Problem 1: Two-Objective Optimization with Inequality Constraints

To maximize two objective functions `f1(x)` and `f2(x)` with the constraints `f1(x) ≥ α1` and `f2(x) ≥ α2`,where these constraints might be infeasible, we can design an evolutionary algorithm with a penalty method to handle the constraints. Here's a step-by-step algorithm:

Algorithm: Evolutionary Multi-Objective Optimization with Penalty Method

1. **Initialization:**
    - Generate an initial population `{xi}` randomly.
    - Evaluate the objective values `f1(xi)` and `f2(xi)` for each solution `xi`.
2. **Constraint-Handling with Penalty:**
    - For each solution `xi`, calculate the penalized fitness value:

      $$\text{Fitness}(x_i) = \begin{cases} (f_1(x_i), f_2(x_i)) & \text{if } f_1(x_i) \geq \alpha_1 \text{ and } f_2(x_i) \geq \alpha_2 \\ (f_1(x_i) - \lambda \cdot \max(0, \alpha_1 - f_1(x_i)), f_2(x_i) - \lambda \cdot \max(0, \alpha_2 - f_2(x_i))) & \text{otherwise} \end{cases}$$

      where `λ` is a penalty coefficient.
3. **Selection:**
    - Use a selection mechanism (e.g., tournament selection) based on the penalized fitness values to select parents.
4. **Crossover and Mutation:**
    - Apply crossover and mutation operations to generate offspring from the selected parents.
5. **Offspring Evaluation:**
    - Evaluate the offspring's objective values `f1(x)` and `f2(x)`.
6. **Population Update:**
    - Combine parents and offspring to form a new population.
    - Select the best individuals based on penalized fitness values to form the next generation.
7. **Termination Check:**
    - If termination criteria are met (e.g., maximum generations or satisfactory convergence), terminate the algorithm.
    - Otherwise, return to step 3.

Pseudocode:

```python
def evolutionary_multi_objective_optimization():
    population = initialize_population()
    evaluate_population(population)

    while not termination_criteria_met():
        parents = selection(population)
        offspring = crossover_and_mutation(parents)
        evaluate_population(offspring)
        population = update_population(population, offspring)

    return best_solution(population)

def evaluate_population(population):
    for individual in population:
        f1 = evaluate_f1(individual)
        f2 = evaluate_f2(individual)
        penalty_f1 = max(0, alpha_1 - f1)
        penalty_f2 = max(0, alpha_2 - f2)
        fitness = (f1 - penalty_coefficient * penalty_f1, f2 - penalty_coefficient * penalty_f2)
        individual.fitness = fitness

# Functions for initialization, selection, crossover, mutation, and population update need to be defined.
```

Conclusion:

This algorithm balances the objectives `f1(x)` and `f2(x)` while attempting to satisfy the constraints. If the constraints are infeasible, the penalty method guides the search towards solutions that are as close as possible to meeting the constraints. The evolutionary principles ensure robust search across the solution space, increasing the likelihood of finding a satisfactory solution for the decision maker.

Problem 2: Two-Objective Optimization with Target Point

To maximize two objective functions `f1(x)` and `f2(x)` given a target (ideal) point `(z1,z2)`, which may be inside or outside the feasible region, we can design an algorithm to find the best solution closest to the target point. We can use an evolutionary algorithm approach, minimizing the distance between the solutions and the target point. Here's the design of such an algorithm:

Algorithm: Evolutionary Algorithm for Target Point Minimization

1. **Initialization:**
   - Generate an initial population of solutions `{xi}` randomly.
   - Evaluate the objective values `f1(xi)` and `f2(xi)` for each solution `xi`.
2. **Distance Calculation:**
   - For each solution `{xi}`, calculate its Euclidean distance to the target point `(z1,z2)`:

$$d(x_i) = \sqrt{(f_1(x_i) - z_1^*)^2 + (f_2(x_i) - z_2^*)^2}$$

3. **Fitness Evaluation:**
   - The fitness value is determined by the distance `d(xi)`. The smaller the distance, the higher the fitness. For use in the evolutionary algorithm, we can define the fitness function as:

$$\text{Fitness}(x_i) = \frac{1}{1 + d(x_i)}$$

     This ensures that smaller distances result in higher fitness values.
4. **Selection:**
   - Use a selection mechanism (e.g., tournament selection) to choose parents based on their fitness values.
5. **Crossover and Mutation:**
   - Apply crossover and mutation operations to the selected parents to generate offspring.
6. **Offspring Evaluation:**
   - Evaluate the objective values `f1(xi)` and `f2(xi)` for the offspring, and compute their distances `d(xi)` and fitness values.
7. **Population Update:**
   - Combine parents and offspring to form a new population.
   - Select the top individuals based on their fitness values to form the next generation.
8. **Check Termination Criteria:**
   - If a stopping criterion is met (e.g., a maximum number of generations or satisfactory convergence level), terminate the algorithm.
   - Otherwise, return to step 4.

