Badania operacyjne: Problem załadunku heterogenicznych dostawczaków

Jakub Wiśniewski, Jakub Karczewski, Jakub Grześ, Mateusz Górski

Omówienie problemu i jego potencjalnego rozwiązania.

Spis treści

1. Wstęp

1.1. Cel i założenia

Celem projektu było opracowanie algorytmu umożliwiającego optymalne rozmieszczenie paczek w pojazdach o różnych wymiarach i udźwigu. Głównym założeniem było maksymalizowanie zysku z dostarczonych paczek. W projekcie zostało zaimplementowane rozwiązanie oparte na algorytmie genetycznym zaadaptowanym do specyfiki problemu.

2. Opis zagadnienia

2.1. Dokładne sformułowanie problemu

Ze względu na specyfikę problemu, na początku należy opisać dwa jego kluczowe elementy: pojazdy oraz paczki.

Do dyspozycji mamy pewną liczbę pojazdów, z których każdy charakteryzuje się ściśle określonym udźwigiem oraz wymiarami przestrzeni ładunkowej: długością, szerokością i wysokością.

Rozważamy także zbiór paczek, z których każda ma dokładnie określoną masę oraz wymiary (długość, szerokość, wysokość). Zakładamy, że paczki mają kształt prostopadłościanów. Dodatkowo, każda paczka posiada przypisaną wartość zysku, jaką uzyskujemy po jej dostarczeniu.

Właściwy problem polega na takim rozmieszczeniu paczek w dostępnych pojazdach, aby zmaksymalizować całkowity zysk z ich dostarczenia, z zachowaniem ograniczeń przestrzennych i wagowych każdego pojazdu.

2.2. Napotkane problemy

Na początku prac nad problemem zakładaliśmy bardziej złożony wariant zadania, uwzględniający dodatkowe ograniczenia oraz rozbudowaną funkcję celu. Planowaliśmy m.in.:

- możliwość obracania paczek w trójwymiarowej przestrzeni ładunkowej, aby lepiej wykorzystać dostępne miejsce,
- przypisywanie paczkom priorytetów rozładunku, określających, w którym punkcie dostawczym powinny zostać wyładowane,
- uwzględnienie tych priorytetów w funkcji celu, tak aby premiować ułożenie paczek w sposób minimalizujący konieczność ich przestawiania.

Niestety, podczas implementacji algorytmu genetycznego napotkaliśmy na szereg trudności. Okazało się, że wprowadzenie wymienionych założeń prowadzi do bardzo skomplikowanej przestrzeni rozwiązań, której nie byliśmy w stanie efektywnie przeszukiwać nawet przy pomocy prostych algorytmów brutalnych, które miały służyć jako punkt odniesienia.

W związku z tym podjęliśmy decyzję o rezygnacji z obracania paczek oraz pominięciu systemu priorytetów. Ostateczna wersja funkcji celu została uproszczona i sprowadza się do maksymalizacji łącznego zysku z załadowanych paczek, bez uwzględniania ich kolejności wyładunku. **Priorytet paczki wciąż jest podawany na wejściu, mimo, że nie wpływa na wynik algorytmu.**

2.3. Model matematyczny

Przesyłki P:
Każdą z $i = 1...|P|$ przesyłek charakteryzuje:

- μ_i - masa,
- ω_i - wymiary (wektor 3 elementowy),
- ζ_i - zysk z dostarczenia,
- π_i - priorytet zabrania paczki.

Pojazdy V:
Każdy z $j = 1...|V|$ pojazdów charakteryzuje:

- U_j - udźwig,
- W_j - wymiary.

2.4. Postać rozwiązania

Definiujemy:

- $i \in \{1, ..., |P|\}$ - indeks paczki
- $j \in \{1, ..., |V|\}$ - indeks pojazdu
- $l \in \mathbb{N}^3$ - położenie ustalonego rogu paczki w przestrzeni pojazdu

Jako rozwiązanie problemu uznajemy znalezienie załadunku Z takiego, że:

$$\mathrm{Z} \subseteq P \times V \times l \tag{1}$$

jest to zbiór trójek oznaczających, że paczka i została umieszczona w pojeździe j na pozycji l.

2.5. Funkcja kosztu

Maksymalizujemy funkcję kosztu

$$C = \sum_{(i,j,l) \in \mathrm{Z}} \zeta_i \tag{2}$$

2.6. Ograniczenia

- Suma mas paczek w załadunku j-tego pojazdu nie może przekroczyć jego maksymalnego udźwigu U_j.

$$\forall_{v_j \in V} : \sum_{\substack{p_i \in P \\ \exists l \in \mathbb{N}^3, \text{ że } (i,j,l) \in Z}} \mu_i \leq U_j \tag{3}$$

- Paczki nie nachodzą na siebie.

$$\forall_{(i_a,j_a,l_a),(i_b,j_b,l_b) \in \mathrm{Z}} \forall_{k \in [1,2,3]} : \\ l_{a_k} \leq l_{b_k} \leq l_{a_k} + \omega_{i_1} \vee l_{a_k} \leq l_{b_k} + \omega_{i_{b_k}} \leq l_{a_k} + \omega_{i_{a_k}} \Rightarrow j_1 \neq j_2 \tag{4}$$

- Paczki nie wychodzą poza obrys pojazdu.

$$\forall_{(i,j,l) \in \mathrm{Z}} \forall_{k \in [1,2,3]} : l_k \geq 0 \wedge l_k + \omega_{i_k} \leq W_{j_k} \tag{5}$$

- Wszystkie paczki mają kształt prostopadłościanu.
- Paczki nie mogą „wisieć" w powietrzu - tzn. pod każdym punktem spodu paczki, musi znajdować się inna paczka, bądź podłoga.

$$\forall_{(i_a,j_a,l_a) \in \mathrm{Z}} : \exists_{\substack{(i_b,j_b,l_b) \in \mathrm{Z} \\ l_a = l_b}} \text{ że } l_{a_3} = 1 \vee l_{a_3} = l_{b_3} + \omega_{i_{b_3}} + 1 \tag{6}$$

2.7. Zastosowanie

Projekt bazuje na rzeczywistym problemie załadunku ciężarówek, zatem można go zastosować w każdej firmie zajmującej się transportem towarów, w celu minimalizacji pracy kierowcy i maksymalizacji zarobków.

3. Opis algorytmu

3.1. Idea algorytmu

Głowna idea algorytmu bazuje na algorytmie genetycznym, który symuluje procesy ewolucyjne w celu znalezienia optymalnych lub bliskich optymalnym rozwiązań problemu. W kolejnych generacjach populacja rozwiązań była oceniana, krzyżowana, mutowana i selekcjonowana, aby poprawić jakość rozwiązań. W implementowanym przez nas rozwiązaniu przyjęliśmy konwencję, w której algorytm genetyczny używany jest w celu odnalezienia dobrego przyporządkowania paczek do pojazdów - możliwość ułożenia paczek we wskazanych pojazdach oraz ich lokalizacja jest ustalana przy pomocy innych algorytmów. Wybór algorytmu znajdującego lokalizację paczki w pojeździe traktujemy jako dobór parametru - wybieramy z kilku autorskich implementacji, opisanych w sekcji *Sekcja 4.4)*.

