Zagadnienie optymalizacji zbiorów w sadzie algorytmem symulowanego wyżarzania i genetycznym

Badania Operacyjne 2
Automatyka i Robotyka 2021/2022

Skład zespołu:

Wojciech Żyła

Magdalena Leonkiewicz

Piotr Hudaszek

Spis treści

1 Wstęp

Celem projektu było zbudowanie modelu matematycznego zbiorów owoców w sadzie, zaimplementowanie go w Pythonie oraz maksymalizacja zysku algorytmami symulowanego wyżarzania i genetycznym.

2 Opis zagadnienia

2.1 Sformułowanie problemu

Poruszany problem dotyczy zagadnienia maksymalizacji zysku ze zbiorów w sadzie owocowym. W sadzie jest kilka rodzajów owoców wiadomo, ile kg owoców jest każdego typu.

Aby zebrać owoce należy zapłacić pracownikom zależnie od zebranej ilości, pracownicy nie są w stanie zebrać więcej niż ustalone dzienne maksimum.

Owoce można sprzedawać na targu lub skupie. Na targu każdy rodzaj owocu ma określony popyt na dany dzień, co oznacza, że nie można sprzedać więcej niż popyt, ale można mniej wtedy w zależności od procent zaspokojenia popytu cena się zmienia. Można sprzedawać też owoce na skupie, gdzie nie ma limitów sprzedaży, ale cena może być niższa. Dana jest cena na każdy dzień i typ owoców w skupie i na targu.

Owoce można przechowywać w magazynie jeden dzień, łączna ilość owoców wszystkich typów w magazynie nie może przekroczyć pojemności magazynu

Rozwiązaniem jest lista dni (standardowo 30) w każdym z nich dla każdego typu owocu, ile owoców zebrać, sprzedać na targu, skupie lub magazynować. Niezależnie od podjętych decyzji ponosimy też stałe koszty utrzymania sadu.

2.2 Model matematyczny

S – ilość typów owoców
s – typ owoców $\in \{1,2,\dots S\}$
N_s - ilość owoców danego typu
t – dzień $\in \{1,2,3,\dots,30\}$

$f_{t\,s}$ – ilość owoców zebranych typu s dnia t
f_t – ilość owoców zebranych wszystkich typów dnia t
f_{max} – max ilość owoców jaką można dziennie zebrać
$cost(f)$ – koszt zebrania f owoców dziennie (zwykła tabela)
$plant\_cost(s)$ – koszt utrzymania danego typu owoców

n_{ts} – ilość sprzedanych na targu owoców typu s dnia t
sk_{ts} – ilość sprzedanych na skupie owoców typu s dnia t
$popyt(s,t)$ – popyt na owoce typu s dnia t
$price(s,t)$ – cena bazowa na targu
$price_{skup}(s,t)$ – cena w skupie
$pzp\big(n_{st}, popyt(s,t)\big)$ – mnożnik ceny bazowej zależny od procentu zaspokojenia popytu

m_{t_s} – ilość owoców typu s przekazanych w dniu t do magazynu (muszą zostać sprzedane następnego dnia)
m_{max} – maksymalna pojemność magazynu
$magaz(m_t)$ – koszt magazynowania m owoców przez noc

Postać rozwiązania

$$\left\{\begin{matrix}(f_{1\,1}, f_{1\,2}, \dots f_{1\,S})\\(n_{1\,1}, n_{1\,2}, \dots n_{1\,S})\\(sk_{1\,1}, sk_{1\,2}, \dots sk_{1\,S})\\(m_{1\,1}, m_{1\,2}, \dots m_{1\,S})\end{matrix}\right\}, \left\{\begin{matrix}(f_{2\,1}, f_{2\,2}, \dots f_{2\,S})\\(n_{2\,1}, n_{2\,2}, \dots n_{2\,S})\\(sk_{2\,1}, sk_{2\,2}, \dots sk_{2\,S})\\(m_{2\,1}, m_{2\,2}, \dots m_{2\,S})\end{matrix}\right\}, \dots \left\{\begin{matrix}(f_{30\,1}, f_{30\,2}, \dots f_{30\,S})\\(n_{30\,1}, n_{30\,2}, \dots n_{30\,S})\\(sk_{30\,1}, sk_{30\,2}, \dots sk_{30\,S})\\(m_{30\,1}, m_{30\,2}, \dots m_{30\,S})\end{matrix}\right\}$$

Funkcja celu (zysk)

$$\sum_{t=1}^{30}\{\sum_{s=1}^{S}[n_{st}\cdot pzp\big(n_{st}, popyt(s,t)\big)\cdot price(s,t) + sk_{st}\cdot price_{skup}(s,t)] - cost(f_t) - magaz(m_t)\} - \sum_{s=1}^{S} plant\_cost(s)$$

Ograniczenia

Dla każdego s: $\sum_{t=1}^{30} f_{s\,t} < N_s$

$f_t \le f_{max}$
$n_{ts} \le popyt(s,t)$
$m_{s\,1} = 0$
$m_{s\_30} = 0$
$m_t \le m_{max}$
$f_{t\,s}, n_{t\,s}, sk_{t\,s}, m_{t\,s} \ge 0$

Dla każdego t i s: $m_{t-1\,s} + f_{t\,s} - n_{t\,s} - sk_{t\,s} - m_{t\,s} = 0$

3 Opis algorytmów

3.1 Elementy wspólne dla obu algorytmów

3.1.1 Reprezentacja rozwiązania

Postać rozwiązania została zaimplementowana jako klasa `Solution` która zawiera listę rozwiązań dla każdego dnia `DaySolution` która zawiera 4 listy:

- zebrane (`harvested`)
- sprzedane na targu (`sold_market`)
- sprzedane na skupie (`sold_wholesale`)
- magazynowane (`warehouse`)

Wszystkie o długości równej ilości typów owoców

```
class DaySolution:
    def __init__(self, fruit_types):
        self.harvested = [0] * fruit_types  # lista zebranych owoców danego dnia (każdy indeks to inny typ owocu)
        self.sold_market = [0] * fruit_types  # lista owoców sprzedanych na targu (każdy indeks to inny typ owocu)
        self.sold_wholesale = [0] * fruit_types  # lista owoców sprzedanych w skupie (każdy indeks to inny typ owocu)
        self.warehouse = [0] * fruit_types  # lista owoców, które po danym dniu trafiły do magazynu (każdy indeks to inny typ owocu)


# postać rozwiązania aby dostać ile owoców sprzedanych na targu 8 dnia typu 2 należy solution.days[7].sold_market[1]
class Solution:
    def __init__(self, fruit_types: int, num_days):
        """
        :param fruit_types: ilość typów owoców
        """
        self.days = []
        for _ in range(num_days):
            d = DaySolution(fruit_types)
            self.days.append(d)

    def __str__(self):
        txt = ""
        for i, day in enumerate(self.days):
            txt += f"Day {i + 1}\nharvested: {day.harvested}\nsold_market: {day.sold_market}\n"
                   f"sold_wholesale: {day.sold_wholesale}\nwarehouse: {day.warehouse}\n\n"
        txt += "\n\n\n\n"
        return txt
```

3.1.2 Sprawdzenie czy rozwiązanie spełnia ograniczenia

```
def check_fruit_limits(solution: Solution, fruit_types: List[FruitTypeInfo]):
    """
    Funkcja sprawdzająca czy łączne zbiory danego
    typu owocu nie przekraczają fizycznej ilości danego typu owocu.

    Zwraca True jeśli limit został spełniony, w innym przypadku
    zwraca False

    :param solution: rozwiązanie
    :param fruit_types: lista typów owoców
    :return: boolean
    """
    # Lista przechowująca łączne zbiory każdego typu.
    # Indeks odpowiada typowi owocu.
```

```python
    total_harvested = [0] * len(fruit_types)

    for day in solution.days:
        harv = day.harvested
        # w poniższej pętli dodaję do łącznej
        # liczby zbiorów zbiory z danego danego dnia
        for i in range(len(total_harvested)):
            total_harvested[i] += harv[i]

    result = True
    for id, fruit in enumerate(fruit_types):
        if total_harvested[id] > fruit.quantity:
            # Jeśli łączna ilość zebranych owoców
            # z danego typu będzie większa niż
            # istniejąca ilość owoców, to znaczy
            # że warunek nie został spełniony i nie musimy dalej wykonywać pętli
            result = False
            break
    return result


def check_harvest_limits(solution: Solution, max_daily_harvest: int):
    """
    Funkcja sprawdzająca czy dzienne zbiory wszystkich
    owoców łącznie nie przekraczają dziennego limitu zbiorów.

    Zwraca True jeśli limit został spełniony, w innym przypadku
    zwraca False

    :param solution: rozwiązanie
    :param max_daily_harvest: maksymalna dzienna ilość zbiorów
    :return: boolean
    """
    result = True
    for day in solution.days:
        # Jeśli któregoś dnia ilość zebranych
        # owoców będzie większa niż dopuszczalny limit
        # to warunek nie jest spełniony i można przerwać
        # dalszą pętlę
        if sum(day.harvested) > max_daily_harvest:
            result = False
            break
    return result


def check_warehouse_limits(solution: Solution, warehouse_capacity: int):
    """
    Funkcja sprawdzająca czy zbiory przekazane do magazynu
    nie przekraczają jego pojemności.

    Zwraca True jeśli limit został spełniony, w innym przypadku
    zwraca False

    :param solution: rozwiązanie
    :param warehouse_capacity: maksymalna pojemność magazynu
    :return: boolean
    """
    result = True
    for day in solution.days:
        # Jeśli któregoś dnia ilość owoców przekazanych
        # do magazynu będzie większa niż jego pojemność
        # to warunek nie jest spełniony i można przerwać
        # dalszą pętlę
        if sum(day.warehouse) > warehouse_capacity:
            result = False
            break
    return result


def check_minimum_amount_sold(solution: Solution, fruit_types: List[FruitTypeInfo]):
    """
    Funkcja sprawdza, czy w ciągu miesiąca zostały
    na targu sprzedane minimlane, z góry założone, ilości
    dla każdego typu owocu.

    Zwraca True jeśli limit został spełniony, w innym przypadku
    zwraca False
```

```
    :param solution:
    :param fruit_types: lista obiektów klasy FruitTypeInfo
    :return: boolean
    """
    # Lista przechowująca łączną sprzedaż na targu każdego typu.
    # Indeks odpowiada typowi owocu.
    total_market_sold = [0] * len(fruit_types)

    for day in solution.days:
        sold = day.sold_market
        # w poniższej pętli dodaję do łącznej
        # liczby sprzedanych owoców sprzedaż z danego dnia
        for i in range(len(total_market_sold)):
            total_market_sold[i] += sold[i]

    result = True
    for id, fruit in enumerate(fruit_types):
        if total_market_sold[id] < fruit.min_market_sold:
            # Jeśli łączna ilość sprzedanych na targu owoców
            # z danego typu będzie mniejsza niż
            # wymagany limit to znaczy, że warunek nie został spełniony
            # i nie musimy dalej wykonywać pętli
            result = False
            break
    return result


def check_if_warehouse_sold(solution: Solution) -> bool:
    """
    Sprawdza czy owoce, które dnia poprzdzającego zostały przekazane do magazyni zostały sprzedane
    w obecym dniu.

    :param solution:
    :return:
    """
    result = True
    for i, day in enumerate(solution.days):
        if i > 0:
            prev_warehouse = sum(solution.days[i - 1].warehouse)
            today_sold = sum(day.sold_market) + sum(day.sold_wholesale)
            if today_sold < prev_warehouse:
                # Jeśli sprzedano mniej niż poprzedniego dnia
                # włożono do magazynu to znaczy, że warunek nie został spełniony
                result = False
                break
    return result


def check_if_today_amount_correct(solution: Solution) -> bool:
    """
    Sprawdza czy w danym dniu rozwiązania ilość owoców sprzedanych oraz przekazanych
    do magazynu jest rówa ilości owoców zebranych plus owoców z poprzedniego dnia.

    :param solution:
    :return:
    """
    result = True
    for i, day in enumerate(solution.days):
        harvest = sum(day.harvested)
        today_sold = sum(day.sold_market) + sum(day.sold_wholesale)
        today_warehouse = sum(day.warehouse)
        if i > 0:
            prev_warehouse = sum(solution.days[i - 1].warehouse)
            if today_sold + today_warehouse != harvest + prev_warehouse:
                result = False
                break
        else:
            if today_sold + today_warehouse != harvest:
                result = False
                break
    return result


def check_if_non_negative(solution: Solution) -> bool:
    """
    Funkcja sprawdzająca, czy wszystkie parametry w rozwiązaniu są
    nieujemne.
```

```
    :param solution:
    :return:
    """

    def is_non_negative(arr):
        result = True
        for el in arr:
            if el < 0:
                result = False
                break
        return result

    result = True
    for i, day in enumerate(solution.days):
        if not (is_non_negative(day.harvested) and is_non_negative(day.sold_market)
                and is_non_negative(day.sold_wholesale) and is_non_negative(day.warehouse)):
            result = False
            break
    return result


def check_if_sold_market_less_than_demand(solution: Solution, fruit_types: List[FruitTypeInfo]) -> bool:
    """
    Funkcja sprawdzająca czy ilość owoców sprzedanych na targu jest
    mniejsza lub równa od popytu
    :param fruit_types:
    :param solution:
    :return:
    """
    result = True
    for i, day in enumerate(solution.days):
        for fruit_id in range(len(fruit_types)):
            # Sprawdź czy ilość owoców sprzedanych na targu
            # nie jest większa niż popyt danego dnia
            if day.sold_market[fruit_id] > fruit_types[fruit_id].demand[i]:
                result = False
                break
    return result


def check_if_sol_acceptable(self, solution: Solution) -> bool:
    """
    Sprawdza wszystkie ograniczenia dla danego rozwiązania w najprostszej wersji
    zwraca bool, w bardziej skomplikowanej informacje gdzie błąd i ewentualnie kara

    :param solution:
    :return:
    """
    one = check_fruit_limits(solution, self.fruit_types)
    two = check_harvest_limits(solution, self.max_daily_harvest)

    # Z ograniczenia trzeciego zrezygnowaliśmy
    #three = check_minimum_amount_sold(solution, self.fruit_types)
    four = check_warehouse_limits(solution, self.warehouse_capacity)
    five = check_if_warehouse_sold(solution)
    six = check_if_today_amount_correct(solution)
    seven = check_if_non_negative(solution)
    eight = check_if_sold_market_less_than_demand(solution, self.fruit_types)

    #print(one, two, four, five, six)

    return one and two and four and five and six and seven and eight
```

3.1.3 Generacja rozwiązania / populacji początkowej

Ta część jest wspólna dla obu algorytmów. W przypadku symulowanego wyżarzania jako rozwiązanie początkowe wybieramy jedno z rozwiązań z wygenerowanej populacji.
Generowanie rozwiązania są tworzone metodą konstrukcyjną na dwa sposoby.

