UNIVERSITY OF EDINBURGH
Business School

2025 - 26

CMSE11431
PRESCRIPTIVE ANALYTICS WITH MATHEMATICAL PROGRAMMING

TITLE: Home Healthcare Routing & Scheduling Problem

EXAM/B NUMBER: B282896

Word count:1199

Answer 1

To arrive at the base model on which we are basing our group project, by removing the complicated layers of the real world of our final MILP formulation. I eliminated the soft constraints on the caregiver overtime, the compatibility of specific skills to match, and the penalty-based goal to enable the unattended patients. The other basic model is the Vehicle Routing Problem with Time Windows (VRPTW). The essential sets and decision variables form the core model that summarizes the logistical constraints of the business: the transportation of a fleet of caregivers to visit several geographically spread patients within time limits and at minimum cost per travel. The set of caregivers is denoted by K and the set of patients and the depot as N. We specify a binary decision variable, that is equal to 1 when caregiver k travels directly between node i and node j and a continuous variable, t_{ik} For node i and caregiver k, t_{ik}, meaning, the time when caregiver k arrives at node i. The general idea is to reduce operational inefficiency, and, in this case, it is the total travel cost:

$$\min Z = \sum_{k \epsilon K} \sum_{i \epsilon N} \sum_{j \epsilon N, j \neq i} c_{ij} x_{ijk}$$

To reflect the underlying logic, three types of hard constraints are necessary. To begin with, there is Demand and Flow Conservation which states that each patient should receive an exact number of visits and each caregiver that arrives at a certain place should leave it.

$$\sum_{k \epsilon K} \sum_{j \epsilon N} x_{ijk} = 1 \quad \forall i \, \epsilon \, patients$$

$$\sum_{j \epsilon N} x_{ijk} = \sum_{j \epsilon N} x_{jik} \quad \forall i \epsilon N, \forall k \epsilon K$$

Second, Time Connectivity is the logical consistency of the arrival times. The arrival time of a next patient j should take into consideration the arrival time of a patient i, service time s_i and travel time.

$$t_{ik} + s_i + d_{ij} - M(1 - x_{ijk}) \leq t_{jk} \, \forall k, \forall i, j$$

Lastly, Time Windows requires that the service should be punctual within the interval required by the patient.

$$e_i \leq t_{ik} \leq l_i$$

It is the key framework of the project it is not possible to mathematically accomplish the basic operational requirement of ensuring that all patients get attention during their assigned time slot without these elements.

Answer 2

Time-Geographic Connectivity is at the basis of the capability of the simplified base model. According to this condition, the only way that the solution is feasible is through the

possibility of a caregiver to travel between the designated locations without violating the continuum of time constraint. Mathematically, given any two successive visits of patient i to patient j, the inequality $t_{ik} + s_i + d_{ij}$ should be true and less than l_j. Since this base model is that the demand satisfaction and time windows are hard constraints, eliminating the available soft choices of overtime or missed visits of the final model, a single violation of this connectivity within the network makes the entire problem Infeasible.

To illustrate this rigidity, we were able to create a special instance of counterexample, namely, The Impossible Split. It is the case of a nurse ($N1$) and two patients ($P1$, $P2$) in need of service within a severely constrained 60 timeframes (08:00-09:00). Patient $P1$ is meant to take 30 minutes of service. The critical parameter that will cause the failure will be the geographic distance between the patients which will be 40 minutes of travel time.

Such an arrangement results in a mathematical paradox. Nurse $N1$ starts the duty at $P1$ as soon as possible (08:00) they will complete the duty at 08:30. Their latest possible arrival at $P2$ is 09:10 with the addition of the compulsory 4 minutes taken in traveling. This is a clear breach of the recent permissible start time of $P2$ which is 09:00 (09:10 ≠ 09:00). As the time (70 minutes) that is required is more than the time (60 minutes) that is available, the solver will give an Infeasible status.

Answer 3

To examine the Price of Integrity in our Home Health Care Base Model, we have built a particular toy example that is meant to put the capacity constraints to the test. We specified a case where there was 1 Nurse (shift capacity: 100 minutes) and 2 Patients (service requirement: 60 minutes each). Our penalty fee is in case of an unvisited patient is set at 1000 dollars. This poses a Bin Packing conflict, in which the total demand (120 mins) cannot be packed efficiently (100 mins) due to granularity of the tasks.

