



Computational Intelligence

CI (DS313/DS351)

Energy Optimization Problem

(Differential Evolution Algorithm)

Instructors

Dr. Ayman Sabry Ghoneim

Dr. Sally Kassem

Name	ID
Omar Yousef Mohamed	20210591
Omar Samir Imam	20200345
Esraa Ahmed Saad	20210062

Table of Contents

<i>Content</i>	<i>Page Number</i>
• Code Documentation	3
• CI Algorithms and linear & Non-Linear mathematical formulation	9
- CI Algorithms	
- Differential Evolution Algorithm	
- Search Paper	
• Problem Definition & Mathematical Model	15
• References	

Code Documentation

Introduction:

In this documentation, we will discuss an optimization problem in the energy field and the solution to this problem using one of the most important Computational Algorithms is the Differential Evolution Algorithm.

The problem is minimizing the total energy cost that three machines need to produce units of solar panels.

Decision Variables:

X1 is the amount of energy used in machine A in KWH.

X2 is the amount of energy used in machine B in KWH.

X3 is the amount of energy used in machine C in KWH.

Objective Function:

Minimize $Z = 0.4X1 + 0.2X2 + 0.3X3$

Subject to some constraints:

$X1 \leq 1000, X2 \leq 640, X3 \leq 420$ (Boundaries of Decision Variables).

$X1 + X2 + X3 \leq 1500$ (Total Energy Consumption).

$0.3X1 + 0.5X2 + 0.2X3 \geq 500$ (Production of Units per KWH).

$0.15X1 + 0.25X2 + 0.1X3 \leq 600$ (Carbon Emissions per KWH).

Parameters:

- 1- Population Size = (50, 30, 20).
- 2- Number of iterations (to end the algorithm if stopping criteria doesn't meet) = 500
- 3- Mutation Factor = 0.7 (Usually is random in the range [0,2.0].
- 4- Crossover Rate = 0.9
- 5- Decision Variables = 3
- 6- Initial Population: is set randomly in the allowed ranges of each decision variable.

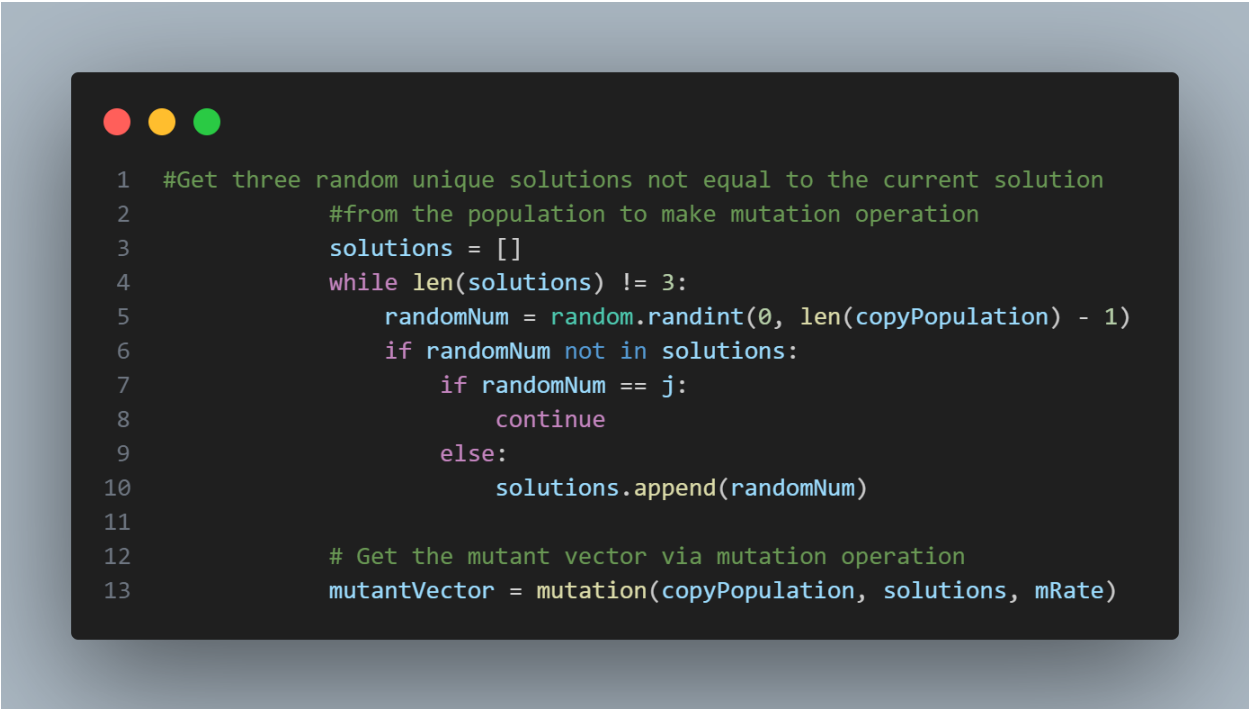
```
1  #Parameters
2  popSize = 50
3  numberOfIterations = 500
4  decisionVariables = 3  # X1, X2, X3
5  F = 0.7  # Mutation Factor
6  crossoverRate = 0.9 #Crossover Probability
7
8  # Randomly initialized x1, x2 and x3 values in the population
9  initialPopulation = [
10     [round(random.uniform(0, 1000), 2),
11      round(random.uniform(0, 640), 2),
12      round(random.uniform(0, 420), 2)] for _ in range(popSize)
13  ]
```

