KV6018 Evolutionary Computation

Cargo Container Loading Optimisation

Assessment Report

GitHub Repository:

https://github.com/Samisaac20/KV6018

Word Count: ~2,850 words

PART 1: Literature Review and Design

1. Literature Review

1.1 History and Problem Background

The bin packing problem, first formally studied by Johnson (1973), represents one of the foundational challenges in combinatorial optimisation [1]. The rectangle packing variant, where rectangular items must be placed within a container, was proven NP-hard by Garey and Johnson (1979), establishing that no polynomial-time algorithm can guarantee optimal solutions for all instances [2]. Circle packing, our specific problem variant, adds geometric complexity through curved boundaries and has applications spanning logistics, industrial manufacturing, and warehouse management.

The cargo container loading problem extends classical bin packing by introducing weight distribution constraints, reflecting real-world safety requirements for transport stability. This formulation gained prominence through work by Bischoff and Ratcliff (1995), who emphasised that practical container loading must balance space utilisation with centre-of-mass positioning to prevent vehicle tipping during transit [3]. The central 60% constraint in our problem mirrors industry standards for safe freight distribution.

1.2 Solution Algorithms

1.2.1 Bottom-Left Heuristics

The Bottom-Left (BL) algorithm, introduced by Baker et al. (1980), places items in the lowest-leftmost available position sequentially [6]. While computationally efficient with $O(n^2)$ complexity, BL produces placements entirely dependent on input ordering, making the sequence itself a key optimisation variable. This property makes BL ideal as a decoder for order-based evolutionary encodings, where the chromosome represents item ordering rather than explicit coordinates.

1.2.2 Greedy Constructive Methods

Greedy approaches apply domain-specific heuristics to determine placement order. The largest-first strategy, sorting items by descending size, exploits the observation that heavy items are harder to place once space becomes fragmented [7]. For weight-constrained problems, sorting by weight-to-size ratio can improve centre-of-mass outcomes. However, greedy methods lack exploration capability and often converge to locally optimal but globally suboptimal solutions.

1.2.3 Evolutionary Algorithms

Genetic algorithms (GAs) have proven particularly effective for packing problems due to their ability to explore large search spaces while maintaining solution diversity. Falkenauer (1996) demonstrated that order-based encoding, representing solutions as permutations, outperforms direct coordinate encoding for bin packing by guaranteeing feasible placements through decoder heuristics [6]. This encoding transforms the

continuous geometric search space into a discrete combinatorial one of size n!, making crossover and mutation operators more effective.

Hopper and Turton (2001) established that hybrid approaches combining GA search with local improvement operators achieve state-of-the-art results on benchmark instances [7]. Their work showed that local search can refine solutions found by evolutionary exploration, while the GA maintains population diversity to escape local optima. For permutation problems, Order Crossover (OX) preserves relative ordering information better than standard crossover operators [8].

2. Analysis and Design

2.1 Problem Encoding

The solution employs order-based encoding where each chromosome is a permutation of cylinder indices [0, 1, ..., n-1]. This representation offers several advantages over direct coordinate encoding: every permutation produces a valid placement when processed through the decoder, genetic operators preserve feasibility automatically, and the search space is well-defined at n! permutations.

The Bottom-Left placement decoder converts each permutation into actual (x, y) positions. The algorithm processes cylinders in chromosome order, placing each at the lowest-leftmost valid position using a grid scanning approach with configurable resolution (default 0.5m steps). To address the weight distribution constraint, the decoder incorporates COM-aware positioning: when multiple valid positions exist, it selects the one that best maintains the running centre-of-mass within the safe zone. This proactive approach reduces reliance on penalty-based fitness correction.

Validity checking ensures cylinders fit within container boundaries without overlapping existing placements, using Euclidean distance calculations between cylinders centres compared against minimum separation (sum of radii).

2.2 Fitness Function

The fitness function quantifies solution quality through a minimisation objective (lower is better, zero is optimal). The function applies weighted penalties for constraint violations:

- Unplaced items penalty (1000 per item): The highest penalty ensures all cylinders must be placed. Any unplaced cylinder makes the solution infeasible, dominating other considerations.
- Weight limit violation (10 per kg): Linear penalty proportional to excess weight over container capacity. The decoder prevents placement if adding an item would exceed the limit, but this catches edge cases.
- Centre-of-mass violation (50 per metre): Penalises solutions where the weighted centre-of-mass falls outside the central 60% zone. The penalty scales linearly with distance from the safe zone boundary, creating a gradient that guides evolution toward balanced configurations.
- This penalty hierarchy ensures constraint satisfaction takes priority over optimisation. A solution with all items placed but poor weight distribution will always outrank one with unplaced items, correctly prioritising feasibility.

