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 3](#_Toc92024501)

[2 Opis zagadnienia 3](#_Toc92024502)

[2.1 Sformułowanie problemu 3](#_Toc92024503)

[2.2 Model matematyczny 3](#_Toc92024504)

[3 Opis algorytmów 5](#_Toc92024505)

[3.1 Elementy wspólne dla obu algorytmów 5](#_Toc92024506)

[3.1.1 Reprezentacja rozwiązania 5](#_Toc92024507)

[3.1.2 Sprawdzenie czy rozwiązanie spełnia ograniczenia 5](#_Toc92024508)

[3.1.3 Generacja rozwiązania / populacji początkowej 8](#_Toc92024509)

[3.2 Algorytm SA (symulowanego wyżarzania) 12](#_Toc92024510)

[3.2.1 Schemat algorytmu 12](#_Toc92024511)

[3.2.2 Wyznaczenie sąsiedniego rozwiązania 12](#_Toc92024512)

[3.3 Algorytm genetyczny 13](#_Toc92024513)

[3.3.1 Schemat algorytmu 13](#_Toc92024514)

[3.3.2 Selekcja 14](#_Toc92024515)

[3.3.2 Krzyżowanie 14](#_Toc92024516)

[3.3.2 Mutacja 14](#_Toc92024517)

[4 Aplikacja 14](#_Toc92024518)

[5 Eksperymenty 15](#_Toc92024519)

[Testowanie losowej wartości parametru *demand\_rate* w funkcji *generate\_satisfy\_demand* służącej do generowania rozwiązania początkowego 16](#_Toc92024520)

[Testowanie algorytmu genetycznego 18](#_Toc92024521)

[6 Podsumowanie 22](#_Toc92024522)

[7 Literatura 22](#_Toc92024523)

[8 Podział pracy 22](#_Toc92024524)

# 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

– ilość typów owoców

– typ owoców

- ilość owoców danego typu

– dzień

– ilość owoców zebranych typu s dnia t

– ilość owoców zebranych wszystkich typów dnia t

– max ilość owoców jaką można dziennie zebrać

– koszt zebrania f owoców dziennie (zwykła tabela)

) – koszt utrzymania danego typu owoców

– ilość sprzedanych na targu owoców typu dnia

– ilość sprzedanych na skupie owoców typu dnia

– popyt na owoce typu s dnia t

– cena bazowa na targu

– cena w skupie

– mnożnik ceny bazowej zależny od procentu zaspokojenia popytu

– ilość owoców typu przekazanych w dniu do magazynu (muszą zostać sprzedane następnego dnia)

– maksymalna pojemność magazynu

– koszt magazynowania owoców przez noc

**Postać rozwiązania**

**Funkcja celu (zysk)**

**Ograniczenia**

Dla każdego s:

Dla każdego i :

# 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.  
 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\_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\_sol\_acceptable(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 3 ostatecznie 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)  
  
 return one and two and four and five and six and seven

### 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.

def generate\_all\_to\_wholesale(self, harvest\_strategies: List[List]):  
# ilość rodzajów owoców  
 fruit\_types\_count = len(self.fruit\_types)

# Hash mapa 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(0, 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[j] = harvest\_per\_type[fruit\_id]  
 fruits\_left[f\_type.name] -= harvest\_per\_type[fruit\_id]  
 else:  
 solution.days[day\_id].harvested[j] = 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:

def generate\_satisfy\_demand(self, harvest\_strategies: List[List], demand\_rate: float,  
 random\_demand\_rate: bool):# ilość rodzajów owoców  
 fruit\_types\_count = len(self.fruit\_types)

# Hash mapa 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)

# 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 range(0, 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[j] = 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[j] = solution.days[day\_id].harvested[fruit\_id]  
 # pozostałości przeznaczone do skupu lub magazynu  
 leftovers = 0  
 else:  
 solution.days[day\_id].sold\_market[j] = 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[j] = 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[j]  
 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) <= 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):  
# 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 = max\_daily\_harvest // fruit\_types\_count  
 har\_per\_type2 = int(0.9 \* max\_daily\_harvest) // fruit\_types\_count  
 har\_per\_type3 = int(0.7 \* max\_daily\_harvest) // fruit\_types\_count  
 har\_per\_type4 = int(0.5 \* max\_daily\_harvest) // fruit\_types\_count

# lista do przechowywania poszczególnych strategii zbiorów

all\_strategies = []  
 x = numer\_of\_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(([numer\_of\_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   
 # ogranieczenia.

result = []  
 for el in solutions:  
 result.append((el, check\_if\_sol\_acceptable(el)))  
 return result

## 3.2 Algorytm SA (symulowanego wyżarzania)

### 3.2.1 Schemat algorytmu

**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 () i typ owoców (). Dla tego dnia i typu wybieramy losowo i zmieniamy ilość zebraną () lub ilośc sprzedaną na targu () lub ilość sprzedaną na skupie (). Zmiana polega na wylosowaniu liczby z zbioru i dodaniu jej.

