Optimization Project in AI Algorithms

Tom Delahaye
Gabriel Carlotti

November, 2024

1 Project Objectives

This project aims to optimize the placement of biscuits on a dough strip while minimizing defects and maximizing profit. The main objectives are:

1. Describe the problem and its challenges.
2. Formulate and implement the problem using Python.
3. Propose multiple problem-solving approaches.
4. Conclude with reflections on the project.

2 Problem Description and Challenges

The problem involves placing biscuits of different lengths and values on a dough strip that contains defects of various classes (a, b, c). The primary challenges are:

- Avoiding overlapping biscuits.
- Ensuring biscuits do not exceed the dough's length.
- Minimizing defects encountered in the dough segment where a biscuit is placed.
- Maximizing the total value of biscuits placed while penalizing empty spaces.

3 Formulation and Implementation

The problem is implemented in Python using the following classes:

- **Dough** (`Dough.py`): Represents the dough and manages defects and biscuit placements.
- **Biscuit** (`Biscuit.py`): Represents biscuits with attributes such as length, value, and maximum allowable defects.

4 Problem-Solving Approaches

4.1 Genetic Approaches

4.1.1 Base Genetic Algorithm

This is the basic implementation of the genetic algorithm, which uses selection, crossover, and mutation to evolve solutions.

Listing 1: Base Genetic Algorithm

```python
def selection(self):
    fitness_values = [self.fitness(individual) for individual in self.population]
    total_fitness = sum(fitness_values)
    pick = random.uniform(0, total_fitness)
    current = 0
    for individual, fitness_value in zip(self.population, fitness_values):
        current += fitness_value
        if current > pick:
            return individual
```

Result: 613

4.1.2 Genetic Algorithm with Elitism

This version of the genetic algorithm preserves the best individuals (elites) to the next generation, ensuring the quality of solutions improves over time.

Listing 2: Elitism Selection

```python
def selection(self):
    elite_size = int(self.population_size * 0.2)
    ranked_population = sorted(self.population, key=lambda ind: self.fitness(ind), reverse=
        True)
    selected_individuals = ranked_population[:elite_size]
```

Result: 630

4.1.3 Genetic Algorithm with Tournament Selection

Tournament selection selects the best individual from a random subset, promoting diversity and avoiding local optima.

Listing 3: Tournament Selection

```python
def selection(self, elite_size):
    ranked_population = sorted(self.population, key=lambda x: self.fitness(x), reverse=True)
    selected_individuals = [individual for individual, _ in ranked_population[:elite_size]]
    for _ in range(len(ranked_population) - elite_size):
        tournament = random.sample(ranked_population, self.tournament_size)
        best_individual = sorted(tournament, key=lambda x: x[1], reverse=True)[0][0]
        selected_individuals.append(best_individual)
```

Result: 639

4.1.4 Genetic Algorithm with Uniform Crossover

Uniform crossover allows genes to be selected randomly from both parents, enhancing genetic diversity.

Listing 4: Uniform Crossover

```python
def uniform_crossover(parent1, parent2):
    child1, child2 = [], []
    for gene1, gene2 in zip(parent1, parent2):
        if random.random() > 0.5:
            child1.append(gene1)
            child2.append(gene2)
        else:
            child1.append(gene2)
            child2.append(gene1)
    return child1, child2
```

Result: 640

4.2 Greedy Search

Greedy search heuristically places biscuits to maximize immediate gains. We explore three heuristics:

4.2.1 Greedy Search with Value Heuristic

Selects biscuits based on their value.

Listing 5: Greedy Search with Value Heuristic

```python
def greedy_value(dough, biscuits):
    biscuits.sort(key=lambda b: b.value, reverse=True)
    return greedy_placement(dough, biscuits)
```

Result: 675

4.2.2 Greedy Search with Value/Length Heuristic

Selects biscuits based on the ratio of value to length.

Listing 6: Greedy Search with Value/Length Heuristic

```python
def greedy_value_length(dough, biscuits):
    biscuits.sort(key=lambda b: b.value / b.length, reverse=True)
    return greedy_placement(dough, biscuits)
```

Result: 690

4.2.3 Greedy Search with Value/(Length * Max Defects) Heuristic

Selects biscuits based on the ratio of value to the product of length and maximum allowable defects.