We solved the Relaxed (LP) instance, in which the operations are fluid ($0 \leq y \leq 1$). The solver used the total 100 minutes of the nurse: visiting Patient 1 completely and the remaining 40 minutes of the total time to visit 2/3 of patient 2. This left us with a patient to go unused (1/3) and the theoretical cost is 333.33 (1/3 x 1000).

Then we solved the Integer (IP) problem with fixed binary constraints ($y \in \{0,1\}$). The nurse remained with a capacity of 40 minutes after attending patient 1 (60 mins). Since this slack was not enough to accommodate an entire patient (40 less than 60), it was dead capacity. The nurse had to leave patient 2 entirely unattended, and the cost would be the difference between the formula and the result would be: In the relaxed model, the slack of 40 minutes was valued at $666.67 (2/3 of penalty was avoided). The same block of time in the integer model had a value of 0 since it could not fit an atomic unit of work. In this way, the loss in efficiency of the integer constraints in logistics due to the clunkiness of the integer constraints itself is mathematically measured by the gap.

Answer 4

Shadow Price Recognition and Decoding.

The shadow price, in the example of a toy with one nurse, 100 minutes capacity limit, and two patients, each requiring 60 minutes of care is based on the nurse-capacity constraint. The relaxed solution using linear program puts the unit value at one of the Gantries, at the relaxed solution, at one of the Gantries at the price of 16.67 per minute.

Theoretical Interpretation

Shadow price is the marginal improvement in the objective function which can be achieved by loosening a constraint by one unit. The fact that the value of time is 16.67 shows that time constraint is binding, the more time used in the shift, the lower the objective cost, which proves that time is a limited resource and it has high marginal utility.

Justification of the Value

This value is based on the linear relationship between penalty avoidance and duration of service. Given that a 60-minute stay will not incur a fine of a thousand dollars, every minute of service costs 1000/60 = 16.67. Using the fluid model, the unattended fraction of a patient decreases by 1/60 when one minute is added, which results in a total of savings of precisely $16.67.

Practical Application

This shadow price is practically used as a break-even point on overtime. It indicates that when a 100-minute limit is strictly enforced the opportunity cost will be 16.67 per minute. When the overtime rate (e.g., half of this standard e.g. 2.00/min) is below this point, the No Overtime policy is economically inefficient.

Improvement Strategy

The shadow price justifies the extension of the shift. Although the linear estimate would be adding 20 minutes would save approximately 333 (20 times 16.67), in the integer reality adding the dead-capacity extension would enable the second visit to be made and the entire 1,000 penalty to be saved.

APPENDIX

Question 2

```
import pulp

def solve_counter_example():

    # 1. Initializing Model

    # here I use LpMinimize, though the objective doesn't matter for feasibility

    prob = pulp.LpProblem("Infeasible_Counter_Example", pulp.LpMinimize)
```

```
# 2. Sets and Parameters
NURSES = ['N1']
PATIENTS = ['P1', 'P2']
NODES = ['0'] + PATIENTS  # '0' is the Depot

# Time Windows (in minutes, e.g., 0 to 60)
# Both patients have the exact same tight window
e = {'P1': 0,  'P2': 0}   # Earliest Start
l = {'P1': 60, 'P2': 60}  # Latest Start
dur = 30                  # Service Duration

# Travel Times
# Distance is 40 mins between patients
dist = {
   ('P1', 'P2'): 40, ('P2', 'P1'): 40,
   ('0', 'P1'): 10,  ('P1', '0'): 10,
   ('0', 'P2'): 10,  ('P2', '0'): 10
}

BIG_M = 1000 # Large constant for linearization

# 3. Decision Variables
# x[k,i,j] = 1 if nurse k travels from i to j
x = pulp.LpVariable.dicts("x",
   ((k, i, j) for k in NURSES for i in NODES for j in NODES if i != j),
   cat='Binary')
```

```
# t[k,i] = Arrival time of nurse k at node i
t = pulp.LpVariable.dicts("t",
    ((k, i) for k in NURSES for i in NODES),
    lowBound=0, upBound=100) # Bounds help solver check ranges


# 4. Objective Function (Minimize Travel)
prob += pulp.lpSum(dist.get((i,j), 0) * x[k,i,j]
                for k in NURSES for i in NODES for j in NODES if i != j)
# Constraint A: All Patients Must Be Visited (Hard Demand)
for p in PATIENTS:
    prob += pulp.lpSum(x[k,i,p] for k in NURSES for i in NODES if i != p) == 1
# Constraint B: Flow Conservation (Arrive node -> Leave node)
for k in NURSES:
    for p in PATIENTS:
        prob += pulp.lpSum(x[k,i,p] for i in NODES if i != p) == \
                pulp.lpSum(x[k,p,j] for j in NODES if j != p)


# Constraint C: Time Connectivity (The specific constraint that will FAIL)
# Arrival at j >= Arrival at i + Service at i + Travel(i,j)
for k in NURSES:
    for i in NODES:
        for j in NODES:
            if i == j: continue


            # Service time is 0 for depot, 'dur' for patients
            s_i = 0 if i == '0' else dur
```

```
            # The inequality
            prob += t[k,j] >= t[k,i] + s_i + dist.get((i,j), 0) - BIG_M * (1 - x[k,i,j])

    # Constraint D: Time Windows
    for k in NURSES:
        for p in PATIENTS:
            prob += t[k,p] >= e[p]
            prob += t[k,p] <= l[p]

    # 6. Solve
    print("--- STARTING SOLVER FOR COUNTER-EXAMPLE ---")
    # msg=0 hides the internal solver logs so we just see the result
    prob.solve(pulp.PULP_CBC_CMD(msg=0))

    # 7. Output Result
    status = pulp.LpStatus[prob.status]
    print(f"Model Status: {status}")

    if status == 'Infeasible':
        print("\nSUCCESS: The model successfully demonstrated INFEASIBILITY.")
        print("Reason: Service (30m) + Travel (40m) = 70m, which exceeds the Time Window (60m).")
    else:
        print(f"Unexpected Feasible Solution Found with Cost: {pulp.value(prob.objective)}")

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

OUTPUT

