0-1 Knapsack Problem using Genetic Algorithms

Advanced Ai mini project

Realised by: Lebga Hanane[1] , Abbou Riyad[1], Haggani Abla[1], Benounene Abdelrahmane[1]

Supervised by : Ms.Mezrar[1]

[1] Ecole National Superieur en Informatique 08/05/1945

- **ABSTRACT**

  The 0-1 Knapsack Problem (KP01) represents a classic challenge in combinatorial optimization, where the goal is to maximize the value of items placed within a knapsack of fixed capacity while adhering to specific constraints. This paper explores various methods for solving KP01, including exact algorithms like dynamic programming and heuristic approaches such as genetic algorithms (GAs). By leveraging a combination of fitness evaluation, selection, mutation, and crossover techniques, GAs provide a robust framework for tackling NP-hard problems like KP01. The results demonstrate the efficacy of genetic algorithms in producing near-optimal solutions, especially when adaptive mutation rates are employed. This study highlights the potential of combining advanced optimization strategies to solve complex real-world problems.

keywords : Combinatorial problem, optimization algorithm, 0-1 knapsack problem, metaheuristic algorithm, genetic algorithm

1 Introduction

Optimization problems are integral to decision-making in diverse domains, from resource allocation to logistics. Among these, the 0-1 Knapsack Problem (KP01) exemplifies a combinatorial challenge requiring a delicate balance between competing objectives under strict constraints. KP01 is defined by a set of items, each with associated weight and value, where the aim is to maximize total value without exceeding the knapsack's capacity. Given its classification as an NP-hard problem, finding an exact solution in polynomial time is infeasible for large instances. Consequently, researchers have turned to heuristic and metaheuristic algorithms, such as genetic algorithms (GAs), to derive satisfactory solutions efficiently. This paper delves into the methodology and effectiveness of these approaches, with particular emphasis on the adaptive capabilities of genetic algorithms.

2 COMBINATORIAL PROBLEM

A combinatorial optimization problem can be formally defined as follows:
given a finite set of feasible solution S and an objective function f: $S \rightarrow R$,
find a solution $s* \epsilon S$ such that :
$f(s*) \leq f(s) \ \forall \ s \in S$ in a minimization problem , or $f(s*) \geq f \ \forall \ s \in S$ in a maximization problem **[2]**.
These problems are often represented by graphs, networks, or sets, and the goal is to find the optimal arrangement or subset that satisfies the given criteria.
For example: the following scenario presents a combinatorial problem: one wants to buy a fashionable car while also keeping the price reasonable and within a certain limit. If we

maximize the first constraint (a good car), we will have a maximum price; conversely, if we minimize the price, we might end up with a lower-quality car within the limits. In this example, we observe that it's challenging to reconcile these two constraints according to our needs.**[3]**
In summary, combinatorial optimization problems are defined by their finite set of feasible solutions and the objective to find the best solution under given constraints. **[3]**

3 0-1 KNAPSACK PROBLEM

The KP01 is an example of a combinatorial optimization problem. It searches for the best solution among many other solutions. The objective of this problem is to maximize the sum of the values of the items in the knapsack such that the sum of the weights is less than or equal to the knapsack's capacity. There is a constraint for each item that is included in the knapsack or not and each item cannot be put into the knapsack more than once or be partially included in the knapsack.**[1]**
The mathematical model for KP01 can be formulated as follows :

0-1 Knapsack Problem:

$$\begin{cases} Maximize \sum_{i=1}^{n} v_i x_i \\ \sum_{i=1}^{n} w_i x_i \leq W \\ x_i \in \{0, 1\} \end{cases}$$

- xi represents the number of instances of the item to include in the knapsack.

KP01 belongs to the NP-hard complexity class which means that this problem can not be solved in polynomial time unless P=NP(There is no known algorithm that can solve the problem in polynomial time for all possible instances)

Example of KP01 : 'Time Management: Allocating Limited Hours to Tasks' :
Suppose that a freelancer needs to decide which projects to take in order to maximize his income, given that he has a limited number of hours available to work.
Objective: The objective is to maximize the total income while staying in the limited hours, by selecting the best project that gives the best incomes .

Project	Time(hours)	income
A	8	500
B	6	400
C	10	600
D	4	300

Figure 01 : Simple KP01

- We want to maximize the total income : $\sum_{i=1}^{4} vixi$ = 500x1+ 400x2+ 600x3 + 300x4

- Subject to this constraints : $\sum_{i=1}^{4} vixi$ = 8x1+ 6x2+ 10x3+ 4x4 $\leq$ 25

  with $xi \in \{0, 1\}$ for i=1,2,....n

4 SOLVING 0-1 KNAPSACK PROBLEM

Several programming methods and approaches are proposed to solve the KP01 problem , which can be categorized into 3 classes: exact, heuristic and metaheuristics, hybrid algorithms .

1. Exact algorithms :

Exact methods guarantee finding the optimal solution by exhaustively exploring the solution space. These methods are typically computationally intensive and are often feasible only for small to moderately sized problems.[3] Example of exact algorithms: Dynamic programming (DP), Branch-and-Bound(BB), Cutting Planes..

Algorithm 1 Pseudocode of the Implemented Dynamic Programming Method

```
for i = 0 to N do
        for j = 0 to Capacity
                if j < Weights[i] then
                Table[i, j] ← Table[i − 1, j]
                else
                        Table[i, j] ← maximum {Table[i − 1, j]
                                        AND
                                Values[i] + Table[i − 1, j − Weights[i]]}
                end
        end
end
return Table[N, Capacity]
```

Figure: Pseudocode of the Implemented Dynamic Programming Method

2. Heuristics and Metaheuristics algorithms :

- **Heuristics algorithms:** Heuristics are approximate techniques that aim to find acceptable solutions, even if they are not necessarily the best or optimal. Heuristics can be developed based on expert knowledge, intuition, trial and error, and they are designed to handle highly complex problems or problems with very large search spaces.[5][6]Among these is the Greedy search algorithm(GSA).

- **Metaheuristics algorithms :**These are more advanced approaches that use iterative search strategies to find the solution space. Among these are simulated annealing(SA), genetic algorithms, and ant colony optimization. These methods provide good solutions for large problems, while careful parameter tweaking may be required.

Algorithm 3 Pseudocode of the Implemented Simulated Annealing Algorithm

```
Initialize parameters of the annealing schedule;
Generate an initial state as the current_state;
k = 1;
repeat
    repeat
        Generate the next_state;
        Δ = value of next_state – value of current_state;
        if Δ > 0 and solution is feasible then
            current_state = next_state
            if value of current_state >value of best_state then
                best_state = current_state;
            end
        else
            acceptance_function = exp(−Δ/T_k);
            if acceptance_function > random [0,1) then
                current_state = next_state;
            end
        end
    until system equilibrium at T_k
    T_{k+1} = T_k * α
until system has been frozen
print out the current_state as the final state.
```

Figure: Pseudocode of the Implemented Simulated Annealing Algorithm

Algorithms	**Choice of techniques based on implementation and performance advantages**
GSA	• It is quite easy to come up with a greedy algorithm (or even multiple greedy algorithms) for a problem. • It requires less computing resources. • It is faster to execute.
DP	• It is well suited for a multi-stage, multi-point and sequential decision process. • It is suitable for linear or non-linear problems, discrete or continuous variables, and deterministic problems.
BB	• It can find an optimal solution (provided the problem is of limited size and enumeration can be computed in reasonable time). • Generally, it will inspect less sub-problems and thus save computational time.
GA	• GA's can work well on mixed discrete/continuous problems. • Support multi-objective optimization problems. • GA's search from a population of points, not a single point. • Makes use of an objective function and not derivatives.
SA	• It can deal with highly non-linear models, chaotic and noisy data. • It is flexible and is able to escape from a local optima.

Figure: Choice of algorithm selection for 0-1 Knapsack problem.[1]

5 GENETIC ALGORITHMS

The GA is a heuristic search and optimization algorithm inspired by natural evolution[7]The GA begins with a set of candidate solutions (chromosomes) called population. A new population is created from solutions of an old population with the hope of getting a better population. Solutions which are chosen to form new solutions (offspring) are selected according to their fitness. The more suitable the solutions are the bigger chances they have to reproduce. This process is repeated until some condition is satisfied. The main components of the GA are the chromosome encoding, the fitness function, selection, recombination and the evolution scheme [8].

