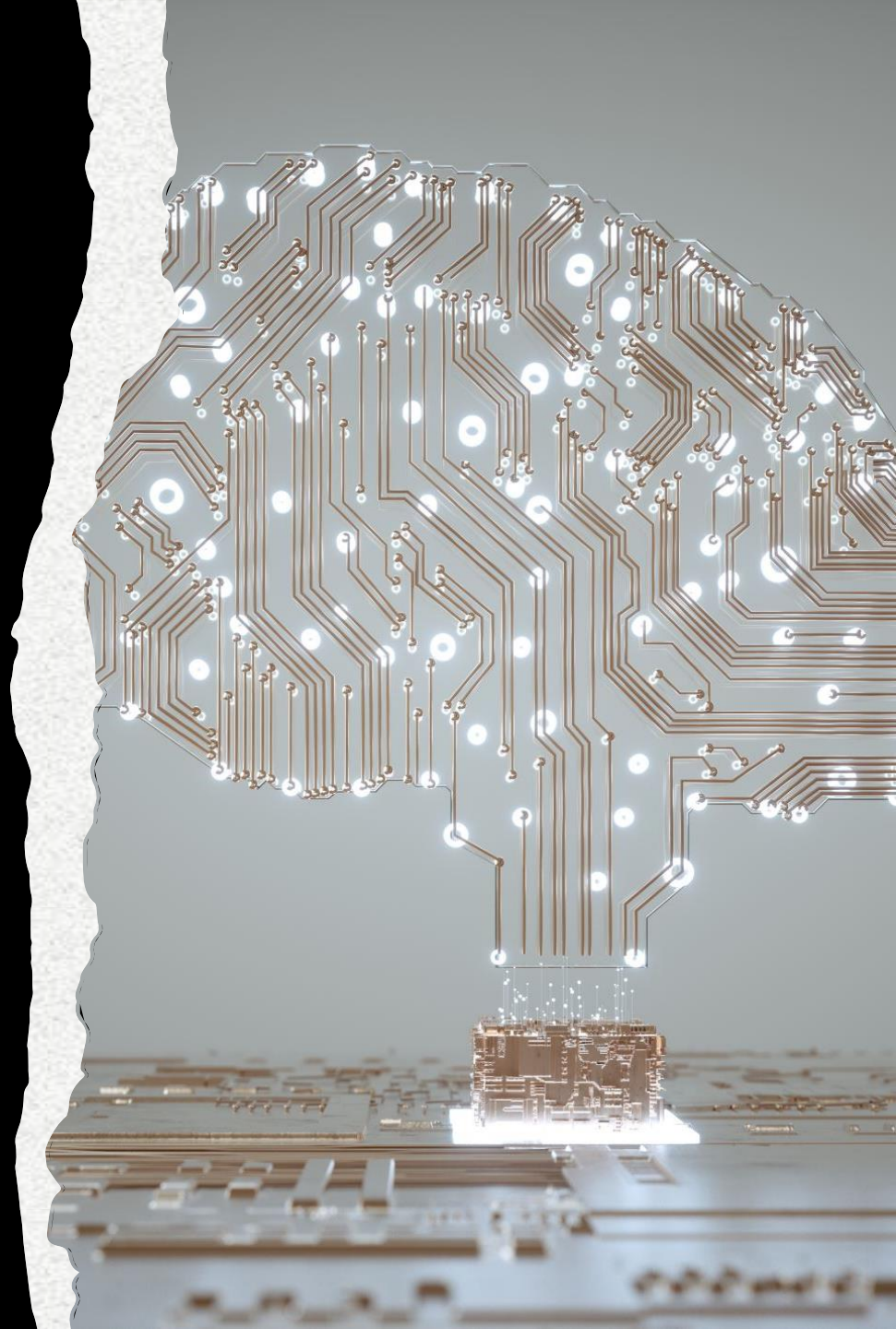
Inteligência Artificial 2023/2024

Topic 3B: Optimization Problems

- Pedro Filipe Vale Gomes (202108825)
- Simão Queirós Rodrigues (202005700)



Book scanning

Objective: develop a strategic plan to optimize the scanning process of books from various libraries within a limited timeframe. This involves:

- **Maximizing the total score** of scanned books, taking into consideration the unique scores assigned to each book;

- **Efficiently managing library signups** by sequencing the registration of libraries such that their combined throughput (number of books scanned per day) is maximized, despite the constraint that only one library can be signed up at a time;

- **Adhering to the time constraints** of the given number of days, ensuring that the scanning operation is completed within this period to maximize the overall score;

- **Optimizing book selection** across libraries to ensure that each book is scanned only once, despite its availability in multiple libraries, to earn its score;

- **Leveraging parallel scanning capabilities** post library signup, to scan books from multiple libraries concurrently, within each library's daily scanning capacity.

Optimization Problem

Solution Representation: A sequence in which libraries are signed up and a selection of books to scan from each library.

Neighborhood/Mutation Functions: Alter the order of library signups or change the selection of books to scan, aiming to find a better solution.