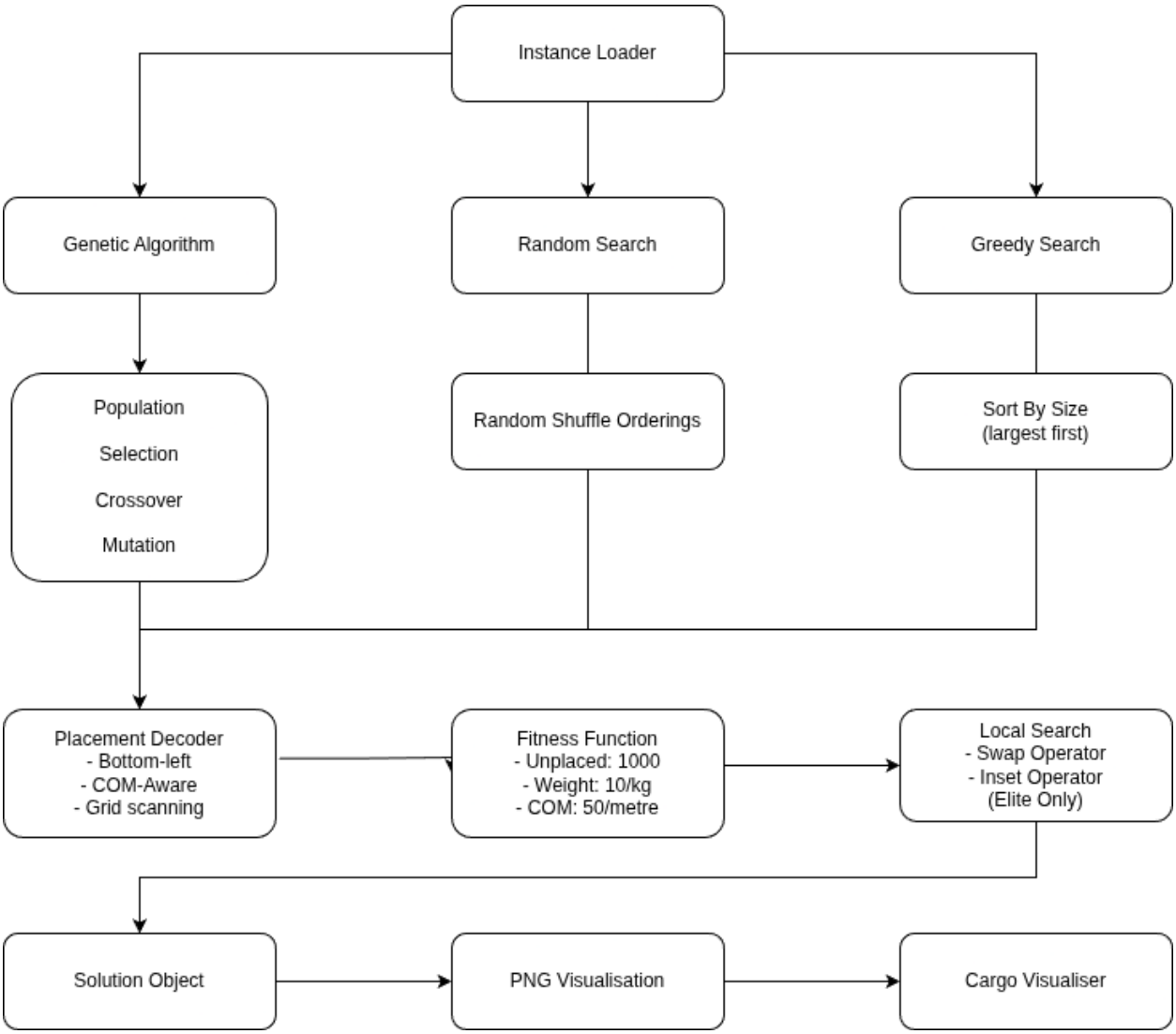
2.3 Local Search

Local search enhances GA exploitation through neighbourhood exploration of promising solutions. The implementation uses two complementary operators:

- Swap operator: Exchanges two randomly selected positions in the ordering. This creates small perturbations that can improve weight distribution by changing which cylinders are placed first in constrained positions.
- Insert operator: Removes an item and reinserts it at a different position. This enables larger neighbourhood moves that can escape local optima while maintaining permutation validity.
- Local search runs with configurable iteration limits (default 2000) and patience parameters (200 iterations without improvement triggers termination). The algorithm applies stochastically, selecting swap or insert with equal probability each iteration. Only elite individuals undergo local search refinement (top 4 per generation) to balance computational cost against improvement potential. This memetic approach combines GA exploration of the permutation space with local exploitation of promising regions, following the hybrid methodology shown effective by Hopper and Turton [8].