Algorithm 1 Genetic algorithm

INPUT: $n, W, P, C, \text{pop}, I, p_c, p_m$

OUTPUT: LOS

Generating initial population O_1 and let $t = 1$.

while $t \leq I$ **do**

//Crossover Operator

for $k = 1 : 2 : \text{pop}$ **do**

if rand $\leq p_c$ **then**

$r_t = \text{rand} * (n-1) + 1, T = O_t^k,$

$O_t^k(r_t + 1 : n) = O_t^{k+1}(r_t + 1 : n),$

$O_t^{k+1}(r_t + 1 : n) = T(r_t + 1 : n).$

end if

end for

//Mutation Operator

$O_t = |O_t - [\text{rand}(\text{pop}, n) \leq p_m]|.$

//Selection Operator

The selection probability of the k-th individual is determined as $\frac{f(O_t^k) - \min f(O_t) + 1}{\sum_{k'=1}^{\text{pop}} f(O_t^{k'}) - \text{pop} \min f(O_t) + \text{pop}}$, and resulting in O_{t+1}.

$t = t + 1.$

end while

LOS $= \max f(O_I)$

Figure: Pseudocode of the Implemented of Genetique algorithms

Chromosome encoding:

In Genetic Algorithms (GAs), the population consists of chromosomes, which represent potential solutions to a specific problem [31]. For the Knapsack Problem (KP), binary encoding is employed, where each chromosome is represented as a sequence of bits—either 0 or 1.

Crossover operation:

New individuals are generated by combining gene segments from each parent, chosen at random. This process helps explore the solution space. After selection, individuals are randomly paired, and parent chromosomes are recombined to create two offspring that inherit traits from both parents. The number of crossover points and the probability of crossover dictate whether the parents are recombined or directly passed to the next generation. Common crossover methods include:

1. **N-Point Crossover :** This involves choosing n crossover points, then exchanging gene fragments. Common types are 1-point and 2-point crossover

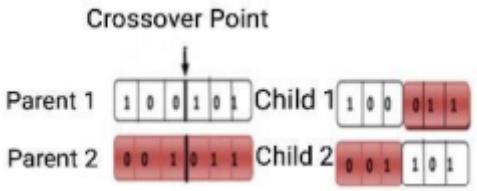


Figure: Example of single point crossover

2. **Uniform Crossover :** Each bit chosen from either of the parents with equal probability.This often results in children having more genetic information from one parent than the other.

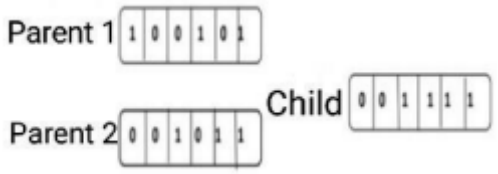


Figure: Example of uniform crossover

Mutation operation :

1. **Bit-Flip Mutation (Mutation par inversion de bit) :**This mutation applies to solutions represented as binary strings (0 and 1). The principle is to invert one or more bits in the binary string, i.e. change a 0 into a 1 or a 1 into a 0 .

   Exemple :
   String before mutation : 1010110
   String after mutation (Bit number 3 inverted) : 1000110

2. **Swap mutation :** Two genes in the chromosome are randomly selected, and their positions are swapped.

   Example: Chromosome 11001 becomes 10101 after swapping the second and fourth genes.

3. **Scramble Mutation :** A subset of genes within a chromosome is randomly shuffled. It is useful when the order of genes matters but not their specific positions.

   Example: Chromosome 11001 may have the middle three genes scrambled to 10101

4. **Upper bound mutation :** A gene's value is changed to its upper limit, ensuring it does not exceed a predefined boundary. Relevant in problems with constrained variables, such as scheduling or optimization.

   4.1 **Linear Relaxation Technique in Upper Bound Mutation:**

   The linear relaxation technique simplifies a problem by relaxing some of its constraints, typically converting discrete or integer constraints into continuous ones. This allows the solution space to expand, making it easier to explore and identify potential upper bounds for optimization.

   In the context of **upper bound mutation**, this technique involves:

   1. **Relaxing Constraints**: For problems where variables are constrained to integer or binary values, the relaxation allows these variables to take any value within a continuous range.
   2. **Calculating Upper Bounds**: By solving the relaxed version of the problem, we approximate an upper bound for the original, constrained problem.

3. **Improving Mutations**: The upper bound derived from the relaxed solution can guide mutations to more promising regions of the solution space.

Formula:

$$\text{mutation rate} = \max(0.01, 0.1 \times (1 - \frac{\text{generation}}{\text{max generations}}) \times \frac{1}{1 + \text{fitness range}})$$

Fitness score: The fitness function is a computation that evaluates the quality of the chromosome as a solution to a particular problem . The fitness function allocates a score to each chromosome in the population. This will help determine how well a particular solution solves a problem.[1]

Replacement : New individuals generated through crossover and mutation are reintegrated into the population based on their fitness. Common methods include:

- **Stationary Replacement**: Replaces parents with offspring regardless of their fitness.
- **Elitist Replacement**: Retains at least the best-performing individual to ensure quality across generations.

Algorithm Convergence : Initially, the algorithm shows rapid improvements due to global exploration of the solution space. Over time, progress slows as the focus shifts to local optimization. Variations in average performance are largely attributed to the effects of mutations.

Stopping Criteria : The process of generating and replacing individuals continues until a predefined stopping condition is met. This could be a set number of iterations, a time limit, or achieving a satisfactory solution. Once the criterion is satisfied, the algorithm outputs the best solution(s) found during the process.

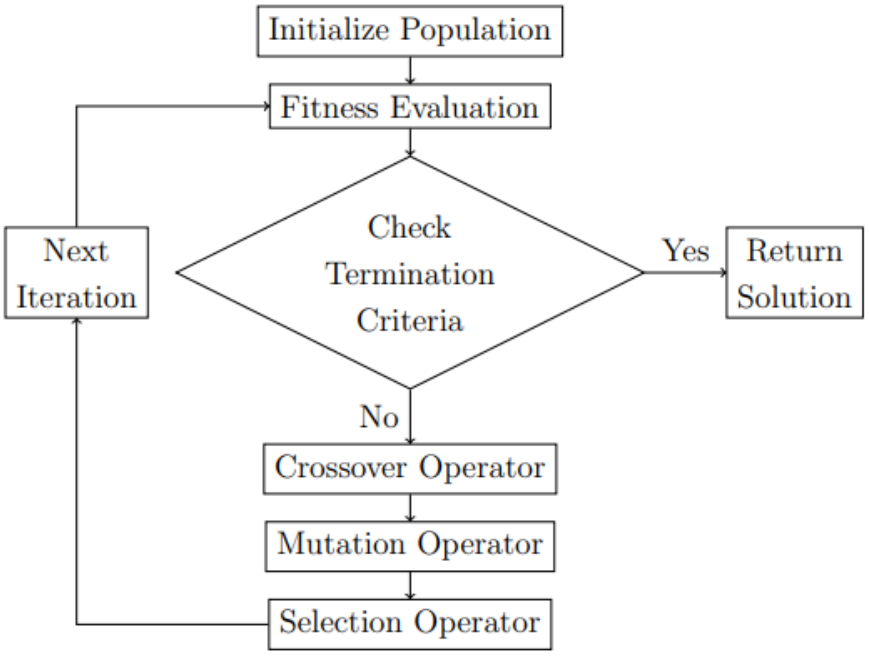


Figure: Flowchart of a basic genetic algorithm [9]

5 Proposed solution :

1. Selection:

Roulette Wheel Selection: Individuals are selected probabilistically, where fitter individuals have a higher chance of being chosen.

```python
def select_parent_roulette(population, fitnesses):
    total_fitness = sum(fitnesses)
    selection_probs = [f / total_fitness for f in fitnesses]
    return random.choices(population, weights=selection_probs, k=1)[0]
```

2. Crossover :

-Single-Point Crossover
-Two-Point Crossover
-Uniform Crossover

```python
def crossover(parent1, parent2, strategy="Single-Point"):
    if strategy == "Single-Point":
        point = random.randint(1, len(parent1) - 1)
        return parent1[:point] + parent2[point:]
    elif strategy == "Two-Point":
        point1, point2 = sorted(random.sample(range(len(parent1)), 2))
        return parent1[:point1] + parent2[point1:point2] + parent1[point2:]
    elif strategy == "Uniform":
        return [p1 if random.random() < 0.5 else p2 for p1, p2 in zip(parent1, parent2)]
```

3. Mutation :

Bit-Flipping Mutation :

```python
def mutate(individual, mutation_rate=0.1):
    for i in range(len(individual)):
        if random.random() < mutation_rate:
            individual[i] = 1 - individual[i]
```

Proposed Solution: Lagrangian Method for Dynamic Mutation Rate : While fixed mutation rates are simple, they may not adapt to the optimization process. A dynamic mutation rate adjusts based on the current state of the population, balancing exploration and exploitation. The **Lagrangian Method** introduces an adaptive mutation rate based on the generation number and the fitness landscape

Lagrangian Mutation Rate : The mutation rate is computed as :

$$\text{mutation rate} = \max(0.01, \frac{1}{1 + (\frac{\text{sum of fitnesses}}{\text{max generations}}) \times (\text{generation} + 1)})$$

Implementation:

```python
def lagrangian_mutation_rate(gen, max_gens, fitnesses):
    return max(0.01, 1 / (1 + (sum(fitnesses) / max_gens) * (gen + 1)))
```

6 Result and discussion :

1. NB of generation = 50 , NB of items = 25, Population size= 100, Knapsack capacity= 50000

crossover/mutation	fixed mutation: 0.1	fixed mutation: 0.01	upper bound: linear relaxation	Lagrangian Mutation Rate
single point	72200	73400	73000	73100
Two point	73100	73200	73500	73300
Uniform	72600	73400	73200	73200

Figure: Fitness score

2. NB of generation = 100 , NB of items = 25, Population size= 100, Knapsack capacity= 50000

crossover/mutation	fixed mutation: 0.1	fixed mutation: 0.01	upper bound: linear relaxation	Lagrangian Mutation Rate
single point	73500	73300	73200	73700
Two point	73300	73200	73300	73200
Uniform	72900	73400	73500	73700

Figure: Fitness score

For 100 generations, the Lagrangian method achieved the highest fitness score (73,700) with single-point crossover, underscoring its potential for adaptive optimization.
Uniform crossover with dynamic mutation displayed consistent performance across generations, making it a reliable choice for large-scale problems

Impact of Mutation Strategies:

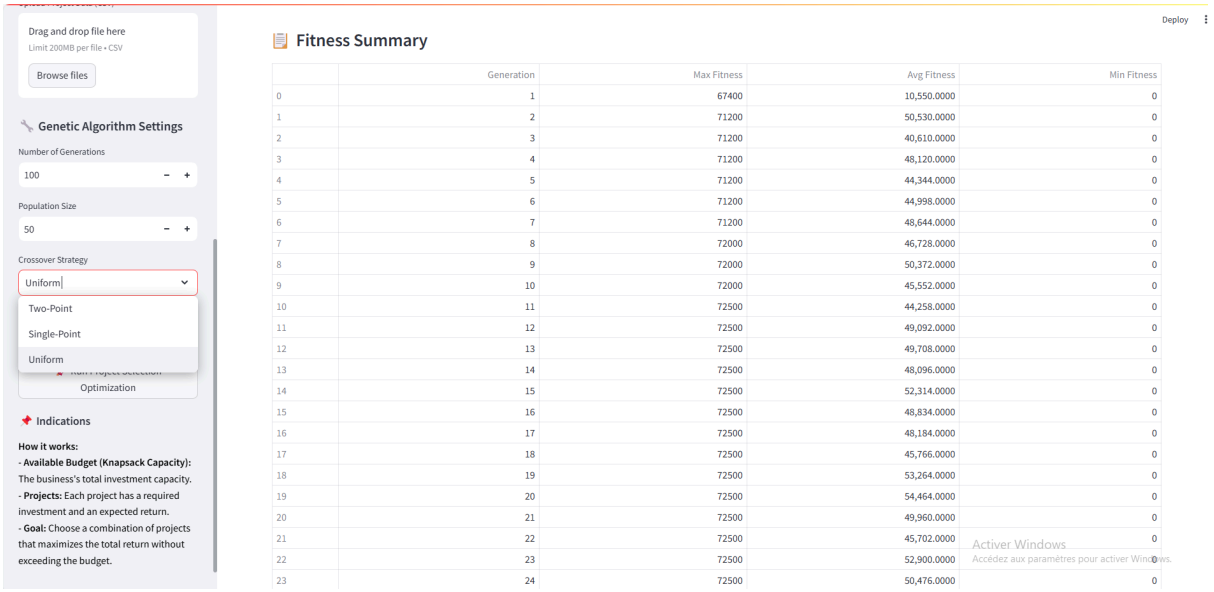
- Adaptive mutation rates, especially the Lagrangian method, consistently outperformed fixed rates in terms of fitness scores.
- Fixed mutation rates (0.1) provided stable but less optimal solutions compared to dynamic methods.

