Assignment 2: Optimization in Practice

Management Science

Assignment Overview

Due: Start of Lecture 10 Weight: 20% of final grade Expected Time: 5-7 hours Work: Groups

Your consulting firm has been hired by "CityExpress," a local delivery company. They need help with: 1. Optimizing delivery routes to reduce costs 2. Creating efficient shift assignments that balance operational needs and worker preferences

Consultants

Who is part of your group?

```
"""
YOUR ANSWER HERE:
"""
```

```
'\nYOUR ANSWER HERE:\n'
```

Setup

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

# Sets a random seed for reproducibility (no need to adjust this for you)
np.random.seed(42)
```

Part A: Smart Delivery Routes (50%)

Scenario

CityExpress has 12 customer orders to deliver tomorrow. They need an efficient route starting and ending at their depot.

The Data

```
# Location coordinates (in km from origin)
locations = {
    0: (10, 10),    # Depot
    1: (8, 15),     # Customer 1
    2: (14, 18),    # Customer 2
```

```
    3: (5, 12),     # Customer 3
    4: (18, 8),     # Customer 4
    5: (12, 5),     # Customer 5
    6: (3, 7),      # Customer 6
    7: (16, 14),    # Customer 7
    8: (7, 3),      # Customer 8
    9: (15, 6),     # Customer 9
    10: (11, 17),   # Customer 10
    11: (4, 16),    # Customer 11
    12: (19, 12)    # Customer 12
}
```

Task 1: Build a Basic Route (20%)

```
def calculate_distance(loc1, loc2):
    """Calculate Euclidean distance between two locations."""
    x1, y1 = loc1
    x2, y2 = loc2
    # YOUR CODE HERE

    return distance

def calculate_total_distance(route, locations):
    """Calculate total distance for a route."""
    total = 0
    # YOUR CODE HERE

    # Sum distances between consecutive locations
    return total

def nearest_neighbor_route(depot, locations):
    """
    Build route using nearest neighbor heuristic.
    Always visit the nearest unvisited customer next.
    """
    route = [depot] # Start with depot
    # YOUR CODE HERE

    route.append(depot)  # End with depot
    return route

# Build your route
# YOUR CODE HERE
```

Task 2: Improve Your Route (20%)

Understanding 2-Opt Improvement:

The 2-opt algorithm improves a route by removing two edges and reconnecting them in a different way. This is like “uncrossing” routes that cross over themselves.

Visual Example:

```
Before:  A → B → C → D → A
If we take edges (A→B) and (C→D) and swap them:
After:   A → C → B → D → A  (reversed the B→C segment)
```

The Algorithm:

1. Start with your current best route and its distance
2. Try swapping every possible pair of edges:
   - Take positions i and j in the route (where i < j)
   - Reverse the segment between i and j
   - Calculate the new distance
3. If the new route is better, keep it as your new best
4. Repeat until no improvement is found

```
def try_swap_improvement(route, locations):
    """
    Try all possible 2-opt swaps and return the best improvement found.

    Args:
        route: Current route (list of location indices)
        locations: Dictionary of location coordinates

    Returns:
        tuple: (best_route, best_distance) or (None, None) if no
improvement
    """
    # YOUR CODE HERE (replace pass)

    pass

def improve_route(initial_route, locations, max_iterations=50):
    """
    Repeatedly apply 2-opt improvements until no improvement is found.

    Args:
        initial_route: Starting route
        locations: Dictionary of location coordinates
        max_iterations: Maximum number of improvement attempts

    """
    # Loop while iteration < max_iterations:
    #   1. Call try_swap_improvement on current_route
    #   2. If no improvement found, break
    #   3. Otherwise, update current_route and current_distance
    #   4. Increment iteration counter
    # YOUR CODE HERE (replace pass)

    pass

# Improve your route
# YOUR CODE HERE
```

Task 3: Visualize and Analyze (10%)

- Visualize both routes (before and after improvement)
- Calculate and compare:
  - Original route distance
  - Improved route distance
  - Percentage improvement
  - Estimated cost savings (€2 per km)

```
# Visualize both routes
# YOUR CODE HERE

# Calculate metrics
# YOUR CODE HERE
```

Business Question: If CityExpress has 50 deliveries per day, how much could they save per month with route optimization? (3-4 sentences)