Mutation:

Mutation operation in DE is essential for implementing the DE algorithm. For everyone in the population, create a mutant vector by adding the weighted difference between two random solutions from the population to a third random solution too.

In detail, for each I solution in the population we should choose three random unique solutions distinct from solution I and add one of them to the weighted difference of the others, the weight used in the mutation operation is the mutation factor F which is in the range of 0.0 and 2.0.

This part is the way that we select the three random solutions P1, P2 and P3



```
1  #Get three random unique solutions not equal to the current solution
2      #from the population to make mutation operation
3      solutions = []
4      while len(solutions) != 3:
5          randomNum = random.randint(0, len(copyPopulation) - 1)
6          if randomNum not in solutions:
7              if randomNum == j:
8                  continue
9              else:
10                 solutions.append(randomNum)
11
12     # Get the mutant vector via mutation operation
13     mutantVector = mutation(copyPopulation, solutions, mRate)
```

Solution is a list that contains the indices of the selected solutions and passes them to the mutation function to return the mutant vector (solution).

Second, Mutation function:

```
1  #Mutation Operation
2  def mutation(pop, solutions, mRate):
3      # The three solutions p1, p2, p3
4      p1, p2, p3 = [pop[i] for i in solutions]
5      mutantVector = [round(p1[i] + mRate * (p2[i] - p3[i]), 2) for i in range(len(p1))]
6      return mutantVector
```

Crossover:

The target solution I and the mutant vector are passed to the crossover function where they are mixed to return the trial solution.

```
1  #Crossover Operation
2  def crossover(targetVector, mutantVector, cRate):
3      solutionLength = len(targetVector)
4      trialVector = np.zeros(solutionLength, dtype=float)
5      for i in range(len(targetVector)):
6          if random.random() < cRate:
7              trialVector[i] = mutantVector[i]
8          else:
9              trialVector[i] = targetVector[i]
10     return list(trialVector)
```

Looping on the two solutions and generate a random number if this random number is less than the crossover rate then take this value in the vector from the mutant solution else take it from the target solution.