Pseudocode:

```
def evolutionary_algorithm_target_point():
    population = initialize_population()
    evaluate_population(population)

    while not termination_criteria_met():
        parents = selection(population)
        offspring = crossover_and_mutation(parents)
        evaluate_population(offspring)
        population = update_population(population, offspring)

    return best_solution(population)

def evaluate_population(population):
    for individual in population:
        f1 = evaluate_f1(individual)
        f2 = evaluate_f2(individual)
        distance = calculate_distance(f1, f2, z1_star, z2_star)
        fitness = 1 / (1 + distance)
        individual.fitness = fitness

def calculate_distance(f1, f2, z1_star, z2_star):
    return ((f1 - z1_star) ** 2 + (f2 - z2_star) ** 2) ** 0.5

# Define initialization, selection, crossover, mutation, and update_population functions.
```

Conclusion:

This algorithm guides the search by minimizing the distance between each solution and the target point. The evolutionary algorithm ensures diversity in the population and a thorough search of the solution space. Ultimately, this algorithm will find the solution closest to the decision maker's target point `(z1,z2)` . If the target point is within the feasible region, the algorithm effectively finds a solution near that target point; if the target point is outside the feasible region, the algorithm will find the closest feasible solution.

Problem 3: Two-Objective Optimization with Target Point and Weight Vector

To find the final solution for the decision maker given the target (ideal) point z^* = `(z1,z2)` and the weight vector $w = (w_1, w_2)$, we can design an algorithm that incorporates both the target point and the weight vector into the objective function. One effective approach is to use a weighted distance metric that measures how far a solution is from the target point while taking into account the importance of each objective through the weights.

Algorithm: Weighted Distance Minimization with Evolutionary Algorithm

1. **Initialization:**
   - Generate an initial population of solutions `{xi}` randomly.
   - Evaluate the objective values `f1(xi)` and `f2(xi)` for each solution `xi`.
2. **Weighted Distance Calculation:**
   - For each solution `xi`, calculate a weighted distance to the target point `(z1,z2)`:

$$d_w(x_i) = \sqrt{w_1(f_1(x_i) - z_1^*)^2 + w_2(f_2(x_i) - z_2^*)^2}$$

3. **Fitness Evaluation:**
   - The fitness value is determined by the weighted distance $d_w(x_i)$

     The smaller the distance, the higher the fitness. The fitness function can be defined as:

$$\text{Fitness}(x_i) = \frac{1}{1 + d_w(x_i)}$$

     This ensures that smaller distances result in higher fitness values.
4. **Selection:**
   - Use a selection mechanism (e.g., tournament selection) to choose parents based on their fitness values.
5. **Crossover and Mutation:**
   - Apply crossover and mutation operations to the selected parents to generate offspring.
6. **Offspring Evaluation:**
   - Evaluate the objective values `f1(xi)` and `f2(xi)` for the offspring, and compute their weighted distances $d_w(x_i)$ and fitness values.

7. **Population Update:**
    - Combine parents and offspring to form a new population.
    - Select the top individuals based on their fitness values to form the next generation.
8. **Check Termination Criteria:**
    - If a stopping criterion is met (e.g., a maximum number of generations or satisfactory convergence level), terminate the algorithm.
    - Otherwise, return to step 4.

Pseudocode:

```
def evolutionary_algorithm_target_point():
    population = initialize_population()
    evaluate_population(population)

    while not termination_criteria_met():
        parents = selection(population)
        offspring = crossover_and_mutation(parents)
        evaluate_population(offspring)
        population = update_population(population, offspring)

    return best_solution(population)

def evaluate_population(population):
    for individual in population:
        f1 = evaluate_f1(individual)
        f2 = evaluate_f2(individual)
        weighted_distance = calculate_weighted_distance(f1, f2, z1_star, z2_star, w1, w2)
        fitness = 1 / (1 + weighted_distance)
        individual.fitness = fitness

def calculate_weighted_distance(f1, f2, z1_star, z2_star, w1, w2):
    return ((w1 * (f1 - z1_star) ** 2 + w2 * (f2 - z2_star) ** 2) ** 0.5)

# Define initialization, selection, crossover, mutation, and update_population functions.
```

Conclusion:

This algorithm integrates both the target point and the weight vector provided by the decision maker. By minimizing the weighted distance to the target point, the algorithm finds a solution that is close to the desired target while respecting the relative importance of each objective as specified by the weights. This approach ensures that the final solution aligns with the decision maker's preferences.