JobSync Code Explanation

This document explains the implementation approach for the JobSync optimization project, detailing how the Python code works and how to interpret the results.

Code Structure

The solution consists of the following main components:

1. **Data Loading and Preprocessing** (`load_data` function)
    - Loads the three CSV files: seekers, jobs, and location distances
    - Converts string representations of lists to actual Python lists
    - Converts numeric fields to appropriate types
2. **Compatibility Checking** (`check_compatibility` function)
    - Implements the five compatibility criteria:
        - Job Type: Seeker's desired type = job's type
        - Salary: Seeker's minimum desired salary ≤ job's maximum offered salary
        - Skills: Seeker possesses all skills required by the job
        - Experience: Seeker's experience level ≥ job's required level
        - Location: Job is remote OR distance ≤ seeker's maximum commute distance
3. **Dissimilarity Calculation** (`calculate_dissimilarity` function)
    - Calculates the average absolute difference between the seeker's and job's questionnaire responses
    - Range: [0, 5] where 0 means perfect match and 5 means maximum dissimilarity
4. **Part 1: Priority-Weighted Matching** (`solve_part1` function)
    - Creates and solves the first ILP model to maximize total priority weight
    - Returns the model, compatible pairs, and maximum priority weight (Mw)
5. **Part 2: Dissimilarity Minimization** (`solve_part2` function)
    - Creates and solves the second ILP model to minimize maximum dissimilarity
    - Takes an omega parameter specifying the minimum percentage of Mw to maintain
    - Returns the model, minimum maximum dissimilarity, and achieved priority weight
6. **Results Visualization** (`plot_results` function)
    - Creates a plot showing how the minimum maximum dissimilarity changes with omega
    - Helps to visualize the trade-off between priority weight and match quality
7. **Solution Analysis** (`analyze_solutions` function)

- Analyzes the results for different omega values
- Identifies a recommended omega value based on the trade-off analysis

Implementation Details

Compatibility Checking

The compatibility check is a critical part of the preprocessing. Rather than adding explicit constraints for each compatibility requirement in the ILP model, we determine compatible pairs upfront and only create decision variables for those pairs. This approach significantly reduces the model size and complexity.

python

```python
def check_compatibility(seeker, job, distances_df):
    # Experience level compatibility
    exp_order = get_experience_level_order()
    seeker_exp = exp_order.get(seeker['Experience_Level'], 0)
    job_exp = exp_order.get(job['Required_Experience_Level'], 6)
    if seeker_exp < job_exp:
        return False

    # Job type compatibility
    if seeker['Desired_Job_Type'] != job['Job_Type']:
        return False

    # Salary compatibility
    if seeker['Min_Desired_Salary'] > job['Salary_Range_Max']:
        return False

    # Skills compatibility
    if not all(skill in seeker['Skills'] for skill in job['Required_Skills']):
        return False

    # Location compatibility
    if job['Is_Remote'] == 1:
        return True
    else:
        seeker_location = seeker['Location']
        job_location = job['Location']
        distance = distances_df.loc[seeker_location, job_location]
        return distance <= seeker['Max_Commute_Distance']
```

ILP Model Creation

For both optimization problems, we create Gurobi models with binary decision variables for each compatible seeker-job pair. The implementation uses Python dictionaries to store the decision variables, which allows for efficient handling of sparse compatibility matrices.

```python
# Decision variables: x[i,j] = 1 if seeker i is assigned to job j, 0 otherwise
x = {}
for seeker_id, job_ids in compatible_pairs.items():
    for job_id in job_ids:
        x[seeker_id, job_id] = model.addVar(vtype=GRB.BINARY, name=f"x_{seeker_id}_{job_id}")
```

Implementing the Minimize Maximum Constraint

In Part 2, we need to minimize the maximum dissimilarity score. This is implemented using a continuous variable `max_dissimilarity` and constraints that force it to be at least as large as any assigned pair's dissimilarity.

```python
# Additional variable for the maximum dissimilarity
max_dissimilarity = model.addVar(vtype=GRB.CONTINUOUS, name="max_dissimilarity")

# Constraint: Maximum dissimilarity
for (seeker_id, job_id), score in dissimilarity_scores.items():
    model.addConstr(
        max_dissimilarity >= score * x[seeker_id, job_id],
        f"max_dissimilarity_{seeker_id}_{job_id}"
    )
```

How to Use the Code

1. **Setup Environment**:
    - Install the required packages: pandas, numpy, matplotlib, gurobipy
    - Ensure you have an academic license for Gurobi
2. **Prepare Data**:
    - Place the three CSV files (seekers.csv, jobs.csv, location_distances.csv) in the same directory as the script
3. **Run the Script**:
    - Execute the main function to run the entire optimization process
    - The script will output results to the console and generate a visualization
4. **Interpret Results**:
    - The script will provide:
        - Maximum priority weight (Mw) from Part 1

- Results for each omega value, including minimum maximum dissimilarity and achieved priority weight
- A recommended omega value with justification
- A visualization showing the trade-off between omega and dissimilarity

Interpreting the Visualization

The generated plot shows how the minimum maximum dissimilarity changes as omega increases. Key points to observe:

1. **Trade-off Curve**: As omega increases (requiring higher priority weights), the minimum achievable maximum dissimilarity typically increases. This illustrates the fundamental trade-off between optimizing for priority weight versus optimizing for match quality.
2. **Elbow Point**: Look for a point where the curve starts to rise more steeply. This "elbow" often represents a good trade-off point, where further increases in omega lead to disproportionately large increases in dissimilarity.
3. **Practical Considerations**: The "best" omega value depends on the specific application context:
    - Higher omega values prioritize filling important jobs (higher priority weight)
    - Lower omega values prioritize better matches between seekers and jobs (lower dissimilarity)

Final Recommendations

Based on the trade-off analysis, the code provides a recommended omega value. However, the final choice should consider:

1. The business impact of unfilled high-priority positions
2. The importance of good matches for long-term job satisfaction and retention
3. The shape of the trade-off curve in your specific dataset

The chosen omega value should represent a balance point that makes sense for the specific job matching context and organizational priorities.