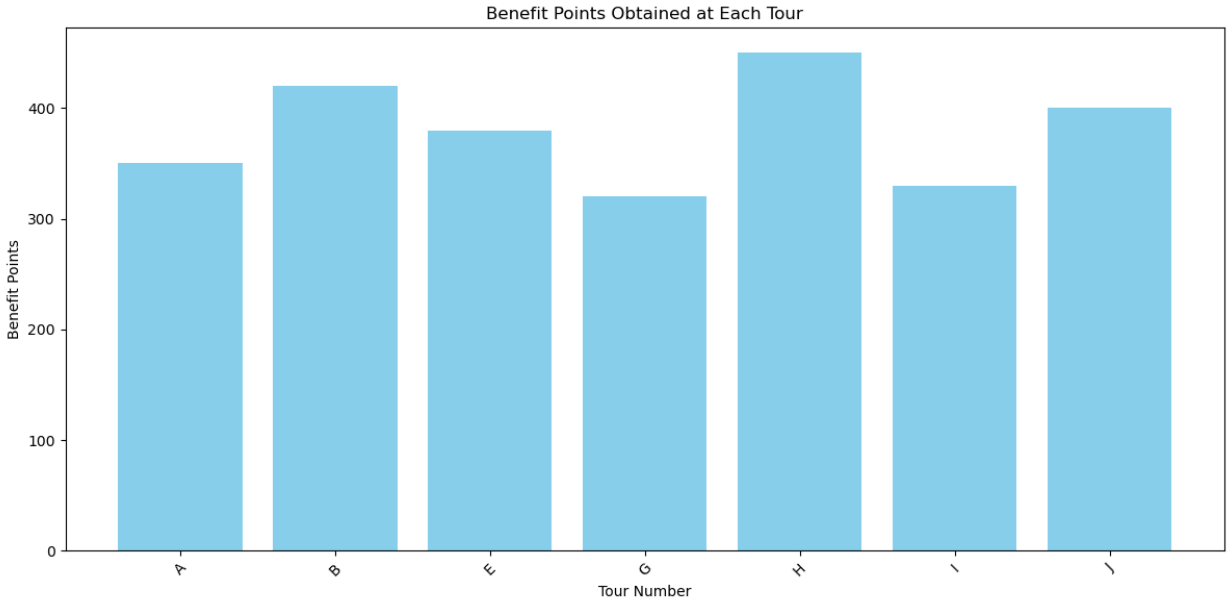

Figure 7: *Benefit point chart for each visit.*

Question II

Ece and Arda are working together to prepare a balanced meal plan for a one-day hiking trip. They have limited space in their bags, and each can carry up to a specific amount of weight. Additionally, Ece and Arda have different preferences for calorie intake. They can share the items in their backpacks once they reach their destination. They aim to select food items from a list so that they have enough food to reach their total calorie goal while considering the capacity requirements and minimizing the combined cost of all selected items.

- Ece's backpack can carry up to 10 kilograms of food, while Arda's can carry up to 8 kilograms.
- Ece wants to consume at least 2000 calories per day, and Arda requires at least 1500 calories per day.

Item	Weight (kg)	Calories	Cost ($)
Granola Bars	2	300	5
Trail Mix	1	800	10
Dried Fruit	2	200	4
Canned Beans	6	800	7
Rice	4	1100	8
Energy Drink	6	150	3
Pasta	5	1200	9
Jerky	2	500	6

Figure 8: *Weight, calorie, and cost chart for the problem.*

As the day of the hike approaches, Ece and Arda gather all the food items they have chosen and start to pack their backpacks. However, as they look at their options, they suddenly realize something important: They only have one item from each type of food. Formulate this as a binary integer programming problem where the decision variable x_i represents whether each item i is included in the meal plan or not.

Decision Variables

Since the implementation for the solution of the second question involves two different approaches, there exists various decision variables.

1. For the Gurobi MIP Model, x_i is the binary decision variable that indicates if item i is selected or not:

   - $x_i = \begin{cases} 1, if\ the\ item\ is\ selected \\ 0, otherwise \end{cases}$

2. y_i indicates if item i is carried by whether Ece or Arda.