Pierwszy, prostszy w swoich założeniach, zakłada, że wszystkie zebrane owoce idą na sprzedaż do skupu tego samego dnia. Podczas tworzenia rozwiązania nie trzeba się zatem martwić o sprzedawanie owoców na targu bądź oddawanie ich do magazynu. Funkcja generująca takie rozwiązanie przyjmuje parametr *harvest_strategies*. Parametr ten to lista list, gdzie wewnętrzne listy to poszczególne strategie zbiorów. Strategia składa się z ilości dni przez ile dana strategia obowiązuje oraz listy odpowiadającej maksymalnej ilości owoców z danego typu zbieranej w ciągu dnia. Przykładowo mamy 3 typy owoców: jabłka, gruszki, śliwki. *harvest_strategies* może wyglądać następująco:

[[15,[20, 30, 25]], [15,[10, 25, 19]]] co oznacza, że przez pierwsze 15 dni zbieramy *dziennie* 20kg jabłek, 30kg gruszek, 25kg śliwek a przez kolejne 15 dni 10kg jabłek, 25kg gruszek, 19kg śliwek. Oczywiście, jeżeli danego w sadzie zostało już mniej owoców niż wynika to z powyższych parametrów to zbieramy tą mniejszą ilość owoców tym samym wykańczając zapasy danego owocu w sadzie. Funkcja ta jest metodą klasy reprezentującej sad, posiada więc dostęp do zmiennych tej klasy.

```python
def generate_all_to_wholesale(self, harvest_strategies: List[List]):

    # ilość rodzajów owoców
    fruit_types_count = len(self.fruit_types)

    # Słownik fruits_left jest wykorzystywany do zapamiętywania
    # ile owoców danego typu zostało w sadzie po każdym dniu zbiorów
    fruits_left = {}
    for f_type in self.fruit_types:
        fruits_left[f_type.name] = f_type.quantity

    solution = Solution(fruit_types_count, self.num_days)

    day_id = 0
    for strategy in harvest_strategies:
        harvest_per_type = strategy[1]
        for i in range(strategy[0]):
            for fruit_id, f_type in enumerate(self.fruit_types):
                # Ile owoców wciąż mamy w sadzie
                f_left = fruits_left[f_type.name]

                # Określenie wielkości zbiorów danego dnia
                if harvest_per_type[fruit_id] <= f_left:
                    solution.days[day_id].harvested[fruit_id] = harvest_per_type[fruit_id]
                    fruits_left[f_type.name] -= harvest_per_type[fruit_id]
                else:
                    solution.days[day_id].harvested[fruit_id] = f_left
                    fruits_left[f_type.name] -= f_left
                solution.days[day_id].sold_wholesale[fruit_id] = solution.days[day_id].harvested[fruit_id]
            day_id += 1

    return solution
```

Drugi sposób zakłada, że każdego dnia staramy się spełnić popyt na dany owoc. Funkcja przyjmuje trzy parametry, *harvest_strategies* który działa tak jak w poprzedniej funkcji, *demand_rate* przyjmujące wartość z zakresu [0-1] i określające jaki procent popytu na dany dzień dla danego owocu chcemy zapełnić oraz *random_demand_rate* przyjmujące wartość True lub False. Jeśli *random_demand_rate* ma wartość True to dla każdego dnia i każdego owocu jest losowana wartość parametru *demand_rate* z zakresu [0.3-1] (testy pokazały, że algorytm znajduje lepsze rozwiązania, jeśli *random_demand_rate* ma wartość False). Po zaspokojeniu popytu, jeśli zostaną nam jeszcze owoce to 70% z tych pozostałych chcemy sprzedać w skupie a resztę wsadzamy do magazynu. Jeśli w magazynie nie ma już miejsca to tą resztę również przeznaczamy na sprzedaż w skupie. W sytuacji, gdy mamy do dyspozycji mniej owoców niż popyt jaki chcemy zaspokoić to sprzedajemy wszystko co mamy i nic nie trafia do skupu ani do magazynu. Kod wygląda następująco:

```python
def generate_satisfy_demand(self, harvest_strategies: List[List], demand_rate: float = 1, random_demand_rate: bool = False):

    # ilość rodzajów owoców
    fruit_types_count = len(self.fruit_types)

    # Słownik fruits_left jest wykorzystywany do zapamiętywania
    # ile owoców danego typu zostało w sadzie po każdym dniu zbiorów
    fruits_left = {}
```

```python
for f_type in self.fruit_types:
    fruits_left[f_type.name] = f_type.quantity

solution = Solution(fruit_types_count, self.num_days)

# Po sprzedaży owoców na targu pewna ilość musi trafić
# do skupu i pewna do magazynu jeśli się tam zmieści.
# percent_to_wholesale określa jaki procent tych owoców
# początkowo chcemy dać do skupu podczas gdy reszta trafi do magazynu.
# Jeśli reszta nie zmieści się w magazynie to na koniec też przeznaczamy
# ją do skupu.
percent_to_wholesale = 0.7

day_id = 0
for strategy in harvest_strategies:
    harvest_per_type = strategy[1]
    for i in range(strategy[0]):
        for fruit_id, f_type in enumerate(self.fruit_types):
            # Ile owoców wciąż mamy w sadzie
            f_left = fruits_left[f_type.name]

            # Określenie wielkości zbiorów danego dnia
            if harvest_per_type[fruit_id] <= f_left:
                solution.days[day_id].harvested[fruit_id] = harvest_per_type[fruit_id]
                fruits_left[f_type.name] -= harvest_per_type[fruit_id]
            else:
                solution.days[day_id].harvested[fruit_id] = f_left
                fruits_left[f_type.name] -= f_left

            if random_demand_rate:
                demand_rate = random.uniform(0.3, 1)

            # Popyt jaki staramy się zaspokoić
            demand = int(f_type.demand[day_id] * demand_rate)
            if day_id == 0:
                # Pierwszy dzień (brak magazynu z dnia poprzedniego)
                if demand >= solution.days[day_id].harvested[fruit_id]:
                    solution.days[day_id].sold_market[fruit_id] = solution.days[day_id].harvested[fruit_id]
                    # pozostałości przeznaczone do skupu lub magazynu
                    leftovers = 0
                else:
                    solution.days[day_id].sold_market[fruit_id] = demand
                    # pozostałości przeznaczone do skupu lub magazynu
                    leftovers = solution.days[day_id].harvested[fruit_id] - demand
                    solution.days[day_id].sold_wholesale[fruit_id] = int(percent_to_wholesale * leftovers)
            else:
                if demand > solution.days[day_id - 1].warehouse[fruit_id]:
                    # Popyt większy niż ilość owoców z magazynu z poprzedniego dnia
                    solution.days[day_id].sold_market[fruit_id] = solution.days[day_id - 1].warehouse[fruit_id]
                    if demand - solution.days[day_id - 1].warehouse[fruit_id] >= solution.days[day_id].harvested[fruit_id]:
                        # Sytuacja gdy owoce z magazynu nie zaspokoiły popytu na targu a ilość
                        # zebranych owoców jest na tyle niska że możemy wszystkie również przeznaczyć
                        # do sprzedaży na targu
                        solution.days[day_id].sold_market[fruit_id] += solution.days[day_id].harvested[fruit_id]
                        # pozostałości przeznaczone do skupu lub magazynu
                        leftovers = 0
                    else:
                        # Sytuacja gdy owoce z magazynu nie zaspokoiły popytu na targu a ilość
                        # zebranych owoców wystarcza na zaspokojenie tego popytu oraz zostaje nam
                        # jeszcze trochę wolnych owoców
                        solution.days[day_id].sold_market[fruit_id] = demand
                        # pozostałości przeznaczone do skupu lub magazynu
                        leftovers = solution.days[day_id].harvested[fruit_id] - (demand - solution.days[day_id - 1].warehouse[fruit_id])
                        solution.days[day_id].sold_wholesale[fruit_id] = int(percent_to_wholesale * leftovers)
                elif demand == solution.days[day_id - 1].warehouse[fruit_id]:
                    # Popyt równy owocom z magazynu
                    solution.days[day_id].sold_market[fruit_id] = solution.days[day_id - 1].warehouse[fruit_id]
                    # pozostałości przeznaczone do skupu lub magazynu
                    leftovers = solution.days[day_id].harvested[fruit_id]
                    solution.days[day_id].sold_wholesale[fruit_id] = int(percent_to_wholesale * leftovers)
                else:
                    # Popyt mniejszy niż owoce z magazynu
                    solution.days[day_id].sold_market[fruit_id] = demand
                    # pozostałości przeznaczone do skupu lub magazynu
                    leftovers = solution.days[day_id].harvested[fruit_id]
                    solution.days[day_id].sold_wholesale[fruit_id] = int(percent_to_wholesale * leftovers) + \
                                                         solution.days[day_id - 1].warehouse[fruit_id] - demand

            # Jeśli część owoców przeznaczona do magazynu zmieści się w nim to możemy je tam wsadzić.
            # W innym wypadku również trafiają one do skupu.
            if leftovers - int(percent_to_wholesale * leftovers) + sum(
                    solution.days[day_id].warehouse) <= self.warehouse_capacity:
                solution.days[day_id].warehouse[fruit_id] = leftovers - int(percent_to_wholesale * leftovers)
            else:
                solution.days[day_id].sold_wholesale[fruit_id] += leftovers - int(percent_to_wholesale * leftovers)
```

```
        day_id += 1

    return solution
```

Mając już opisane dwie metody tworzenia rozwiązań początkowych można przejść do opisania funkcji która tworzy początkową populację z wykorzystaniem obu powyższych funkcji. W tej funkcji najpierw należy utworzyć różne strategie zbiorów *harvest_strategies* które były omówione przy funkcji *generate_all_to_wholesale*. Następnie wywoływane są obie powyższe funkcje które tworzą rozwiązania początkowe dla różnych parametrów.

```
def create_initial_population(self, random_demand_rate: bool = False):
    # ilość rodzajów owoców
    fruit_types_count = len(self.fruit_types)    # lista do przechowywania rozwiązań
    solutions = []


    # Zmienne har_per_type określają ile owoców danego typu
    # chcemy zbierać. Zakładamy tutaj, że wszystkich owoców zbieramy
    # po równo. Musimy również pamiętać że łączne zbiory nie mogą
    # przekroczyć dziennego limitu zbiorów. Symbol // oznacza dzielenie
    # całkowite.
    har_per_type1 = self.max_daily_harvest // fruit_types_count
    har_per_type2 = int(0.9 * self.max_daily_harvest) // fruit_types_count
    har_per_type3 = int(0.7 * self.max_daily_harvest) // fruit_types_count
    har_per_type4 = int(0.5 * self.max_daily_harvest) // fruit_types_count


    # lista do przechowywania poszczególnych strategii zbiorów
    all_strategies = []


    x = self.num_days//4
    # harvest_strategies1 to lista będąca strategią zbiorów
    harvest_strategies1 = []


    # zapis [har_per_type1] * fruit_types_count oznacza stworzenie
    # listy o długości fruit_types_count wypełnionej wartością har_per_type1.
    harvest_per_type = [har_per_type1] * fruit_types_count
    harvest_strategies1.append([x, harvest_per_type])
    harvest_per_type = [har_per_type2] * fruit_types_count
    harvest_strategies1.append(([x, harvest_per_type]))
    harvest_per_type = [har_per_type3] * fruit_types_count
    harvest_strategies1.append(([x, harvest_per_type]))
    harvest_per_type = [har_per_type4] * fruit_types_count
    harvest_strategies1.append(([self.num_days-3*x, harvest_per_type]))

   all_strategies.append(harvest_strategies1)



 # W identyczny sposób jak harvest_strategies_1 w kodzie jest tworzonych jeszcze
 # kilka strategii. Następnie wywoływane są funkcje generate_all_to_wholesale oraz
  # oraz generate_satisfy_demand z różnymi parametrami oraz strategiami. Przykładowe
  # wywołania zostały pokazane poniżej. Znalezione rozwiązania są dodawane do listy
 # rozwiązań.


 solution = self.generate_all_to_wholesale(all_strategies[0])
  solutions.append(deepcopy(solution))
  solution = self.generate_satisfy_demand(all_strategies[0], 0.6,
                              random_demand_rate=random_demand_rate)
  solutions.append(deepcopy(solution))


 # Następnie stworzone powyżej strategie zbiorą są edytowane na
  # dwa sposoby, aby powstały strategie, w których zbieramy różne ilości owoców
  # każdego typu.





all_strategies2 = deepcopy(all_strategies)
# Edycja listy strategii zbiorów w taki sposób, że owoców
# pierwszego typu zbieramy najwięcej a każdych następnych
# coraz mniej.
for i in range(len(all_strategies)):
```

```
    # Pętla po listach ze strategiami (elementy z all_strategies)
    for strat_id in range(len(all_strategies[i])):
        # Pętla po danych strategiach w danej liście ze strategiami
        # (elementy na przykład z harvest_strategies1)
        for fruit_id in range(len(all_strategies[i][strat_id][1])):
            # Pętla po zbiorach danego typu owocu w danej strategii
            i_end = fruit_types_count-1-fruit_id
            if fruit_id < i_end:
                multiplier = (fruit_types_count-1-fruit_id)/fruit_types_count
                all_strategies[i][strat_id][1][fruit_id] += int(all_strategies[i][strat_id][1][i_end]*multiplier)
                all_strategies[i][strat_id][1][-fruit_id-1] -= int(all_strategies[i][strat_id][1][i_end]*multiplier)
            else:
                break


# Edycja listy strategii zbiorów w taki sposób, że
# dla sąsiadujących ze sobą typów owoców w sposób losowy
# dobieramy różnicę z pewnego zakresu i dla jednego owocu
# z pary dodajemy tą różnicę a dla drugiego odejmujemy.
# Przykładowo ze strategii [20, 20, 20, 20, 20] może powstać
# strategia [23, 17, 16, 24, 20]. Dla pierwszej pary różnica
# to -3, dla drugiej 4 a piąta liczba nie ma pary więc została taka jak oryginalnie.
for i in range(len(all_strategies2)):
    # Pętla po listach ze strategiami (elementy z all_strategies)
    for strat_id in range(len(all_strategies2[i])):
        # Pętla po danych strategiach w danej liście ze strategiami
        # (elementy na przykład z harvest_strategies1)
        for fruit_id in range(0,len(all_strategies2[i][strat_id][1]),2):
            # Pętla po zbiorach danego typu owocu w danej strategii
            if fruit_id <= len(all_strategies2[i][strat_id][1])-2:
                percent = random.uniform(0.85, 1.15)
                defaul_fruits = all_strategies2[i][strat_id][1][fruit_id]
                fruit_delta = defaul_fruits-int(defaul_fruits * percent)
                all_strategies2[i][strat_id][1][fruit_id] -= fruit_delta
                all_strategies2[i][strat_id][1][fruit_id+1] += fruit_delta


# Następnie ponownie wywoływane są funkcje generate_all_to_wholesale oraz
# oraz generate_satisfy_demand z różnymi parametrami i zedytowanymi strategiami.


# Na koniec przechdzimy w pętli po wszystkich rozwiązaniach i do ostatecznej listy
# z rozwiązaniami dodawane jest rozwiązanie wraz ze sprawdzeniem czy spełnia ono
# ograniczenia.


result = []
for el in solutions:
    result.append((el, self.check_if_sol_acceptable(el)))
return result
```