```
--- STARTING SOLVER FOR COUNTER-EXAMPLE ---
Model Status: Infeasible

SUCCESS: The model successfully demonstrated INFEASIBILITY.
Reason: Service (30m) + Travel (40m) = 70m, which exceeds the Time Window (60m).
```

Figure 1: Output for question 2

Question 3

```
import pandas as pd

import pulp

import numpy as np

import os

import shutil

import warnings

warnings.filterwarnings("ignore")

DATA_DIR = 'dataset_gap_analysis'

NUM_NURSES = 1

NUM_PATIENTS = 2


# Costs

COST_UNATTENDED = 1000.0  # Penalty when they miss a patient

COST_OVERTIME = 10000.0   # Strict limit (Higher cost prevents overtime)


# Time Parameters

SHIFT_DURATION = 100      # Nurse has 100 minutes

PATIENT_DURATION = 60     # Each patient needs 60 minutes

def generate_toy_data():
```

```
    if os.path.exists(DATA_DIR): shutil.rmtree(DATA_DIR)
    os.makedirs(DATA_DIR, exist_ok=True)

    print(f"Generating Instance: 1 Nurse (Cap {SHIFT_DURATION}m), 2 Patients (Req
{PATIENT_DURATION}m)...")

    # 1. Caregivers (1 Nurse)
    cg = [{'caregiver_id': 'N1', 'skill': 'A'}]
    pd.DataFrame(cg).to_csv(f"{DATA_DIR}/caregivers.csv", index=False)

    # 2. Patients (2 Patients)
    pats = []
    for i in range(NUM_PATIENTS):
        pats.append({
            'patient_id': f'P{i+1}',
            'skill_required': 'A',
            'service_duration': PATIENT_DURATION,
            'x': 0, 'y': 0
        })
    pd.DataFrame(pats).to_csv(f"{DATA_DIR}/patients.csv", index=False)

    # 3. Nodes
    nodes = [{'node_id': '0', 'type': 'depot', 'x': 0, 'y': 0}] + \
            [{'node_id': p['patient_id'], 'type': 'patient', 'x': 0, 'y': 0} for p in pats]
    pd.DataFrame(nodes).to_csv(f"{DATA_DIR}/nodes.csv", index=False)

    # 4. Travel Times (Zero to focus on capacity)
    tt = []
```

```
    node_ids = [n['node_id'] for n in nodes]
    for n1 in node_ids:
        for n2 in node_ids:
            tt.append({'from_node': n1, 'to_node': n2, 'travel_time': 0})
    pd.DataFrame(tt).to_csv(f"{DATA_DIR}/travel_times.csv", index=False)

    # 5. Compatibility
    compat = [{'caregiver_id': c['caregiver_id'], 'patient_id': p['patient_id'], 'compat': 1}
              for c in cg for p in pats]
    pd.DataFrame(compat).to_csv(f"{DATA_DIR}/compatibility.csv", index=False)