   - $y_i = \begin{cases} 1, if\ carried\ by\ Ece \\ 0, if\ carried\ by\ Arda \end{cases}$

3. For the custom implementation of the Branch and Bound Method, each item i is assigned a state as follows:

   - $s_i = 0$, if the item i is not chosen.
   - $s_i = 1$, if the item i is chosen by Ece.
   - $s_i = 2$, if the item i is chosen by Arda.

Objective Function

In this problem, the objective is to minimize the combined cost of all the selected items, which can be mathematically represented as follows:

$$Minimize \sum_i cost_i \times x_i$$

Constraints

For the proper formulation of the given problem, one must include the constraints given below into the mathematical formulation:

1. **Weight Constraints:**
    - The maximum weight that Ece can carry is 10 kilograms:

$$\sum_i weight_i \times x_i \leq 10$$

    - The maximum weight that Arda can carry is 8 kilograms:

$$\sum_i weight_i \times x_i \leq 8$$

2. **Calorie Constraints:**
    - Ece wants to consume at least 2000 calories per day:

$$\sum_i calories_i \times x_i \geq 2000$$

    - Arda requires at least 1500 calories per day:

$$\sum_i calories_i \times x_i \geq 1500$$

3. **Stock Constraint:** There exists a single piece from each item.

$$x_i \in \{0,1\}$$

Results

The implemented code compares two optimization methods—Branch & Bound (B&B) and Gurobi, a mathematical optimization solver—for solving a constrained knapsack problem. The primary goal of this problem is to minimize the cost of selected items while meeting specific calorie and weight constraints. The constraints include satisfying a minimum calorie requirement of 3500 and ensuring that the combined weight of the items does not exceed the carrying capacities of two carriers: Ece (10 kg) and Arda (8 kg). By solving this problem, we aimed to evaluate the performance and efficiency of a custom algorithm compared to an optimization solver.

For the Branch & Bound approach, we have implemented a recursive algorithm that explores all possible combinations of item assignments to either Ece, Arda, or neither. The algorithm uses pruning techniques to eliminate branches of the search tree that cannot yield optimal solutions. In contrast, the Gurobi solver formulates the problem as a mixed-integer linear program. Using binary decision variables, the solver efficiently determines the optimal selection and assignment of items.

Both methods produced identical results in terms of cost, ensuring that the calorie and weight constraints were met while minimizing the total cost. This outcome validates the accuracy of the Branch & Bound implementation when compared to the Gurobi solver. The figure given below display the obtained results.

```
B&B: Found best cost = 30
  Ece carries: ['Canned Beans', 'Rice'] (Weight=10 kg)
  Arda carries: ['Pasta', 'Jerky'] (Weight=7 kg)
  Combined calories = 3600 (≥ 3500)
B&B runtime = 0.000405 seconds

Restricted license - for non-production use only - expires 2026-11-23
Gurobi: Found best cost = 30.0
  Ece carries: ['Canned Beans', 'Rice'] (Weight=10 kg)
  Arda carries: ['Pasta', 'Jerky'] (Weight=7 kg)
  Combined calories = 3600 (≥ 3500)
Gurobi runtime = 0.010872 seconds

=== Comparison Summary ===
 B&B cost = 30, runtime = 0.000405 s
 Gurobi cost = 30.0, runtime = 0.010872 s
Note: Both should ideally yield the same minimal cost (if optimal).
```

Figure 9: *Obtained results and comparison of two implementations.*

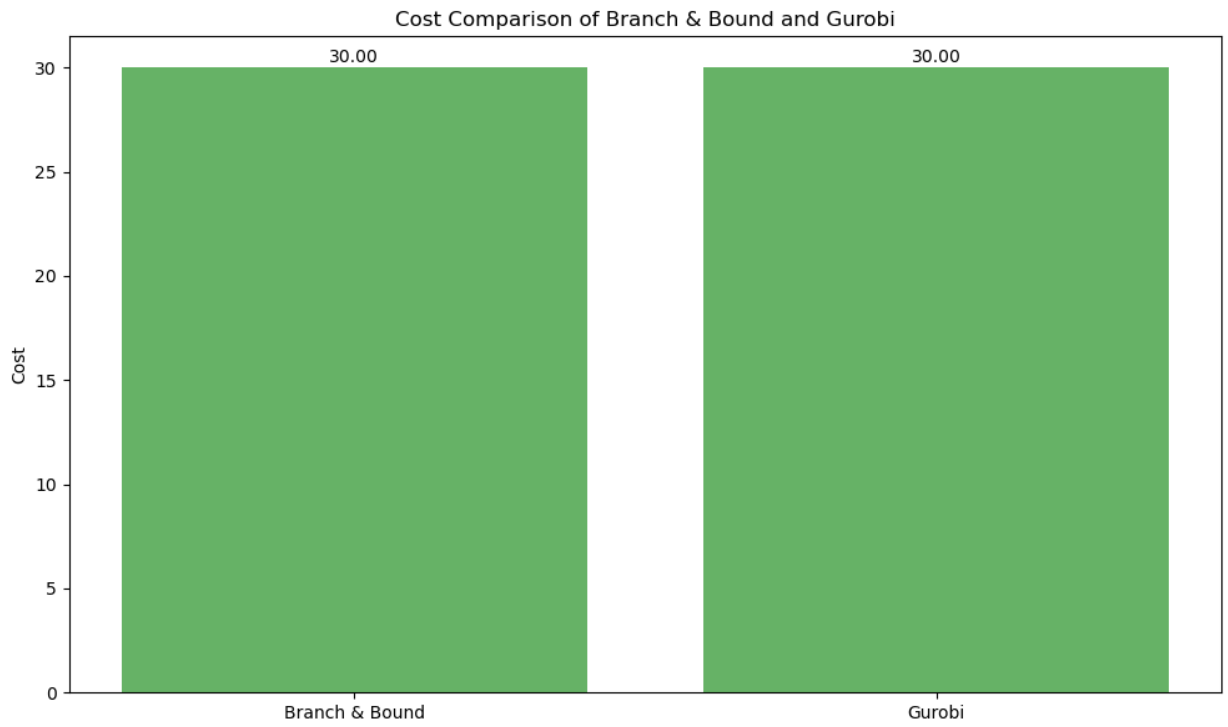

Figure 10: *Cost comparison of the two implementations.*

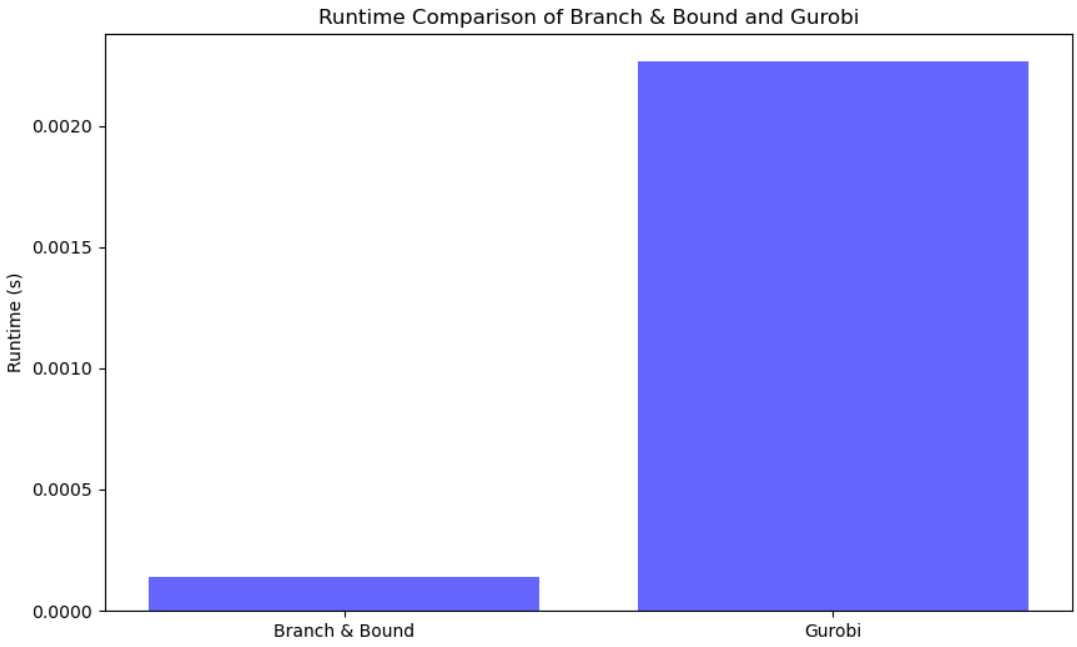

Figure 11: *Runtime comparison of the two implementations.*

In conclusion, our analysis demonstrated that both Branch and Bound algorithm and the optimization library provided the identical results, whereas the Branch and Bound algorithm has yielded better runtime performance.