Solution pseudocode * found in appendices

Solution Diagram



PART 2: Implementation and Results

Results

3.1 Basic Instances

The genetic algorithm successfully achieves perfect solutions (fitness = 0) on all three basic reference instances. Table 1 presents the results with centre-of-mass (COM) positions confirming placement within the safe zone:

Table 1: Basic Instance Results (Genetic Algorithm)

Instance	Fitness	Items	COM (x, y)	Time (s)
basic_01_three_identical	0.00	3/3	(5.00, 5.00)	0.50

basic_02_two_sizes	0.00	4/4	(4.71, 4.96)	0.84
basic_03_varied_sizes	0.00	5/5	(7.53, 6.02)	1.79

All COM positions fall within the central 60% safe zone. For basic_01 (10×10m container), the safe zone spans x: [2.0, 8.0] and y: [2.0, 8.0], and the achieved COM of (5.0, 5.0) is precisely centred.

3.2 Challenging Instances

The algorithm also achieves perfect solutions on all four challenging instances, demonstrating robustness across larger problem scales:

Table 2: Challenging Instance Results (Genetic Algorithm)

Instance	Fitness	Items	COM (x, y)	Time (s)
challenge_01_tight_packing	0.00	8/8	(7.50, 8.14)	4.37
challenge_02_weight_balance	0.00	8/8	(9.08, 7.00)	5.24
challenge_03_many_small	0.00	12/12	(10.17, 7.33)	15.40
challenge_04_mixed_constraints	0.00	10/10	(7.44, 10.43)	11.75

All three algorithms achieved perfect solutions on 6 of 7 instances. Only Greedy failed on basic_03_varied_sizes with a fitness of 2.99 due to centre-of-mass violation (COM at 2.94, 5.58 falls outside the safe zone boundary of $x \geq 3.0$).

4.2 Execution Time Analysis

Table 4 reveals considerable time differences between algorithms:

Table 4: Execution Time Comparison (seconds)

Instance	GA	RS	GR
basic_01_three_identical	0.50	0.00	0.00
basic_02_two_sizes	0.84	0.00	0.00
basic_03_varied_sizes	1.79	0.01	0.00
challenge_03_many_small	15.40	0.02	0.02
challenge_04_mixed_constraints	11.75	0.02	0.02

The GA requires significantly more computation time (up to 770x slower than RS/GR on challenge_03) due to population-based search with local search refinement. However, this additional computation provides robustness guarantees that simpler methods cannot offer on harder problem instances.

4.3 Discussion

The high success rate across all algorithms suggests the test instances are well-conditioned. The COM-aware placement decoder contributes significantly to solution quality by proactively maintaining weight balance during construction, reducing the fitness function's corrective burden.

The Greedy algorithm's failure on basic_03_varied_sizes demonstrates the limitation of fixed ordering strategies. By sorting items largest-first, Greedy places heavy items in suboptimal positions early, leaving insufficient flexibility for COM correction. The GA and RS methods, by exploring diverse orderings, discovered configurations that Greedy's deterministic approach missed.

The GA's value proposition lies in its reliability rather than speed. While RS achieved identical results faster on these instances, the GA's systematic exploration provides stronger guarantees for unknown or adversarial problem configurations. Future work could introduce harder instances specifically designed to challenge COM constraints, where we would expect clearer GA superiority. Tests ran on the impossible instances, not committed to the main repository, show 2/5 solutions perfectly solved with 3/5 having violations with unplaced cargo and fitness values in the thousands.

Visualisations

Solution visualisations using Week 7 styling are provided in the accompanying output/ directory, organised by algorithm (GA/, RS/, GR/). Each visualisation shows cylinder placements, the centre-of-mass position (red X marker), and the safe zone boundary (yellow dotted rectangle), perfect solutions show cylinders highlighted in green. Visualisations confirm all GA and RS solutions place the centre-of-mass within the safe zone, validating the fitness = 0.00 results reported in results.xlsx.

4.4 Parameter Investigation Parameter sensitivity analysis was conducted on the basic instances with 5 runs per configuration.