3.2 Algorytm SA (symulowanego wyżarzania)

3.2.1 Schemat algorytmu

W tym algorytmie użyliśmy 2 kryteria stopu:

1. Osiągnięcie temperatury minimalnej (`T_stop`)
2. Przez określoną liczbę iteracji (`iter_epsilon`) wartość funkcji celu nie zmieniała się o więcej niż `epsilon`

```
def simulated_annealing(T_start, T_stop, iter_in_temp, epsilon, iter_epsilon, alpha):

    solution = create_initial_population()[0] #Znalezienie rozw. początkowego
    best_solution = solution
    best_profit = self.calculate_objective_fun(solution)
    T = T_start                         #Temperatura początkowa

    while T > T_stop:                   #1 kryterium stopu
        for j in range(iter_in_temp):
            cand_sol = draw_solution(solution)        #losowanie sąsiada z otoczenia
            cand_sol_fun = calc_objective_fun(cand_sol) #wyznaczenie fun celu dla wylosowanego

            delta = cand_sol_fun - self.calc_objective_fun(solution)

            #obliczenie różnicy wart funkcji celu pomiędzy starym a nowym rozw
```

```
            if delta >= 0:      #polepszenie rozwiazania
                solution = cand_sol
                profit_lst.append(cand_sol_fun)
                if cand_sol_fun > best_profit:    #Uzyskano nowe najlepsze rozwiązanie
                    best_solution = solution
                    best_profit = cand_sol_fun
            else:
                drawn_num = np.random.rand()
                if drawn_num < math.exp(delta/T):
                    solution = cand_sol    #przyjęcie jako gorszego rozwiązania jako aktualne
        T = alpha * T       #liniowa zmiana tempertury
        if spełnione_2_kryt_stopu:
            return best_solution, best_profit

return best_solution, best_profit
```

3.2.2 Wyznaczenie sąsiedniego rozwiązania

Aby znaleźć rozwiązanie sąsiednie wybieramy z rozwiązania oryginalnego losowo dzień (t) i typ owoców (s). Dla tego dnia i typu wybieramy losowo i zmieniamy ilość zebraną ($f_{t\,s}$) lub ilośc sprzedaną na targu ($n_{t\,s}$) lub ilość sprzedaną na skupie ($sk_{t\,s}$). Zmiana polega na wylosowaniu liczby z zbioru $\{-2, -1, 0, 1, 2\}$ i dodaniu jej.

Niestety takie generowanie rozwiązań sprawiało, że prawie wszystkie rozwiązania były niedopuszczalne. Głównie z powodu niespełnienia ograniczenia:

Dla każdego t i s: $m_{t-1\,s} + f_{t\,s} - n_{t\,s} - sk_{t\,s} - m_{t\,s} = 0$

Dlatego po zmianie rozwiązania zostaje wywołana metoda, która w zmienionym dniu i typie liczy, ile kg owoców powinno zostać przekazane do magazynu zgodnie z wzorem:

$$m_{t\,s} = m_{t-1\,s} + f_{t\,s} - n_{t\,s} - sk_{t\,s}$$

Następnie liczy deltę między starym stanem magazynowym w tym dniu a wyliczonym. Jeśli delta jest większa od zera to powiększamy sprzedaż dnia następnego o deltę i dzięki temu nie musimy zmieniać kolejnych dni. Zwiększamy sprzedaż najpierw na targu, jeśli popyt zaspokojony to resztę na skupie. Analogicznie, jeśli delta jest mniejsza od zera to zmniejszamy sprzedaż robimy to najpierw na skupie, następnie (jeśli na skupie 0) to na targu.

Mimo tych operacji nie każde wyznaczone rozwiązanie spełnia ograniczenia, dlatego po wylosowaniu sąsiada sprawdzamy, czy jest dopuszczalny, jeśli nie to losujemy następnego. Robimy tak aż znajdziemy dopuszczalnego, jeśli w 100 iteracjach się to nie uda to program zwraca błąd. Po tych zmianach program działał i około 90% losowanych sąsiadów było dopuszczalnych. Zastanawialiśmy się nad zaimplementowaniem funkcji kary, aby ominąć odrzucanie rozwiązań, ale uznaliśmy, że byłoby to bardzo skomplikowane.

3.3 Algorytm genetyczny

3.3.1 Schemat algorytmu

```
def genetic_algorithm(self, max_iter_no_progress, max_iter, replacement_rate=0.5, mutation_proba=0.2,
                      verbose: bool=True, random_demand_rate: bool=False, return_best_results: bool=False):
    """
```

```
Metoda znajdująca rozwiązanie optymalne za pomocą algorytmu genetycznego

:param bruteforce_comapre: lista stretegii zbiorów przekazywana podczas porównania z ręczym obliczeniem
                 rozwiązania
:param return_best_results: Parametr używany podczas eksperymentów. Jeśli jest ustawiony na True
                 to funkcja zwraca dodatkową listę z najlepszymi wynikami w każdej iteracji.
:param random_demand_rate: Parametr random_demand_rate przekazywaniy do funkcji generującej
                rozwiązanie początkowe.
:param max_iter_no_progress: Maksymalna ilość iteracji bez poprawy funkcji celu
:param max_iter: Łączna maksymalna ilość iteracji algorytmu
:param replacement_rate: Procent populacji jaki jest zastępowany przez potomków
               w każdej iteracji algorytmu (liczba z zakresu 0-1).
:param mutation_proba: Prawdopodobieństwo wystąpienia mutacji u dziecka
              (liczba z zakresu 0-1).
:param verbose: wyświetlaj numer iteracji i dotychczas najlepszy wynik
:return: Znalezione rozwiązanie, koszt rozwiązania, ilość wykonanych iteracji
"""
solutions = self.create_initial_population(random_demand_rate=random_demand_rate)


# population to lista list, w której przechowujemy rozwiązania.
# Poszczególne elementy listy population to dwuelementowe
# listy o następującej postaci [rozwiązanie, funkcja celu dla rozwiązania]
population = [[sol[0], self.calculate_objective_fun(sol[0])] for sol in solutions]
# sortowanie populacji po wartości funkcji celu
population = sorted(population, key=lambda x: x[1])


# licznik iteracji bez poprawy funkcji celu
iter_with_no_progress = 0
# licznik wszystkich iteracji
iterations = 0
# wartość funkcji celu dla najleoszego rozwiązania
best_cost = -np.inf


# lista best_results przechowuje najlepsze wyniki w każdej iteracji
best_results = []


while iter_with_no_progress <= max_iter_no_progress and iterations <= max_iter:
    # Kryterium stopu algorytmu jest osiągnięcie maksymalnej liczby iteracji bez poprawy
    # lub osiągnięcie maksymalnej iteracji w ogóle.
    iterations += 1


    # licznik dzieci utworzonych w danej iteracji
    children_count = 0
    # lista przechowująca nowe rozwiązania (dzieci)
    children = []
    # aktualny procent populacji, która zostanie
    # zastąpiona przez nowych członków
    replaced = 0


    while replaced < replacement_rate:
        # nowych potomków tworzymy tak długo dopóki procent
        # populacji jaki zostanie zastąpiony przez potomków
        # jest mniejszy niż replacement_rate


        # w każdej iteracji tworzę 2 nowych potomków
        # więc aktualizuję children_count i replaced
        children_count += 2
        replaced = children_count/len(population)


        parents = self.selection(population)
        parents = [parents[i][0] for i in range(len(parents))]


        child1 = self.crossover(parents[0], parents[1])
        child2 = self.crossover(parents[1], parents[0])


        if child1 is None or child2 is None:
            children_count -= 2
            replaced = children_count / len(population)
            continue


        # następnie losujemy liczbę z zakresu 0-1 i sprawdzamy
        # czy mamy dokonać mutacji jednego oraz drugiego dziecka.
        if random.uniform(0, 1) <= mutation_proba:
            try:
                child1 = self.draw_solution(child1, 1)
            except:
                pass
        if random.uniform(0, 1) <= mutation_proba:
            try:
                child2 = self.draw_solution(child2, 1)
            except:
```

```
            pass

        # dołączenie dzieci do listy children
        children.append(deepcopy(child1))
        children.append(deepcopy(child2))

    for i in range(len(children)):
        # podmienienie najsłabszych elementów z populacji przez
        # nowych potomków
        population[i] = [deepcopy(children[i]), self.calculate_objective_fun(children[i])]
    # ponowne sortowanie populacji zawierającej nowych członków
    population = sorted(population, key=lambda x: x[1])

    if population[-1][1] > best_cost:
        # jeśli funkcja celu najlepszego członka obecnej populacji jest
        # lepsza niż dotychczasowo najlepsza to podmień najlepszy koszt
        # oraz zresetuj licznik iteracji bez poprawy
        best_cost = population[-1][1]
        iter_with_no_progress = 0
    else:
        # w innym przypadku zwiększamy licznik iteracji bez poprawy
        iter_with_no_progress += 1

    if verbose:
        print(f"best profit: {population[-1][1]} | iteration number: {iterations}")
    if return_best_results:
        best_results.append(population[-1][1])

if return_best_results:
    return population[-1][0], population[-1][1], iterations, best_results
else:
    return population[-1][0], population[-1][1], iterations
```

3.3.2 Selekcja

Do wybrania rodziców w pierwszej kolejności losowani są dwaj różni kandydaci z całej populacji, następnie do listy rodziców trafia to rozwiązanie, dla którego wartość funkcji celu była większa, procedura kończy się, gdy w liście znajduje się dwójka rodziców.

3.3.2 Krzyżowanie

Aby skrzyżować dwa rozwiązania najpierw losujemy ze zbioru $\{3,4,5,\dots 28\}$ dzień a, następnie ze zbioru $\{a+1, a+2, \dots, 29\}$ dzień b. Dzielimy rozwiązania rodziców ($rodzic_1$ i $rodzic_2$) na 3 przedziały:

1. $rodzic_1$ dni 1 do $a-1$
2. $rodzic_2$ dni a do $b-1$
3. $rodzic_1$ dni b do 30

Następnie tworzymy potomka z tych 3 części rozwiązania. Po połączeniu pojawi się problem ze stanami magazynowymi więc dla każdego typu owoców, jeśli w dniu $a-1$ $rodzic_1$ ma więcej w magazynie niż $rodzic_2$ to potomek musi w dniu a sprzedać więcej, w przeciwnym wypadku mniej. Analogicznie trzeba postąpić dla drugiego łączenia. Algorytm zmniejszania lub zwiększania sprzedaży jest taki sam jak używany podczas generacji sąsiada.

W wyniku tego krzyżowania dostaniemy jednego potomka, można uzyskać 2, jeśli wywołamy tę funkcję 2 razy z zamienionymi kolejnością rodzicami.

3.3.2 Mutacja

Jako mutacji użyliśmy funkcji generującej sąsiada rozwiązania z algorytmu symulowanego wyżarzania.

4 Aplikacja

Do poprawnego działania programu konieczne jest zainstalowanie pakietów z pliku `requirements.txt`. Na komputerze musi być również zainstalowany język `Python` w wersji 3 i powyżej.

Aby móc korzystać z terminala do obsługi programu należy w terminalu przejść folderu zawierającego m.in. foldery `project_app` oraz `wyniki`.

Przykładowa komenda uruchamiająca algorytm symulowanego wyżarzania: `python -m project_app annealing --t_start 5000 --t_stop 800 --iter_in_temp 100 --epsilon 2 --iter_epsilon 10 --alpha 0.99 --initial_sol 11 -–verbose`

Znaczenie poszczególnych parametrów:

- `--t_start` temperatura początkowa
- `--t_stop` temperatura końcowa
- `--iter_in_temp` ilość iteracji wykonywanych dla jednej temperatury
- `--epsilon` minimalna wartość, o którą musi zmienić się funkcja celu przez okres iteracji określony *iter_epsilon*, aby algorytm nie zakończył działania
- `--iter_epsilon` ilość iteracji, po której algorytm zakończy działanie jeśli funkcja celu nie zmieniła się o więcej niż *epsilon*
- `--alpha` współczynnik o jaki zmniejszana jest temperatura
- `--initial_sol` numer rozwiązania początkowego
- `--verbose` parametr opcjonalny, jego użycie skutkuje wyświetlaniem aktualnej temperatury oraz najlepszego zysku w trakcie pracy algorytmu

Przykładowa komenda uruchamiająca algorytm genetyczny: `python -m project_app genetic --iter_no_progress 600 --max_iter 3000 --replacement_rate 0.6 --mutation_proba 0.7 --random_demand_rate --verbose`

Znaczenie poszczególnych parametrów:

- `--iter_no_progress` maksymalna liczba iteracji bez poprawy wyniku
- `--max_iter` maksymalna liczba iteracji
- `--replacement_rate` procent populacji zastępowany przez nowe pokolenie po każdej iteracji
- `--mutation_proba` prawdopodobieństwo wystąpienia mutacji
- `--random_demand_rate` parametr opcjonalny, jego użycie skutkuje ustawieniem tego parametru na wartość True w algorytmie tworzenia rozwiązania. Testy pokazały że algorytm działa lepiej bez używania tego parametru
- `--verbose` parametr opcjonalny, jego użycie skutkuje wyświetlaniem aktualnej iteracji oraz najlepszego zysku w trakcie pracy algorytmu

Ustawienie parametrów sadu oraz owoców
Parametry sadu oraz różne rodzaje owoców, koszty magazynowania i koszty pracowników można ustawić bezpośrednio w pliku programu pod ścieżką `project_app/app_settings.py`.

Wyniki
Wyniki działania algorytmu są zapisywanie w folderze `wyniki` w pliku `algorytm_genetyczny.txt` lub `algorytm_wyrzarzania.txt`.

5 Eksperymenty

Wszystkie pliki używane do przeprowadzenia testów można znaleźć w folderze testy.

5.1 Testowanie losowej wartości parametru *demand_rate* w funkcji *generate_satisfy_demand* służącej do generowania rozwiązania początkowego

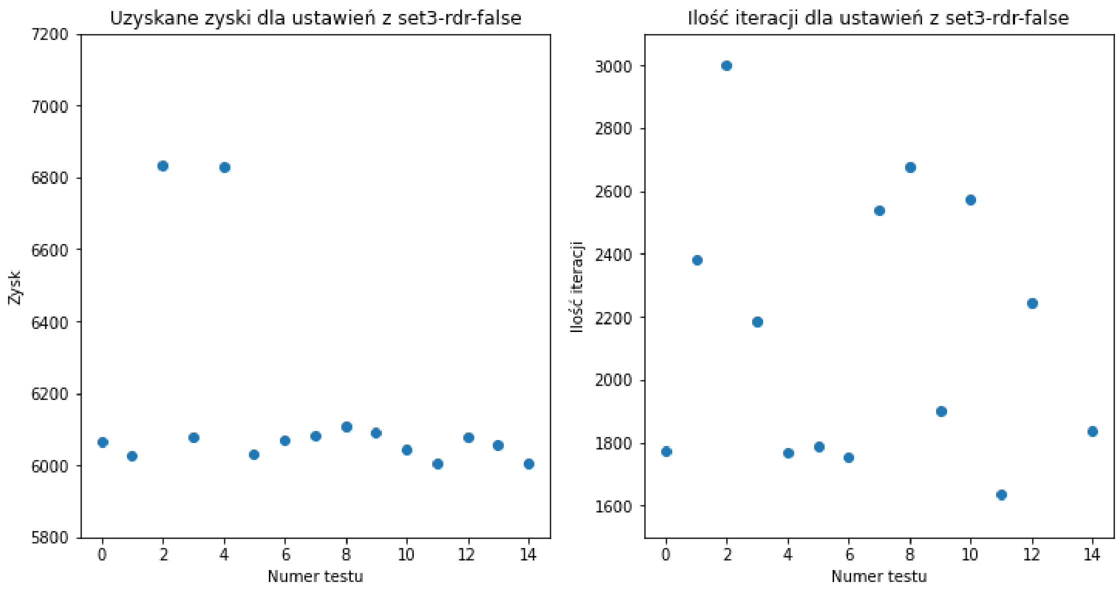
Wpływ działania losowego popytu podczas tworzenia początkowych rozwiązań został przetestowany dla algorytmu genetycznego. Za każdym razem ustawienia algorytmu były takie same, różnił się jedynie parametr *random_demad_rate*. W ramach testu dla każdego zestawu parametrów 15 razy uruchomiliśmy algorytm genetyczny i sprawdziliśmy jakie ostateczne wyniki udało się uzyskać.

```
algorithm_settings = {
    "set3-rdr-false":{
            "max_iter_no_progress": 400,
            "max_iter": 3000,
            "replacement_rate": 0.7,
            "mutation_proba": 0.4,
            "random_demand_rate": False,
            "verbose": False},

    "set3-rdr-true":{
            "max_iter_no_progress": 400,
            "max_iter": 3000,
            "replacement_rate": 0.7,
            "mutation_proba": 0.4,
            "random_demand_rate": True,
            "verbose": False}
}
```

Rys. 1. Dwa zestawy ustawień algorytmu podczas testu.

```
Wyniki dla zestawu ustawień z set3-rdr-false
Zysk: 6064.800000000002 | Iteracje: 1776
Zysk: 6028.300000000003 | Iteracje: 2380
Zysk: 6831.600000000002 | Iteracje: 3001
Zysk: 6077.000000000004 | Iteracje: 2188
Zysk: 6828.200000000002 | Iteracje: 1770
Zysk: 6030.700000000002 | Iteracje: 1790
Zysk: 6068.600000000002 | Iteracje: 1753
Zysk: 6081.200000000002 | Iteracje: 2539
Zysk: 6110.200000000004 | Iteracje: 2675
Zysk: 6092.9000000000015 | Iteracje: 1900
Zysk: 6043.000000000002 | Iteracje: 2573
Zysk: 6003.100000000002 | Iteracje: 1638
Zysk: 6079.700000000002 | Iteracje: 2247
Zysk: 6057.200000000001 | Iteracje: 1297
Zysk: 6005.500000000002 | Iteracje: 1838
Średni zysk: 6160.133333333335
Mediana zysku: 6068.600000000002
```

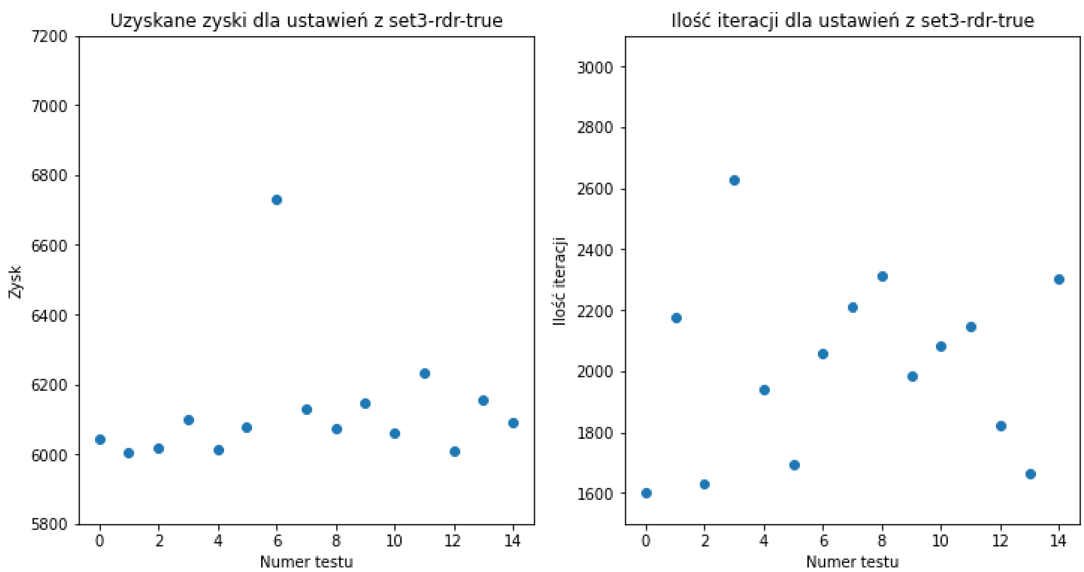
Rys. 2. Wyniki testów dla *random_demad_rate* ustawionego na False.



Rys. 3. Wizualizacja wyników z rys. 2.

```
Wyniki dla zestawu ustawień z set3-rdr-true
Zysk: 6044.9000000000015 | Iteracje: 1601
Zysk: 6003.800000000001 | Iteracje: 2174
Zysk: 6016.700000000002 | Iteracje: 1631
Zysk: 6097.700000000001 | Iteracje: 2627
Zysk: 6014.9000000000015 | Iteracje: 1939
Zysk: 6077.600000000002 | Iteracje: 1695
Zysk: 6729.299999999999 | Iteracje: 2059
Zysk: 6129.700000000001 | Iteracje: 2208
Zysk: 6072.300000000002 | Iteracje: 2315
Zysk: 6144.800000000001 | Iteracje: 1986
Zysk: 6062.000000000002 | Iteracje: 2082
Zysk: 6232.500000000001 | Iteracje: 2148
Zysk: 6009.4000000000015 | Iteracje: 1822
Zysk: 6155.0 | Iteracje: 1668
Zysk: 6089.800000000002 | Iteracje: 2303
Średni zysk: 6125.360000000001
Mediana zysku: 6077.600000000002
```

Rys. 4. Wyniki testów dla *random_demad_rate* ustawionego na True.



Rys. 5. Wizualizacja wyników z rys.4

Jak można zauważyć wyniki działania algorytmu genetycznego są podobne niezależnie od parametru *random_demad_rate*. Jednak w przypadku późniejszych testów wybraliśmy *random_demad_rate=False* ponieważ dla tych ustawień udało się dwukrotnie uzyskać wysokie wyniki odstające od reszty podczas gdy dla *random_demad_rate=True* taka sytuacja miała miejsce tylko raz a uzyskany wynik i tak był mniejszy niż te dla parametru równego False.

5.2 Testowanie algorytmu genetycznego

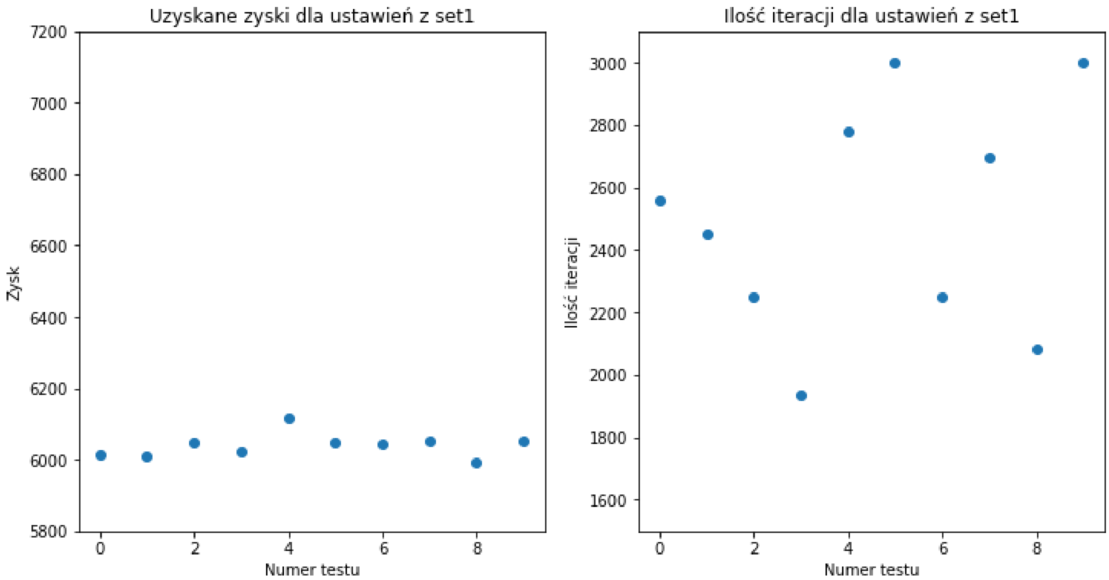
Testy algorytmu genetycznego polegały na sprawdzeniu 4 zestawów parametrów. Dla każdego zestawu algorytm został uruchomiony 10 razy.

```
algorithm_settings = {
    "set1":{"max_iter_no_progress": 400,
           "max_iter": 3000,
           "replacement_rate": 0.4,
           "mutation_proba": 0.4,
           "random_demand_rate": False,
           "verbose": False},

    "set2":{"max_iter_no_progress": 400,
           "max_iter": 3000,
           "replacement_rate": 0.4,
           "mutation_proba": 0.7,
           "random_demand_rate": False,
           "verbose": False},

    "set3":{"max_iter_no_progress": 400,
           "max_iter": 3000,
           "replacement_rate": 0.7,
           "mutation_proba": 0.4,
           "random_demand_rate": False,
           "verbose": False},

    "set4":{"max_iter_no_progress": 600,
           "max_iter": 3000,
           "replacement_rate": 0.7,
           "mutation_proba": 0.7,
           "random_demand_rate": False,
           "verbose": False},
}
```

Rys. 6. Zestawy ustawień algorytmu podczas testu.

```
Wyniki dla zestawu ustawień z set1
Zysk: 6015.200000000002 | Iteracje: 2557
Zysk: 6011.000000000002 | Iteracje: 2450
Zysk: 6046.400000000002 | Iteracje: 2250
Zysk: 6024.200000000002 | Iteracje: 1937
Zysk: 6117.100000000002 | Iteracje: 2779
Zysk: 6048.000000000002 | Iteracje: 3001
Zysk: 6045.000000000004 | Iteracje: 2252
Zysk: 6050.500000000002 | Iteracje: 2694
Zysk: 5993.600000000002 | Iteracje: 2083
Zysk: 6051.4000000000015 | Iteracje: 3001
Średni zysk: 6040.240000000003
Mediana zysku: 6045.700000000003
```

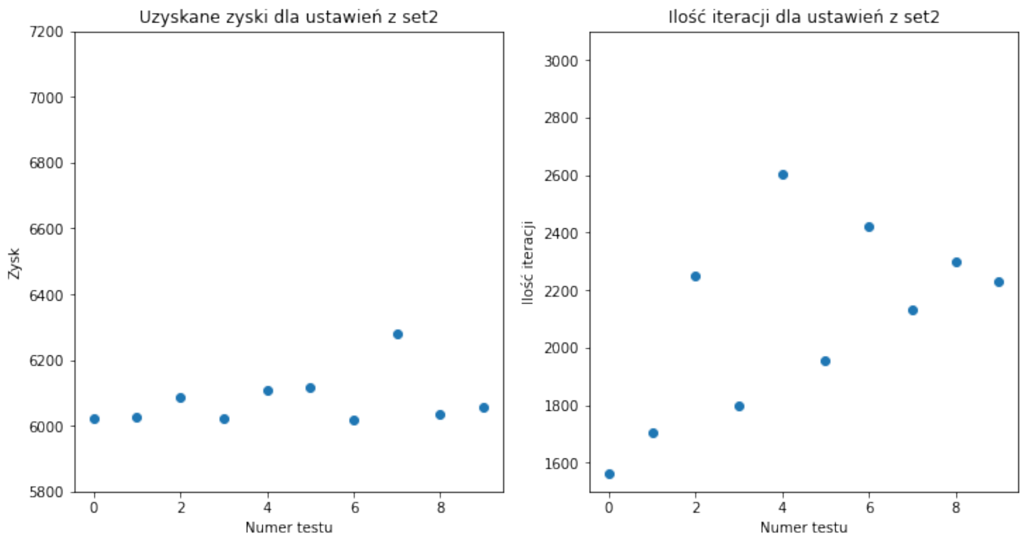
Rys. 7. Wyniki działania algorytmu.