Listing 7: Greedy Search with Value/(Length * Max Defects)

```python
def greedy_value_defects(dough, biscuits):
    biscuits.sort(key=lambda b: b.value / (b.length * sum(b.max_defects.values())), reverse=
        True)
    return greedy_placement(dough, biscuits)
```

Result: 672

4.3 Constraint Satisfaction Problem (CSP) with OR-Tools

Constraint Satisfaction Problems (CSPs) involve finding a solution that satisfies a set of constraints. In this context, the constraints are:

- Biscuits must not overlap.
- Biscuits must fit within the dough's length.
- Defects must not exceed the allowable thresholds for each biscuit type.

We use **Google OR-Tools**, a powerful library for solving CSPs, to model and solve this problem. The approach involves:

1. Defining integer variables for the positions of each biscuit.
2. Adding constraints to ensure biscuits do not overlap and respect defect thresholds.
3. Solving the model using the OR-Tools CP-SAT solver.

Listing 8: CSP with OR-Tools

```python
from ortools.sat.python import cp_model

def csp_placement(dough, biscuits):
    model = cp_model.CpModel()
    positions = [model.NewIntVar(0, dough.LENGTH - biscuit.length, f'pos_{i}') for i,
        biscuit in enumerate(biscuits)]
    for i in range(len(biscuits)):
        for j in range(i + 1, len(biscuits)):
            model.Add(positions[i] + biscuits[i].length <= positions[j])

    solver = cp_model.CpSolver()
    status = solver.Solve(model)

    if status == cp_model.OPTIMAL:
        return [(solver.Value(pos), biscuits[i].biscuit_type) for i, pos in enumerate(
            positions)]
```

Result: 715

This approach guarantees an optimal solution by exploring all possible placements within the constraints. However, the computational cost may become prohibitive for very large problem instances.

4.4 Dynamic Programming (DP)

Dynamic Programming (DP) is a method for solving complex problems by breaking them down into simpler subproblems and storing their solutions. In this problem, DP is used to determine the optimal placement of biscuits on the dough.

Steps for Dynamic Programming Approach

1. **Define a DP Table**: The table stores the maximum value achievable for each position on the dough. 2. **Recurrence Relation**: For each position, decide whether to place a biscuit or leave the space empty. 3. **Update Values**: Update the table based on the best possible placement of biscuits.

Listing 9: Dynamic Programming for Biscuit Placement

```
def dp_placement(dough, biscuits):
    dp = [0] * (dough.LENGTH + 1)
    for i in range(dough.LENGTH):
        for biscuit in biscuits:
            if i + biscuit.length <= dough.LENGTH:
                dp[i + biscuit.length] = max(dp[i + biscuit.length], dp[i] + biscuit.value)
    return dp[dough.LENGTH]
```

Result: 715

This approach ensures an optimal solution by considering all possible placements and storing intermediate results to avoid redundant calculations. DP is efficient for structured problems but may face scalability issues with large datasets or complex constraints.

5 Results Review

- **Basic Genetic Algorithm**: 613
- **Genetic Algorithm with Elitism**: 630
- **Genetic Algorithm with Tournament Selection and Crossover**: 639
- **Genetic Algorithm with Uniform Crossover**: 640
- **Greedy Search with Value Heuristic**: 675
- **Greedy Search with Value/Length Heuristic**: 690
- **Greedy Search with Value/(Length * Max Defects) Heuristic**: 672
- **CSP with OR-Tools**: 715
- **Dynamic Programming**: 715

6 Conclusion and Reflections

This project demonstrated the effectiveness of various optimization techniques. The key takeaways are:

- **Genetic Algorithms** provide flexible solutions but require fine-tuning of parameters and are computationally intensive.
- **Greedy Search** is fast and effective, with the *Value/Length* heuristic performing the best among the greedy approaches.
- **CSP and Dynamic Programming** achieved the best results (715) by ensuring optimal solutions, though they may struggle with scalability for larger datasets.

Selecting the appropriate method depends on the trade-off between solution quality and computational efficiency.