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

Dla każdego i :

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:

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 było by 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):  
 *"""  
 Metoda znajdująca rozwiązanie optymalne za pomocą algorytmu genetycznego* ***: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  
  
 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:  
 # jeślli 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}")  
  
 return population[-1][0], population[-1][1], iterations

### 3.3.2 Selekcja

### 3.3.2 Krzyżowanie

Aby skrzyżować dwa rozwiązania najpierw losujemy ze zbioru dzień , następnie ze zbioru dzień b. Dzielimy rozwiązania rodziców ( i ) na 3 przedziały:

1. dni 1 do
2. dni do
3. dni 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 ma więcej w magazynie niż to potomek musi w dniu 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 tą 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 jęyk 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 wyrzarzania: python -m project\_app annealing --t\_start 5000 --t\_stop 800 --iter\_in\_temp 100   
--epsilon 2 --iter\_epsilon 10 --alpha 0.99 --neighbour\_type 1 --initial\_sol 11 --random\_demand\_rate –verbose

Znaczenie poszczególnych parametrów:

* --t\_start temperatura początkowa
* --t\_stop temperatura końcowa
* --iter\_in\_temp ilość iteracji wykonywanych dla jednej temperatury
* --epsilon
* --iter\_epsilon
* --alpha współczynnik o jaki zmniejszana jest temperatura
* --neighbour\_type metoda wyboru sąsiada
* --initial\_sol numer rozwiązania początkowego
* --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 temperatury oraz najlpeszego 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 najlpeszego zysku w trakcie pracy algorytmu

**Ustawienie parametrów sadu oraz owoców**  
Parametry sadu oraz różne rodzaje owoców, koszta magazynowania i koszta 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.

## Testowanie losowej wartości parametru *demand\_rate* w funkcji *generate\_satisfy\_demand* służącej do generowania rozwiązania początkowego

Text

Description automatically generatedWpływ 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.

## 

Rys. <numer>. Dwa zestawy ustawień algorytmu podczas testu.

Text

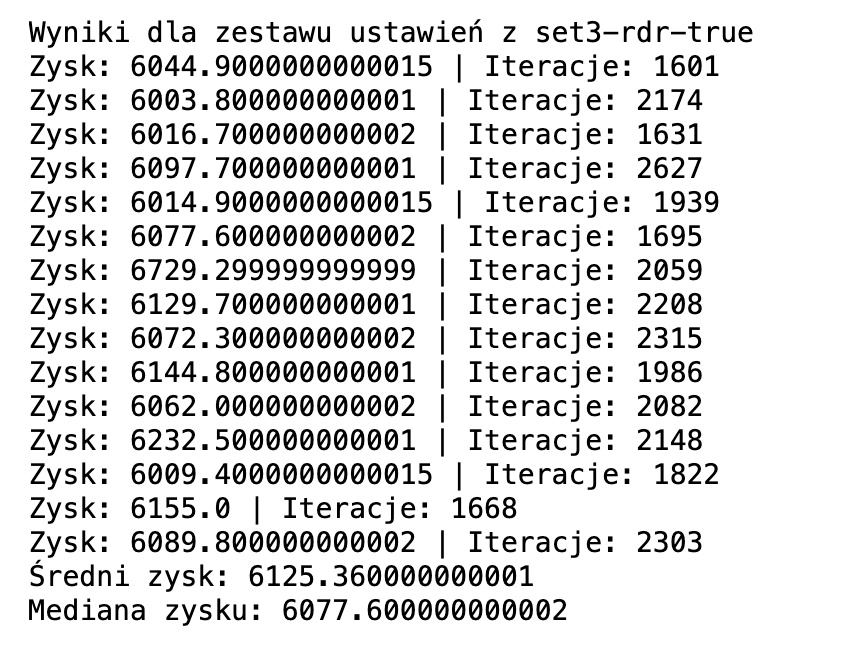
Description automatically generated

Rys. <numer>. Wyniki testów dla *random\_demad\_rate* ustawionego na False.

A picture containing icon

Description automatically generated

Rys. <numer>. Wizualizacja wyników z rys. <poprzedni numer>.



Rys. <numer>. Wyniki testów dla *random\_demad\_rate* ustawionego na True.

Chart, scatter chart

Description automatically generated

Rys. <numer>. Wizualizacja wyników z rys. <poprzedni numer>.

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.

## Testowanie algorytmu genetycznego

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

Text

Description automatically generated

Rys. <numer>. Zestawy ustawień algorytmu podczas testu.

Text

Description automatically generated

Rys. <numer>. Wyniki działania algorytmu.

Chart, scatter chart

Description automatically generated

Rys. <numer>. Wizualizacja wyników z rys. <poprzedni numer>.

Text

Description automatically generated

Rys. <numer>. Wyniki działania algorytmu.

Chart, scatter chart

Description automatically generated

Rys. <numer>. Wizualizacja wyników z rys. <poprzedni numer>.

Text

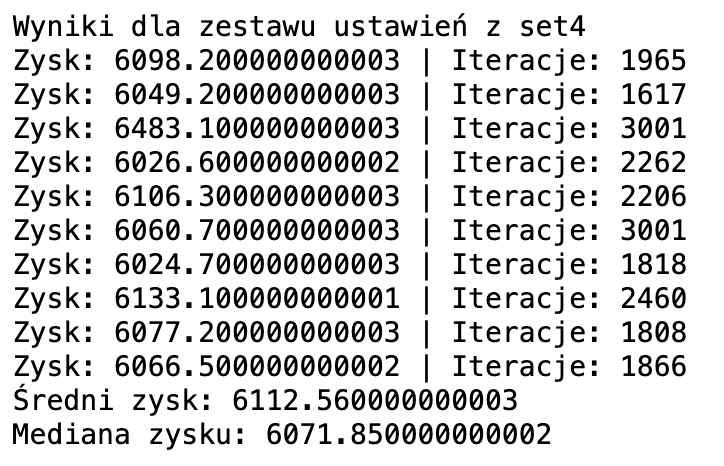
Description automatically generated

Rys. <numer>. Wyniki działania algorytmu.

Scatter chart

Description automatically generated

Rys. <numer>. Wizualizacja wyników z rys. <poprzedni numer>.



Rys. <numer>. Wyniki działania algorytmu.

Chart, scatter chart

Description automatically generated

Rys. <numer>. