A Selection Hyper-Heuristic Approach: Combining Genetic Algorithms and Greedy Methods for Efficient Bin Packing

Abdessamed Seddiki[1], Lydia Mouhoun[1], Serine Gheddou[1], Hind Ledra[1], Ines Bouais[1], Elamine Selmane[1], Mohammed Arab[1]
[1] Ecole nationale Supérieure d'Informatique, ESI, Algiers, Algeria

ABSTRACT

We propose a novel hyper-heuristic approach for solving the bin packing problem, which involves packing items into bins to minimize the number used. This problem is challenging (NP-hard) (1) and crucial in logistics, resource allocation, and data storage optimization. Our approach combines four construction heuristics (First Fit, Best Fit, Next Fit, Worst Fit) at a lower level, each with specific optimality and complexity characteristics. At an upper level, a Genetic Algorithm (GA) dynamically selects and combines these heuristics to optimize bin utilization and execution time. Experimental results show our method significantly reduces bin usage while outperforming individual heuristics, offering an efficient solution for diverse bin packing scenarios

Keywords: bin packing, hyper-heuristic, construction heuristics, genetic algorithm, NP-hard, optimality, computational complexity, logistics, resource allocation.

1 INTRODUCTION

We focus on creating an efficient hybridization scheme by combining approximate methods. Hybridization, in this context, refers to the integration of two or more optimization methods to achieve superior results compared to using each method independently. Our proposed hybridization approach utilizes an evolutionary algorithm, the Genetic Algorithm (GA), in conjunction with four heuristic algorithms: First Fit (FF), Best Fit (BF), Next Fit (NF), and Worst Fit (WF).

The Genetic Algorithm is an evolutionary technique inspired by the process of natural selection. It belongs to the class of evolutionary algorithms and operates by simulating the natural selection process to iteratively improve solutions. GAs are notable for their efficiency in exploring and exploiting the search space, flexibility in handling complex optimization problems without restrictive assumptions, and resilience against getting trapped in local optima.

The four heuristic algorithms—FF, BF, NF, and WF—are established approximate methods frequently employed to tackle NP-hard problems such as bin packing. The FF algorithm assigns each item to the first bin that can accommodate it, offering simplicity and speed. The BF algorithm places items in the bin with the least remaining space, optimizing space utilization. The NF algorithm is similar to FF but opens a new bin when the current one is full, which can be advantageous for sequential item processing. Lastly, the WF algorithm places items in the bin with the most remaining space, creating larger gaps that may accommodate future items more flexibly. Each heuristic has its own optimality bounds and computational complexities, making them suitable for different scenarios.

By hybridizing the GA with these construction heuristics, we aim to develop an efficient hyper-heuristic approach that dynamically combines their strengths. The GA orchestrates the selection and application of these heuristics to effectively address various instances of the bin packing problem. This hybrid approach not only seeks to optimize the number of bins used but also aims to balance solution quality with computational efficiency. Our goal is to demonstrate that this hyper-heuristic method can outperform individual heuristic applications, providing a robust solution for real-world bin packing challenges in logistics, resource allocation, and data storage optimization.

2 Literature Review

The development of heuristics has led to various algorithms tailored to specific problems, yet no single heuristic provides high-quality results across all problem instances. Certain problems exhibit features that make particular heuristics effective, but these features might not be present in other problems, reducing the heuristic's performance. Hyper-heuristics have emerged as a research focus, aiming to create more general algorithms capable of efficiently solving various problems.

2.1 HyFlex Framework

Ochoa et al. (2012) introduced HyFlex, a software framework designed to develop cross-domain search methodologies. HyFlex offers a common interface for handling different combinatorial problems and provides problem-specific algorithm components, acting as a benchmark for developing and comparing the generality of selection hyper-heuristics. This framework has been used to test algorithms across domains such as maximum satisfiability, bin packing, flow shop scheduling, personnel scheduling, traveling salesman, and vehicle routing (2).

HyFlex has inspired numerous studies:

- Burke et al. (2010b) compared various hyper-heuristics combining heuristic selection and acceptance approaches (3).
- Burke et al. (2012) proposed a general packing methodology for 1D, 2D, and 3D bin packing and knapsack packing. They developed a genetic programming system to generate high-quality heuristics for different problems, albeit with a significant computational cost per instance (4).

2.2 Selection Hyper-Heuristics

HyFlex and similar systems typically employ a selection hyper-heuristic approach that perturbs complete candidate solutions to improve quality, involving limited search. In contrast, Sim and Hart (2013) discussed a selection hyper-heuristic approach constructing solutions incrementally, which requires significant search effort to create the solution-builder but results in a lower cost for generating solutions to unseen problems compared to HyFlex methods (5). This incremental approach has also been used for solving constraint satisfaction problems (6).

2.3 Generalized Solutions in Combinatorial Optimization

Much of the research on combinatorial optimization problems focuses on developing methods to produce the best solutions for benchmark problem instances rather than generalized solutions. Hyper-heuristics aim to provide general solutions by exploring a heuristic space comprising low-level heuristics, which can be either constructive (building solutions) or perturbative (improving initial solutions). For example, in the one-dimensional bin-packing problem, construction heuristics include first-fit, best-fit, next-fit, and worst-fit. Hyper-heuristics may use methodologies like variable neighborhood search, tabu search, and genetic programming to select or generate low-level heuristics for different problem domains.

Despite its importance, the two-dimensional irregular bin-packing problem (2DIBPP) has received less attention in the literature compared to other bin-packing problems. Hyper-heuristic methods, particularly those based on genetic algorithms, have been applied to both regular and irregular 2D bin-packing problems. Terashima-Marín et al. (2010) developed heuristics for selecting and placing

pieces in bins (7), and Lopez-Camacho et al. (2013, 2014) extended one-dimensional heuristics to two-dimensional cases, achieving good results on convex polygon instances (8).

2.4 Evolutionary Algorithms in Operations Research

Evolutionary algorithms (EAs) are widely used in operations research due to their effectiveness in searching large spaces, although they often lack performance guarantees and may appear as black-box algorithms. To address acceptability issues, EAs can be used to generate solution processes applicable to many problem instances, built from simple heuristics. For instance, in large exam timetabling problems, a system might switch heuristics based on problem characteristics. An example is using a modern learning classifier system, XCS, to learn rules associating problem states with specific heuristics, successfully solving a large set of one-dimensional bin-packing problems.

In conclusion, hyper-heuristics represent a significant step towards creating generalized solution processes for a variety of combinatorial optimization problems. They offer a way to construct and select heuristics dynamically, adapting to different problem instances, and thus provide a versatile approach to solving complex problems efficiently.

3 Formalization of the Bin Packing Problem

The bin packing problem (BPP) is a classic combinatorial optimization problem. It is known to be NP-hard (1), meaning that no polynomial-time algorithm is known to solve all instances of the problem optimally.

Given that our research case falls under the category of one-dimensional bin packing, we are dealing with two key dimensions: the items with varying weights and the uniform capacity of the bins. The primary objective is to allocate these items into a minimum number of bins without exceeding the capacity of any bin.

This section formalizes the problem and discusses its mathematical formulation, key concerns, and practical implications:

3.1 Formal Mathematical Definition

The one-dimensional bin packing problem (1D-BPP) can be formally defined as follows:

- **Inputs:**
  - A set of n items, $I = \{i_1, i_2, \ldots, i_n\}$.
  - Each item i_j has a weight w_j where $w_j \in R^+$ for $j = 1, \ldots, n$.
  - A bin capacity $C \in R^+$.
- **Output:**
  - A partition of I into m subsets (bins) $B_1, B_2, \ldots, B_m$ such that $\sum_{i \in B_k} w_i \leq C$ for all $k = 1, 2, \ldots, m$, and m is minimized.

3.2 Objective Function

The objective is to minimize the number of bins m. Formally, we can write the objective function as:

$$\min m$$

subject to the constraints:

$$\sum_{i \in B_k} w_i \leq C \quad \forall k = 1, 2, \ldots, m$$

3.3 Decision Version

A related decision problem asks whether a given set of items can be packed into a fixed number of bins m without exceeding the bin capacity C. This can be stated as:

- **Given:** A set of items with weights $\{w_1, w_2, \ldots, w_n\}$, a bin capacity C, and an integer m.
- **Question:** Can the items be packed into m or fewer bins of capacity C?

By clearly defining the 1D-BPP, its formalism, and the practical concerns, we establish a foundation upon which further discussions on solution approaches, algorithm performance, and real-world applications can be built. This provides a comprehensive understanding of the problem's significance and the challenges involved in addressing it.

4 Methodology

In this section, we will develop a solution for the bin packing problem using a hyper-heuristic with a constructive approach.

The following diagram shows the general architecture of our solution

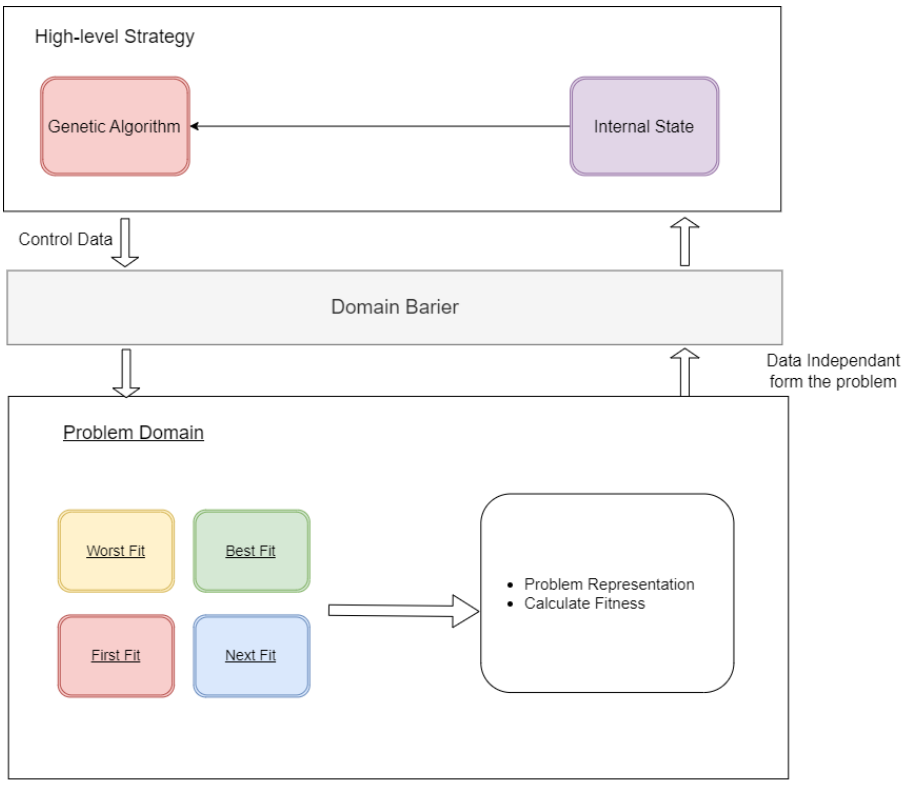

Figure 1: General Solution Framework

4.1 Problem Domain: Heuristics

This level is problem specific, we have chosen four heuristics for the bin packing problem.
In the complexity calculation, n is the number of items and m is the number of bins used.

First Fit Algorithm The First Fit (FF) algorithm places each item into the first available bin that has sufficient space. If no such bin exists, a new bin is opened.

- **Optimality Bound:** $\text{FF}(L) \leq \lfloor 1.7\text{OPT} \rfloor$ (9)
- **Computational Complexity:** $O(n \cdot m)$

Best Fit Algorithm The Best Fit (BF) algorithm places each item into the bin that will leave the least remaining space after the item is added. If no suitable bin is found, a new bin is opened.

- **Optimality Bound:** $\text{BF}(L) \leq \lfloor 1.7\text{OPT} \rfloor$ (10)
- **Computational Complexity:** $O(n \cdot m)$

Next Fit Algorithm The Next Fit (NF) algorithm places each item into the current bin. If the item does not fit, a new bin is opened, and the item is placed there.

- **Optimality Bound:** $\text{NF}(L) \leq 2\text{OPT}$ (11)
- **Computational Complexity:** $O(n)$

Worst Fit Algorithm The Worst Fit (WF) algorithm places each item into the bin that will leave the most remaining space after the item is added. If no suitable bin is found, a new bin is opened.

- **Optimality Bound:** WF(L) $\leq$ 2OPT - 2 (12)
- **Computational Complexity:** $O(n \cdot m)$

4.2 Heuristics Selection Criteria

Performance vs. Execution Time

- **Best Fit (BF) and First Fit (FF):** These heuristics are efficient in terms of the number of bins used, with moderately good execution times.
- **Next Fit (NF) and Worst Fit (WF):** These heuristics are slightly less efficient in bin usage but offer superior execution times.

Space Utilization

- **Best Fit (BF):** This heuristic is optimized to maximize bin utilization by minimizing unused space.

Placement Strategy Diversity

- **Worst Fit (WF):** This approach aims to distribute items in a way that leaves room for larger items in future bins, thus introducing strategic diversity in item placement.

4.3 High Level: Genetic Algorithm (GA)

The higher level is managed by the Genetic Algorithm (GA). to be specific, the Genetic Algorithm will be used not in selecting the destination bin for each item, but for selecting the heuristic to use. allowing more diversity in the solution.

Initial Population The initial population in the genetic algorithm consists of randomly generated chromosomes, even the population size is generated randomly. Each chromosome represents a unique combination of heuristics. For example, a chromosome might be represented as "NNFBWF," where each letter corresponds to a specific heuristic. The chromosome length is variable but does not exceed the number of items in the dataset.

Objective Function The objective function evaluates each chromosome by balancing the optimality and complexity of the chosen heuristics. It is defined as a weighted combination:

$$\text{Fitness} = a \times \text{Optimality} + b \times \text{Complexity}$$

where a and b are adjustable coefficients. This balance facilitates the prioritization of efficient solutions while maintaining manageable complexity levels.

Here is a detailed pseudo-algorithm for the High Level

Algorithm 1 Hyper-heuristic Algorithm for Bin Packing (Part 1)

```
1: procedure HyperHeuristic(population_size, initial_population, num_bins, items, max_iteration)
2:     solution ← []                                              ▷ Initialize an empty solution
3:     current_population ← initial_population                    ▷ Set the initial population
4:     current_items ← items                                      ▷ Set the items to be packed
5:     while current_items is not empty do
6:         selected_chromosome ← BestChromosome(current_population)   ▷ Select the best chromosome
7:         for gene in selected_chromosome do
8:             if current_items is not empty then
9:                 if gene == 'B' then
10:                    solution ← BestFit(solution, current_items[0])      ▷ Apply Best Fit heuristic
11:                else if gene == 'F' then
12:                    solution ← FirstFit(solution, current_items[0])     ▷ Apply First Fit heuristic
13:                else if gene == 'W' then
14:                    solution ← WorstFit(solution, current_items[0])     ▷ Apply Worst Fit heuristic
15:                else if gene == 'N' then
16:                    solution ← NextFit(solution, current_items[0])      ▷ Apply Next Fit heuristic
17:                end if
18:                Remove current_items[0] from current_items              ▷ Remove packed item
19:            end if
20:        end for
21:        current_population ← GeneticAlgorithm(current_population, population_size, max_iteration)   ▷ Evolve the population
22:    end while
23:    return solution                                            ▷ Return the final solution
24: end procedure

25: procedure GeneticAlgorithm(population, population_size, max_iteration)
26:    for iteration from 1 to max_iteration do
27:        parents ← TournamentSelection(population, 2)           ▷ Select parents
28:        children ← Crossover(parents[0], parents[1])           ▷ Perform crossover
29:        mutated_children ← [Mutation(child) for child in children]   ▷ Apply mutation
30:        population ← SelectNextGeneration(population, mutated_children, population_size)   ▷ Select next generation
31:    end for
32:    return population                                          ▷ Return evolved population
33: end procedure
```

```
procedure TournamentSelection(population, tournament_size)
    contestants ← RandomSample(population, tournament_size)      ▷ Select random contestants
    winner ← Contestant with minimum fitness in contestants      ▷ Select the best contestant
    return winner                                                 ▷ Return the winner
end procedure

procedure BestChromosome(chromosomes)
    best_chromosome ← None
    best_fitness ← ∞                                              ▷ Initialize to infinity
    for chromosome in chromosomes do
        fitness ← FitnessFunction(chromosome)                     ▷ Calculate fitness
        if fitness < best_fitness then
            best_fitness ← fitness
            best_chromosome ← chromosome
        end if
    end for
    return best_chromosome                                        ▷ Return the best chromosome
end procedure

procedure FitnessFunction(chromosome)
    optimalities ← 'N': 1, 'F': 3, 'B': 3, 'W': 2
    complexities ← 'N': 1, 'F': 1, 'B': 2, 'W': 2
    total_optimality ← Sum of optimalities[gene] for each gene in chromosome
    total_complexity ← Sum of complexities[gene] for each gene in chromosome
    fitness ← weight_optimality × total_optimality + weight_complexity × total_complexity
    return fitness                                                ▷ Return the fitness value
end procedure

procedure Mutation(child)
    mutation_rate ← 0.1
    if random number < mutation_rate then
        index ← Random integer from 0 to length of child − 1
        new_gene ← Random choice from heuristics
        child ← Replace gene at index in child with new_gene
    end if
    return child                                                  ▷ Return mutated child
end procedure
```

Another way to view how the algorithm works is through the following diagram:

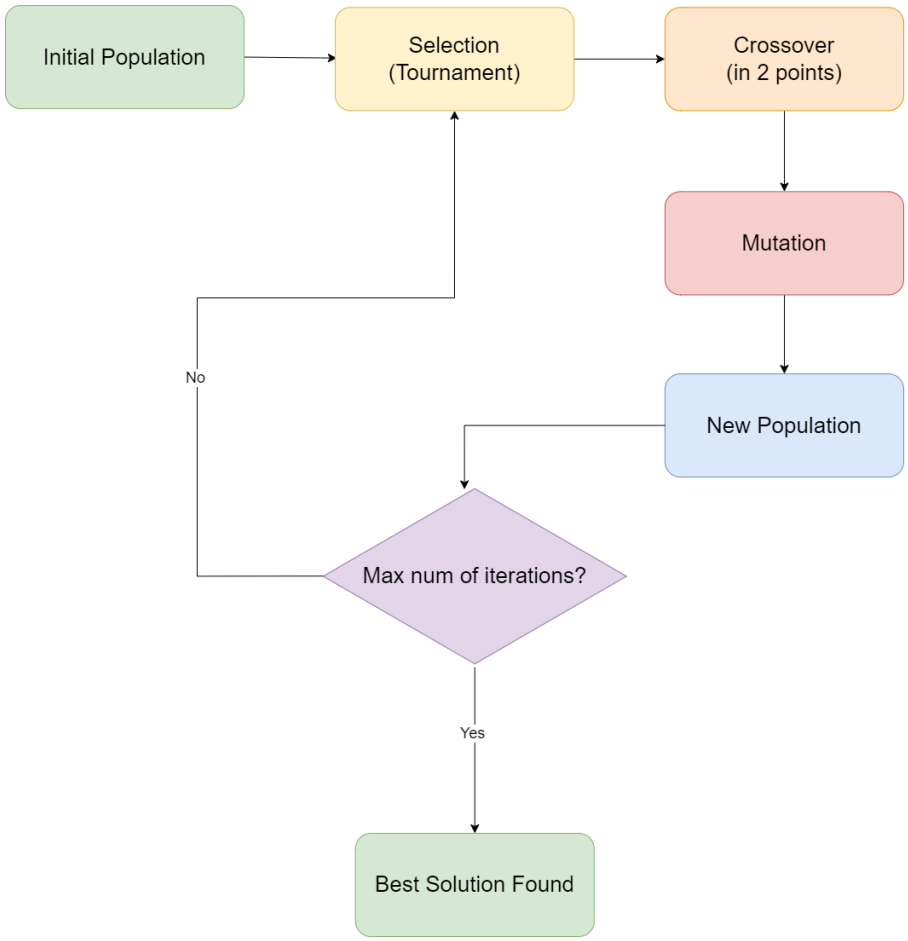

Figure 2: Genetic Algorithm Framework

4.4 Parameters

These are the parameters needed for the solution to work:

- a, b: Fitness Coefficients
- **Items list:** List of items to be treated
- **Bin Capacity:** Capacity of a single bin
- **Chromosome Length:** Length of a single chromosome (less than or equal to the items list length)
- **Initial Population Size:** The starting number of chromosomes in the genetic algorithm.
- **Maximum number of iterations**: the maximum number of iterations for the genetic algorithm.

4.5 Example

The following example will help understanding how the algorithm works.
we suppose that b=0 as we will only be interested by the optimality of the solution.

For ease we will suppose the optimalities to be as follows:

- Best Fit: noted by B, optimality = 3
- First Fit: noted by F, optimality = 3
- Worst Fit: noted by W, optimality = 2
- Next Fit: noted by N, optimality = 1

Initial State Starting from this initial population

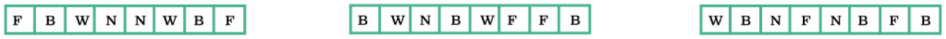

Figure 3: Example-Initial population

Selection We calculate the Fitness for each chromosome, and then We use the tournament strategy for selection

Figure 4: Example-Selection

we randomly select 2 chromosomes for a tournament, the one with a higher fitness wins, suppose this was the case:

In the first iteration, the chromosome with fitness 19 wins against the one with fitness 18 so we take the chromosome with fitness=19. In the second iteration, the chromosome with fitness 20 wins.

Crossover in This step we will cross the parents in 2 points, the resulting offsprings are as follows

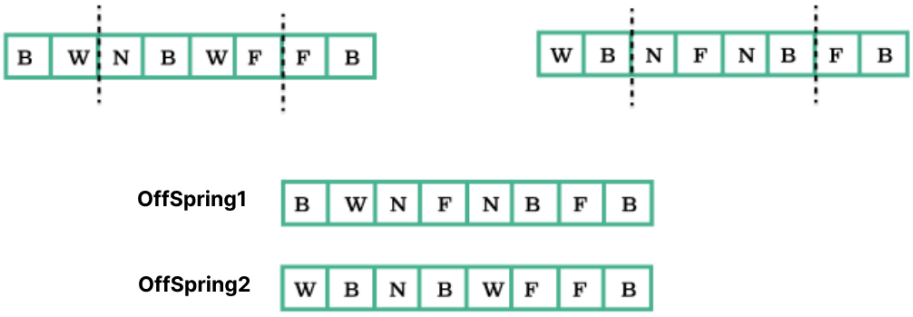

Figure 5: Example-Crossover

Mutation Now we cause a little mutation in the current population

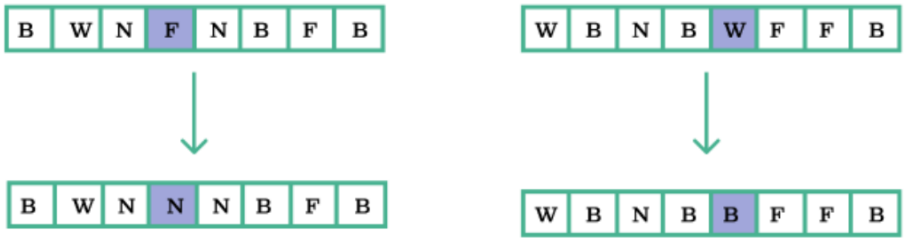

Figure 6: Example-Mutation

Final Step Reorder the resulting population by the fitness and choose the 3 firsts chromosomes. when getting to the final iteration, only pick one chromosome and that's the best solution.

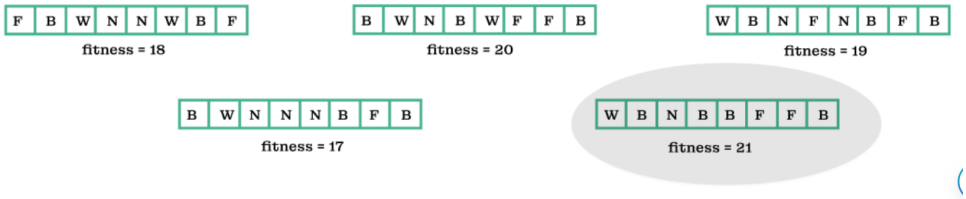

Figure 7: Example-Final Iteration

Apply the result Apply the resulting solutions on items, for example, the first item should be placed using the Worst fit strategy, the fourth one with the Best Fit Strategy.. etc