Appendices

Appendix A: Python Script for the First Problem

```
# Import the required libraries.
from gurobipy import Model, GRB, quicksum
import matplotlib.pyplot as plt
import networkx as nx

# Define the labels for cities.
cities = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J']
numCities = len(cities)

# Define the cost matrix for travel between cities.
travelCost = [
    [0, 300, 450, 150, 225, 330, 495, 600, 315, 405],
    [300, 0, 180, 330, 315, 375, 510, 195, 420, 240],
    [450, 180, 0, 210, 495, 135, 240, 435, 180, 345],
    [150, 330, 210, 0, 360, 195, 300, 405, 270, 285],
    [225, 315, 495, 360, 0, 240, 465, 180, 285, 225],
    [330, 375, 135, 195, 240, 0, 270, 300, 345, 390],
    [495, 510, 240, 300, 465, 270, 0, 135, 255, 210],
    [600, 195, 435, 405, 180, 300, 135, 0, 285, 315],
    [315, 420, 180, 270, 285, 345, 255, 285, 0, 150],
    [405, 240, 345, 285, 225, 390, 210, 315, 150, 0]
]

# Define an array for benefit points.
benefitPoints = [350, 420, 270, 300, 380, 410, 320, 450, 330, 400]

# Create the model.
m = Model("Tan-Tech Investor Tour")

# Define the binary variables for travel routes.
x = m.addVars(numCities, numCities, vtype=GRB.BINARY, name="x")
y = m.addVars(numCities, vtype=GRB.BINARY, name="y")

# Define the objective function, which is to maximize the benefit points.
```

```
m.setObjective(quicksum(benefitPoints[i] * y[i] for i in range(numCities)), GRB.MAXIMIZE)

# Define the constraints of the problem.

# Constraint 1: The total travel cost cannot exceed 1500 USD.
m.addConstr(quicksum(travelCost[i][j] * x[i, j] for i in range(numCities) for j in range(numCities)) <= 1500, "Budget")

# Constraint 2: Due to logistical and team limitations, Tan-Tech can visit a maximum of 7 cities in total.
m.addConstr(quicksum(y[i] for i in range(numCities)) <= 7, "MaxCities")

# Constraint 3: The tour must start and end in City A.
m.addConstr(quicksum(x[0, j] for j in range(1, numCities)) == 1, "StartAtA")
m.addConstr(quicksum(x[i, 0] for i in range(1, numCities)) == 1, "EndAtA")

# Constraint 4: Flow constraints to ensure each visited city is entered and exited exactly once.
for i in range(1, numCities):
    m.addConstr(quicksum(x[i, j] for j in range(numCities) if j != i) == y[i], f"VisitOut_{i}")
    m.addConstr(quicksum(x[j, i] for j in range(numCities) if j != i) == y[i], f"VisitIn_{i}")

# Constraint 5: Subtour elimination (MTZ formulation).
u = m.addVars(numCities, vtype=GRB.INTEGER, name="u")
for i in range(1, numCities):
    for j in range(1, numCities):
        if i != j:
            m.addConstr(u[i] - u[j] + numCities * x[i, j] <= numCities - 1)

# Solve the model.
m.optimize()

# Output the results and prepare for plotting.
optimal_routes = []
visited_cities = set()
tour_benefits = []
tour_number = 1
if m.status == GRB.OPTIMAL:
    print("Optimal route found:")
    for i in range(numCities):
```

```
        for j in range(numCities):
            if x[i, j].x > 0.5:
                print(f"Travel from {cities[i]} to {cities[j]}")
                optimal_routes.append((cities[i], cities[j]))
                visited_cities.add(cities[i])
                visited_cities.add(cities[j])
                tour_benefits.append((tour_number, cities[i], benefitPoints[cities.index(cities[i])]))
                tour_number += 1
    print(f"Total Benefit Points: {m.objVal}")
else:
    print("No feasible solution found.")

# Plotting the optimal route.
if optimal_routes:
    G = nx.DiGraph()
    G.add_edges_from(optimal_routes)

    pos = nx.spring_layout(G)
    plt.figure(figsize=(10, 8))
    nx.draw_networkx_nodes(G, pos, nodelist=list(visited_cities), node_size=700, node_color='lightblue')
    nx.draw_networkx_edges(G, pos, edgelist=optimal_routes, arrowstyle='->', arrowsize=20, edge_color='black')
    nx.draw_networkx_labels(G, pos, labels={city: city for city in visited_cities}, font_size=12, font_color='black')
    plt.title('Optimal Tour for Tan-Tech Investor Tour')
    plt.show()

# Plotting the benefit points for each tour.
if tour_benefits:
    tour_numbers, cities_visited, benefits = zip(*tour_benefits)
    plt.figure(figsize=(12, 6))
    plt.bar(tour_numbers, benefits, color='skyblue')
    plt.xlabel('Tour Number')
    plt.ylabel('Benefit Points')
    plt.xticks(tour_numbers, cities_visited, rotation=45)
    plt.title('Benefit Points Obtained at Each Tour')
    plt.tight_layout()
    plt.show()
```