Effectiveness of Crossover Techniques:

- Single-point crossover demonstrated rapid convergence in early generations but plateaued, suggesting limited exploration in later stages.
- Uniform crossover offered a balanced exploration-exploitation trade-off, particularly effective when combined with dynamic mutation.

Deploiement :

Using streamlit and the GA algorithms the 0-1 knapsack problem was deployed to solve : Project Selection Optimization problem where the purpose of this problem is to to decide which projects to fund to maximize overall return on investment while adhering to the budget limit.



	Generation	Max Fitness	Avg Fitness	Min Fitness
0	1	67400	10,550.0000	0
1	2	71200	50,530.0000	0
2	3	71200	40,610.0000	0
3	4	71200	48,120.0000	0
4	5	71200	44,344.0000	0
5	6	71200	44,998.0000	0
6	7	71200	48,644.0000	0
7	8	72000	46,728.0000	0
8	9	72000	50,372.0000	0
9	10	72000	45,552.0000	0
10	11	72500	44,258.0000	0
11	12	72500	49,092.0000	0
12	13	72500	49,708.0000	0
13	14	72500	48,096.0000	0
14	15	72500	52,314.0000	0
15	16	72500	48,834.0000	0
16	17	72500	48,184.0000	0
17	18	72500	45,766.0000	0
18	19	72500	53,264.0000	0
19	20	72500	54,464.0000	0
20	21	72500	49,960.0000	0
21	22	72500	45,702.0000	0
22	23	72500	52,900.0000	0
23	24	72500	50,476.0000	0

Figure: Streamlit interface for Project Selection using genetic algorithm

Conclusion:

The 0-1 Knapsack Problem underscores the complexities inherent in combinatorial optimization, requiring innovative approaches to achieve practical solutions. This study demonstrates that genetic algorithms, equipped with dynamic mutation rates and tailored crossover techniques, provide a versatile and efficient means of addressing KP01. Experimental results validate the adaptability and scalability of these methods, making them well-suited for applications ranging from project selection to resource management. While exact algorithms offer theoretical guarantees, heuristic and metaheuristic strategies like GAs bridge the gap between computational feasibility and solution quality. Future work could explore hybrid models that integrate exact and heuristic methods to further enhance performance across a wider array of optimization problems.

RÉFÉRENCES :

[1]A Comparative Study of Meta-Heuristic Optimization Algorithms for 0 - 1 Knapsack Problem: Some Initial Results

[2] Kellerer, H., Pferschy, U., & Pisinger, D. (2004). Knapsack Problems. Springer.

[3] Optimizing the Multidimensional Knapsack Problem: A Hybrid Approach of Genetic Algorithm and Branch-and-Cut

[4]0-1 Knapsack Problem Solving using Genetic Optimization Algorithm

[5] COMPARISON OF METAHEURISTICS IN SOLVING THE KNAPSACK PROBLEM: AN EXPERIMENTAL ANALYSIS

[6] GOLDBARG et al., 2015

[7] C. Changdar, G. S. Mahapatra, and R. K. Pal, "An improved genetic algorithm based approach to solve constrained Knapsack problems in fuzzy environment," Expert Syst. Appl., vol. 42, no. 4, pp. 2276–2286, 2015.

[8] M. Hristakeva and D. Shrestha, "Solving the 0-1 Knapsack problem with genetic algorithms," in Proc. Midwest Instruct. Comput. Symp., Apr. 2004, pp. 8–10.

[9] An upper bound of the mutation probability in the genetic algorithm for general 0-1 knapsack problem

[10] ADDRESSING THE KNAPSACK CHALLENGE THROUGH CULTURAL ALGORITHM OPTIMIZATION

[11] Genetic Algorithms for 0/1 Multidimensional Knapsack Problems 1996

[12] Nature-inspired algorithms for 0-1 knapsack problem: A survey 2023