```
"""
YOUR ANSWER HERE:
"""
```

```
'\nYOUR ANSWER HERE:\n'
```

Part B: Smart Shift Assignment (50%)

Scenario

CityExpress warehouse needs to assign 6 workers to 6 different shifts this week. Each worker works exactly one shift. Your job: maximize worker satisfaction by matching them to their preferred shifts.

Connection to Lecture 6

Remember greedy heuristics from job shop scheduling (SPT, EDD)? You'll design similar rules here—but for workers and shifts instead of jobs and machines.

Think about:

- SPT prioritized shortest jobs first
- EDD prioritized jobs with earliest deadlines first
- What should you prioritize for shift assignment?

The Data

```
# 6 shifts available this week
shifts = [
    'Monday-Morning',
    'Monday-Evening',
```

```
    'Tuesday-Morning',
    'Tuesday-Evening',
    'Wednesday-Morning',
    'Wednesday-Evening'
]

# Worker shift preferences (in order of preference: 1st choice, 2nd choice,
3rd choice)
worker_preferences = {
    0: ['Monday-Morning', 'Tuesday-Morning', 'Wednesday-Morning'],     #
Morning person
    1: ['Monday-Evening', 'Tuesday-Evening', 'Wednesday-Evening'],     #
Prefers evenings
    2: ['Monday-Morning', 'Wednesday-Morning'],                        #
Flexible, fewer preferences
    3: ['Tuesday-Evening', 'Wednesday-Evening'],                       #
Evening only, fewer preferences
    4: ['Monday-Morning', 'Monday-Evening', 'Tuesday-Morning'],        #
Busy early week
    5: ['Wednesday-Morning', 'Wednesday-Evening']                      #
Wednesday preferred
}

# Assignment representation:
# assignment = [shift_index for each worker]
# Example: assignment = [0, 1, 2, 3, 4, 5]
#   Worker 0 gets shift 0 (Monday-Morning)
#   Worker 1 gets shift 1 (Monday-Evening)
#   Worker 2 gets shift 2 (Tuesday-Morning)
#   etc.
```

Task 1: Design Your Greedy Heuristic (15%)

Challenge: Create a function that builds an assignment using YOUR OWN greedy strategy.

Need a starting point?

Ask yourself:

1. What made SPT different from EDD?
2. What attributes do workers have?
3. What attributes do shifts have?
4. Pick one attribute to prioritize. That's your greedy rule!

```
def my_greedy_assignment(worker_preferences, shifts):
    """
    Build an assignment using YOUR greedy strategy.

    Args:
        worker_preferences: Dict of worker_id → list of preferred shift
names
        shifts: List of 6 shift names
```

```
    Returns:
        list: assignment where assignment[worker_id] = shift_index
              Example: [0, 1, 2, 3, 4, 5] means worker 0→shift 0, worker
1→shift 1, etc.
    """
    assignment = [-1] * 6  # -1 means unassigned
    available_shifts = list(range(6))  # Track which shifts are still open

    # YOUR GREEDY STRATEGY HERE
    # Questions to guide you:
    # 1. In what order will you process the workers? (0,1,2,3,4,5 or
different order?)
    # 2. For each worker, how do you pick their shift from available ones?
    # 3. What if their preferred shifts are all taken?

    # YOUR CODE HERE

    return assignment

# Test your greedy heuristic
# YOUR CODE HERE
```

Deliverable:

1. Working greedy function that produces a valid assignment (all assigned, no duplicate shifts)
2. Written explanation (3-4 sentences)

```
"""
YOUR EXPLANATION HERE:
My greedy strategy: [describe your rule]
Reasoning: [why did you choose this approach?]
"""
```

```
'\nYOUR EXPLANATION HERE:\nMy greedy strategy: [describe your
rule]\nReasoning: [why did you choose this approach?]\n'
```

Task 2: Build an Evaluation Function (15%)

Challenge: Create a function that measures how GOOD an assignment is.

Design questions:

- Should you give more points for 1st choice vs 2nd choice vs 3rd choice?
- Should all workers count equally, or weight some more?
- What if a worker gets a shift they didn't list as preferred?

```
def calculate_satisfaction(assignment, worker_preferences, shifts):
    """
    Calculate how good an assignment is.
```

```
    Design YOUR OWN scoring system!

