TECHNICAL SCIENCES

OPTIMIZATION ALGORITHMS FOR PROCESSING DOCUMENT FLOW DATA ON THE HADOOP PLATFORM

Mammadli A.
Abdullazade N.
Computer engineering department
Azerbaijan State Oil and Industry University
https://doi.org/10.5281/zenodo.19573269

Abstract

In the era of digital transformation, document flow systems generate rapidly increasing volumes of structured, semi-structured, and unstructured data. Traditional data processing approaches are insufficient to efficiently manage, analyze, and extract meaningful insights from such large-scale datasets. This paper explores the application of optimization algorithms for processing document flow data on the Apache Hadoop platform. The study examines the effectiveness of Greedy and Genetic algorithms, along with other optimization techniques including Simulated Annealing, Ant Colony Optimization, and Particle Swarm Optimization. A comparative analysis is conducted based on key performance metrics such as execution time, memory usage, accuracy, and scalability. The findings indicate that while Greedy algorithms provide faster results, Genetic algorithms achieve higher accuracy and better scalability. Furthermore, the integration of hybrid optimization models demonstrates significant potential in improving both efficiency and performance in distributed big data environments. The results highlight the importance of combining optimization techniques with scalable platforms like Hadoop for advanced document flow management systems.

In the digital transformation era, document flow systems generate large volumes of diverse data that require efficient processing. Traditional methods are insufficient for handling such data at scale. This study investigates the use of optimization algorithms on the Hadoop platform for document flow data processing. Greedy and Genetic algorithms, along with other techniques, are analyzed based on performance metrics such as execution time, memory usage, accuracy, and scalability. The results show that Greedy algorithms are faster, while Genetic algorithms provide better accuracy and scalability. The study highlights the effectiveness of hybrid approaches in improving performance in distributed big data environments.

Keywords: Hadoop, Greedy, Genetic, Simulated Annealing, Ant Colony, Particle Swarm

Introduction

Modern DMS provide functionalities such as capturing, storing, retrieving, indexing, classifying, and distributing digital documents. Key features include audit trails, version control, access management, and integration with ERP systems. Recent research focuses on the use of artificial intelligence, particularly natural language processing (NLP), for automated classification and summarization. However, many DMS still face challenges in scalability and integration with big data analytics.

However, despite the advanced capabilities of modern Document Management Systems (DMS), the rapid growth of data volume and complexity has exposed their limitations, particularly in terms of scalability and integration with big data analytics. As a result, there is a growing need for more powerful and distributed computing frameworks that can efficiently handle large-scale document flow data. In this context, Apache Hadoop emerges as a suitable solution, providing scalable and fault-tolerant infrastructure for processing and analyzing massive datasets. [2, p.256-257]

Hadoop platform and its components

Apache Hadoop is an open-source framework designed for distributed storage and processing of big data. Its core components—HDFS (Hadoop Distributed File System), MapReduce (for data processing), and YARN (for resource management)—provide a robust, scalable, and fault-tolerant system. Hadoop supports both structured and unstructured data formats. Its ecosystem includes powerful tools such as Hive (for SQL queries), Pig (for data transformation), and Spark (for fast in-memory analytics). The modular and scalable architecture of Hadoop makes it suitable for integration with machine learning pipelines and real-time data streaming frameworks such as Kafka.

The Hadoop ecosystem includes the following core components:

- **HDFS:** Stores large files by dividing them into blocks across multiple nodes
- **MapReduce:** Enables parallel processing by splitting tasks across machines
- **YARN:** Manages computational resources and task scheduling
- **Hive and Pig:** Provide high-level data processing using SQL and scripting
- **Spark:** Enables fast analytics for real-time and iterative tasks

Studies in healthcare, e-government, and finance sectors have demonstrated the effectiveness of Hadoop in document-oriented analytics. [1, p.107-113]

Optimization Algorithms in Big Data

Optimization algorithms help improve execution time, resource utilization, and result quality in large-scale systems:

- **Greedy Algorithms:** Simple and fast, selecting locally optimal solutions at each step; suitable when speed is a priority
- **Genetic Algorithms:** Based on natural selection and evolution, effective in exploring large solution spaces; suitable for multi-objective problems
- **Simulated Annealing:** Can accept worse solutions probabilistically to escape local minima
- **Ant Colony Optimization:** Inspired by ant behavior, useful for path and routing optimization
- **Particle Swarm Optimization:** Based on social behavior models, effective in clustering and classification problems

Hybrid forms of these algorithms are increasingly being used. [4, p.314-315]

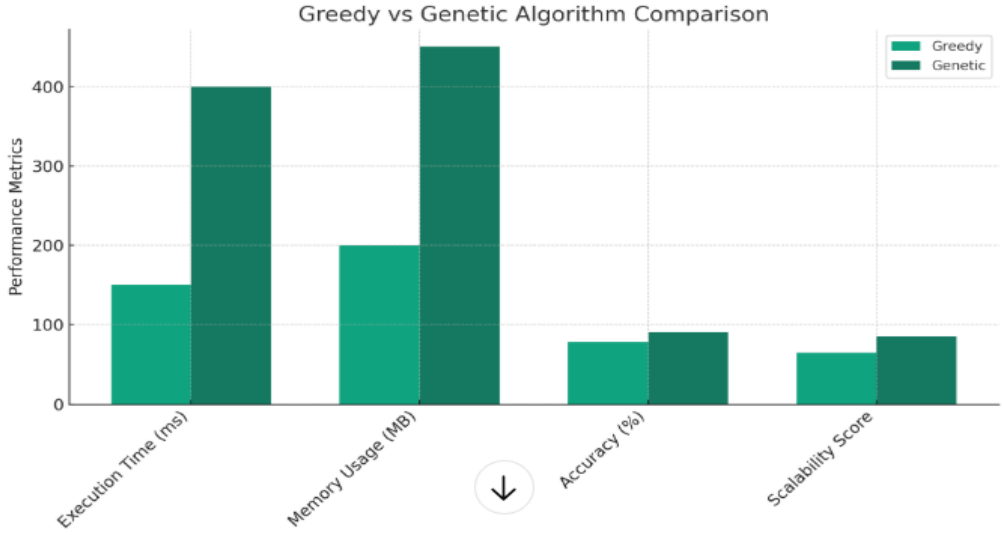

Fig. 1. Greedy vs Genetic Algorithm Comparison

The comparative performance graph of Greedy and Genetic algorithms for document flow optimization is presented above (Fig. 1). The comparison is based on the following criteria:

- **Execution Time:** Greedy is faster
- **Memory Usage:** Genetic requires more memory
- **Accuracy:** Genetic provides higher accuracy
- **Scalability:** Genetic is more flexible

Example-1. The following code implements a simple document ranking system in Java (Fig. 2).

It stores documents with their scores, then sorts them in descending order based on those scores, and prints the results. The logic follows a greedy approach, meaning it prioritizes the highest-scoring documents first without applying any complex optimization or constraints.

In short, the program's purpose is to rank and display documents from most relevant (highest score) to least relevant (lowest score).

```java
// GreedyDocumentSorter.java (sorts documents based on simple scores)
public class GreedyDocumentSorter {
    public static void main(String [] args) {
      // Simulated list of document scores.
      Map<String, Integer> docScores = new HashMap<>();
      docScores.put("doc1.txt", 34);
      docScores.put("doc2.txt", 87);
      docScores.put("doc3.txt", 12);
      docScores.put("doc4.txt", 65);

      docScores.entrySet().stream()
          .sorted(Map.Entry.<String, Integer>comparingByValue().reversed())
          .forEach(e -> System.out.println(e.getKey() + " -> " + e.getValue()));
      }
}
```

Fig. 2. Greedy Algorithm with MapReduce (Java)

Example-2. The following code implements a Genetic Algorithm for optimizing document ordering using Apache Spark (PySpark).