3.2. Adaptacja do problemu

3.2.1. Reprezentacja osobnika

Osobnik jest reprezentowany jako wektor, w którym wartość *j* na *i-tej* pozycji oznacza, że *i-ta* paczka trafia do *j-tego* pojazdu. Przykładowo:

$$[0, 2, 1, 2, 0]$$

oznacza, że pierwsza paczka trafia do pierwszego pojazdu, druga do trzeciego, trzecia do drugiego itd. Należy zwrócić uwagę na fakt, iż w celu zachowania zgodności rozwiązania z ograniczeniami, nie wszystkie paczki zostaną zabrane przez powiązany samochód - decyzja ta należy do osobnego algorytmu, który zostanie opisany w dalszej części sprawozdania.

3.2.2. Generowanie populacji początkowej

Wstępna populacja jest losowo generowana poprzez przypisanie każdej paczce losowego pojazdu przy pomocy funkcji `generate_random_assignment`. Procedura wybiera jednostajnie z puli wszystkich pojazdów. Rozmiar populacji początkowej jest parametrem uruchomieniowym programu o nazwie „population_size".

3.2.3. Ocena osobnika

Jako że osobnik w naszym algorytmie genetycznym to przyporządkowanie paczek do pojazdów to aby wyznaczyć funkcję celu, należy najpierw wywołać procedurę budującą rozwiązanie na podstawie tego właśnie przyporządkowania. Dzieje się to poprzez wywołanie procedury `build_solution_from_assignment`, która dla danego przydziału paczek, generuje ich rozłożenie w ciężarówce. Jej pseudokod został opisany w sekcji *Sekcja 3.3)*. Etap układania paczek realizowany jest przy pomocy jednego z zaimplementowanych algorytmów wskazanego jako parametr uruchomieniowy. Mamy do dyspozycji algorytmy:

- heurystyka układająca paczki od najmniejszej
- heurystyka układająca paczki od najdroższej
- zmodyfikowany wielowymiarowy „knapsack"

Dla otrzymanego w ten sposób rozwiązania obliczenie funkcji celu jest już sprawą trywialną - iterujemy się po wszystkich paczkach, które zostały zabrane i sumujemy zysk z ich dostarczenia.

3.2.4. Selekcja

W celu wybrania osobników do rozmnażania konstruujemy ranking. Zawiera on całą aktualnie rozważaną populację. Nasz ranking może być rozumiany poprzez uszeregowanie wszystkich osobników po malejącej wartości wyliczonej na ich podstawie funkcji celu. Nad tym rankingiem budujemy rozkład „Half-Cauchy" o eksperymentalnie dobranych parametrach.
Wybór „rodziców" polega na dwukrotnym spróbkowaniu wspo-

mnianego rozkładu (co w szczególności może prowadzić do dwukrotnego wybrania tego samego osobnika).

3.2.5. Krzyżowanie

Przyjęliśmy 3 różne metody krzyżowań, aby móc porównać jak ich wybór wpływa na wyniki algorytmu.

a) **Uniform crossover**, dla każdej paczki losowo wybierane jest przypisanie z jednego z dwóch rodziców z prawdopodobieństwem 50%. Proste, pozwala przeszukać przestrzeń rozwiązań.
b) **One-point crossover**, wybierany jest jeden punkt podziału, przed którym sekwencja genów pochodzi od pierwszego rodzica, a po którym – od drugiego. Pozwala to zachować większe fragmenty dobrze działających przypisań.
c) **Heuristic crossover**, dla każdej paczki sprawdzane są dwa pojazdy: ten przypisany przez pierwszego i drugiego rodzica. Z tych dwóch wybierany jest ten, który ma aktualnie mniej przypisanych paczek, w przypadku remisu wybieramy losowo. Premiuje to równomierne rozłożenie paczek.

3.2.6. Mutacje

Mutowanie „dzieci" zaimplementowaliśmy jako proces dwuetapowy: najpierw losujemy czy nowopowstały osobnik będzie „mutantem" (na co istnieje szansa „mutation_percentage" - parametr uruchomieniowy), a następnie „mutanci" podlegają procedurze mutacji genów. Polega ona na tym, że każdy ich gen ma szanse (podaną jako parametr uruchomieniowy „gen_mutation_percentage") na zmianę przypisania na dowolny pojazd (wybrany z rozkładu równomiernego nad całym zbiorem pojazdów). W szczególności może to być ten sam pojazd.

3.2.7. Tworzenie nowej populacji

Generowanie nowej populacji odbywa się pzy użyciu procedury `reproduce_best` w następujący sposób: z aktualnej populacji brane jest „parent_percentage” (parametr uruchomieniowy) procent najlepszych rozwiązań, pozostałe (100 - „parent_percentage”) procent rozwiązań, to „dzieci” generowane poprzez proces krzyżowania opisany powyżej. Następnie, każde z dzieci ma szanse na zmutowanie w sposób również już poprzednio opisany.

3.2.8. Warunek stopu

Program wykonuje się przez liczbę iteracji wskazaną jako parametr uruchomieniowy „iters”.

3.2.9. Wybór najlepszego rozwiązania

W każdej iteracji procesu genetycznego jako potencjalny finalny wynik działania algorytmu rozważany jest osobnik pierwszy według rankingu. Jako ostateczny wynik zwracany jest najlepszy spośród tych pretendentów. Otrzymujemy również plik w formacie CSV, który w każdym wierszu przechowuje: numer generacji, najlepszego osobnika w danej generacji oraz najlepszego osobnika dotychczas. Na podstawie tych danych można określić, w której iteracji po raz pierwszy odkryto najlepszego osobnika oraz jak przebiegał proces jego wyszukiwania.