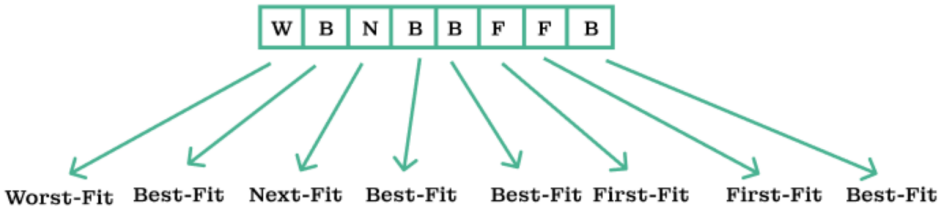

Figure 8: Example-Application

5 Categories of Tests and Benchmarks

5.1 Test Categories

Category	Objective	Description
Tests of Solution Accuracy	Evaluate the accuracy of our hyper-heuristic by comparing it to exact solutions.	Use various benchmarks with complexity reduced to zero to focus on optimality. Compare the solutions obtained with those provided by exact algorithms.
Population Size and Iteration Number Evaluation Test	Study the effect of different hyperparameters on the performance of the hyper-heuristic.	Examine the impact of the number of iterations and chromosome size on performance using diverse benchmarks to ensure representativeness of results.
Tests on the Impact of Complexity and Optimality	Analyze the impact of complexity and optimality weights on solution performance.	Modify weights a and b in the objective function to observe changes in results, aiming to understand trade-offs between complexity and optimality.

5.2 Benchmarks

Benchmark	Objects	Application Scope	Reference
Falkenauer U	Uniform items	General framework	Emanuel Falkenauer
Falkenauer T	Triplets (1 large, 2 small)	Real-world scenarios	Benchmark by Falkenauer

6 Comparison of Solution Methods

6.1 First Category: Comparison with other solutions

6.1.1 Bin Utilization Comparison

I use the following symbols to represent algorithms:

- B&B: Branch and Bound
- RT: Recherche Tabou (Tabu Search)
- AG: Algo génétique (Genetic Algorithm)
- HC: Hill Climbing
- HH: Our Hyper-heuristic

Benchmark	B&B	RT	AG	HC	HH
B-50	25 bins	25 bins	29 bins	25 bins	25 bins
B-100	47 bins	47 bins	56 bins	47 bins	47 bins
B-200	-	80 bins	114 bins	80 bins	81 bins
B-500	-	-	309 bins	202 bins	202 bins
B-1000	-	-	611 bins	402 bins	402 bins

Table 1: Number of bins used across different benchmarks for B&B, RT, AG, HC, and HH methods.

6.1.2 Execution Time Comparison

Benchmark	B&B	RT	AG	HC	HH
B-50	0.70 ms	149 ms	81 ms	346 ms	13 ms
B-100	69.00 ms	352 ms	46 ms	354 ms	14 ms
B-200	-	526 ms	250 ms	408 ms	28 ms
B-500	-	-	267 ms	302 ms	79 ms
B-1000	-	-	332 ms	302 ms	190 ms

Table 2: Execution time across different benchmarks for B&B, RT, AG, HC, and HH methods.

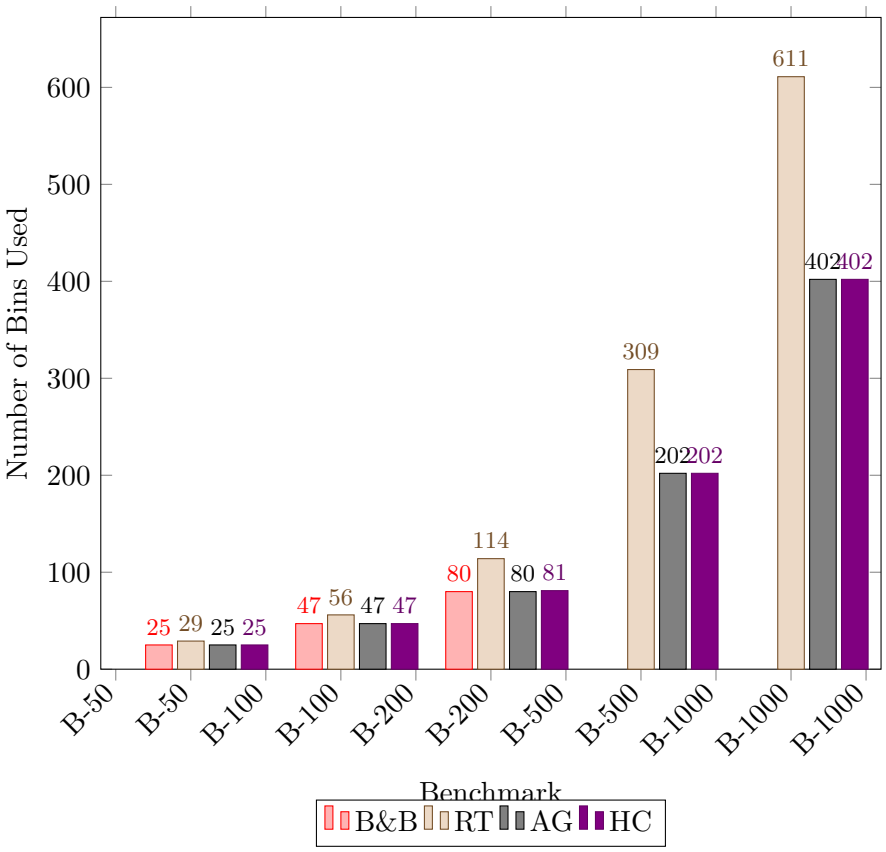

Figure 9: Number of bins used across different benchmarks for B&B, RT, AG, HC, and HH methods.