Appendix B: Python Script for the Second Problem

```
import time
import gurobipy as gp
from gurobipy import GRB

items_data = [
    (2, 300, 5, "Granola Bars"),
    (1, 800, 10, "Trail Mix"),
    (2, 200, 4, "Dried Fruit"),
    (6, 800, 7, "Canned Beans"),
    (4, 1100, 8, "Rice"),
    (6, 150, 3, "Energy Drink"),
    (5, 1200, 9, "Pasta"),
    (2, 500, 6, "Jerky")
]

REQUIRED_CALORIES = 3500
CAPACITY_ECE = 10
CAPACITY_ARDA = 8

best_cost_BNB = float('inf')
best_solution_BNB = None

max_cal_suffix = [0] * len(items_data)
running_sum = 0
for i in reversed(range(len(items_data))):
    running_sum += items_data[i][1]
    max_cal_suffix[i] = running_sum


def branch_and_bound_2subset_knapsack(
        index,
        ece_weight,
        arda_weight,
        total_calories,
        total_cost,
```

```
    selected_items
):
    global best_cost_BNB, best_solution_BNB

    # Prune if capacity exceeded
    if ece_weight > CAPACITY_ECE or arda_weight > CAPACITY_ARDA:
        return

    # Prune if cost already worse than best
    if total_cost >= best_cost_BNB:
        return

    if index == len(items_data):
        # Reached a leaf node; check calorie requirement
        if total_calories >= REQUIRED_CALORIES:
            if total_cost < best_cost_BNB:
                best_cost_BNB = total_cost
                best_solution_BNB = selected_items[:]
        return

    # Bound: if even taking all remaining items can't hit 3500 calories, prune
    potential_max_cal = total_calories + max_cal_suffix[index]
    if potential_max_cal < REQUIRED_CALORIES:
        return

    w_i, cal_i, cost_i, name_i = items_data[index]

    # BRANCH 1: NOT CHOSEN
    selected_items.append(0)
    branch_and_bound_2subset_knapsack(
        index + 1,
        ece_weight,
        arda_weight,
        total_calories,
        total_cost,
        selected_items
    )
```

```
    selected_items.pop()

    # BRANCH 2: CHOSEN BY ECE
    selected_items.append(1)
    branch_and_bound_2subset_knapsack(
        index + 1,
        ece_weight + w_i,
        arda_weight,
        total_calories + cal_i,
        total_cost + cost_i,
        selected_items
    )
    selected_items.pop()

    # BRANCH 3: CHOSEN BY ARDA
    selected_items.append(2)
    branch_and_bound_2subset_knapsack(
        index + 1,
        ece_weight,
        arda_weight + w_i,
        total_calories + cal_i,
        total_cost + cost_i,
        selected_items
    )
    selected_items.pop()


def solve_custom_BNB():
    global best_cost_BNB, best_solution_BNB
    best_cost_BNB = float('inf')
    best_solution_BNB = None

    branch_and_bound_2subset_knapsack(
        index=0,
        ece_weight=0,
        arda_weight=0,
        total_calories=0,
```

```
        total_cost=0,
        selected_items=[]
    )
    return best_cost_BNB, best_solution_BNB


def print_BNB_solution(cost, solution):
    if solution is None:
        print("No feasible B&B solution found.")
        return
    print(f"B&B: Found best cost = {cost}")
    ece_list = []
    arda_list = []
    total_cals = 0
    total_w_ece = 0
    total_w_arda = 0

    for i, assign in enumerate(solution):
        w_i, cal_i, cost_i, name_i = items_data[i]
        if assign == 1:
            ece_list.append(name_i)
            total_w_ece += w_i
            total_cals += cal_i
