Ant Colony Optimization for Text Summarization: Technical Report

Executive Summary

This report analyzes a Python implementation that applies Ant Colony Optimization (ACO) to the problem of extractive text summarization. The system selects important sentences from a document to create concise summaries by simulating how ant colonies find optimal paths through pheromone deposition and evaporation. The implementation demonstrates how concepts from swarm intelligence can be effectively applied to natural language processing tasks.

1. Introduction to ACO for Text Summarization

Ant Colony Optimization is a metaheuristic inspired by the foraging behavior of ant colonies. In nature, ants deposit pheromones along paths as they search for food, with shorter, more efficient paths accumulating stronger pheromone trails over time, which influences other ants to follow these paths.

In the context of text summarization, ACO works by:

1. Treating sentences as nodes in a graph
2. Using "ants" to traverse the document and select important sentences
3. Updating pheromone trails based on solution quality
4. Converging toward an optimal summary after multiple iterations

2. System Architecture Overview

The system consists of three main components:

1. **Sample Documents Module** (`sample_documents.py`): Contains example documents and their human-written summaries
2. **ACO Summarizer Module** (`aco_summarizer.py`): Implements the core ACO algorithm for text summarization
3. **Main Application** (`main.py`): Demonstrates the summarizer's functionality

2.1 Sample Documents

The sample documents cover three topics: Artificial Intelligence, Climate Change, and Quantum Computing. Each document includes the original text and a human-written summary for comparison.

python

```python
DOCUMENTS = [
    {
        "id": 1,
        "title": "Artificial Intelligence",
        "text": """Artificial Intelligence (AI) is intelligence demonstrated by machines, as op
        AI research has been defined as the field of study of intelligent agents, which refers
        ...""",
        "summary": """Artificial Intelligence (AI) is machine-demonstrated intelligence, define
        ..."""
    },
    # Additional documents...
]
```

3. ACO Summarizer Implementation

3.1 Core Parameters

The ACO algorithm uses several parameters that control its behavior:

python

```python
def __init__(self, num_ants=10, alpha=1.0, beta=2.0, rho=0.1, q0=0.9,
             max_iterations=30, compression_ratio=0.3):
    """
    Initialize the ACO Summarizer with simplified parameters.

    Parameters:
    -----------
    num_ants : int
        Number of ants in the colony (more ants = more exploration)
    alpha : float
        Importance of pheromone (higher = follow other ants more)
    beta : float
        Importance of heuristic information (higher = greedier selection)
    rho : float
        Pheromone evaporation rate (how quickly trails fade)
    q0 : float
        Probability of exploitation vs exploration (higher = more greedy)
    max_iterations : int
        Maximum number of iterations (more = better but slower)
    compression_ratio : float
        Target summary length as a fraction of original document length
    """
```

These parameters control the exploration-exploitation tradeoff and have significant impact on the quality of the generated summaries.

3.2 Text Preprocessing

The system preprocesses text by:

1. Tokenizing the document into sentences
2. Cleaning each sentence (removing special characters, converting to lowercase)
3. Creating sentence vectors using a bag-of-words approach

```python
def preprocess_text(self, text):
    """
    Preprocess the text by tokenizing into sentences and cleaning.
    """
    # Split text into sentences
    sentences = sent_tokenize(text)

    # Clean sentences
    cleaned_sentences = []
    for sentence in sentences:
        # Remove special characters and numbers
        sentence = re.sub(r'[^a-zA-Z\s]', '', sentence)
        # Convert to lowercase
        sentence = sentence.lower()
        # Remove extra whitespace
        sentence = re.sub(r'\s+', ' ', sentence).strip()

        if sentence:  # Only add non-empty sentences
            cleaned_sentences.append(sentence)

    return sentences, cleaned_sentences
```

3.3 Sentence Scoring

A key component is the calculation of heuristic scores for each sentence based on multiple features:

python

```python
def calculate_sentence_scores(self, sentences, cleaned_sentences):
    """
    Calculate importance scores for each sentence based on multiple features.
    """
    # Feature 1: Position score (first and last sentences are important)
    position_scores = np.zeros(n)
    for i in range(n):
        # Sentences at the beginning and end get higher scores
        position_scores[i] = 1.0 - abs(i - n/2) / (n/2)

    # Feature 2: Length score (not too short, not too long)
    length_scores = np.zeros(n)
    for i in range(n):
        words = cleaned_sentences[i].split()
        length = len(words)
        # Penalize very short or very long sentences
        if length < 3:
            length_scores[i] = 0.3
        elif length > 30:
            length_scores[i] = 0.5
        else:
            length_scores[i] = 1.0

    # Feature 3: Centrality score (how similar a sentence is to others)
    centrality_scores = np.sum(similarity_matrix, axis=1) / n

    # Combine all features with weights
    for i in range(n):
        scores[i] = (
            0.3 * position_scores[i] +  # Position weight
            0.2 * length_scores[i] +    # Length weight
            0.5 * centrality_scores[i]  # Centrality weight
        )
```

The system uses three key features to score sentences:

1. **Position**: Sentences at the beginning and end are given higher scores (weight: 0.3)
2. **Length**: Moderate-length sentences (3-30 words) are preferred (weight: 0.2)
3. **Centrality**: Sentences similar to many others receive higher scores (weight: 0.5)

3.4 Ant Solution Generation

Each "ant" in the colony generates a candidate summary by selecting sentences:

```python
def ant_solution(self, n, heuristic_scores, pheromones, target_length):
    """
    Generate a solution by a single ant.
    """
    # Start with a random sentence
    current = random.randint(0, n-1)
    selected = [current]

    # Select remaining sentences until we reach target length
    while len(selected) < target_length:
        # Calculate probability of selecting each sentence
        probabilities = np.zeros(n)

        for j in range(n):
            if j not in selected:  # Only consider unselected sentences
                # ACO formula: τ^α × η^β
                # τ (tau) = pheromone level
                # η (eta) = heuristic value
                tau = pheromones[current, j]
                eta = heuristic_scores[j]

                probabilities[j] = (tau ** self.alpha) * (eta ** self.beta)
```

The probability of selecting each sentence follows the standard ACO formula: $\tau^\alpha \times \eta^\beta$, where:

- τ (tau) is the pheromone level
- η (eta) is the heuristic value
- α (alpha) controls the importance of pheromone trails
- β (beta) controls the importance of heuristic scores

3.5 Pheromone Update

After each iteration, pheromone levels are updated based on the quality of solutions:

python

```python
def update_pheromones(self, pheromones, all_solutions, best_solution, best_quality):
    """
    Update pheromone levels based on the best solution.
    """
    # Step 1: Evaporation - reduce all pheromones
    pheromones = (1 - self.rho) * pheromones

    # Step 2: Deposit new pheromones for the best solution
    for i in range(len(best_solution) - 1):
        current = best_solution[i]
        next_sentence = best_solution[i + 1]
        # Add pheromone proportional to solution quality
        pheromones[current, next_sentence] += self.rho * best_quality

    return pheromones
```

This follows the standard ACO approach:

1. **Evaporation**: All pheromone levels are reduced by factor (1-ρ)
2. **Deposition**: New pheromones are added along the best solution path

4. Main Summarization Process

The full summarization process combines all these components:

```python
def summarize(self, text):
    """
    Generate a summary using ACO.
    """
    # Step 1: Preprocess text
    original_sentences, cleaned_sentences = self.preprocess_text(text)
    n = len(original_sentences)

    # Step 2: Calculate target summary length (30% of original by default)
    target_length = max(1, int(n * self.compression_ratio))

    # Step 3: Calculate sentence importance scores
    heuristic_scores = self.calculate_sentence_scores(original_sentences, cleaned_sentences)

    # Step 4: Initialize pheromone matrix
    pheromones = np.ones((n, n)) * 0.1

    # Step 5: Initialize best solution tracking
    best_solution = None
    best_quality = 0

    # Step 6: Main ACO loop - multiple iterations of ant colony
    for iteration in range(self.max_iterations):
        all_solutions = []

        # Each ant finds a solution
        for ant in range(self.num_ants):
            # Get this ant's solution
            solution, quality = self.ant_solution(n, heuristic_scores, pheromones, target_lengt
            all_solutions.append((solution, quality))

            # Update best solution if better
            if quality > best_quality:
                best_solution = solution
                best_quality = quality

        # Update pheromones based on results
        pheromones = self.update_pheromones(pheromones, all_solutions, best_solution, best_qual
```

5. Performance Analysis

The system processes documents with the following outputs:

python

```python
def print_summary_result(doc, aco_summary, selected_indices):
    """Print the ACO-generated summary."""
    # Calculate compression ratio
    original_words = len(doc['text'].split())
    summary_words = len(aco_summary.split())
    compression = summary_words / original_words * 100

    print(f"\nSUMMARY STATISTICS:")
    print(f"- Original length: {original_words} words")
    print(f"- Summary length: {summary_words} words")
    print(f"- Compression ratio: {compression:.1f}%")
```

The system aims for a compression ratio of approximately 30% of the original document length, as specified by the `compression_ratio` parameter.

6. Advantages and Limitations

Advantages:

1. **Adaptability**: The ACO approach can adapt to different document types
2. **Multi-criteria selection**: Uses multiple features to evaluate sentence importance
3. **Parameter tuning**: Allows adjustment of parameters to optimize for different summarization goals
4. **No training required**: Works without needing labeled training data

Limitations:

1. **Computationally intensive**: Multiple iterations of the entire ant colony can be slow for large documents
2. **Extractive only**: Cannot paraphrase or generate new content
3. **Basic sentence representation**: Uses simple bag-of-words vectors rather than more advanced embeddings
4. **Language specific**: Currently focused on English with English stopwords

7. Potential Improvements

Several enhancements could improve the system:

1. **Advanced sentence embeddings**: Replace bag-of-words with transformers or word embeddings
2. **Additional features**: Consider sentence similarity to title, presence of named entities, etc.
3. **Parameter optimization**: Use techniques like grid search to find optimal ACO parameters
4. **Multi-document summarization**: Extend the approach to summarize multiple related documents

5. **Abstractive elements**: Combine with neural generation to rephrase selected content

8. Conclusion

The ACO text summarization system demonstrates an effective application of swarm intelligence to natural language processing. By simulating how ant colonies find optimal paths, the algorithm can identify and extract the most important sentences from a document to create concise summaries.

The implementation provides a solid foundation for extractive summarization and could be further extended with more advanced features and techniques. The modular design allows for easy experimentation with different parameters and sentence scoring mechanisms.