It simulates a set of documents and tries to find the best possible arrangement (solution) based on a defined fitness function. The algorithm works by generating multiple candidate solutions (populations), evaluating them, and iteratively improving them through genetic operations like selection, crossover, and mutation (Fig. 3).

In general, the program:

- Creates random document sequences (solutions)

- Evaluates how “good” each sequence is using a fitness function
- Selects the best solutions
- Combines them (crossover) and slightly changes them (mutation)
- Repeats this process over several generations

At the end, it outputs the best document sequence found.

```
import random
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName("GeneticDocFlow").getOrCreate()

# Observed documents
documents = [f"doc{i}" for i in range(1, 11)]

# Fitness function
def fitness(solution):
    return sum(ord(c) for doc in solution for c in doc) % 100

# Generate initial population
def generate_population(size):
    return [random.sample(documents, len(documents)) for _ in range(size)]

# Crossover operation
def crossover(p1, p2):
    point = len(p1) // 2
    return p1[:point] + [x for x in p2 if x not in p1[:point]]

# Mutation operation
def mutate(solution):
    i, j = random.sample(range(len(solution)), 2)
    solution[i], solution[j] = solution[j], solution[i]
    return solution

population = generate_population(10)

for generation in range(20):
    population = sorted(population, key=fitness, reverse=True)
    parents = population[:2]
    children = [mutate(crossover(parents[0], parents[1])) for _ in range(8)]
    population = parents + children

# Best result
best_solution = max(population, key=fitness)
print("Best solution:", best_solution)
```

Fig. 3. Genetic Algorithm (Python with PySpark)

Simulated Annealing (SA)

A practical application of the Simulated Annealing approach involves optimizing the distribution of documents across nodes in a Hadoop cluster. The main objective of this problem is to allocate documents in such a way that the overall execution time is minimized while maintaining proper load balancing among the nodes (Fig. 4).

The process begins with an initial random distribution of documents across the available nodes. Then, iterative improvements are performed by making small adjustments, such as transferring a document from one node to another. Unlike traditional optimization methods, Simulated Annealing allows the acceptance of worse solutions with a certain probability, especially at higher temperature levels. This mechanism helps the system avoid local optima and increases the likelihood of reaching a more optimal global solution over time. [3, p.137-144]

```
documents = [10, 20, 30, 40, 50] # sizes
nodes = 3

def cost(sol):
    loads = [0*nodes
    for d, n in zip(documents, sol):
        loads[n] += d
    return max(loads) - min(loads)  # imbalance

current = [random.randint(0, nodes-1) for _ in documents]
T = 100
for _ in range(1000):
    new = current[:]
    i = random.randint(0, len(documents)-1)
    new[i] = random.randint(0, nodes-1)
    delta = cost(new) - cost(current)
       if delta < 0 or random.random) < math.exp(-delta / T):
           current = new
    T *= 0.99 # cooling
print('Optimized': current)
```

Fig. 4. Simulated Annealing algorithm

Ant Colony Optimization (ACO)

A practical application of the Ant Colony Optimization approach involves finding the optimal processing path in document workflows. In this problem, documents are required to pass through multiple stages, such as parsing, classification, and storage. The main objective is to determine the shortest and most efficient path that minimizes processing time (Fig. 5).

The method is based on the behavior of ants exploring possible paths between nodes. During the process, ants probabilistically select different routes, and paths that lead to better results accumulate higher levels of pheromones. Over time, these pheromone trails guide subsequent selections toward more optimal solutions, while less efficient paths gradually lose their significance and disappear. This iterative process enables the system to converge toward an optimal or near-optimal processing path. [5, p.157-156]

```
import numpy as np
pheromone = np.ones(4,4))
distance = np.array([
  [0, 2, 2, 5],
  [2, 0, 3, 4],
  [2, 3, 0, 1],
  [5, 4, 1, 0]]
alpha, beta = 1, 2
def choose_path(currentt):
  probs = ]]
  for 1 in range({_pheromone<current[i]*ᐞgᵐᵃ) *({1/distance(current][1])**
      ᴱᶻba if distance(current[[i] else 0))
  probs = probs / np.sum(probs)
return np.random.choice(range(4), p-probs)
path = [0]
for _ in range(3)
path.append(choose_path(path[-1]))
```

Fig. 5. Ant Colony Optimization (ACO) algorithm

Particle Swarm Optimization (PSO)

A practical application of the Particle Swarm Optimization approach involves determining the optimal allocation of computational resources, such as CPU and RAM, for document processing tasks. The primary objective is to minimize latency while maximizing system throughput (Fig. 6).