Rys. 8. Wizualizacja wyników z rys. 7.

```
Wyniki dla zestawu ustawień z set2
Zysk: 6024.000000000002 | Iteracje: 1563
Zysk: 6027.000000000002 | Iteracje: 1703
Zysk: 6085.100000000003 | Iteracje: 2251
Zysk: 6020.500000000002 | Iteracje: 1796
Zysk: 6106.400000000003 | Iteracje: 2603
Zysk: 6117.800000000003 | Iteracje: 1955
Zysk: 6017.100000000002 | Iteracje: 2422
Zysk: 6279.499999999998 | Iteracje: 2130
Zysk: 6033.200000000003 | Iteracje: 2299
Zysk: 6056.000000000003 | Iteracje: 2229
Średni zysk: 6076.660000000002
Mediana zysku: 6044.600000000002
```

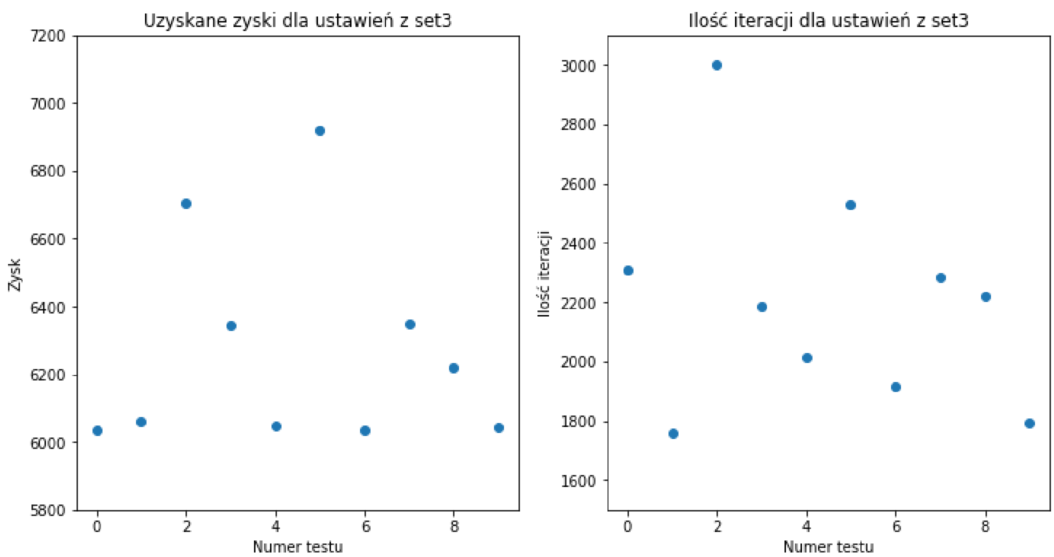
Rys. 9. Wyniki działania algorytmu.



Rys. 10. Wizualizacja wyników z rys. 9.

```
Wyniki dla zestawu ustawień z set3
Zysk: 6036.000000000002 | Iteracje: 2307
Zysk: 6062.9000000000015 | Iteracje: 1761
Zysk: 6706.900000000002 | Iteracje: 3001
Zysk: 6343.4000000000015 | Iteracje: 2185
Zysk: 6048.100000000002 | Iteracje: 2012
Zysk: 6918.6 | Iteracje: 2528
Zysk: 6036.500000000002 | Iteracje: 1915
Zysk: 6346.699999999999 | Iteracje: 2284
Zysk: 6218.200000000002 | Iteracje: 2218
Zysk: 6043.600000000002 | Iteracje: 1793
Średni zysk: 6276.090000000001
Mediana zysku: 6140.550000000001
```

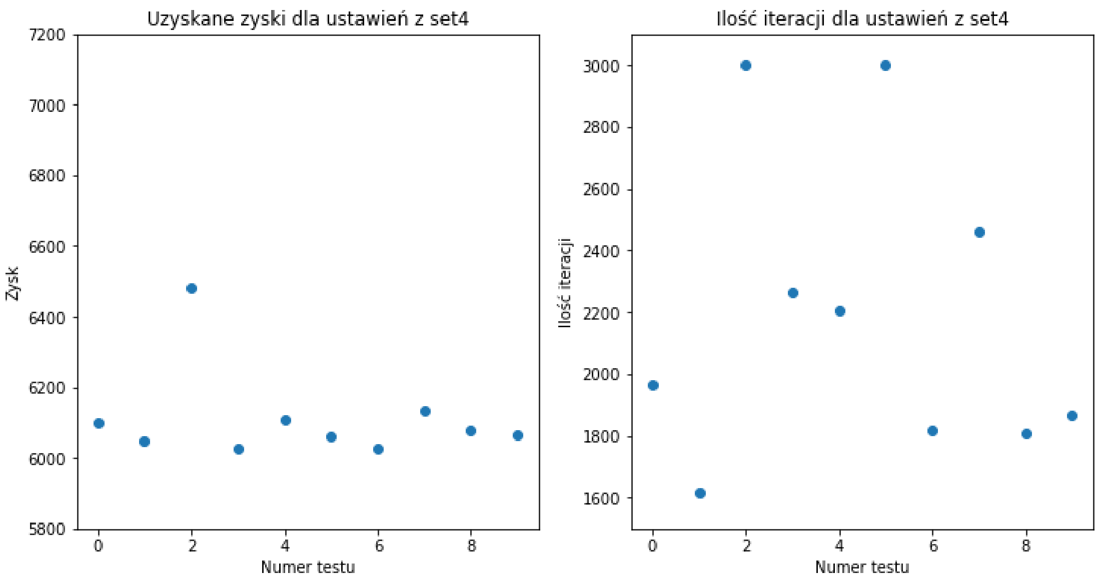
Rys. 11. Wyniki działania algorytmu.



Rys. 12. Wizualizacja wyników z rys. 11.

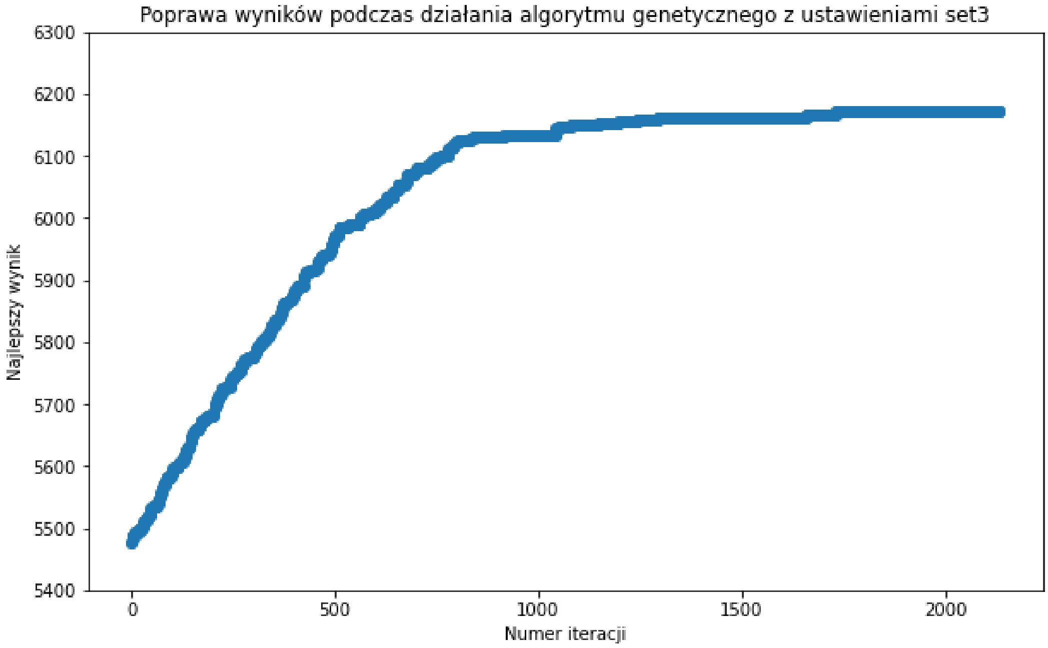
```
Wyniki dla zestawu ustawień z set4
Zysk: 6098.200000000003 | Iteracje: 1965
Zysk: 6049.200000000003 | Iteracje: 1617
Zysk: 6483.100000000003 | Iteracje: 3001
Zysk: 6026.600000000002 | Iteracje: 2262
Zysk: 6106.300000000003 | Iteracje: 2206
Zysk: 6060.700000000003 | Iteracje: 3001
Zysk: 6024.700000000003 | Iteracje: 1818
Zysk: 6133.100000000001 | Iteracje: 2460
Zysk: 6077.200000000003 | Iteracje: 1808
Zysk: 6066.500000000002 | Iteracje: 1866
Średni zysk: 6112.560000000003
Mediana zysku: 6071.850000000002
```

Rys. 13. Wyniki działania algorytmu.



Rys. 14. Wizualizacja wyników z rys. 13.

Jak można zauważyć, najlepszą średnią, medianę oraz najlepszy pojedynczy wynik uzyskaliśmy dla ustawień z *set3*. Kolejnym eksperymentem było zwizualizowanie poprawy rozwiązania wraz z kolejnymi iteracjami algorytmu. Wizualizację przeprowadziliśmy dla ustawień algorytmu z *set3*.



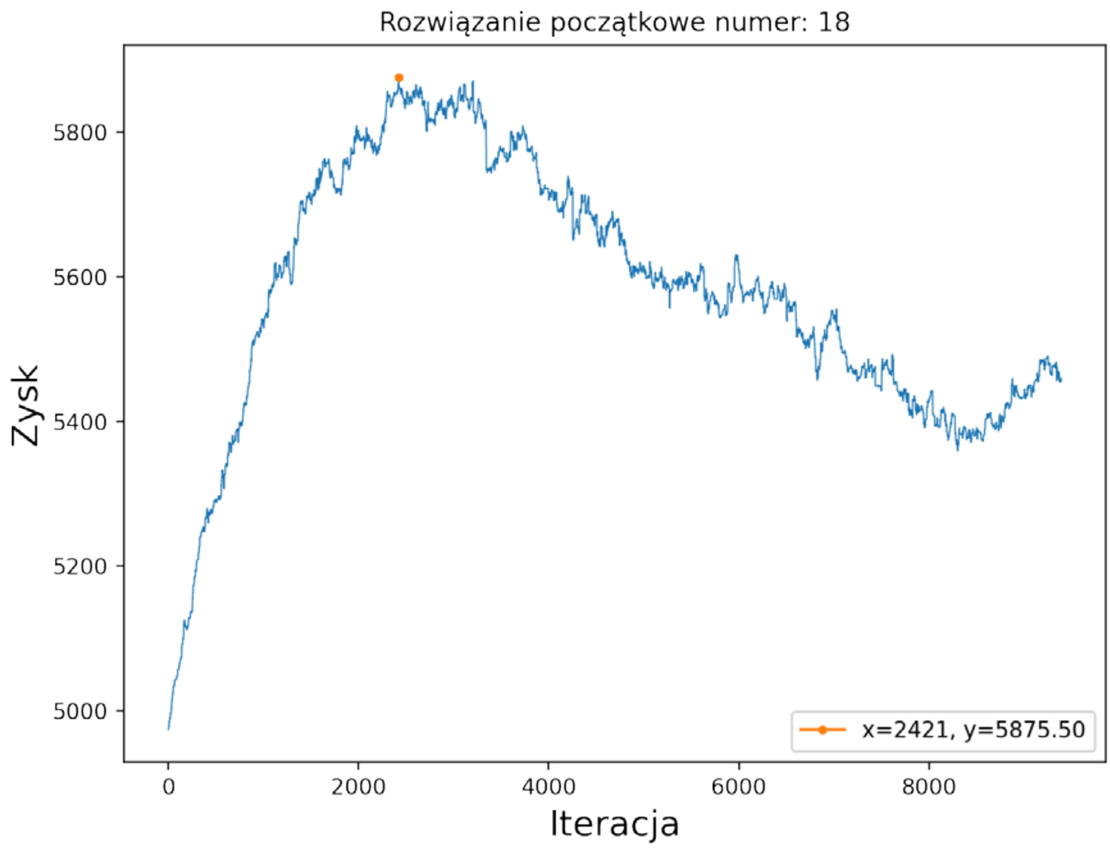
Rys. 15. Poprawa rozwiązania podczas działania algorytmu genetycznego.

5.3 Testowanie algorytmu Symulowanego Wyżarzania

Pierwsza faza testów algorytmu symulowanego wyżarzania polegała na sprawdzeniu zachowania się algorytmu w zależności od wyboru populacji początkowej przy następującym zestawie wartości parametrów:

- `T_start = 3000`
- `t_stop = 1000`
- `iter_in_temp = 100`
- `epsilon = 4`
- `iterations_epsilon = 10`
- `alpha = 0.99`
- `initial_sol - testowane było każde rozwiązanie początkowe`
- `verbose = False`

Przy zadanych wartościach argumentów najlepsze rozwiązanie zostało uzyskane dla populacji początkowej numer 18. Wykresy z wynikami dla pozostałych rozwiązań początkowych znajdują się w folderze testy/visuals/SA.