Selection:

The selection operation chooses the best solution to keep in the next generation between the trial solution and target solution (solution I) but before this, we should validate the values trial solution if it is out of the range or boundary or not if yes repair it to keep it feasible.

```
1 # Pass the target and mutant vector to crossover operation to return the trial vector
2 trialVector = crossover(copyPopulation[j], mutantVector, cRate)
3 repair(trialVector) # If the new Solution violates the boundary repair it
4 # If the trial vector is better than the target vector then take it else keep the target vector
5 # Minimization Problem
6 if objective(trialVector) < fitness[j]:
7     nextGeneration[j] = trialVector
8 else:
9     nextGeneration[j] = copyPopulation[j]
```

Constraints Handling:

To handle the constraints in the problem there are two ways first to make a repair function that converts the infeasible solutions into feasible solutions, second to add a penalty that is very large to the objective function to force the algorithm to keep searching only in the feasible areas. In this implementation, we used both techniques first, repair function to repair the boundary second, we added a penalty to the solution that violates other constraints.

```
1
2 # Repair function repairs the solution values x1, x2 and x3 if one of them exceed its boundary
3 def repair(solution):
4     boundaries = [(0, 1000), (0, 640), (0, 420)]
5     for i in range(3):
6         lower, upper = boundaries[i]
7         solution[i] = max(lower, min(solution[i], upper))
```

Objective Function:

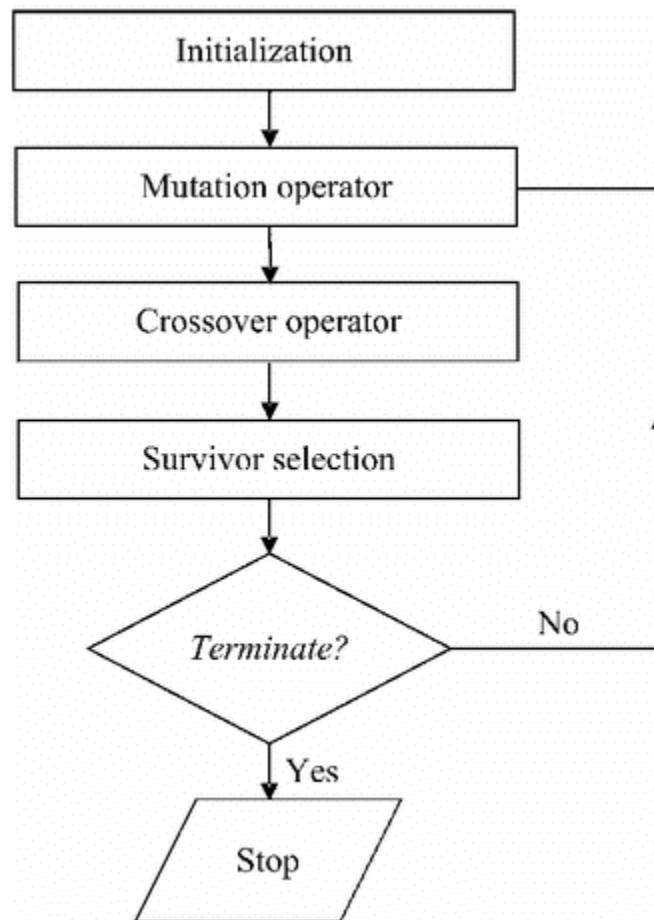
```
1 #Objective Function with penalty added if the solution violates at least one constraint
2 def objective(solution):
3     x1, x2, x3 = solution
4     if isFeasible(solution):
5         return round(0.4 * x1 + 0.2 * x2 + 0.3 * x3, 2)
6     else:
7         penalty = (max(0, (x1 + x2 + x3 - 1500)) +
8                   abs(min(0, 0.3 * x1 + 0.5 * x2 + 0.2 * x3 - 500)) +
9                   max(0, 0.15 * x1 + 0.25 * x2 + 0.1 * x3 - 600))
10        penaltyFactor = 1000 # Increased penalty factor
11        return round((0.4 * x1 + 0.2 * x2 + 0.3 * x3) + penaltyFactor * penalty, 2)
```

Termination Criteria:

The stopping Criteria in this implementation is the number of iterations which is equal to 500 iterations and we add a check if there is no big change over 30 generations to stop the algorithm preventing waste of resources.

```
1
2 for i in range(iterations):
3     '''
4     This condition is the stopping criteria of the algorithm
5     if there is no big change over the 30 iterations then stop the algorithm
6     '''
7     if i >= 30:
8         if abs(bestFitness[i - 30] - bestFitness[i - 1]) < bestFitness[i - 30] * 0.01:
9             break
10
11 #Rest of the algorithm below
```


DE Algorithm Flow Chart



CI Algorithms and linear & Non-Linear mathematical formulation

CI Algorithms

Computational Intelligence (CI) refers to a subset of artificial intelligence (AI) techniques that focus on problem-solving and decision-making inspired by biological and natural systems. CI algorithms are designed to model and solve complex problems by mimicking the behavior of natural systems, such as evolutionary processes, swarm intelligence, or neural networks. Here are some key areas within Computational Intelligence:

Evolutionary Algorithms: These algorithms are inspired by the process of natural selection and evolution. Examples include Genetic Algorithms (GA), Genetic Programming (GP), Evolutionary Strategies (ES), and Differential Evolution (DE). They are often used for optimization and search problems.

Swarm Intelligence: Swarm intelligence algorithms are inspired by the collective behavior of social insects or other animal societies. Examples include Particle Swarm Optimization (PSO), Ant Colony Optimization (ACO), and Bee Colony Optimization (BCO). They are used for optimization, routing, and clustering problems.

Artificial Neural Networks (ANNs): ANNs are computational models inspired by the structure and function of biological neural networks. Deep Learning, a subset of ANNs, has gained prominence in recent years due to its success in various tasks such as image recognition, natural language processing, and reinforcement learning.

Fuzzy Systems: Fuzzy logic is a form of multi-valued logic that deals with reasoning that is approximate rather than exact. Fuzzy systems are used for modeling and controlling systems with uncertainty and imprecision.

Hybrid Systems: Many real-world problems require the integration of multiple computational intelligence techniques. Hybrid systems combine different CI algorithms to leverage their complementary strengths and address complex problems more effectively.

Differential Evolution Algorithm

Differential Evolution (DE) is an evolutionary algorithm that optimizes a problem by iteratively improving a population of candidate solutions according to a specified fitness function. Here's how DE can be applied to solve linear programming problems:

Initialization: Generate an initial population of candidate solutions (vectors of decision variables).

Evaluation: Evaluate the fitness of each candidate solution using the objective function (Z) and the constraint functions.

Selection: Select the best candidate solutions based on their fitness values.

Mutation: Generate new candidate solutions by perturbing existing solutions using differential mutation operators.

Crossover: Combine information from different candidate solutions to create new candidate solutions.

Replacement: Replace the current population with the newly generated candidate solutions.

Termination: Repeat steps 2-6 until a termination condition is met (e.g., a maximum number of iterations or a satisfactory solution is found).

In the context of linear programming, DE can be used to optimize the coefficients of the objective function (c_1, c_2, \dots, c_n) and/or the decision variables (x_1, x_2, \dots, x_n) while satisfying the linear constraints.

DE operates by iteratively improving candidate solutions through mutation and crossover operations, aiming to find the best combination of decision variables that

minimizes or maximizes the objective function while satisfying the given constraints.

While DE is more commonly used for non-linear optimization problems, it can also be adapted to solve linear programming problems by appropriately defining the fitness function and mutation/crossover operators to handle linear constraints. However, it's worth noting that other optimization algorithms, such as the simplex method or interior point methods, are more commonly used for linear programming due to their efficiency and specific handling of linear constraints. DE may be used in cases where the problem has non-linear components or constraints.

Formal Summary

Title: "Solving Linear Programming Problems with Differential Evolution"

Year: 2022

Summary

This paper presents a comprehensive investigation into the application of Differential Evolution (DE) algorithms for solving linear programming (LP) problems, which are fundamental in various domains such as operations research, economics, and engineering. The authors address the challenge of optimizing linear objective functions subject to linear equality and inequality constraints, a class of problems commonly encountered in real-world optimization scenarios.