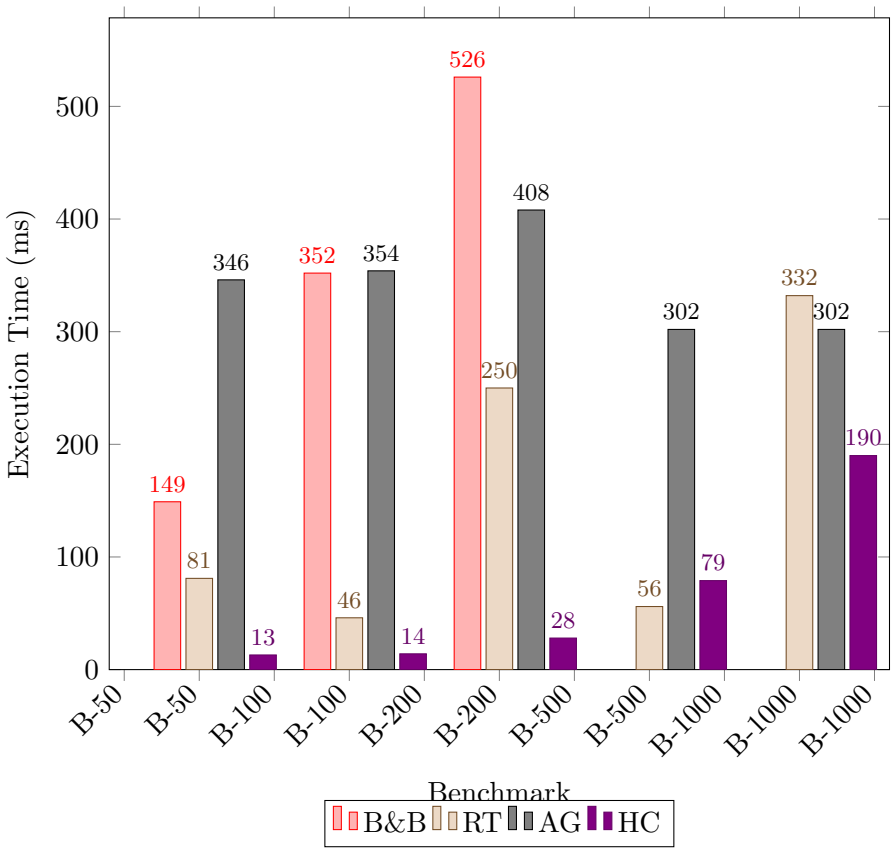

Figure 10: Execution time across different benchmarks for B&B, RT, AG, HC, and HH methods.

6.1.3 Discussion

The tables above provide a comprehensive comparison of the number of bins used and the execution times for various methods applied to bin packing problems.

Bin Utilization For smaller benchmarks (B-50 and B-100), all methods except AG used the same number of bins. AG used slightly more bins, that indicates less efficiency.

For medium benchmarks (B-200), RT used 80 bins, while AG required significantly more bins (114). HH used 81 bins, showing competitive performance.

For larger benchmarks (B-500 and B-1000), HC and HH showed consistent performance, using the same number of bins. AG required more bins, indicating scalability issues. B&B and RT results are unavailable for larger benchmarks, suggesting they are impractical for large-scale problems.

Execution Time B&B performed well for very small problems but did not scale to larger problems. RT showed significant execution times, making it less practical. AG and HC had moderate execution times for smaller benchmarks, but times increased significantly for larger benchmarks. HH maintained reasonable execution times across all benchmark sizes, highlighting its scalability and efficiency.

6.1.4 Overall Performance

Combining bin utilization and execution time metrics, the HH method stands out as the most balanced and scalable solution. It consistently uses a minimal number of bins and maintains efficient execution

times across all benchmark sizes. AG and HC perform well for smaller problems but do not scale as effectively as HH.

In conclusion, the hyper-heuristic method provides the best overall performance, making it suitable for a wide range of bin packing problems from small to large scales.

6.2 Second Category: Population Size and Iteration Number Evaluation Test

Balancing optimality and complexity is crucial for developing effective and practical algorithms in this tests, we set : complexity= 1 , optimality=1

- **Population Size:** larger population sizes lead to worse solutions and require more execution time.where small size gives better result and execution time.

Benchmark	Population Size	Solution	Execution Time
B-50	10	25 bins	0.00800 s
B-50	100	25 bins	0.03496 s
B-50	1000	25 bins	0.54263 s
B-50	10000	25 bins	5.02556 s
B-180	10	68 bins	0.06798 s
B-180	100	68 bins	0.35399 s
B-180	1000	68 bins	1.54301 s
B-180	10000	68 bins	17.58510 s
B-200	10	81 bins	0.06304 s
B-200	100	81 bins	0.36 s
B-200	1000	81 bins	2.08612 s
B-200	10000	81 bins	19.80647 s

Table 3: Benchmark results showing population size, solution, and execution time.

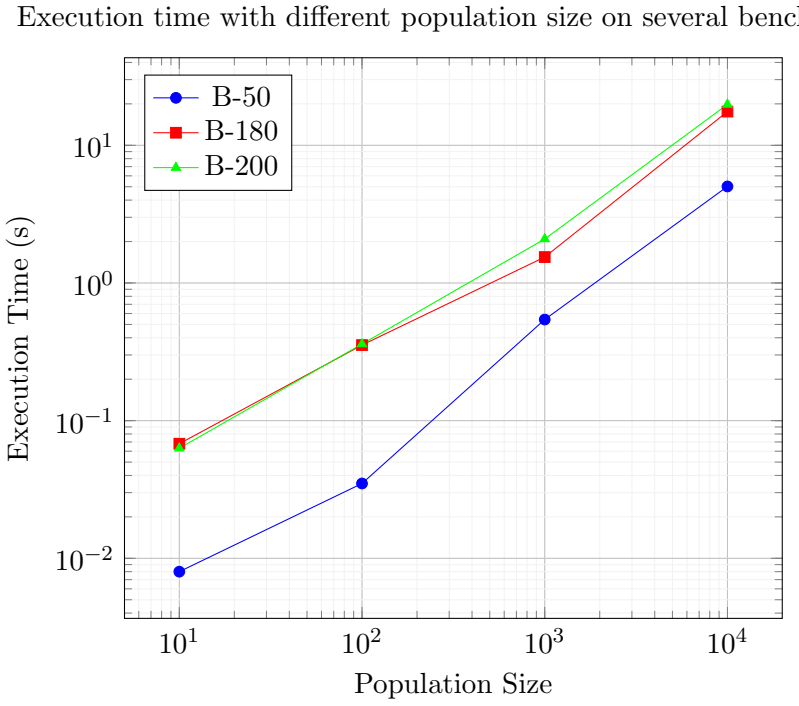

Figure 11: Execution time across different population sizes for B-50, B-180, and B-200 benchmarks.

- **Number of Iterations:** a small number of iterations gives a good result, however increasing the number of iterations beyond a certain point does not significantly improve the solution but increases execution time.