3.3. Pseudokod

```
function generate_random_assignment(num_of_packages, num of vehicles):
  assignment <- array of size num_of_packages
  for i from 0 to num_of_packages - 1:
    assignment[i] <- random int from range [0, num_of_vehicles]
  return assignment

function build_solution_from_assignment(assignment, pacakages,vehicles):
  packages_per_vehicle <- array of pacakage arrays, of size lenght(vehicles)
  package_ids_per_vehicle <- array of package id arrays, of size lenght(vehicles)

  for i from 0 to length(assignment) - 1:
    vehicle_id <- assignment[i]
    packages_per_vehicle[vehicle_id].add(packages[i])
    packages_ids_per_vehicle[vehicle_id].add(i)

    solutions = (empty solutions array, cost = 0)

    for i from 0 to lenght(vehicles) - 1:
    if packages_per_vehicle[i] is not empty:
      tmp1 <- array of vehicles[i]
      (computed_solutions, cost) ← compute_solutions(packages_per_vehicle[i], tmp1)
      solutions.cost ← solutions.cost + cost

      for j from 0 to length(computed_colutions) - 1:
        computed_solutions[j].vehicle_id ← i
        computed_solutions[j].package_id ← package_ids_per_vehicle[i][j]

      add all computed_solutions to solutions.solutions

  return solutions

function mutate_assignment(assigment, num_of_vehicles)
  mutated <- copy of assignment

  for i from 0 to lenght(assignment) - 1:
    if random int from range [0, 99] < GENE_MUTATION_PERCENTAGE
      mutated[i] <- random int from range [0, num_of_vehciles]

  return mutated

function initialize_population(population_size, num_packages, num vehicles)::
  population <- empty assignment array
  for i from 0 to population_size - 1:
    assignment ← generate_random_assignment(num_packages, num_vehicles)
        population.add(assignment)
  return population
```

```
function sort_population_by_evaluation_in_place(population, packages, vehicles):
    scored_population ← empty array of pairs (cost, assignment)

    for each assignment in population:
         (solutions, cost) ← build_solution_from_assignment(assignment, packages,
vehicles)
        scored_population.add((cost, assignment))

    sort scored_population descending by cost

    population ← empty array
    for every pair (cost, assignment) in scored_population:
        population.add(assignment)

function reproduce_best(best_assignments, population_size, num_vehicles, packages,
vehicles, crossover_type):
    new_population ← empty arrray

    n ← population_size * (100 - CHILDREN_PERCENTAGE) / 100

    add n first assignments from best_assignments to new_population

    if crossover_type == "uniform":
        crossover ← UniformCrossover
    else jeśli crossover_type == "one_point":
        crossover ← OnePointCrossover
    else jeśli crossover_type == "heuristic":
        crossover ← HeuristicCrossover(packages, vehicles)

    center ← (length(best_assignments) - 1) / 2
    scale ← center / 3
    rng ← random number generator
    dist ← Cauchy distibution

    while length(new_population) < population_size:
        index_1 ← index generated from using dist,center,scale and rng
        index_2 ← index generated from using dist,center,scale and rng

        child ← crossover(best_assignments[index_1], best_assignments[index_2])

        if random int from range [0, 99] < MUTATION_PERCENTAGE:
            child ← mutate_assignment(child, num_vehicles)

        new_population.add(child)

    return new_population
```

```
function genetic_algorithm(packages, vehicles, crossover_type):
    generations ← GENERATIONS
    num_packages ← length(packages)
    num_vehicles ← length(vehicles)

    best_ever ← (empty assignment, cost = 0)

    population ← initialize_population(POPULATION_SIZE, num_packages, num_vehicles)

    for gen from 0 to generations - 1:
        sort_population_by_evaluation_in_place(population, packages, vehicles)

                                      (best_solutions,    current_best_cost)    ←
build_solution_from_assignment(population[0], packages, vehicles)

        if current_best_cost > best_ever.koszt:
            best_ever ← (population[0], current_best_cost)

           population ← reproduce_best(population, POPULATION_SIZE, num_vehicles,
packages, vehicles, crossover_type)

    return build_solution_from_assignment(best_ever.assignment, packages, vehicles)
```

4. Aplikacja

4.1. Format wejścia

Wejście aplikacji wygląda w następujący sposób i jest podawany na stdin:

rodzaj_krzyżowania
N
X Y Z W
...
n
x y z p w c
...

gdzie:

- rodzaj_krzyżowania dotyczy wyłącznie algorytmu genetycznego (nie powinien być on podany dla pozostałych metod), to string „uniform" / „one_point" / „heuristic", metody te zostały opisane w *Sekcja 3.2.5)*.
- N to liczba ciężarówek
- X, Y, Z to wymiary ciężarówki
- W to maksymalny udźwig ciężarówki
- n to liczba paczek
- x, y, z to wymiary paczki
- p to priorytet paczki
- w to maksymalny udźwig ciężarówki
- c to zysk za dostarczenie paczki

Przykładowo:
uniform
3
13 19 15 190
16 15 13 188
14 12 16 197
3
10 8 2 3 47 19
2 9 8 1 22 85
8 2 4 4 47 43

4.2. Format wyjścia

Wyjście aplikacji wygląda w następujący sposób:

package_id: p vehicle_id: v package_position: (x, y, z)
...
Total profit: c

gdzie:

- p to id paczki
- v to id pojazdu
- x, y, z to położenie paczki
- c to całkowity zysk

Przykładowo:
package_id: 0 vehicle_id: 0 package_position: (7, 15, 0)
package_id: 1 vehicle_id: 0 package_position: (10, 6, 0)
package_id: 2 vehicle_id: 0 package_position: (1, 7, 0)
Total profit: 848

4.3. Jak uruchomić program

Program był konstruowany z myślą o systemie Linux. W celu uruchomienia programu należy odpowiednio dobrać parametry umieszczone w nagłówku pliku „genetic.cpp". Następnie należy uruchomić skrypt „compile_all.sh" w celu kompilacji całego projektu. Następnie można uruchomić program wynikowy „solve.out" oraz podać input na standardowym wejściu. Wynik zostanie obliczony i wysłany na standardowe wyjście automatycznie po zakończeniu działania.

4.4. Alternatywne podejścia

Oprócz opisanego wyżej podejścia algorytmu genetycznego zaimplementowaliśmy także dwie alternatywne metody znajodwania rozwiązań:

- Heurystyka

Algorytm heurystyczny działa on w sposób zachłanny – dla każdej paczki próbuje znaleźć pierwsze dostępne miejsce w jednym z pojazdów, zaczynając od końca przestrzeni ładunkowej (czyli od najwyższych współrzędnych x i y) i przeszukując poziomy od dołu do góry (w osi z). Dla każdej paczki algorytm sprawdza, czy spełnia ograniczenia z sekcji *Sekcja 2.6)*

Jeśli wszystkie te warunki są spełnione, paczka zostaje dodana do rozwiązania, a zajmowane przez nią pozycje są oznaczane jako zajęte.

Ze względu na zachłanny charakter działania, nie gwarantuje on znalezienia globalnie najlepszego możliwego rozmieszczenia paczek.

- Knapsack

Algorytm oparty na problemie plecakowym ma na celu optymalne przypisanie paczek do pojazdów w taki sposób, aby zmaksymalizować łączny zysk, jednocześnie spełniając ograniczenia wagowe i przestrzenne pojazdów. Ze względu na te wymogi, nie jest możliwe zastosowanie trywialnej memoizacji jak w zwykłym wielowymiarowym problemie plecakowym. Wynika to z warunku konieczność całkowitego „podparcia" paczek, jak i z tego, że nie zajmujemy się kilkoma „cechami" opisującymi jakieś miary (tak jakby to było w przypadku wielowymiarowego problemu plecakowego, gdzie każda paczka miałaby przykładowo wagę, koszt, wysokość, odporność, i każda z tych cech miałaby jakieś górne ograniczenie jako „pojemność" plecaka, wtedy każdą cechę zwiększaliśmy osobno przy decyzji o dołożeniu paczki), tylko przestrzenią 3D, której opisanie za pomocą programowania dynamicznego generowałoby ogromną liczbę stanów.