    Possible approaches:
    - 1st choice = 3 pts, 2nd choice = 2 pts, 3rd choice = 1 pt, other = 0
pts
    - 1st choice = 10 pts, 2nd choice = 5 pts, 3rd choice = 1 pt, other =
-5 pts
    - Binary: preferred shift = 1 pt, non-preferred = 0 pts
    - Your own scoring!

    Args:
        assignment: List where assignment[worker_id] = shift_index
        worker_preferences: Dict of worker_id → list of preferred shift
names
        shifts: List of shift names

    Returns:
        float or int: Total satisfaction score (higher is better)
    """
    # Design your own scoring system!
    # YOUR EVALUATION LOGIC HERE


    return total_satisfaction

# Test your evaluation function
# YOUR CODE HERE
```

Deliverable:

1. Working evaluation function
2. Written explanation (2-3 sentences):
   - What scoring system did you design?
   - Why did you choose this approach?
   - How does it relate to metrics from Lecture 6?

```
"""
YOUR EXPLANATION HERE:
My scoring system: [describe how you calculate satisfaction]
Reasoning: [why this approach?]
"""
```

```
'\nYOUR EXPLANATION HERE:\nMy scoring system: [describe how you calculate
satisfaction]\nReasoning: [why this approach?]\n'
```

Task 3: Improve with Local Search (15%)

Challenge: Take your greedy solution and improve it using local search.

Think about Lecture 7:

- 2-opt tried swaps and kept improvements
- It kept searching until no improvement was found
- Can you apply similar logic here?

Your task: Implement a local search that tries swapping workers' shifts.

```
def improve_with_local_search(initial_assignment, worker_preferences,
shifts):
    """
    Improve an assignment using local search (like 2-opt from Lecture 7).
    Strategy: Try swapping pairs of workers' shifts, keep if it improves
satisfaction.

    Think about:
    - How do you generate "neighbor" solutions? (swap two workers)
    - How do you know if a neighbor is better? (use your evaluation
function!)
    - When do you stop? (no improvement found, or max iterations)

    Args:
        initial_assignment: Starting assignment (from your greedy)
        worker_preferences: Dict of worker_id → preferred shifts
        shifts: List of shift names

    Returns:
        list: improved assignment
    """

    # YOUR LOCAL SEARCH LOGIC HERE


    # YOUR CODE HERE

    return improved_assignment

# Apply local search
# YOUR CODE HERE
```

Deliverable:

1. Working local search function that improves the solution
2. Visualization showing before/after
3. Written explanation (3-4 sentences):
   - How does your local search work?
   - How much did you improve the greedy solution?
   - Could you improve it further?

```
"""
YOUR EXPLANATION HERE:
Local search method: [describe your approach]
Results: [how much improvement?]
```

```
Further improvements?: [yes/no and why?]
"""
```

```
'\nYOUR EXPLANATION HERE:\nLocal search method: [describe your approach]\nResults: [how much improvement?]\nFurther improvements?: [yes/no and why?]\n'
```

Task 4: Business Reflection (5%)

Question 1: Imagine one worker calls in sick at the last minute. How would you quickly reassign the remaining 5 workers to 6 shifts (one shift will be unfilled)? Would your greedy strategy still work? (3-4 sentences)

```
"""
YOUR ANSWER HERE:
"""
```

```
'\nYOUR ANSWER HERE:\n'
```

Question 2: Your manager says "I don't care about worker preferences, just fill all shifts as quickly as possible." How would this change your approach? What would you lose? (2-3 sentences)

```
"""
YOUR ANSWER HERE:
"""
```

```
'\nYOUR ANSWER HERE:\n'
```

Submission Checklist

- ? All code cells run without errors
- ? Part A: Routes are properly visualized
- ? Part B: Greedy heuristic, evaluation function, and local search all implemented
- ? Written explanations completed for all tasks
- ? Business questions answered
- ? Code is commented and clear
- ? Names added to top of notebook

Tips

- Use AI tools to help understand concepts and program, but make sure you understand the code
- Start simple - get something working before optimizing
- Experiment! - there's no single "correct" approach

- Focus on reasoning - I care more about your thinking than perfect code
- The goal is good solutions, not perfect ones

Bibliography