In this approach, each particle represents a potential resource allocation configuration within the system.

The particles move through the solution space by continuously updating their positions based on both their individual best-known solutions and the globally best solution found by the swarm. This collective behavior enables the system to efficiently converge toward an optimal or near-optimal resource distribution, improving overall system performance in distributed environments. [6, p.2-3]

```
import random
particles = [radom.uniform(0, 10) for _ in range(10)]
velocity = [0]*10
best_personal = particles[:]
best_global = min(particles)
def cost(x):
  return (x-5)**>2  # optimal resurs 5
for _ in range(100):
  for i in range(10):
    velocity[] = 0.5*velocity[i] + 0.8*(best_personal[i]-particles[i])
    particles[i] += velocity[i]
    if cost(particles[i]) < cost(best_personal[i]):
      best_personal[i] = particles[i]
  best_global = min(best_personal, key=cost)
print("Optimal resource value:." best_global)
```

Fig. 6. Particle Swarm Optimization (PSO) algorithm

Results

The results of this study demonstrate that optimization algorithms play a crucial role in improving the efficiency of document flow data processing on the Hadoop platform. The comparative analysis of Greedy and Genetic algorithms, along with other optimization techniques, highlights significant differences in performance based on key evaluation metrics such as execution time, memory usage, accuracy, and scalability.

The findings indicate that the Greedy algorithm performs efficiently in terms of execution time and computational simplicity. It is particularly suitable for scenarios where quick decision-making is required. However, its limitation lies in its tendency to reach only locally optimal solutions, which may reduce overall system performance in complex problem spaces.

In contrast, the Genetic algorithm provides higher accuracy and better scalability, making it more suitable for large-scale and complex optimization problems. Despite its advantages, it requires greater computational resources and memory usage, which may affect system efficiency in resource-constrained environments.

Additionally, optimization techniques such as Simulated Annealing, Ant Colony Optimization, and Particle Swarm Optimization have shown promising results in addressing specific challenges such as load balancing, routing optimization, and resource allocation in distributed systems. These methods are particularly effective in dynamic environments where adaptive solutions are required.

Overall, the results suggest that relying on a single optimization algorithm is not sufficient to address all challenges associated with large-scale document flow processing. Hybrid approaches that combine multiple algorithms can significantly enhance both performance and accuracy while ensuring better resource utilization.

In conclusion, the integration of optimization algorithms within the Hadoop ecosystem provides an effective solution for managing and processing large-scale document flow data. The use of hybrid optimization models, combined with scalable distributed frameworks, offers a promising direction for improving system performance and supporting advanced data-driven decision-making processes.

References:

1. Dean, J., & Ghemawat, S. (2008). MapReduce: Simplified Data Processing on Large Clusters. Communications of the ACM, 51(1), 107–113.

2. White, T. (2015). Hadoop: The Definitive Guide (4th ed.). O'Reilly Media.

3. Gandomi, A. H., & Haider, M. (2015). Beyond the Hype: Big Data Concepts, Methods, and Analytics. International Journal of Information Management, 35(2), 137–144.

4. Goldberg, D. E. (1989). Genetic Algorithms in Search, Optimization, and Machine Learning. Addison-Wesley.

5. Dorigo, M., & Stützle, T. (2004). Ant Colony Optimization. MIT Press.

6. Kennedy, J., & Eberhart, R. (1995). Particle Swarm Optimization. Proceedings of IEEE International Conference on Neural Networks.