Hard Constraints:

- A book can be scanned only once;
- Only one library can be signed up at a time;
- Scanning and signup processes must be completed within the given number of days.

Evaluation Functions: The sum of scores of all scanned books.

Input File Structure:

The first line contains three integers: the total number of books (*B*), the total number of libraries (*L*), and the total number of days (*D*).

The second line lists *B* integers, representing the scores of each book.

Following that, for each library, there are two lines:

- The first line for a library contains three integers: the number of books in the library (*N*), the signup process duration for the library (*T*), and the number of books that can be shipped per day from the library (*M*). one library can be signed up at a time; and signup processes must be completed within the given number of days.
- The second line lists the *N* book IDs present in the library.

Implementation Overview

Programming Languague: Python

Development Environment: PyQt5 for GUI development, providing a user-friendly interface for importing library data and executing the book scanning algorithm

Data Structures:

Library Class:

- Represents each library with an ID, signup time, daily book scanning capacity, and a list of books.
- Facilitates managing the signup process and the selection of books for scanning from each library

Book Class:

- Represent each book with na ID and a score
- User to keep track of the individual scores of books

```python
class Book:
    def __init__(self, id, score):
        self.id = id
        self.score = score

    def __repr__(self):
        return f"ID: {self.id} - Score: {self.score}"
```

```python
class Library:
    def __init__(self, id, signup_time, books_per_day):
        self.id = id
        self.signup_time = signup_time
        self.books_per_day = books_per_day
        self.books = []

    def add_book(self, book):
        self.books.append(book)

    def __repr__(self):
        return f"Library(ID: {self.id}, Signup Time: {self.
        signup_time}, Books/Day: {self.books_per_day}, Books:
        {len(self.books)})"
```

Code Snippets

- Reading Input and Initial Setup
- Library Signup and Book Scanning Logic

```
num_books = int(a[0])
num_libraries = int(a[1])
num_days = int(a[2])
```

```
books = []
for book in b:
    books.append(Book(len(books), int(book)))
```

```
libraries = []
added_lib = False
for line in lines:
    if len(line):
        line = line.split(' ')
        if not added_lib:
            libraries.append(Library(len(libraries), int(line[1]),
            int(line[2])))
            added_lib = True
        else:
            for book_id in line:
                libraries[-1].add_book(books[int(book_id)])
            added_lib = False
```

Bibliography

- Online Qualification Round of Hash Code 2020

Greedy Algorithm

```python
def apply_greedy_algorithm(input_file):
    D, libraries = read_input_file(input_file)

    for library in libraries:
        library.sort_books()
    libraries.sort(key=lambda lib: sum(book.score for book in lib.books) / lib.signup_time, reverse=True)

    days_remaining = D
    signup_process = []
    books_scanned = set()
    total_score = 0

    for library in libraries:
        if days_remaining <= 0 or days_remaining < library.signup_time:
            break
        days_remaining -= library.signup_time

        books_to_scan = []
        for book in library.books:
            if book.id not in books_scanned:
                books_to_scan.append(book.id)
                books_scanned.add(book.id)
                total_score += book.score
        signup_process.append((library, books_to_scan))

    base_name = os.path.splitext(os.path.basename(input_file))[0]
    output_path = f'proj/output/{base_name}_greedy.txt'
    write_solution(output_path, signup_process)

    print(f'Greedy algorithm applied for {base_name} with total score: {total_score}')
```

Simulated Annealing

```python
def apply_simulated_annealing(input_file):
    D, libraries = read_input_file(input_file)

    current_solution = initial_solution(D, libraries)
    current_score = score_solution(current_solution, D)

    T = 1.0
    T_min = 0.001
    alpha = 0.9

    while T > T_min:
        i = 1
        while i <= 100:
            new_solution = neighbor_solution(current_solution, libraries, D)
            new_score = score_solution(new_solution, D)

            delta = new_score - current_score

            acceptance_probability = math.exp(delta / T) if delta < 0 else 1

            if acceptance_probability > random.random():
                current_solution = new_solution
                current_score = new_score

            i += 1

        T *= alpha

    base_name = os.path.splitext(os.path.basename(input_file))[0]
    output_path = f'proj/output/{base_name}_simulated_annealing.txt'
    write_solution(output_path, current_solution)

    print(f'Simulated Annealing applied for {base_name} with a total score of {current_score}')
```

Hill Climbing

```python
def apply_hill_climbing(input_file):
    D, libraries = read_input_file(input_file)

    current_solution = initial_solution(D, libraries)
    current_score = score_solution(current_solution, D)

    best_solution = current_solution
    best_score = current_score

    while True:
        neighbor_sol = neighbor_solution(current_solution, libraries, D)
        neighbor_score = score_solution(neighbor_sol, D)

        if neighbor_score > current_score:
            current_solution = neighbor_sol
            current_score = neighbor_score

            if current_score > best_score:
                best_solution = current_solution
                best_score = current_score
        else:
            break

    base_name = os.path.splitext(os.path.basename(input_file))[0]
    output_path = f'proj/output/{base_name}_hill_climbing.txt'
    write_solution(output_path, best_solution)

    print(f'Hill Climbing applied for {base_name} with a total score of {best_score}')
```