Więcej o tym znajduje się w sekcji 9, na samym końcu sprawozdania.

W pierwszym etapie, paczki przypisywane są do poszczególnych pojazdów zgodnie z ustalonym wektorem przypisań. Dla każdej grupy paczek przypisanych do danego pojazdu wywoływana jest funkcja rozwiązująca problem plecakowy z uwzględnieniem ograniczeń przestrzennych.

Zastosowanie tego podejścia pozwala na znalezienie optimum, kosztem większego czasu obliczeń.

Powyższe metody można przetestować po wykonaniu plików knapsack_solve.out oraz heuristic_solve.out (są one kompilowane przy okazji wykonania skryptu compile_all.sh).

5. Eksperymenty

5.1. Dobór parametrów

5.1.1. Zmieniane parametry

Testowaniu podlegały następujące parametry:

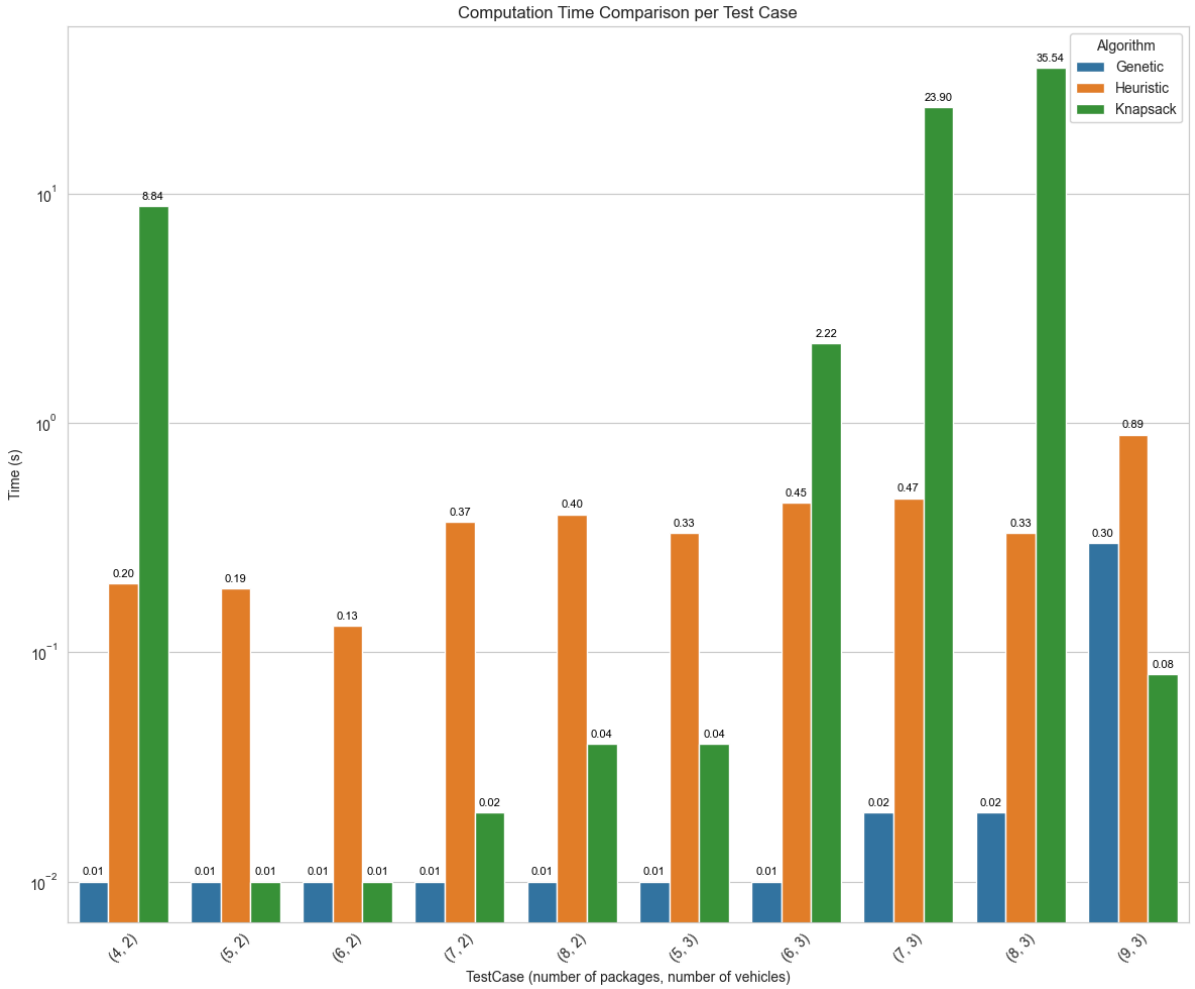
- Wykorzystany algorytm
  (do wyboru z heurystyki, genetycznego i knapsack)
- Typ krzyżowania, procent populacji potomnej
  i szanse na mutacje (w przypadku zastosowania algorytmu genetycznego)
- Liczba oraz przedziały wymiarów pojazdów
  (po zdefiniowaniu losowane z rozkładem jednostajnym)
- Liczba oraz przedziały wymiarów paczek
  (po zdefiniowaniu też losowane z rozkładem jednostajnym)