Benchmark	Max Number of Iterations	Solution	Execution Time
B-50	10	25 bins	0.00699 s
B-50	100	25 bins	0.03102 s
B-50	1000	26 bins	0.54851 s
B-50	10000	25 bins	4.35686 s
B-180	10	68 bins	0.03004 s
B-180	100	68 bins	0.11700 s
B-180	1000	68 bins	1.62287 s
B-180	10000	68 bins	14.70123 s
B-1000	10	402 bins	0.13000 s
B-1000	100	402 bins	0.89253 s
B-1000	1000	402 bins	8.50841 s
B-1000	10000	402 bins	1mn 30.55423s

Table 4: Benchmark results for different max number of iterations.

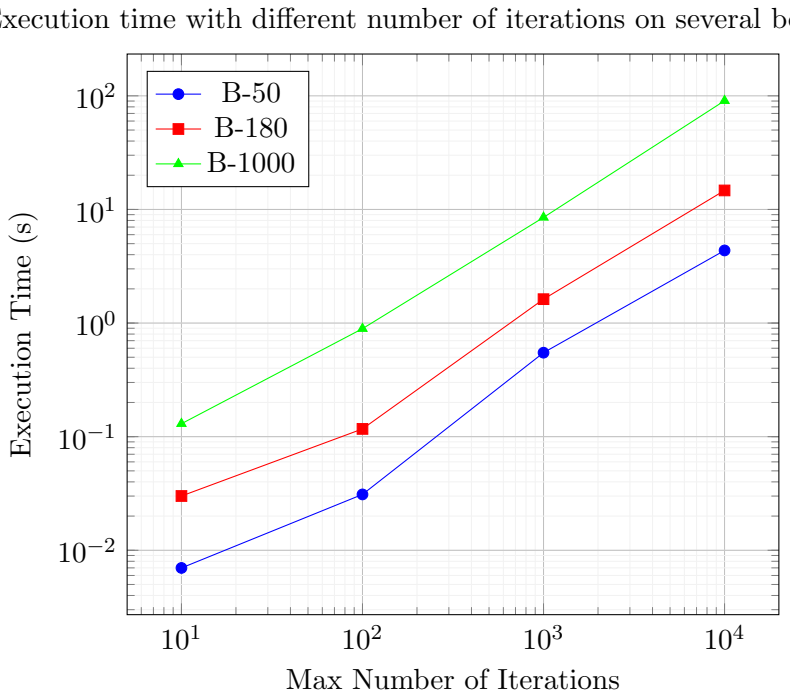

Figure 12: Execution time across different max number of iterations for B-50, B-180, and B-1000 benchmarks.

6.2.1 Discussion

Population Size The table on population size demonstrates a clear relationship between population size, solution quality, and execution time across all benchmarks:

- **Solution Quality**: The solution quality, as indicated by the number of bins used, remains consistent across different population sizes for each benchmark. This suggests that increasing the population size does not lead to an improvement in the solution.

- **Execution Time**: Execution time increases significantly as the population size grows. For example, execution time for B-50 increases from 0.00800 seconds with a population size of 10 to 5.02556 seconds with a population size of 10,000.

Overall Analysis: Smaller population sizes are more efficient. They achieve optimal solutions with significantly shorter execution times. Increasing the population size leads to diminishing returns in solution quality and a significant increase in execution time, indicating that it is unnecessary to use large populations for effective problem-solving in these benchmarks.

Number of Iterations The table on the number of iterations shows how varying the maximum number of iterations impacts solution quality and execution time:

- **Solution Quality**: The number of bins used remains largely unchanged across different numbers of iterations for each benchmark. This indicates that increasing the number of iterations beyond a certain point does not significantly improve the solution.
- **Execution Time**: Execution time increases dramatically as the number of iterations grows. For instance, execution time for B-1000 increases from 0.13000 seconds with 10 iterations to over a minute with 10,000 iterations.

Overall Analysis: A smaller number of iterations (e.g., 10 or 100) is sufficient to achieve optimal solutions. Increasing the number of iterations results in longer execution times without any significant improvement in solution quality. This suggests that setting a moderate number of iterations can lead to efficient and effective problem-solving.

Conclusion Combining insights from both tables, it is evident that:

1. **Efficiency**: Smaller population sizes and fewer iterations are more efficient. They provide optimal solutions with significantly shorter execution times.
2. **Scalability**: Larger population sizes and more iterations do not scale well. They lead to substantial increases in execution time without improving solution quality.
3. **Optimal Strategy**: An optimal strategy for solving bin packing problems involves using smaller population sizes and moderate numbers of iterations to balance solution quality and execution time effectively.

6.3 Third Category: Tests on the Impact of Complexity and Optimality

In evaluating our hyperheuristic approach, the parameters a and b in the fitness function $f = a \times \text{optimality} + b \times \text{complexity}$ play a crucial role in determining the effectiveness of the solution.

We did the tests on the benchmark of 1000 binsand we got the following results:

Complexity	Optimality	Number of Bins	Execution Time (s)
0.1	0.1	408	0.21399
0.1	1.0	406	0.08300
1.0	1.0	405	0.06703
1.0	3.0	403	0.06203
2.0	1.0	404	0.06204
2.0	3.0	404	0.06599
3.0	0.1	427	0.09405
3.0	1.0	408	0.06499
3.0	4.0	404	0.06500
4.0	0.1	428	0.06603
4.0	1.0	408	0.06000
4.0	4.0	404	0.06503

Table 5: Results for different combinations of complexity and optimality parameters.

Discussion

Low Complexity with Low Optimality

Scenario: Optimality = 0.1, Complexity = 0.1
Results: High number of bins (408) and long execution time (0.21399 s).
Summary: Poor performance in both solution quality and time.

Low Complexity with High Optimality

Scenario: Optimality = 1.0, Complexity = 0.1
Results: Moderate number of bins (406) and lower execution time (0.08300 s).
Summary: Fast but not the best solution quality.

Equal Complexity and Optimality

Scenario: Optimality = 1.0, Complexity = 1.0
Results: Lower number of bins (405) and reasonable execution time (0.06703 s).
Summary: Balanced performance in both time and solution quality.

High Complexity with Low Optimality

Scenario: Optimality = 0.1, Complexity = 3.0
Results: Very high number of bins (427) and moderate execution time (0.09405 s).
Summary: Inefficient use of resources, poor solution quality.