Rys. 16. Najlepsze rozwiązanie przy jednorazowym wywołaniu dla zadanych wartości parametrów

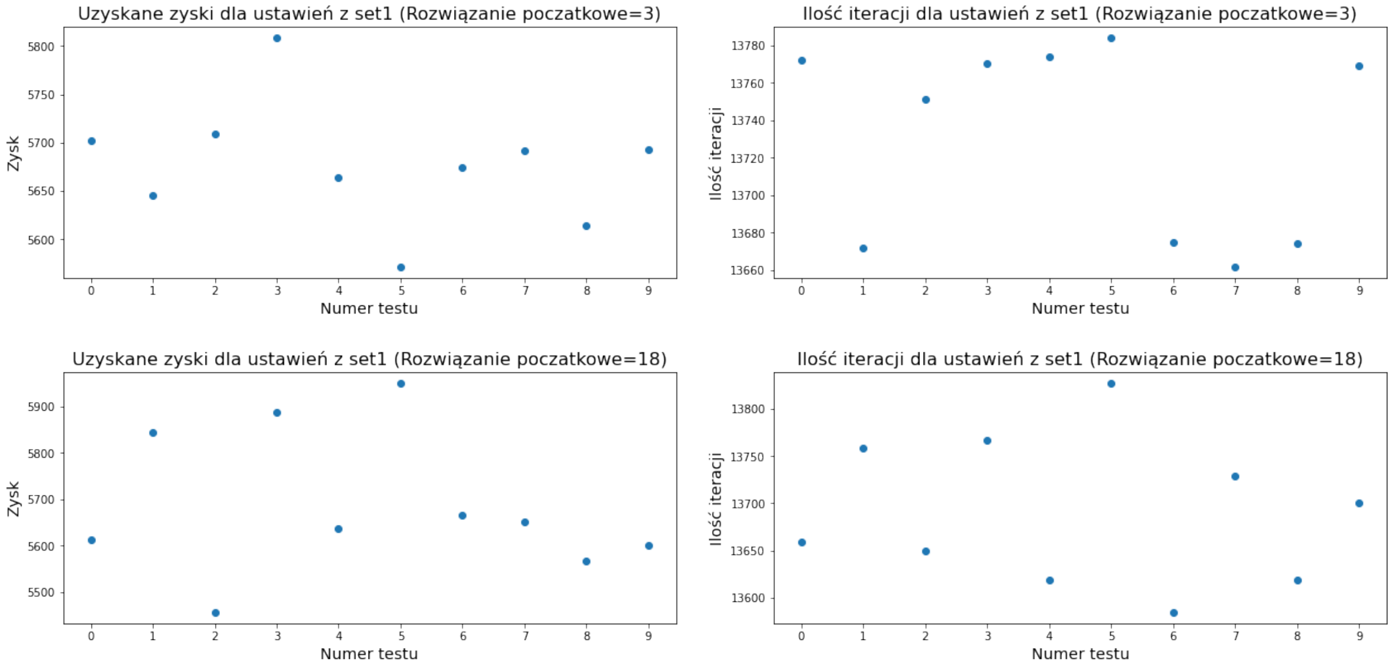
Następnie algorytm symulowanego wyżarzania był testowany za pomocą 5 zestawów parametrów dla dwóch najbardziej obiecujących rozwiązań początkowych tj. 3 i 18. Dla każdego zestawu algorytm został uruchomiony 10 razy.

```python
# Pięć zestawów parametrów
algorithm_settings = {
    "set1":{"T_start": 5000,
            "T_stop": 1000,
            "iterations_in_temp": 100,
            "epsilon": 5,
            "iterations_epsilon": 100,
            "alpha": 0.99,
            "initial_sol": [3, 18],
            "verbose": False},

    "set2":{"T_start": 5000,
            "T_stop": 1000,
            "iterations_in_temp": 500,
            "epsilon": 5,
            "iterations_epsilon": 500,
            "alpha": 0.99,
            "initial_sol": [3, 18],
            "verbose": False},

    "set3":{"T_start": 5000,
            "T_stop": 1000,
            "iterations_in_temp": 100,
            "epsilon": 5,
            "iterations_epsilon": 100,
            "alpha": 0.95,
            "initial_sol": [3, 18],
            "verbose": False},

    "set4":{"T_start": 5000,
            "T_stop": 1000,
            "iterations_in_temp": 100,
            "epsilon": 5,
            "iterations_epsilon": 100,
            "alpha": 0.999,
            "initial_sol": [3, 18],
            "verbose": False},

    "set5":{"T_start": 1000,
            "T_stop": 10,
            "iterations_in_temp": 100,
            "epsilon": 5,
            "iterations_epsilon": 100,
            "alpha": 0.99,
            "initial_sol": [3, 18],
            "verbose": False},
}
```

Rys. 17. Zestawy ustawień algorytmu podczas testu.

```
Wyniki dla zestawu ustawień z set1 (Rozwiązanie początkowe=3)
Zysk: 5702.40 | Iteracje: 13772
Zysk: 5645.60 | Iteracje: 13672
Zysk: 5709.60 | Iteracje: 13751
Zysk: 5808.40 | Iteracje: 13770
Zysk: 5663.60 | Iteracje: 13774
Zysk: 5572.00 | Iteracje: 13784
Zysk: 5674.80 | Iteracje: 13675
Zysk: 5691.50 | Iteracje: 13662
Zysk: 5614.00 | Iteracje: 13674
Zysk: 5693.20 | Iteracje: 13769
Średni zysk: 5677.51
Mediana zysku: 5683.15

Wyniki dla zestawu ustawień z set1 (Rozwiązanie początkowe=18)
Zysk: 5612.00 | Iteracje: 13659
Zysk: 5844.40 | Iteracje: 13759
Zysk: 5456.40 | Iteracje: 13649
Zysk: 5887.60 | Iteracje: 13767
Zysk: 5637.30 | Iteracje: 13619
Zysk: 5950.40 | Iteracje: 13827
Zysk: 5664.30 | Iteracje: 13585
Zysk: 5651.00 | Iteracje: 13729
Zysk: 5565.50 | Iteracje: 13619
Zysk: 5601.10 | Iteracje: 13700
Średni zysk: 5687.00
Mediana zysku: 5644.15
```

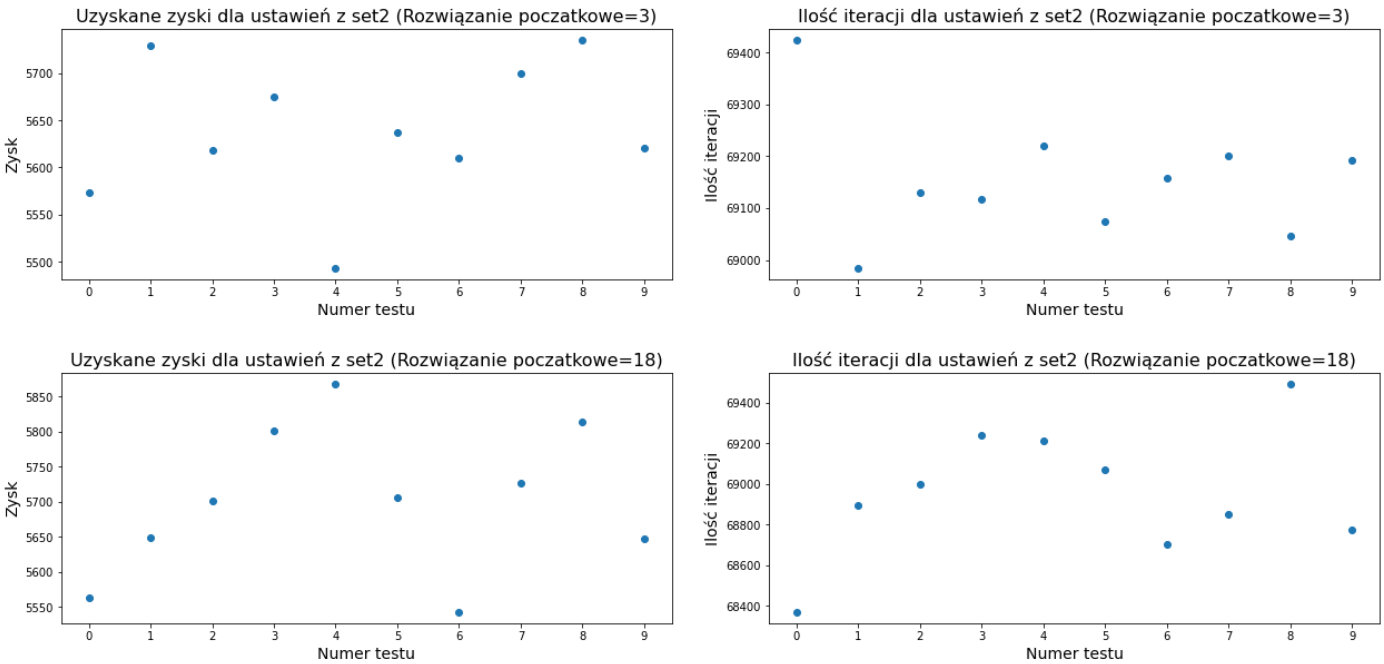
Rys. 18. Wyniki działania algorytmu.



Rys. 19. Wizualizacja wyników z rys. 18.

```
Wyniki dla zestawu ustawień z set2 (Rozwiązanie początkowe=3)
Zysk: 5573.80 | Iteracje: 69424
Zysk: 5728.90 | Iteracje: 68985
Zysk: 5618.40 | Iteracje: 69130
Zysk: 5675.40 | Iteracje: 69118
Zysk: 5493.60 | Iteracje: 69221
Zysk: 5636.80 | Iteracje: 69075
Zysk: 5609.50 | Iteracje: 69157
Zysk: 5700.10 | Iteracje: 69201
Zysk: 5735.20 | Iteracje: 69046
Zysk: 5620.10 | Iteracje: 69193
Średni zysk: 5639.18
Mediana zysku: 5628.45

Wyniki dla zestawu ustawień z set2 (Rozwiązanie początkowe=18)
Zysk: 5563.20 | Iteracje: 68372
Zysk: 5648.60 | Iteracje: 68896
Zysk: 5701.80 | Iteracje: 68998
Zysk: 5801.80 | Iteracje: 69242
Zysk: 5867.90 | Iteracje: 69212
Zysk: 5706.00 | Iteracje: 69071
Zysk: 5543.10 | Iteracje: 68701
Zysk: 5726.30 | Iteracje: 68851
Zysk: 5813.30 | Iteracje: 69492
Zysk: 5646.80 | Iteracje: 68777
Średni zysk: 5701.88
Mediana zysku: 5703.90
```

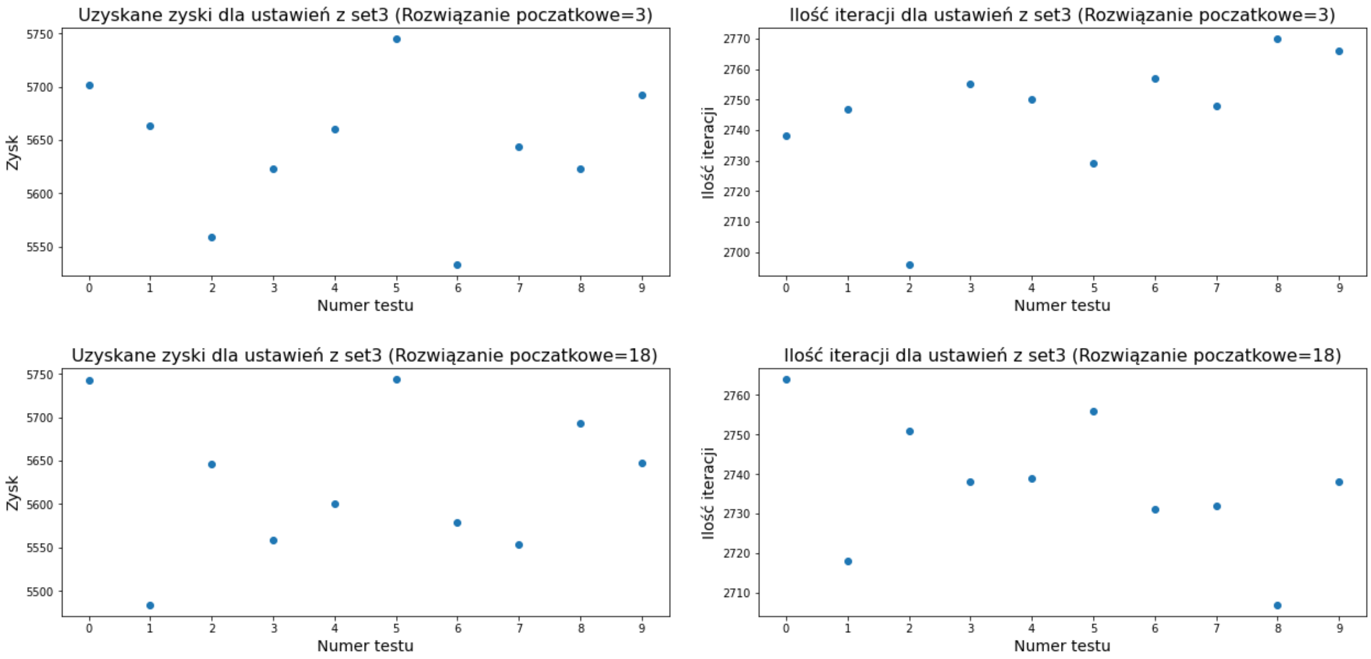
Rys. 20. Wyniki działania algorytmu.



Rys. 21. Wizualizacja wyników z rys. 20.

```
Wyniki dla zestawu ustawień z set3 (Rozwiązanie początkowe=3)
Zysk: 5701.80 | Iteracje: 2738
Zysk: 5663.40 | Iteracje: 2747
Zysk: 5558.60 | Iteracje: 2696
Zysk: 5622.90 | Iteracje: 2755
Zysk: 5660.20 | Iteracje: 2750
Zysk: 5745.10 | Iteracje: 2729
Zysk: 5533.50 | Iteracje: 2757
Zysk: 5644.00 | Iteracje: 2748
Zysk: 5623.50 | Iteracje: 2770
Zysk: 5692.30 | Iteracje: 2766
Średni zysk: 5644.53
Mediana zysku: 5652.10

Wyniki dla zestawu ustawień z set3 (Rozwiązanie początkowe=18)
Zysk: 5742.90 | Iteracje: 2764
Zysk: 5484.20 | Iteracje: 2718
Zysk: 5645.70 | Iteracje: 2751
Zysk: 5558.90 | Iteracje: 2738
Zysk: 5599.90 | Iteracje: 2739
Zysk: 5743.90 | Iteracje: 2756
Zysk: 5579.00 | Iteracje: 2731
Zysk: 5553.20 | Iteracje: 2732
Zysk: 5693.20 | Iteracje: 2707
Zysk: 5647.00 | Iteracje: 2738
Średni zysk: 5624.79
Mediana zysku: 5622.80
```

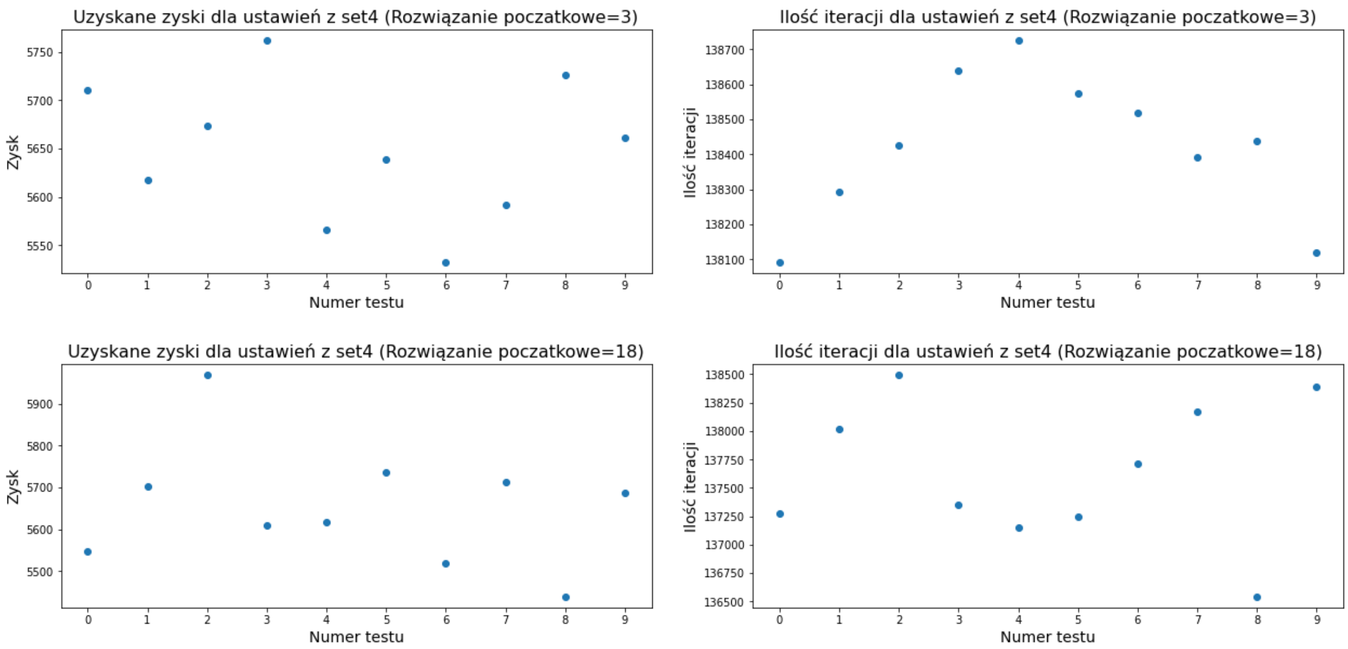
Rys. 22. Wyniki działania algorytmu.