5.1.2. Obserwacje i wybrane wartości

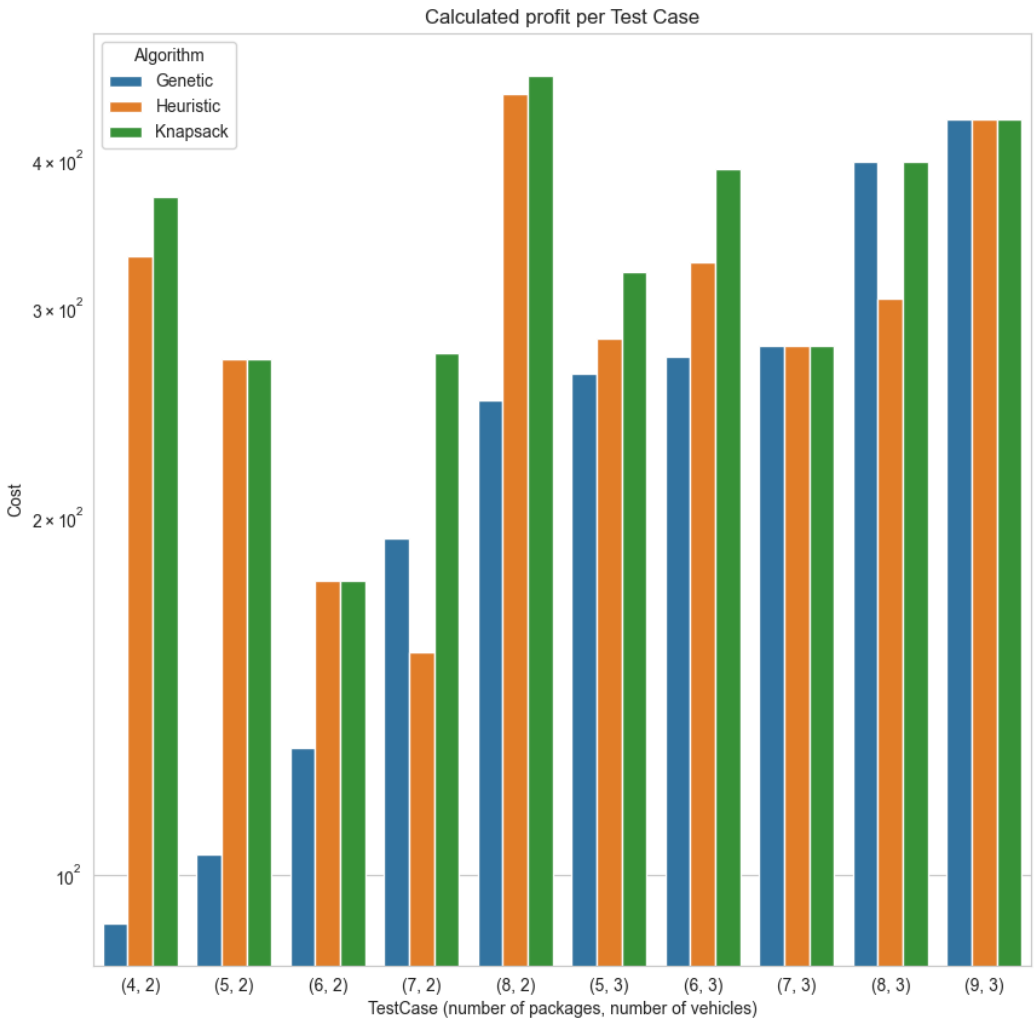
5.1.2.1. Wykresy dla małych wartości

Poniższe wykresy zostały wygenerowane dla niewielkiej liczby paczek i pojazdów, a także niewielkich wymiarów pojazdów. Dla algorytmu genetycznego zostały przypisane w obu przypadkach następujące parametry:

- POPULATION SIZE = 200
- CHILDREN_PERCENTAGE = 80
- MUTATION_PERCENTAGE = 25
- GENE_MUTATION_PERCENTAGE = 40
- RODZAJ_KRZYŻOWANIA = 'uniform' (dotyczy tylko algorytmu genetycznego)



Rysunek 1: Porównanie czasu wykonania dla niewielkich danych



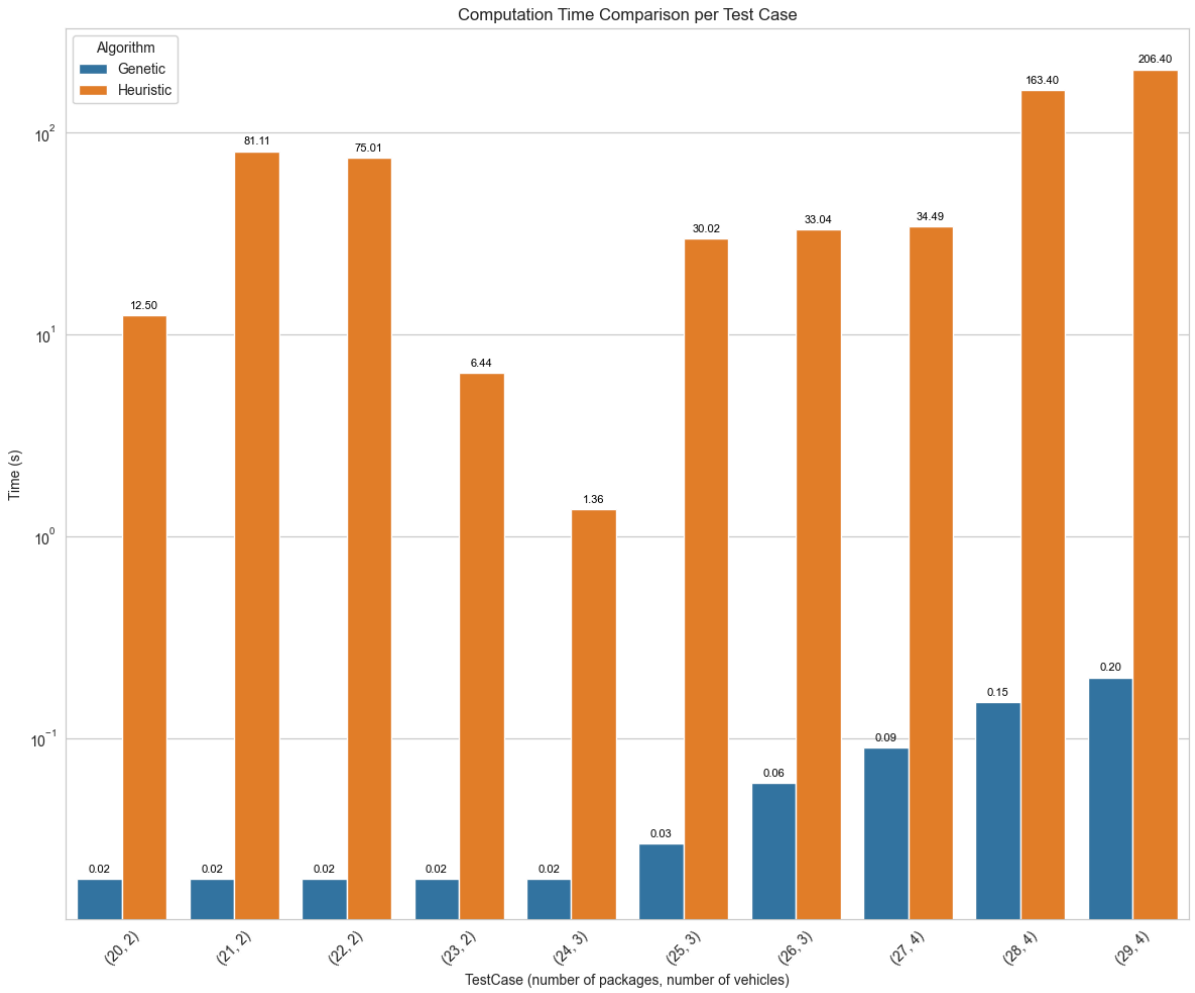
Rysunek 2: porównanie jakości algorytmów dla małych danych , w kontekście sumarycznego kosztu

5.1.2.2. Wykresy dla większych danych

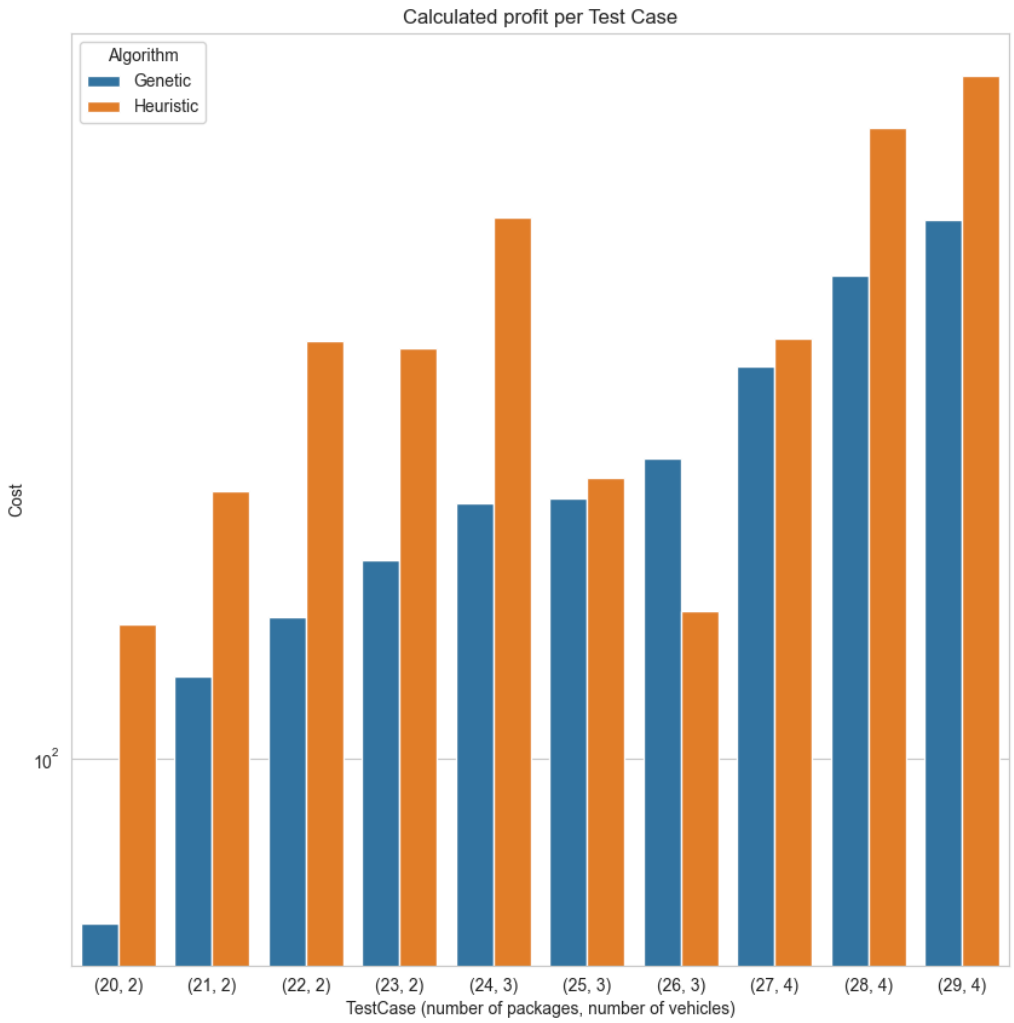
W tym zestawieniu wykonane testy dla danych z przedziałów:

- 20-30 paczek
- 2-4 pojazdy, średnie wymiary pojazdów

pominięte zostało wykonanie „knapsacka", gdyż nie poradziłby sobie z takimi danymi w sensownym czasie.



Rysunek 3: porównania czasu wykonania dla dużych danych



Rysunek 4: porównanie jakości algorytmów dla dużych danych , w kontekście sumarycznego kosztu

5.1.2.3. Obserwacje na podstawie wykresów

Na podstawie pierwszych dwóch wykresów można zauważyć, że pod względem obliczonej wartości rozwiązania, algorytmy: genetyczny i heurystyczny radzą sobie trochę gorzej niż knapsack.

Jednak można zauważyć, że już dla takich niewielkich przedziałów danych testowych, jego czas wykonania jest bardzo duży w porównaniu do pozostałych metod. Dodatkowo, już dla tak małych danych obserwuje się, że algorytm genetyczny wykonuje się wielokrotnie szybciej w porównaniu do heurystyki.

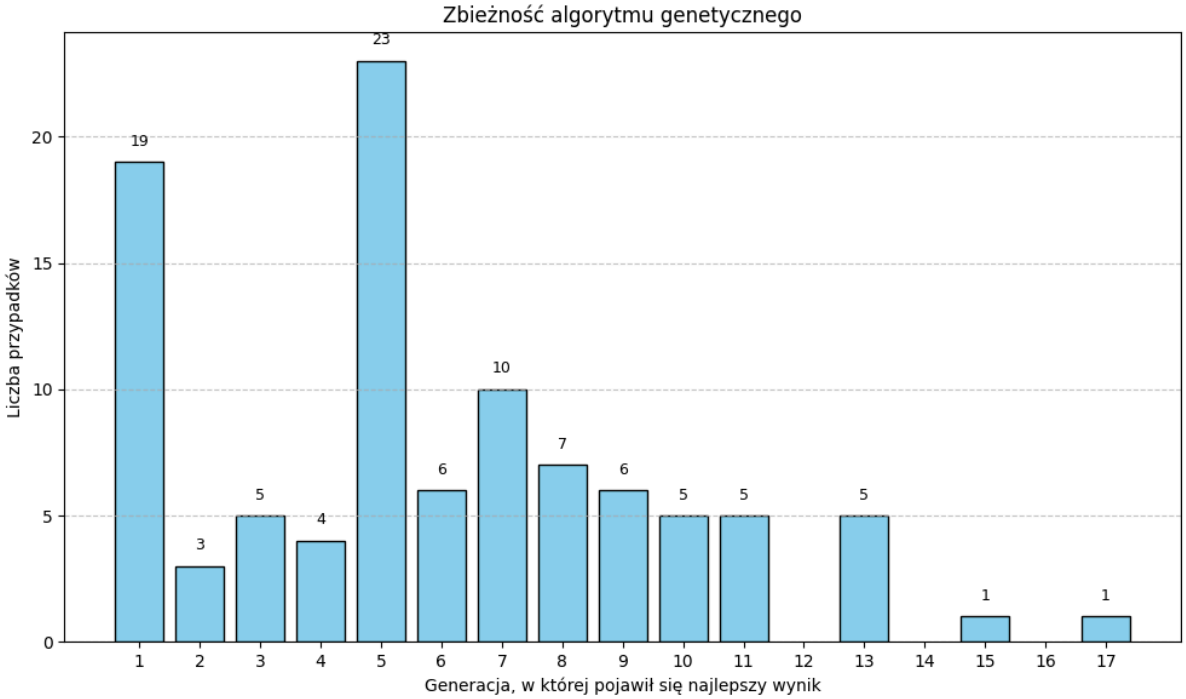
Na podstawie wykresów dla większych danych, można dostrzec jeszcze większą przepaść między czasami wykonania genetycznego i heurystyki, gdzie genetyczny potrafi się wykonywać nawet ponad 100 razy szybciej niż heurystyka.

Chociaż uzyskiwane rozwiązania są w około 90% przypadków gorsze (przynajmniej w zakresie przedstawionym na wykresie), spadek jakości jest stosunkowo niewielki w porównaniu do znacznych oszczędności czasowych.

Algorytm genetyczny wykazuje zauważalną przewagę czasową, co sugeruje, że przy znacznie większych zbiorach danych będzie wielokrotnie bardziej efektywny, nie tracąc przy tym istotnie na jakości uzyskiwanych wyników.

5.2. Zbieżność algorytmu genetycznego

W celu zbadania zbieżności algorytmu genetycznego tj. generacji, w której pojawił się najlepszy algorytm, został uruchomiony 100 razy na pojedynczym przypadku testowym, gdzie maksymalna liczba iteracji wynosiła 200. Wyniki tych testów prezentują się następująco:



Można zaobserwować, że w zdecydowanej większości przypadków najlepsze rozwiązanie zostało znalezione już w 1. lub 5. generacji, a liczba generacji, w której osiągnięto najlepszy wynik, nigdy nie przekroczyła 17. Tak wczesne znalezienie optymalnego rozwiązania wynika z dobrze zaprojektowanej heurystyki do układania paczek w pojazdach, która dzięki swojej skutecznej implementacji często znajduje rozwiązanie optymalne już we wczesnych generacjach.

5.2.1. Wykresy i wnioski z doboru parametrów dla algorytmu genetycznego

5.2.1.1. Opis eksperymentu

Algorytm genetyczny został dla wielu zestawów danych przetestowany w następujący sposób:

```
# Definicje początkowe
start_params ← [
  [0, 30, 25, 40],
  [0, 60, 15, 40],
  [0, 60, 25, 20]
]
params_steps ← [
  [0, 10, 0, 0],
  [0, 0, 10, 0],
  [0, 0, 0, 10]
]
populations ← [50, 100, 150, 200, 250]
strategies  ← ["uniform", "one_point", "heuristic"]

# Główna pętla eksperymentów
FOR z IN 0 .. LENGTH(populations) - 1 DO
  population ← populations[z]

  FOR l IN 0 .. LENGTH(strategies) - 1 DO
    strategy ← strategies[l]

    FOR k IN 0 .. LENGTH(params_steps) - 1 DO

      FOR i IN 0 .. NUM_TESTS - 1 DO

        # Skopiuj parametry bazowe
        params ← COPY(start_params[k])

        # Ustaw rozmiar populacji
        params[0] ← population

        # Zastosuj kroki do pozostałych parametrów
        FOR j IN 1 .. LENGTH(params) - 1 DO
          params[j] ← params[j] + i * params_steps[k][j]
        END

        # Uruchom solver
        # Każde wywołanie obsługuje te same dane wejściowe
        run_solver(strategy, params, input_file)

      END
```

Poniżej jest przedstawiona tabelka reprezentująca najlepsze 30 wyników pod względem parametru $\frac{\text{koszt}}{\text{czas wykonania}}$. Czas został podany w sekundach.

	cross_type	pop_size	child_perc	mut_perc	gen_mut_perc	cost	exec_time
0	heuristic	50	60	25	50	626	5.560000
1	heuristic	50	60	45	40	561	5.630000
2	heuristic	50	60	55	40	570	5.730000
3	heuristic	50	60	25	40	649	6.620000
4	uniform	50	40	25	40	566	5.780000
5	heuristic	50	70	25	40	493	5.110000
6	one_point	50	60	25	40	424	4.700000
7	one_point	50	60	45	40	619	6.920000
8	uniform	50	50	25	40	513	5.770000
9	uniform	50	60	55	40	599	6.740000
10	one_point	50	70	25	40	519	5.970000
11	one_point	50	60	25	30	485	5.580000
12	uniform	50	60	25	40	592	6.990000
13	heuristic	50	50	25	40	492	5.920000
14	one_point	50	60	25	20	425	5.180000
15	uniform	50	60	45	40	416	5.110000
16	uniform	50	60	25	30	489	6.050000
17	one_point	50	60	25	40	478	5.920000
18	heuristic	50	60	25	60	462	5.780000
19	uniform	50	60	15	40	523	6.610000
20	one_point	50	50	25	40	510	6.620000
21	one_point	50	60	55	40	543	7.280000
22	uniform	50	70	25	40	358	4.860000
23	one_point	50	60	25	40	527	7.240000
24	uniform	50	60	25	40	639	9.030000
25	heuristic	50	40	25	40	639	9.220000
26	heuristic	50	60	15	40	537	7.900000
27	heuristic	50	60	25	40	500	7.370000
28	heuristic	50	60	25	40	362	5.410000
29	one_point	50	60	15	40	563	8.450000

Rysunek 5:
Top 30 wyników

Na podstawie tabelki można wywnioskować, że ustawienie :

- POPULATION_SIZE jedynie na 50
- MUTATION_PERCENTAGE na 25 %
- CHILD_PERCENTAGE na około 70 %
- GENE_MUTATION_PERCENTAGE na około 40 %

pozwalało uzyskiwać najlepsze wyniki.

Warto również zaznaczyć, że spośród 10 najlepszych rozwiązań przedstawionych w powyższej tabeli, aż 5 wykorzystało metodę krzyżowania typu heuristic. Patrząc jednak całościowo, rozkład procentowy zastosowania poszczególnych metod krzyżowania wydaje się dość wyrównany.

Zważywszy na ogromną liczbę możliwych kombinacji parametrów, nie sposób było znaleźć konfigurację absolutnie optymalną, jednak uzyskane rezultaty prawdopodobnie znajdują się bardzo blisko optimum. Należy również pamiętać, że wszystkie eksperymenty zostały przeprowadzone na jednym, tym samym zestawie danych, składającym się z 5 pojazdów i 20 paczek o szerokim zakresie wymiarów.

6. Wnioski

Nasze rozwiązanie adaptujące algorytm genetyczny zdecydowanie przyspiesza rozwiązywanie zadanego problemu, jednocześnie go nie komplikując. Daje on jednak gorsze wyniki niż zasobożerne rozwiązanie brutalne. Na podstawie przeprowadzonych przez nas eksperymentów i porównań wnioskujemy, że algorytm ten nadaje się do rozwiązania tego problemu, ale tylko jeżeli nie zależy nam na rozwiązaniu dokładnym, a jedynie na szybkim otrzymaniu rozwiązania bliskiego optymalnego. Jeśli mielibyśmy podejść do tej adaptacji jeszcze raz od początku, to rzeczą, którą moglibyśmy zrobić inaczej jednocześnie, być może zmieniając efektywność algorytmu, byłoby przedefiniowanie osobnika w algorytmie genetycznym tak aby nie reprezentował on jedynie przydziału paczka - pojazd, a np. przydział paczka - pojazd - położenie.

7. Podział pracy

Imie i Nazwisko	Model matematyczny	Heurystyka	Adaptacja do algorytmu genetycznego	Testowanie wariantów	Suma
Jakub Grześ	40%	10%	45%	5%	25%
Jakub Karczewski	10%	10%	25%	55%	25%
Jakub Wiśniewski	40%	35%	25%	0%	25%
Mateusz Górski	10%	45%	5%	40%	25%

8. Podejście do problemu przy użycie ILP

Poniżej przedstawione zostały wyniki pracy kolegi, który zaczął rozważania dotyczące implementacji ILP w projekcie, ale nie zdążył doprowadzić ich do końca.

8.1. Zmienne binarne

- $X(v_{\text{id}}, x, y, z, p_{\text{id}})$ - pozycja (x, y, z) w pojeździe o id: v_{id} jest zapełniona przez paczkę o id: p_{id}
- $Y(v_{\text{id}}, x, y, z, p_{\text{id}})$ - paczka o id: p_{id} jest „zaczepiona" w pojeździe v_{id} na pozycji (x, y, z)
- $Z(p_{\text{id}})$ - paczka o id: p_{id} została wykorzystana, czyli trafiła do któregoś pojazdu, zmienna pomocnicza

8.2. Warunki:

Dla każdego $Y(i, x, y, z, j)$, jeśli ma wartość 1:

- suma po wszystkich $X(i, x_1, y_1, z_1, l)$ musi być równa 0,
  co zapewnia że żadna inna paczka nie zajmuje tych pól.
  (l != j oraz
  x_1 należy do $[x, x + \delta_x]$ oraz
  y_1 należy do $[y, y + \delta_y]$ oraz
  z_1 należy do $[z, z + \delta_z]$,
  $(\delta_x, \delta_y, \delta_z)$ to wymiary paczki o id j)

- Każdy $X(i, x_1, y_1, z_1, j)$ musi być równy 1, bo skoro umieszczamy tą paczkę to wypełniamy za jej pomocą wszystkie pola.
  (j jest takie samo jak poziom wyżej, przedziały wymiarów jak podpunkt wyżej)
- Dla każdego pola (x, y) pod powierzchnią paczki, którą chcemy umieścić, suma po $X(i, x, y, z - 1, l)$ musi być równa 1.
  (zmienia się tutaj tylko l)
  Aby łatwo pozbyć się problemu z przypadkiem z = 0 dodajemy na dole sztuczną „warstwę" zmiennych o przypisanej z góry wartości 1.

Dla każdej paczki o id j:

- Suma po wszystkich $Y(i, x, y, z, j)$ jest równoważna $Z(j)$
  (wszystkie zmienne w X poza j dowolnie dobrane, zgodnie z warunkami zadania)

8.3. Funkcja celu:

$$\sum_{j=0}^{n} Z_j * k_j \tag{7}$$

,
Gdzie Z_j to zmienna dotycząca wykorzystania paczki, a k_j to jej wartość.

8.4. Środowisko

Mozna wykorzystać bibliotekę ORTools od Google, a dokładnie:

- linear_solver typu „cbc_mixed_integer_programming”,
  w przypadku sprowadzenia naszego problemu danego SAT-em do ILP
- CP-SAT solver,
  w przypadku pozostawienia sformułowania problemu poprzez SAT

9. Dlaczego memoizacja nie zadziała w przypadku problemu załadunku paczek

W problemie klasycznego plecaka, gdzie mamy paczki z pewnymi atrybutami (takimi jak waga, objętość, koszt, odporność), każda decyzja sprowadza się do prostego wyboru: czy dodać paczkę do „obecnego stanu plecaka", reprezentowanego przez wartości wektorów (np. pozostała waga, miejsce, budżet itd.), czy ją pominąć. Te stany są łatwe do jednoznacznego opisania jako kombinacje liczb — np. DP[pozostała_waga][pozostała_objętość][pozostały_budżet]. Dzięki temu memoizacja jest wydajna — przestrzeń stanów da się sensownie ograniczyć i przeglądać.

W przypadku 3D-problemu układania paczek sytuacja jest znacznie trudniejsza — i fundamentalnie inna.

9.1. Stan nie da się łatwo zapisać jako proste wektory liczbowych zasobów

W klasycznym DP stan plecaka to liczba (lub kilka liczb). W przypadku 3D-załadunku stanem jest… kształt zajętej przestrzeni w trójwymiarze, czyli zbiór nieciągłych bloków w przestrzeni. Każda paczka może zostać umieszczona w wielu miejscach, a możliwych konfiguracji geometrycznych zajętej przestrzeni jest astronomicznie wiele — o wiele więcej niż kombinacji liczbowych wektorów.

Nie istnieje łatwy sposób, by porównać, czy dwa stany są równoważne: zajęta przestrzeń o tym samym „wolumenie" może wyglądać zupełnie inaczej i generować inne możliwości dalszego dołożenia paczek.

9.2. „Szatkowanie" przestrzeni 3D jest nieregularne

Dodanie jednej paczki do przestrzeni 3D nie redukuje dostępnego miejsca w sposób łatwy do przewidzenia (np. nie „zmniejsza" pojemności jak w plecaku). Przestrzeń jest „szatkowana" nieregularnie: paczka może być dołożona w narożniku, wzdłuż ściany, albo jako most nad innymi paczkami. Każda taka operacja zmienia topologię wolnej przestrzeni w trudny do opisania sposób. To oznacza, że rekurencyjne rozgałęzienie nie daje się sprowadzić do prostych stanów i przejść między nimi.

9.3. Reguła podparcia (stabilność) czyni każdy stan zależnym od ułożenia poprzednich paczek

Nie można dodać paczki „w powietrzu” — musi być podparta (czyli musi leżeć na podłodze lub na innych paczkach). To oznacza, że decyzja o tym, czy można dodać paczkę, zależy od konkretnego ułożenia wcześniej dodanych paczek. Zatem nie tylko zajętość przestrzeni jest istotna, ale topologia tej zajętości — które paczki są na których poziomach i gdzie. To czyni pamiętanie stanów (memoizację) praktycznie niemożliwym, bo dwa stany o identycznym rozmiarze zajętej przestrzeni mogą mieć zupełnie inne znaczenie, jeśli chodzi o dalsze możliwości ułożenia.

9.4. Liczba możliwych stanów rośnie wykładniczo z wymiarami pojazdu

W klasycznym DP mamy skończoną i kontrolowaną liczbę stanów — z góry ustaloną przez rozmiary plecaka. W przypadku przestrzeni 3D, każda zmiana prowadzi do nowego, potencjalnie unikalnego rozkładu wolnych miejsc. Gdybyśmy chcieli każdy z tych stanów zapamiętać (memoizować), szybko zabrakłoby pamięci — nawet dla bardzo małych instancji problemu.