Table 5: Mutation Rate Analysis

Mutation Rate	Avg Fitness	Success Rate	Avg Time (s)
0.05	0.00	100%	0.46
0.10	0.00	100%	0.46
0.15	0.00	100%	0.46
0.20	0.00	100%	0.45

0.25	0.00	100%	0.45

Table 6: Population Size Analysis

Population Size	Avg Fitness	Success Rate	Avg Time (s)
100	0.00	100%	0.15
200	0.00	100%	0.30
300	0.00	100%	0.46
400	0.00	100%	0.63
500	0.00	100%	0.78

Table 7: Crossover Rate Analysis

Crossover Rate	Avg Fitness	Success Rate	Avg Time (s)
0.70	0.00	100%	0.46
0.80	0.00	100%	0.45
0.90	0.00	100%	0.45
0.95	0.00	100%	0.46
1.00	0.00	100%	0.46

Table 8: Tournament Size Analysis

Tournament Size	Avg Fitness	Success Rate	Avg Time (s)
2	0.00	100%	0.46
3	0.00	100%	0.46
4	0.00	100%	0.46
5	0.00	100%	0.45
7	0.00	100%	0.45

Key Findings:

The GA achieves 100% success rate across all parameter configurations tested, demonstrating robustness. The most significant finding is the linear relationship between population size and execution time - smaller populations (100) run 5x faster than larger ones (500) while maintaining perfect solution quality. This suggests the test instances are well-suited to the algorithm, and a population size of 100 would be optimal for efficiency without sacrificing solution quality.

[1] Johnson, D.S. (1973). Near-Optimal Bin Packing Algorithms. PhD thesis, MIT.

[2] Garey, M.R. and Johnson, D.S. (1979). Computers and Intractability: A Guide to NP-Completeness. W.H. Freeman.

[3] Bischoff, E.E. and Ratcliff, M.S.W. (1995). Issues in the development of approaches to container loading. Omega, 23(4).

[4] Baker, B.S., Coffman Jr, E.G. and Rivest, R.L. (1980). Orthogonal packings in two dimensions. SIAM Journal on Computing, 9(4).

[5] Lodi, A., Martello, S. and Vigo, D. (2002). Recent advances on two-dimensional bin packing problems. Discrete Applied Mathematics, 123(1-3).

[6] Falkenauer, E. (1996). A hybrid grouping genetic algorithm for bin packing. Journal of Heuristics, 2(1).

[7] Hopper, E. and Turton, B.C.H. (2001). An empirical investigation of meta-heuristic and heuristic algorithms for a 2D packing problem. European Journal of

Operational Research, 128(1).

[8] Davis, L. (1985). Applying adaptive algorithms to epistatic domains. Proceedings of the International Joint Conference on Artificial Intelligence.

Appendices

Genetic Algorithm

```
INPUT: cargo_items, container
OUTPUT: best_solution

population = []
FOR i = 1 TO 300:
    genome = random_permutation([0, 1, ..., n-1])
    population.append(genome)

FOR generation = 1 TO 1500:
    sort population by fitness
    new_population = top 8 elites

    WHILE new_population.size < 300:
        parent1 = tournament_select(population)
        parent2 = tournament_select(population)
        child = order_crossover(parent1, parent2)
        child = swap_mutation(child, rate=0.2)
        new_population.append(child)

    apply local_search to top 4
    population = new_population

    IF best_fitness = 0:
        RETURN best_solution

RETURN best_solution
```

Order Crossover

```
INPUT: parent1, parent2
OUTPUT: child
```

```
start, end = random_points()
child = copy parent1[start:end]
FOR item IN parent2:
    IF item NOT IN child:
        child.append(item)

RETURN child
```

Fitness Function

```
INPUT: solution
OUTPUT: fitness

fitness = 0
fitness += unplaced_items × 1000
fitness += excess_weight × 10
fitness += com_outside_safezone × 50

RETURN fitness
```

Placement Decoder

```
INPUT: genome
OUTPUT: solution

placed = []

FOR cargo_id IN genome:
    FOR y = 0 TO container.depth:
        FOR x = 0 TO container.width:
            IF valid_position(x, y) AND within_weight_limit:
                place cargo at best COM position
                placed.append(cargo)
                BREAK
```

```
RETURN solution
```

Local Search

```
INPUT: solution
OUTPUT: improved_solution

best = solution

FOR i = 1 TO 2000:
    IF random < 0.5:
        neighbour = swap_two_items(best)
    ELSE:
        neighbour = insert_item(best)

    IF fitness(neighbour) < fitness(best):
        best = neighbour

RETURN best
```