    # 6. Shift
    cg_day = [{'caregiver_id': 'N1', 'day': 1, 'alpha': 0, 'beta': SHIFT_DURATION}]
    pd.DataFrame(cg_day).to_csv(f"{DATA_DIR}/caregiver_days.csv", index=False)

    # 7. Patient Requirements
    pt_day = []
    for p in pats:
        pt_day.append({'patient_id': p['patient_id'], 'day': 1, 'required': 1, 'dur':
PATIENT_DURATION})
    pd.DataFrame(pt_day).to_csv(f"{DATA_DIR}/patient_days.csv", index=False)


# 2. SOLVER FUNCTION
def solve_instance(relaxed=False):
    # Load Data
    nodes_df = pd.read_csv(f"{DATA_DIR}/nodes.csv").set_index('node_id')
    caregivers_df = pd.read_csv(f"{DATA_DIR}/caregivers.csv").set_index('caregiver_id')
    patients_df = pd.read_csv(f"{DATA_DIR}/patients.csv").set_index('patient_id')
```

```
    tt_df = pd.read_csv(f"{DATA_DIR}/travel_times.csv").set_index(['from_node',
'to_node'])

    pt_day_df = pd.read_csv(f"{DATA_DIR}/patient_days.csv").set_index(['patient_id', 'day'])

    cg_day_df = pd.read_csv(f"{DATA_DIR}/caregiver_days.csv").set_index(['caregiver_id',
'day'])


    K = list(caregivers_df.index)

    P = list(patients_df.index)

    N = ['0'] + P

    D = [1]


    # Parameters

    dur = pt_day_df['dur'].to_dict()

    start = cg_day_df['alpha'].to_dict()

    end = cg_day_df['beta'].to_dict()


    # Model

    prob = pulp.LpProblem("Gap_Analysis", pulp.LpMinimize)

    cat_type = pulp.LpContinuous if relaxed else pulp.LpBinary


    # Variables

    x = pulp.LpVariable.dicts("x", ((k, d, i, j) for k in K for d in D for i in N for j in N if i != j),

                              lowBound=0, upBound=1, cat=cat_type)

    y = pulp.LpVariable.dicts("y", ((k, d, p) for k in K for d in D for p in P),

                              lowBound=0, upBound=1, cat=cat_type)

    unattended = pulp.LpVariable.dicts("unattended", ((d, p) for d in D for p in P),

                                       lowBound=0, upBound=1, cat=cat_type)

    ot = pulp.LpVariable.dicts("ot", ((k, d) for k in K for d in D), lowBound=0)
```

```
# Objective
prob += (
   pulp.lpSum(COST_UNATTENDED * unattended[d,p] for d in D for p in P) +
   pulp.lpSum(COST_OVERTIME * ot[k,d] for k in K for d in D)
)

# Constraints
for d in D:
   for p in P:
      prob += pulp.lpSum(y[k,d,p] for k in K) + unattended[d,p] == 1

   for k in K:
      prob += pulp.lpSum(x[k,d,'0',j] for j in P) <= 1
      prob += pulp.lpSum(x[k,d,'0',j] for j in P) == pulp.lpSum(x[k,d,i,'0'] for i in P)

      work_load = pulp.lpSum(dur.get((p,d),0) * y[k,d,p] for p in P)
      capacity = end.get((k,d), 100) - start.get((k,d), 0)
      prob += work_load <= capacity + ot[k,d]

      for p in P:
         prob += pulp.lpSum(x[k,d,i,p] for i in N if i!=p) == y[k,d,p]
         prob += pulp.lpSum(x[k,d,p,j] for j in N if j!=p) == y[k,d,p]

solver = pulp.PULP_CBC_CMD(msg=0)
prob.solve(solver)
```

```
    return pulp.value(prob.objective)


# 3. MAIN


if __name__ == "__main__":
    generate_toy_data()


    print("\n--- Solving Relaxed (Fluid) ---")
    cost_lp = solve_instance(relaxed=True)


    print("\n--- Solving Integer (Rigid) ---")
    cost_ip = solve_instance(relaxed=False)


    # Gap Calculations
    absolute_gap = cost_ip - cost_lp


    # Avoid division by zero
    if cost_lp == 0:
        percent_str = "Infinite (LP Cost is 0)"
    else:
        percent_gap = (absolute_gap / cost_lp) * 100
        percent_str = f"{percent_gap:.2f}%"


    print(f"\n======================================")
    print(f" GAP ANALYSIS RESULTS")
    print(f"======================================")
    print(f" Relaxed Cost (LP): ${cost_lp:,.2f}")
```

```
print(f" Integer Cost (IP): ${cost_ip:,.2f}")
print(f"----------------------------------")
print(f" Absolute Gap:     ${absolute_gap:,.2f}")
print(f" Percentage Gap:    {percent_str}")
print(f"==================================")
```

Output

```
Generating Instance: 1 Nurse (Cap 100m), 2 Patients (Req 60m)...