High Complexity with High Optimality

Scenario: Optimality = 1.0, Complexity = 3.0
Results: Moderate number of bins (408) and reasonable execution time (0.06499 s).
Summary: Good balance, effective solution quality with acceptable time.

Equal High Complexity and Optimality

Scenario: Optimality = 4.0, Complexity = 4.0
Results: Low number of bins (404) and reasonable execution time (0.06503 s).
Summary: Efficient solution quality and good time performance.

Conclusion

The balance between optimality and complexity significantly impacts both the solution quality and execution time:

Scenario	Summary
Low Complexity, Low Optimality:	Poor performance overall.
Low Complexity, High Optimality:	Fast execution but suboptimal solutions.
Equal Complexity and Optimality:	Balanced and effective performance.
High Complexity, Low Optimality:	Acceptable time with poor solutions.
High Complexity, High Optimality:	Good balance, effective solutions with acceptable time.
Equal High Complexity and Optimality:	Most efficient solution quality and good time performance.

Table 6: Impact of optimality and complexity on solution quality and execution time.

For the best results, a balanced approach with equal or higher levels of both optimality and complexity is recommended.

- **High Optimality, Low Complexity:** This setting yields fast but moderately efficient solutions in terms of bin usage. It is suitable for time-sensitive applications where quick, near-optimal solutions are needed.
- **Equal Optimality and Complexity:** This balanced setting ensures efficient bin usage and reasonable execution times. It is ideal for applications where both the quality of the solution and the time to find it are critical.

Recommended Strategy: A balanced approach with moderate to high levels of both complexity and optimality generally yields the best results. This strategy ensures a thorough search process while also focusing on optimality, leading to efficient bin usage and reasonable execution times. This balance can be fine-tuned based on specific application needs and constraints.

7 Conclusion

In this study, we explored the application of a hyper-heuristic approach combining Genetic Algorithm (GA) with a greedy selection strategy to tackle the Bin Packing Problem (BPP). The BPP, a classic NP-hard problem, requires efficient allocation of items into bins to minimize wasted space and optimize resource utilization.

Our investigation began by evaluating traditional heuristic methods such as First Fit (FF), Best Fit (BF), Next Fit (NF), and Worst Fit (WF). While effective in specific scenarios, these heuristics exhibited limitations in scalability and optimality for larger and more complex problem instances.

To address these challenges, we implemented a hyper-heuristic framework that integrates GA with a greedy selection mechanism. The GA employed a population-based evolutionary approach to evolve solutions iteratively, while the greedy strategy dynamically selected heuristics based on their past performance and immediate suitability.

Our experimental results demonstrated that the hyper-heuristic approach outperformed traditional heuristics and metaheuristic methods like GA alone. By dynamically combining heuristic selection and evolutionary optimization, our method achieved competitive results in terms of both computational efficiency and bin utilization. It effectively balanced exploration of new heuristic combinations with exploitation of known effective strategies, adapting flexibly to different problem sizes and complexities.

In conclusion, the hyper-heuristic approach leveraging GA with a greedy selection mechanism proved to be a robust and efficient solution for the Bin Packing Problem. It offers significant advantages over traditional methods in terms of scalability, solution quality, and computational efficiency. Future research could explore further enhancements and adaptations of hyper-heuristic frameworks to tackle other complex combinatorial optimization problems.

8 Bibliography

References

[1] Garey, M. R., & Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co Ltd.

[2] Ochoa, G., Hyde, M., Curtois, T., Vazquez-Rodriguez, J. A., Walker, J., Gendreau, M., Kendall, G., McCollum, B., Parkes, A. J., Petrovic, S., & Burke, E. K. (2012). HyFlex: *A benchmark framework for cross-domain heuristic search*. Evolutionary Computation in Combinatorial Optimization (EvoCOP 2012) (pp. 136–147). Springer.

[3] Burke, E. K., Hyde, M., Kendall, G., & Woodward, J. (2010). *A genetic programming hyper-heuristic approach for evolving 3D packing heuristics*. In Proceedings of the IEEE Congress on Evolutionary Computation (CEC) (pp. 1-8). IEEE.

[4] Burke, E. K., Hyde, M., Kendall, G., Ochoa, G., & Ozcan, E. (2010). *A classification of hyper-heuristic approaches*. In Handbook of Metaheuristics (pp. 449-468). Springer.

[5] Sim, K., & Hart, E. (2016). *A Combined Generative and Selective Hyper-heuristic for the Vehicle Routing Problem*. In GECCO '16: Proceedings of the Genetic and Evolutionary Computation Conference 2016 (pp. 1093-1100). July 2016

[6] Terashima-Marín, H., Ortiz-Bayliss, J. C., Ross, P., & Valenzuela-Rendón, M. (2008). *Hyper-heuristics for the dynamic variable ordering in constraint satisfaction problems*. In GECCO '08: Proceedings of the 10th annual conference on Genetic and evolutionary computation (pp. 571-578). July 2008

[7] Gomez, J. C., & Terashima-Marín, H. (2018). *Evolutionary hyper-heuristics for tackling bi-objective 2D bin packing problems*. Genetic Programming and Evolvable Machines, 19(1), 1-31.

[8] López-Camacho, E., Terashima-Marin, H., Ross, P., & Ochoa, G. (2014). *A unified hyper-heuristic framework for solving bin packing problems*. Expert Systems with Applications, 41(15), 6876-6889.

[9] Dósa, György; Sgall, Jiri (2013), *First Fit bin packing: A tight analysis*, 30th International Symposium on Theoretical Aspects of Computer Science (STACS 2013). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, pp. 538–549.

[10] György, Dósa; Sgall, Jirí (2014). *Optimal Analysis of Best Fit Bin Packing*, Automata, Languages, and Programming. Lecture Notes in Computer Science. Vol. 8572. pp. 429–441

[11] Vazirani, Vijay V. (2003), *Approximation Algorithms*, Berlin: Springer.

[12] Johnson, David S (1973), *Near-optimal bin packing algorithms*, Massachusetts Institute of Technology.

[13] Nelishia Pillay, *A Study of Evolutionary Algorithm Selection Hyper-Heuristics for the One-Dimensional Bin-Packing Problem*, June 2012, School of Mathematics, Statistics and Computer Science, University of KwaZulu-Natal, South Africa.