Rys. 23. Wizualizacja wyników z rys. 22.

```
Wyniki dla zestawu ustawień z set4 (Rozwiązanie początkowe=3)
Zysk: 5710.30 | Iteracje: 138093
Zysk: 5617.20 | Iteracje: 138293
Zysk: 5673.20 | Iteracje: 138425
Zysk: 5761.90 | Iteracje: 138639
Zysk: 5566.10 | Iteracje: 138726
Zysk: 5638.20 | Iteracje: 138574
Zysk: 5532.70 | Iteracje: 138519
Zysk: 5592.00 | Iteracje: 138393
Zysk: 5725.80 | Iteracje: 138438
Zysk: 5660.60 | Iteracje: 138120
Średni zysk: 5647.80
Mediana zysku: 5649.40

Wyniki dla zestawu ustawień z set4 (Rozwiązanie początkowe=18)
Zysk: 5547.90 | Iteracje: 137278
Zysk: 5703.60 | Iteracje: 138022
Zysk: 5969.00 | Iteracje: 138494
Zysk: 5610.30 | Iteracje: 137347
Zysk: 5615.80 | Iteracje: 137147
Zysk: 5736.00 | Iteracje: 137245
Zysk: 5518.90 | Iteracje: 137709
Zysk: 5713.20 | Iteracje: 138172
Zysk: 5439.50 | Iteracje: 136545
Zysk: 5686.40 | Iteracje: 138394
Średni zysk: 5654.06
Mediana zysku: 5651.10
```

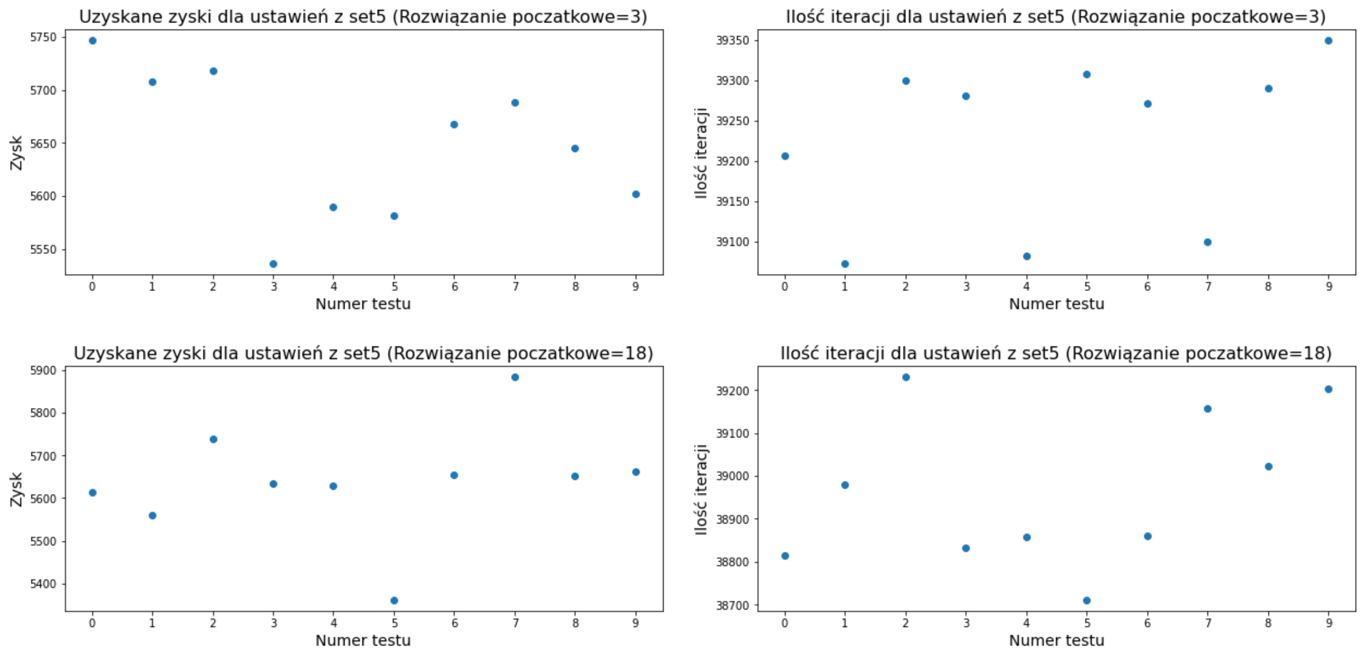
Rys. 24. Wyniki działania algorytmu.



Rys. 25. Wizualizacja wyników z rys. 24.

```
Wyniki dla zestawu ustawień z set5 (Rozwiązanie początkowe=3)
Zysk: 5746.80 | Iteracje: 39207
Zysk: 5707.40 | Iteracje: 39073
Zysk: 5718.20 | Iteracje: 39300
Zysk: 5536.80 | Iteracje: 39281
Zysk: 5589.40 | Iteracje: 39082
Zysk: 5581.90 | Iteracje: 39308
Zysk: 5668.00 | Iteracje: 39272
Zysk: 5688.10 | Iteracje: 39099
Zysk: 5645.30 | Iteracje: 39290
Zysk: 5602.30 | Iteracje: 39350
Średni zysk: 5648.42
Mediana zysku: 5656.65

Wyniki dla zestawu ustawień z set5 (Rozwiązanie początkowe=18)
Zysk: 5613.70 | Iteracje: 38815
Zysk: 5560.80 | Iteracje: 38980
Zysk: 5739.50 | Iteracje: 39231
Zysk: 5635.30 | Iteracje: 38832
Zysk: 5629.30 | Iteracje: 38857
Zysk: 5362.90 | Iteracje: 38712
Zysk: 5653.80 | Iteracje: 38861
Zysk: 5883.80 | Iteracje: 39157
Zysk: 5651.40 | Iteracje: 39022
Zysk: 5662.70 | Iteracje: 39202
Średni zysk: 5639.32
Mediana zysku: 5643.35
```

Rys. 26. Wyniki działania algorytmu.



Rys. 27. Wizualizacja wyników z rys. 26.

W przypadku rozwiązania początkowego numer 3 najlepszą średnią, medianę i pojedynczy wynik (5808,40) uzyskano wykorzystując pierwszy zestaw.

Natomiast dla rozwiązania początkowego numer 18 najlepszą średnią i medianę uzyskano dla drugiego zestawu, natomiast najlepszy pojedynczy wynik (5969) uzyskany został w *set4*. Pomimo obiecujących wyników przy pojedynczych wywołaniach funkcji okazało się, że rozwiązanie numer 18 daje o wiele lepsze rezultaty od rozwiązania 3.

5.4 Sprawdzenie algorytmów pod kątem prawidłowej implementacji

Sprawdzenie przykładowego wyniku (algorytm genetyczny):

Dzień pierwszy:

	Wiśnie	Jabłka	Gruszki	Śliwki
Ilość	38	33	11	6
Sprzedano na rynku	0	0	0	0
Sprzedano w skupie	38	33	11	6
Magazynowane	0	0	0	0
Popyt	10	5	2	10
Cena bazowa na targu	7	3	3	5
Cena w skupie	5	3	3	4
Koszt zasadzenia	110	62	85	91

Zysk ze sprzedaży w skupie:

$$38 \times 5 + 33 \times 3 + 11 \times 3 + 6 \times 4 = 190 + 99 + 33 + 24 = 346\ zł$$

Koszt zatrudnienia pracowników:

Łącznie zebrano:

$$38 + 33 + 11 + 6 = 88\ kg\ owoców$$

co daje łączny koszt zatrudnienia pracowników równy 100 złotych. Dodając do tego koszt wynajmu magazynu - 15 złotych otrzymujemy zysk po pierwszym dniu równy:

$$346 - (100 + 15) = 231\ złotych$$

Dzień drugi:

	Wiśnie	Jabłka	Gruszki	Śliwki
Ilość	38	33	11	9
Sprzedano na rynku	7	2	1	7
Sprzedano w skupie	31	31	8	2
Magazynowane	0	0	2	0
Popyt	10	8	4	10
Cena bazowa na targu	7	3	3	5
Cena w skupie	5	3	3	4

Zysk ze sprzedaży w skupie:

$$31 \times 5 + 31 \ \times 3 + 8 \times 3 + 2 \times 4 = 155 + 93 + 24 + 8 = 280 \ zł$$

Zysk ze sprzedaży na rynku:

$$7 \times 7 \times 1{,}1 + 2 \ \times 3 \times 1{,}5 + 1 \times 3 \times 1{,}5 + 7 \times 5 \times 1{,}1 = 53{,}9 + 9 + 4{,}5 + 38{,}5 = 105{,}9 \ zł$$

Koszt zatrudnienia pracowników:

Łącznie zebrano:

$$38 + 33 + 11 + 9 = 91 \ kg \ owoców$$

co daje łączny koszt zatrudnienia pracowników równy 100 złotych. Dodając do tego koszt wynajmu magazynu - 15 złotych otrzymujemy zysk po drugim dniu równy:

$$280 + 105{,}9 - (100 + 15) = 270{,}9 \ złotych$$

Dzień trzeci:

	Wiśnie	Jabłka	Gruszki	Śliwki
Ilość	43	37	13	7
Sprzedano na rynku	7	2	3	7

Sprzedano w skupie	36	35	12	0
Magazynowane	0	0	0	0
Popyt	10	3	4	10
Cena bazowa na targu	7	3	3	5
Cena w skupie	5	3	3	4

Zysk ze sprzedaży w skupie:

$$36 \times 5 + 35 \ \times 3 + 12 \times 3 + 0 \times 4 = 180 + 105 + 36 = 321\ zł$$

Zysk ze sprzedaży na rynku:

$$7 \times 7 \times 1{,}1 + 2 \ \times 3 \times 1{,}1 + 3 \times 3 \times 1{,}1 + 7 \times 5 \times 1{,}1 = 53{,}9 + 6{,}6 + 9{,}9 + 38{,}5 = 108{,}9\ zł$$

Koszt zatrudnienia pracowników:

Łącznie zebrano:

$$43 + 37 + 13 + 7 = 100\ kg\ owoców$$

co daje łączny koszt zatrudnienia pracowników równy 100 złotych. Dodając do tego koszt wynajmu magazynu - 15 złotych otrzymujemy zysk po trzecim dniu równy:

$$321 + 108{,}9 - (100 + 15) = 314{,}9\ złotych$$

Dzień czwarty:

	Wiśnie	Jabłka	Gruszki	Śliwki
Ilość	38	33	11	9
Sprzedano na rynku	10	2	3	7
Sprzedano w skupie	16	20	3	0
Magazynowane	12	11	5	2
Popyt	10	3	4	10
Cena bazowa na targu	7	3	3	5
Cena w skupie	5	3	3	4

Zysk ze sprzedaży w skupie:

$$16 \times 5 + 20\ \times 3 + 3 \times 3 = 80 + 60 + 9 = 149\ zł$$

Zysk ze sprzedaży na rynku:

$$10 \times 7 \times 1 + 2\ \times 3 \times 1{,}1 + 3 \times 3 \times 1{,}1 + 7 \times 5 \times 1{,}1 = 70 + 6{,}6 + 9{,}9 + 38{,}5 = 125\ zł$$

Koszt zatrudnienia pracowników:

Łącznie zebrano:

$$38 + 33 + 11 + 9 = 91\ kg\ owoców$$

co daje łączny koszt zatrudnienia pracowników równy 100 złotych. Dodając do tego koszt wynajmu magazynu - 25 złotych (koszt magazynowania 30 kilogramów owoców) otrzymujemy zysk po czwartym dniu równy:

$$149 + 125 - (100 + 25) = 149\ złotych$$

Dzień piąty:

	Wiśnie	Jabłka	Gruszki	Śliwki
Ilość	29	25	9	5
Sprzedano na rynku	10	3	4	3
Sprzedano w skupie	31	33	10	4
Magazynowane	0	0	0	0
Popyt	10	3	4	10
Cena bazowa na targu	7	3	3	5
Cena w skupie	5	3	3	4

Zysk ze sprzedaży w skupie:

$$31 \times 5 + 33\ \times 3 + 10 \times 3 + 4 \times 4 = 155 + 99 + 30 + 16 = 300\ zł$$

Zysk ze sprzedaży na rynku:

$$10 \times 7 \times 1 + 3\ \times 3 \times 1 + 4 \times 3 \times 1 + 3 \times 5 \times 1{,}1 = 70 + 9 + 12 + 16{,}5 = 107{,}5\ zł$$

Koszt zatrudnienia pracowników:

Łącznie zebrano:

$$29 + 25 + 9 + 5 = 68\ kg\ owoców$$

co daje łączny koszt zatrudnienia pracowników równy 100 złotych. Dodając do tego koszt wynajmu magazynu - 15 złotych (koszt magazynowania 30 kilogramów owoców) otrzymujemy zysk po piątym dniu równy:

$$300 + 107{,}5 - (100 + 15) = 292{,}5\ złotych$$

Zysk po pięciu dniach:

$$292{,}5 + 149 + 314{,}9 + 270{,}9 + 231 = 1258{,}3\ zł$$

Zysk po odjęciu kosztów zasadzenia:

$$1258{,}3 - 85 - 62 - 91 - 110 = 910{,}3\ zł$$

Otrzymany wynik pokrywa się z wynikiem otrzymanym przez algorytm genetyczny:

```
Day 1
harvested: [38, 33, 11, 6]
sold_market: [0, 0, 0, 0]
sold_wholesale: [38, 33, 11, 6]
warehouse: [0, 0, 0, 0]

Day 2
harvested: [38, 33, 11, 9]
sold_market: [7, 2, 1, 7]
sold_wholesale: [31, 31, 8, 2]
warehouse: [0, 0, 2, 0]

Day 3
harvested: [43, 37, 13, 7]
sold_market: [7, 2, 3, 7]
sold_wholesale: [36, 35, 12, 0]
warehouse: [0, 0, 0, 0]

Day 4
harvested: [38, 33, 11, 9]
sold_market: [10, 2, 3, 7]
sold_wholesale: [16, 20, 3, 0]
warehouse: [12, 11, 5, 2]

Day 5
harvested: [29, 25, 9, 5]
sold_market: [10, 3, 4, 3]
sold_wholesale: [31, 33, 10, 4]
warehouse: [0, 0, 0, 0]




 910.3
```

Sprawdzenie przykładowego wyniku (algorytm SA):

Dzień pierwszy:

	Wiśnie	Jabłka	Gruszki	Śliwki
Ilość	17	17	17	17
Sprzedano na rynku	0	0	0	0
Sprzedano w skupie	17	17	17	17
Magazynowane	0	0	0	0
Popyt	10	5	2	10
Cena bazowa na targu	7	3	3	5
Cena na skupie	5	3	3	4

Zysk ze sprzedaży w skupie:

$$17 \times 5 + 17 \times 3 + 17 \times 3 + 17 \times 4 = 85 + 51 + 51 + 68 = 255\ zł$$

Koszt zatrudnienia pracowników:

Łącznie zebrano:

$$17 + 17 + 17 + 17 = 68\ kg\ owoców$$

co daje łączny koszt zatrudnienia pracowników równy 100 złotych. Dodając do tego koszt wynajmu magazynu - 15 złotych otrzymujemy zysk po pierwszym dniu równy:

$$255 - (100 + 15) = 140\ złotych$$

Dzień drugi:

	Wiśnie	Jabłka	Gruszki	Śliwki
Ilość	20	20	20	20
Sprzedano na rynku	10	2	3	10
Sprzedano w skupie	8	17	14	9
Magazynowane	2	1	3	1
Popyt	10	8	4	10
Cena bazowa na targu	7	3	3	5
Cena na skupie	5	3	3	4

Zysk ze sprzedaży w skupie:

$$8 \times 5 + 17 \times 3 + 14 \times 3 + 9 \times 4 = 40 + 51 + 42 + 36 = 169\ zł$$

Zysk ze sprzedaży na rynku:

$$10 \times 7 \times 1 + 2 \times 3 \times 1{,}5 + 3 \times 3 \times 1{,}1 + 10 \times 5 \times 1 = 70 + 9 + 9{,}9 + 50 = 138{,}9\ zł$$

Koszt zatrudnienia pracowników:

Łącznie zebrano:

$$20 + 20 + 20 + 20 = 91\ kg\ owoców$$

co daje łączny koszt zatrudnienia pracowników równy 100 złotych. Dodając do tego koszt wynajmu magazynu - 15 złotych otrzymujemy zysk po drugim dniu równy:

$$169 + 138{,}9 - (100 + 15) = 192{,}9\ złotych$$

Dzień trzeci:

	Wiśnie	Jabłka	Gruszki	Śliwki
Ilość	25	25	25	25
Sprzedano na rynku	10	3	4	10
Sprzedano w skupie	16	11	23	12
Magazynowane	1	12	1	4
Popyt	10	3	4	10
Cena bazowa na targu	7	3	3	5
Cena w skupie	5	3	3	4

Zysk ze sprzedaży w skupie:

$$16 \times 5 + 11 \ \times 3 + 23 \times 3 + 12 \times 4 = 80 + 33 + 69 + 48 = 230\ zł$$

Zysk ze sprzedaży na rynku:

$$10 \times 7 \times 1 + 3 \ \times 3 \times 1 + 4 \times 3 \times 1 + 10 \times 5 \times 1 = 70 + 9 + 12 + 50 = 141\ zł$$

Koszt zatrudnienia pracowników:

Łącznie zebrano:

$$25 + 25 + 25 + 25 = 100\ kg\ owoców$$

co daje łączny koszt zatrudnienia pracowników równy 100 złotych. Dodając do tego koszt wynajmu magazynu - 20 złotych otrzymujemy zysk po trzecim dniu równy:

$$230 + 141 - (100 + 20) = 251\ złotych$$

Dzień czwarty:

	Wiśnie	Jabłka	Gruszki	Śliwki
Ilość	22	22	22	22
Sprzedano na rynku	10	2	1	10
Sprzedano w skupie	12	31	2	8
Magazynowane	1	1	20	8

Popyt	10	3	4	10
Cena bazowa na targu	7	3	3	5
Cena w skupie	5	3	3	4

Zysk ze sprzedaży w skupie:

$$12 \times 5 + 31 \ \times 3 + 2 \times 3 + 8 \times 4 = 60 + 93 + 6 + 32 = 191 \text{ zł}$$

Zysk ze sprzedaży na rynku:

$$10 \times 7 \times 1 + 2 \ \times 3 \times 1{,}1 + 1 \times 3 \times 1{,}5 + 10 \times 5 \times 1 = 70 + 6{,}6 + 4{,}5 + 50 = 131{,}1 \text{ zł}$$

Koszt zatrudnienia pracowników:

Łącznie zebrano:

$$22 + 22 + 22 + 22 = 88 \ kg \ owoców$$

co daje łączny koszt zatrudnienia pracowników równy 100 złotych. Dodając do tego koszt wynajmu magazynu - 25 złotych otrzymujemy zysk po czwartym dniu równy:

$$191 + 131{,}1 - (100 + 25) = 197{,}1 \ złotych$$

Dzień piąty:

	Wiśnie	Jabłka	Gruszki	Śliwki
Ilość	22	22	22	22
Sprzedano na rynku	10	3	4	10
Sprzedano w skupie	13	20	38	20
Magazynowane	0	0	0	0
Popyt	10	3	4	10
Cena bazowa na targu	7	3	3	5
Cena w skupie	5	3	3	4

Zysk ze sprzedaży w skupie:

$$13 \times 5 + 20 \ \times 3 + 38 \times 3 + 20 \times 4 = 65 + 60 + 114 + 80 = 319 \text{ zł}$$

Zysk ze sprzedaży na rynku:

$$10 \times 7 \times 1 + 3 \ \times 3 \times 1 + 4 \times 3 \times 1 + 10 \times 5 \times 1 = 70 + 9 + 12 + 50 = 141 \ zł$$

Koszt zatrudnienia pracowników:

Łącznie zebrano:

$$22 + 22 + 22 + 22 = 88 \ kg \ owoców$$

co daje łączny koszt zatrudnienia pracowników równy 100 złotych. Dodając do tego koszt wynajmu magazynu - 15 złotych otrzymujemy zysk po piątym dniu równy:

$$319 + 141 - (100 + 15) = 345 \ złotych$$

Zysk po pięciu dniach:

$$345 + 197{,}1 + 251 + 192{,}9 + 140 = 1126 \ zł$$

Zysk po odjęciu kosztów zasadzenia:

$$1126 \ - 85 - 62 - 91 - 110 = 778 \ zł$$

Otrzymany wynik pokrywa się z wynikiem otrzymanym przez algorytm symulowanego wyżarzania:

```
best profit: 778.0 | temperature: 1.023548201039937
best profit: 778.0 | temperature: 1.0133127190295377
best profit: 778.0 | temperature: 1.0031795918392423
kryt stopu 1
Solution: Day 1
harvested: [17, 17, 17, 17]
sold_market: [0, 0, 0, 0]
sold_wholesale: [17, 17, 17, 17]
warehouse: [0, 0, 0, 0]

Day 2
harvested: [20, 20, 20, 20]
sold_market: [10, 2, 3, 10]
sold_wholesale: [8, 17, 14, 9]
warehouse: [2, 1, 3, 1]

Day 3
harvested: [25, 25, 25, 25]
sold_market: [10, 3, 4, 10]
sold_wholesale: [16, 11, 23, 12]
warehouse: [1, 12, 1, 4]

Day 4
harvested: [22, 22, 22, 22]
sold_market: [10, 2, 1, 10]
sold_wholesale: [12, 31, 2, 8]
warehouse: [1, 1, 20, 8]

Day 5
harvested: [22, 22, 22, 22]
sold_market: [10, 3, 4, 10]
sold_wholesale: [13, 20, 38, 20]
warehouse: [0, 0, 0, 0]





778.0
```

5.5 Porównanie z dokładnym rozwiązaniem

W celu znalezienia dokładnego rozwiązania rozważyliśmy prosty przypadek o następujących danych:

- dni zbiorów: 2
- pojemność magazynu: 1
- dzienny limit zbiorów: 2
- 2 typy owoców o następujących danych
    a. Gruszki:
        - kilogramy w sadzie: 3
        - koszt zasadzenia: 1.5
        - ceny podstawowe na targu: [2, 1.5]
        - ceny w skupie: [1, 1]
        - popyt: [1, 3]
    b. Jabłka:
        - kilogramy w sadzie: 3
        - koszt zasadzenia: 2
        - ceny podstawowe na targu: [2, 3]
        - ceny w skupie: [1, 1.5]

        - popyt: [2, 2]
- mnożnik ceny:

```
def multiplier(k):
    if k < 30:
        return 1.5
    if k < 50:
        return 1.2
    if k < 80:
        return 1.1
    if k <= 100:
        return 1
```

- koszty pracowników:

```
def employee_cost(kilograms):
    if 0 <= kilograms <= 1:
        cost = 1.5
    elif 1 < kilograms <= 2:
        cost = 2
    else:
        cost = 3
    return cost
```

- koszty magazynowania:

```
def warehouse_cost(kilograms):
    if 0 <= kilograms <= 1:
        cost = 1
    elif 1 < kilograms <= 2:
        cost = 2
    else:
        cost = 3
    return cost
```

Następnie ręcznie wprowadzono wszystkie sposoby na jakie w ciągu dnia można ułożyć zbiory, sprzedaż i magazyn z założeniem, że nie posiadamy dodatkowych owoców z magazynu z dnia poprzedniego. Następnie w pętli po tych dziennych rozwiązaniach dodano wszystkie opcje z założeniem, że posiadamy jakieś owoce w magazynie z poprzedniego dnia.

Kolejnym krokiem było sprawdzenie wszystkich rozwiązań rozwiązań metodą brute force i wybranie tych, które spełniały ograniczenia.

```
Day 1
harvested: [1, 1]
sold_market: [1, 1]
sold_wholesale: [0, 0]
warehouse: [0, 0]

Day 2
harvested: [0, 2]
sold_market: [0, 2]
sold_wholesale: [0, 0]
warehouse: [0, 0]

Zysk: 0.7000000000000002
Poprawne rozwiązania: 1237
Ilość dziennych rozwiązań: 125
```

Rys. 28. Dokładne wyniki

Powyżej widać, że nawet dla tak prostych założeń istniało aż 1237 rozwiązań spełniających warunki.

```
Wynik algorytmu genetycznego
Day 1
harvested: [2, 0]
sold_market: [1, 0]
sold_wholesale: [0, 0]
warehouse: [1, 0]

Day 2
harvested: [0, 2]
sold_market: [1, 2]
sold_wholesale: [0, 0]
warehouse: [0, 0]

Zysk: 0.2999999999999998
```

Rys. 29. Algorytm genetyczny, który w rozwiązaniach początkowych nie zawiera optymalnej strategii zbiorów.

```
Wynik algorytmu genetycznego
Day 1
harvested: [1, 1]
sold_market: [1, 1]
sold_wholesale: [0, 0]
warehouse: [0, 0]

Day 2
harvested: [0, 2]
sold_market: [0, 2]
sold_wholesale: [0, 0]
warehouse: [0, 0]

Zysk: 0.7000000000000002
```

Rys. 30. Algorytm genetyczny, który w rozwiązaniach początkowych zawiera optymalną strategię zbiorów.

```
Wynik algorytmu wyrzarzania
Day 1
harvested: [2, 0]
sold_market: [0, 0]
sold_wholesale: [2, 0]
warehouse: [0, 0]

Day 2
harvested: [0, 2]
sold_market: [0, 0]
sold_wholesale: [0, 2]
warehouse: [0, 0]

Zysk: -4.5
```

Rys. 31. Algorytm symulowanego wyżarzania, który startuje z rozwiązania innego niż optymalne.

```
Wynik algorytmu wyrzarzania
Day 1
harvested: [1, 1]
sold_market: [1, 1]
sold_wholesale: [0, 0]
warehouse: [0, 0]

Day 2
harvested: [0, 2]
sold_market: [0, 2]
sold_wholesale: [0, 0]
warehouse: [0, 0]

Zysk: 0.7000000000000002
```

Rys. 32. Algorytm symulowanego wyżarzania, który startuje z rozwiązania optymalnego.

Oba algorytmy nie były w stanie znaleźć optymalnego rozwiązania w momencie gdy wśród rozwiązań początkowych nie było rozwiązania optymalnego. Może to wynikać z faktu, że nawet dla tak prostego przypadku ilość dopuszczalnych rozwiązań jest bardzo duża.

6 Podsumowanie

Udało nam się zaimplementować algorytm symulowanego wyżarzania i genetyczny oraz je przetestować.

Główne problemy jakie napotkaliśmy to:

- Utrudnione szukanie błędów w kodzie z powodu losowości w algorytmach
- Uzyskiwanie rozwiązań niedopuszczalnych po mutacji kub krzyżowaniu

Podczas pracy nad projektem użyliśmy między innymi Python, Jupyter notebook oraz Github co pozwoliło nam szybko pisać kod, rysować wykresy, oraz kontrolować postępy.

Nie tylko nauczyliśmy się jak działają te algorytmy, ale także rozwinęliśmy naszą wiedzę z programowania obiektowego w Pythonie oraz pracy w grupie.

Kierunki rozwoju projektu:

- więcej rodzajów mutacji, krzyżowania i selekcji
- interfejs graficzny
- optymalizacja kodu

7 Literatura

Wykłady oraz materiały udostępnione na platformie UPeL przez dr hab. inż. Joanna Kwiecień oraz dr hab. inż. Wojciech Chmiel.

Strony internetowe:

https://www.obitko.com/tutorials/genetic-algorithms/selection.php

https://www.tutorialspoint.com/genetic_algorithms/index.htm

8 Podział pracy

Etap	Piotr Hudaszek [%]	Wojciech Żyła [%]	Magdalena Leonkiewicz [%]
Model zagadnienia	70	25	5
Implementacja modelu	35	30	35
Algorytm symulowanego wyżarzania	50	50	0
Algorytm genetyczny	33	33	33
Implementacja aplikacji	0	100 (prosty i krótki etap)	0
Testy	0	40	60
Dokumentacja	33	33	33