Genetic Algorithm

```python
NUM_GENERATIONS = 100
POPULATION_SIZE = 50
MUTATION_RATE = 0.1
def apply_genetic_algorithm(input_file):
    num_generations = NUM_GENERATIONS
    population_size = POPULATION_SIZE

    D, libraries = read_input_file(input_file)
    population = generate_initial_population(population_size, libraries)

    for generation in range(num_generations):
        for chromosome in population:
            calculate_fitness(chromosome, D)

        population.sort(key=lambda x: x.score, reverse=True)

        best_solution = population[0]

    best_solution = population[0]

    # Extract the solution from the best chromosome
    solution = [(library, []) for library in best_solution.libraries]

    # Populate the books to scan for each library based on the actual solution generated
    days_remaining = D
    for i, (library, _) in enumerate(solution):
        days_remaining -= library.signup_time
        if days_remaining <= 0:
            solution = solution[:i]  # Remove libraries for which there's not enough time
            break
        books_to_scan = []
        for book in library.books:
            if book.id not in [book_id for _, book_ids in solution for book_id in book_ids]:
                books_to_scan.append(book.id)
                if len(books_to_scan) >= library.books_per_day * days_remaining:
                    break
        solution[i] = (library, books_to_scan)

    base_name = os.path.splitext(os.path.basename(input_file))[0]
    output_path = f'proj/output/{base_name}_genetic_algorithm.txt'
    write_solution(output_path, solution)
    print(f'Genetic Algorithm applied for {base_name} with a total score of {best_solution.score}')
```

Helper Functions

initial_solution: Generates a random starting point for the optimization algorithm by selecting a list of libraries to sign up.

neighbor_solution: Creates a variation of the current solution, attempting minor changes to explore nearby solutions.

score_solution: Calculates the total score of a given solution based on the unique books scanned within the time constraints.

write_solution: Writes the details of the signup process and the selected books to an output file in the specified format.

```python
def initial_solution(D, libraries):
    solution = []
    days_remaining = D
    for library in libraries:
        if days_remaining - library.signup_time >= 0:
            solution.append((library, []))
            days_remaining -= library.signup_time
    random.shuffle(solution)
    return solution
```

```python
def neighbor_solution(solution, libraries, D):
    if not solution:
        return solution

    neighbor = solution[:]
    idx = random.randrange(len(neighbor))
    library, _ = neighbor[idx]

    if random.random() < 0.5:
        idx_swap = random.randrange(len(neighbor))
        neighbor[idx], neighbor[idx_swap] = neighbor[idx_swap], neighbor[idx]
    else:
        random_books = random.sample(library.books, min(len(library.books), library.books_per_day * (D - library.signup_time)))
        neighbor[idx] = (library, random_books)

    return neighbor
```

```python
def score_solution(solution, D):
    score = 0
    books_scanned = set()
    days_remaining = D
    for library, books in solution:
        days_remaining -= library.signup_time
        if days_remaining <= 0:
            break

        num_scanned_books = min(days_remaining * library.books_per_day, len(books))
        for book in books[:num_scanned_books]:
            if book.id not in books_scanned:
                score += book.score
                books_scanned.add(book.id)
    return score
```

```python
def write_solution(output_path, signup_process):
    if not os.path.exists('proj/output'):
        os.makedirs('proj/output')
    with open(output_path, 'w') as f:
        f.write(f"{len(signup_process)}\n")
        for library, book_ids in signup_process:
            f.write(f"{library.id} {len(book_ids)}\n")
            book_ids_str = ' '.join(str(book_id) for book_id in book_ids)
            f.write(f"{book_ids_str}\n")
```

Conclusions

Algorithm / File	A	B	C	D	E	F
Greedy	21	9000000	5645747	4815395	7616025	5561306
Simulated Annealing	21	5581800	894135	1288950	813619	950410
Hill Climbing	21	1694800	115041	13975	366151	948113
Genetic Algorithm	21	8700000	982711	4359095	2736447	1544906

Greedy Algorithm:

Achieves the highest scores in most files, indicating good performance in terms of solution quality.

Despite its simplicity and speed, it manages to provide competitive solutions in most cases.

Suitable for scenarios where quick decision-making and high scores are prioritized over finding the absolute optimal solution.

Simulated Annealing (SA):

While not consistently achieving the highest scores, it still provides respectable results across all files.

Offers a good balance between solution quality and computational time.

Performs relatively well in larger problem instances, indicating its scalability and adaptability to different problem sizes.

Hill Climbing:

Generally produces lower scores compared to Greedy and Simulated Annealing.

Despite its speed, it sacrifices solution quality, especially evident in larger problem instances.

Best suited for scenarios where speed is crucial and finding the absolute optimal solution is not the primary objective.

Genetic Algorithm (GA):

Offers competitive scores across all files, especially in larger problem instances.

Takes longer to execute compared to Greedy and Hill Climbing but provides better-quality solutions.

Strikes a balance between solution quality and computational time, making it suitable for various problem sizes and complexities.