```python
import numpy as np
import matplotlib.pyplot as plt

# Define parameters
population_size = 100
num_resources = 3
generations = 50
mutation_rate = 0.1

# Fitness function: Maximize profit while ensuring the sum of allocations <= 1
def fitness_function(allocation):
    revenue = allocation[0] * 100 + allocation[1] * 80 + allocation[2] * 50
    cost = sum(allocation) * 30
    return revenue - cost if sum(allocation) <= 1 else -np.inf  # Constraint: sum <= 1

# Initialize population with normalized allocations
population = np.random.rand(population_size, num_resources)
population = population / population.sum(axis=1, keepdims=True)

# Evolutionary loop
for generation in range(generations):
    # Evaluate fitness of each individual
    fitness = np.array([fitness_function(ind) for ind in population])

    # Handle invalid fitness values
    if np.all(np.isinf(fitness)):
        print("All solutions are invalid. Check constraints or population initialization.")
        break

    # Replace -inf with a small value to allow selection
    fitness = np.where(np.isinf(fitness), -1e10, fitness)

    # Normalize fitness to make it non-negative
    min_fitness = np.min(fitness)
    if min_fitness < 0:
        fitness = fitness - min_fitness  # Shift all fitness values to be non-negative

    # Selection (Roulette Wheel Selection)
    fitness_sum = fitness.sum()
    if fitness_sum == 0:
        probabilities = np.ones(population_size) / population_size  # Equal probability
    else:
        probabilities = fitness / fitness_sum

    parents_idx = np.random.choice(np.arange(population_size), size=population_size, p=probabilities)
    parents = population[parents_idx]

    # Crossover
    offspring = []
    for i in range(0, population_size, 2):
        parent1, parent2 = parents[i], parents[i + 1]
        crossover_point = np.random.randint(1, num_resources)
        child1 = np.concatenate((parent1[:crossover_point], parent2[crossover_point:]))
        child2 = np.concatenate((parent2[:crossover_point], parent1[crossover_point:]))
        offspring.extend([child1, child2])
    offspring = np.array(offspring)

    # Mutation
    for i in range(population_size):
        if np.random.rand() < mutation_rate:
            mutate_idx = np.random.randint(num_resources)
            offspring[i, mutate_idx] += np.random.uniform(-0.1, 0.1)
            offspring[i] = np.clip(offspring[i], 0, 1)  # Ensure allocation is valid
            offspring[i] = offspring[i] / offspring[i].sum()  # Re-normalize allocation

    # Replace population with new offspring
    population = offspring
```

```python
    # Mutation
    for i in range(population_size):
        if np.random.rand() < mutation_rate:
            mutate_idx = np.random.randint(num_resources)
            offspring[i, mutate_idx] += np.random.uniform(-0.1, 0.1)
            offspring[i] = np.clip(offspring[i], 0, 1)  # Ensure allocation is valid
            offspring[i] = offspring[i] / offspring[i].sum()  # Re-normalize allocation

    # Replace population with new offspring
    population = offspring

# Find the best solution in the final population
fitness = np.array([fitness_function(ind) for ind in population])
best_idx = np.argmax(fitness)
best_allocation = population[best_idx]

# Print and visualize the results
print("Optimal Resource Allocation:", best_allocation)
print("Maximum Profit Achieved:", fitness[best_idx])

# Bar chart visualization
plt.bar(range(num_resources), best_allocation)
plt.title("Optimal Resource Allocation")
plt.xlabel("Resources")
plt.ylabel("Proportion Allocated")
plt.xticks(range(num_resources), [f"Resource {i+1}" for i in range(num_resources)])
plt.show()
```

```
Optimal Resource Allocation: [0.4123063 0.5876937 0.       ]
Maximum Profit Achieved: 58.246125979225354
```