The study begins by outlining the formulation of DE as an optimization algorithm inspired by the principles of natural evolution, emphasizing its ability to efficiently explore solution spaces and handle complex, nonlinear, and multimodal optimization problems. Unlike traditional LP solvers, which often rely on specialized techniques tailored to linear constraints, DE offers a versatile and robust approach that can adapt to diverse problem structures and solution spaces.

To apply DE to LP problems, the authors propose a novel fitness evaluation mechanism tailored to the characteristics of linear constraints. The fitness function assesses candidate solutions based on their adherence to the constraints and the optimality of the objective function. Through a series of experiments on benchmark LP problem instances, the efficacy and efficiency of the proposed DE approach are thoroughly evaluated and compared against state-of-the-art LP solvers.

Experimental results demonstrate the ability of DE to efficiently navigate the search space and converge to high-quality solutions, often outperforming or matching the

performance of traditional LP solvers in terms of solution accuracy, convergence speed, and robustness to problem complexity. Moreover, the study explores the impact of various DE parameters, such as population size, mutation strategies, and crossover operators, on the algorithm's performance, providing valuable insights into the design and tuning of DE for LP optimization tasks.

The paper concludes with discussions on the implications of the findings for practitioners and researchers in the field of evolutionary computation and optimization. The successful application of DE to LP problems opens up new avenues for tackling complex optimization challenges in diverse domains, ranging from resource allocation and logistics to finance and industrial engineering. Furthermore, the study highlights potential directions for future research, including the integration of hybrid optimization techniques, the development of parallel and distributed DE algorithms, and the exploration of multi-objective and dynamic LP formulations.

Problem Definition & Mathematical Model

Definition:

Greentech Manufacturing operates three key machines (A, B, and C) to produce solar panels and seeks to minimize energy costs while meeting production and environmental requirements. Machine A can operate for up to 10 hours per day (1000 kWh), Machine B for up to 8 hours per day (640 kWh), and Machine C for up to 6 hours per day (420 kWh). Together, these machines must produce at least 500 units of solar panels daily while A produces 0.3 units of solar panels per kWh, B produces 0.5 units of solar panels per kWh, and C produces 0.2 units of solar panels per kWh. The total carbon emissions from these machines must not exceed 600 units know that Machine A emits 0.15 carbon units per kWh, Machine B emits 0.25 carbon units per kWh and Machine C emits 0.1 carbon units per kWh, and the total energy consumption should be within 1500 kWh per day. The cost function to minimize involves the energy costs of each machine, where Machine A costs \$0.40 per kWh, Machine B costs \$0.20 per kWh, and Machine C costs \$0.30 per kWh. The challenge is to find the optimal energy allocation that minimizes costs while adhering to these operational, production, and environmental constraints.

Decision Variables

X_1 is the amount of energy used in machine A in KWH.

X_2 is the amount of energy used in machine B in KWH.

X_3 is the amount of energy used in machine C in KWH.

Objective Function

Minimize $Z = 0.4X_1 + 0.2X_2 + 0.3X_3$

Constraints

$X_1 \leq 1000, X_2 \leq 640, X_3 \leq 420$ (Boundaries of Decision Variables).

$X_1 + X_2 + X_3 \leq 1500$ (Total Energy Consumption).

$0.3X_1 + 0.5X_2 + 0.2X_3 \geq 500$ (Production of Units per KWH).

$0.15X_1 + 0.25X_2 + 0.1X_3 \leq 600$ (Carbon Emissions per KWH).

References

- <https://ieeexplore.ieee.org/xpl/RecentIssue.jsp?punumber=4235>
- <https://article.sapub.org/10.5923.j.ijee.20190902.03.html>
- [Differential Evolution from Scratch in Python - MachineLearningMastery.com](#)
- Michalewicz, Z., & Schoenauer, M. (1996). "Evolutionary algorithms for constrained parameter optimization problems." *Evolutionary Computation* ۛ