--- Solving Relaxed (Fluid) ---

--- Solving Integer (Rigid) ---

==================================
 GAP ANALYSIS RESULTS
==================================
 Relaxed Cost (LP): $333.33
 Integer Cost (IP): $1,000.00
----------------------------------
 Absolute Gap:      $666.67
 Percentage Gap:    200.00%
==================================
```

Figure 2: Output for question 3

Question 4

```
import pulp


# Configuration
# 1 Nurse, 2 Patients, 100 min Capacity, 60 min Demand each
SHIFT_CAPACITY = 100
PATIENT_DEMAND = 60
PENALTY_COST = 1000.0
# ensuring the constraint binds ,Overtime cost must be high enough to be avoid
OT_COST = 10000.0
```

```
def solve_and_get_shadow_price():
    print(f"--- Setting up LP for Shadow Price Calculation ---")
    print(f"Capacity: {SHIFT_CAPACITY}m | Demand: 2x {PATIENT_DEMAND}m | Penalty: ${PENALTY_COST}")

    # 1. Creating the LP Problem (Minimization)
    prob = pulp.LpProblem("Shadow_Price_Analysis", pulp.LpMinimize)

    # 2. Decision Variables (Continuous / Relaxed)
    # y1, y2: Fraction of patient 1 and 2 visited (0.0 to 1.0)
    y1 = pulp.LpVariable("y1", lowBound=0, upBound=1, cat=pulp.LpContinuous)
    y2 = pulp.LpVariable("y2", lowBound=0, upBound=1, cat=pulp.LpContinuous)

    # ot: Overtime minutes used (Continuous)
    ot = pulp.LpVariable("ot", lowBound=0, cat=pulp.LpContinuous)

    # 3. Objective Function
    # Minimize: (Unvisited Portion * Penalty) + (Overtime * OT Cost)
    # Unvisited portion = (1 - y)
    cost_unattended = PENALTY_COST * (1 - y1) + PENALTY_COST * (1 - y2)
    cost_overtime = OT_COST * ot
    prob += cost_unattended + cost_overtime

    # 4. Constraints

    # Total work time <= Capacity + Overtime
    # We assign this constraint to a python variable 'cap_constraint' to access its shadow price later
```

```
total_work = (PATIENT_DEMAND * y1) + (PATIENT_DEMAND * y2)
cap_constraint = (total_work <= SHIFT_CAPACITY + ot)

# Add it to the problem with a name
prob += cap_constraint, "Nurse_Capacity_Constraint"

# 5. Solve
# We use GLPK or CBC. Standard CBC usually populates .pi (dual values).
prob.solve(pulp.PULP_CBC_CMD(msg=0))

# 6. Extract Results & Shadow Price
print(f"\n--- Solution Status: {pulp.LpStatus[prob.status]} ---")
print(f"Objective Cost: ${pulp.value(prob.objective):.2f}")
print(f"y1 (Patient 1): {pulp.value(y1):.2f}")
print(f"y2 (Patient 2): {pulp.value(y2):.2f}")

# Accessing the Shadow Price
shadow_price = cap_constraint.pi

print(f"\n--- Shadow Price Analysis ---")
print(f"Constraint Name: {cap_constraint.name}")
print(f"Shadow Price (.pi): {shadow_price}")
print(f"Theoretical Interpretation: ${abs(shadow_price):.2f} per minute")

# 7. Verification
value_per_minute = PENALTY_COST / PATIENT_DEMAND
print(f"Manual Verification ($1000 / 60m): ${value_per_minute:.2f}")
```

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

Output

```
--- Setting up LP for Shadow Price Calculation ---
Capacity: 100m | Demand: 2x 60m | Penalty: $1000.0

--- Solution Status: Optimal ---
Objective Cost: $333.33
y1 (Patient 1): 0.67
y2 (Patient 2): 1.00

--- Shadow Price Analysis ---
Constraint Name: Nurse_Capacity_Constraint
Shadow Price (.pi): -16.666667
Theoretical Interpretation: $16.67 per minute
Manual Verification ($1000 / 60m): $16.67
```

Figure 3: Output for question 4