        elif assign == 2:
            arda_list.append(name_i)
            total_w_arda += w_i
            total_cals += cal_i

    print(f"  Ece carries: {ece_list} (Weight={total_w_ece} kg)")
    print(f"  Arda carries: {arda_list} (Weight={total_w_arda} kg)")
    print(f"  Combined calories = {total_cals} (≥ {REQUIRED_CALORIES})")

def solve_with_gurobi():



    n = len(items_data)
```

```
w = [items_data[i][0] for i in range(n)]
cal = [items_data[i][1] for i in range(n)]
cost = [items_data[i][2] for i in range(n)]

m = gp.Model("MealPlan2SubsetKnapsack")

x = m.addVars(n, vtype=GRB.BINARY, name="x")
y = m.addVars(n, vtype=GRB.BINARY, name="y")

# Objective
m.setObjective(gp.quicksum(cost[i] * x[i] for i in range(n)), GRB.MINIMIZE)

# Constraint: total calories >= 3500
m.addConstr(gp.quicksum(cal[i] * x[i] for i in range(n)) >= REQUIRED_CALORIES, name="CalReq")

# Ece capacity
m.addConstr(gp.quicksum(w[i] * y[i] for i in range(n)) <= CAPACITY_ECE, name="EceCap")

# Arda capacity
m.addConstr(gp.quicksum(w[i] * (x[i] - y[i]) for i in range(n)) <= CAPACITY_ARDA, name="ArdaCap")

# Logical link: y[i] <= x[i]
for i in range(n):
    m.addConstr(y[i] <= x[i], name=f"logic_{i}")

# Solve
m.setParam("OutputFlag", 0)  # silent
m.optimize()

if m.status == GRB.OPTIMAL:
    xSol = [int(round(x[i].X)) for i in range(n)]
    ySol = [int(round(y[i].X)) for i in range(n)]
    return m.objVal, (xSol, ySol)
else:
    return None, (None, None)
```

```
def print_gurobi_solution(cost, xSol, ySol):

    if cost is None or xSol is None:
        print("No feasible Gurobi solution found.")
        return
    print(f"Gurobi: Found best cost = {cost}")
    ece_list = []
    arda_list = []
    total_cals = 0
    total_w_ece = 0
    total_w_arda = 0

    for i in range(len(items_data)):
        w_i, cal_i, cost_i, name_i = items_data[i]
        if xSol[i] == 1:
            # item is chosen, check who carries it
            if ySol[i] == 1:
                # carried by Ece
                ece_list.append(name_i)
                total_w_ece += w_i
                total_cals += cal_i
            else:
                # carried by Arda
                arda_list.append(name_i)
                total_w_arda += w_i
                total_cals += cal_i

    print(f"  Ece carries: {ece_list} (Weight={total_w_ece} kg)")
    print(f"  Arda carries: {arda_list} (Weight={total_w_arda} kg)")
    print(f"  Combined calories = {total_cals} (≥ {REQUIRED_CALORIES})")


def main():
    start_bnb = time.time()
    cost_bnb, sol_bnb = solve_custom_BNB()
    end_bnb = time.time()
```

```
    print_BNB_solution(cost_bnb, sol_bnb)
    bnb_time = end_bnb - start_bnb
    print(f"B&B runtime = {bnb_time:.6f} seconds\n")


    start_gurobi = time.time()
    cost_gurobi, (xSol, ySol) = solve_with_gurobi()
    end_gurobi = time.time()

    print_gurobi_solution(cost_gurobi, xSol, ySol)
    gurobi_time = end_gurobi - start_gurobi
    print(f"Gurobi runtime = {gurobi_time:.6f} seconds\n")


    print("=== Comparison Summary ===")
    print(f" B&B cost = {cost_bnb}, runtime = {bnb_time:.6f} s")
    print(f" Gurobi cost = {cost_gurobi}, runtime = {gurobi_time:.6f} s")
    print("Note: Both should ideally yield the same minimal cost (if optimal).")


if __name__ == "__main__":
    main()
```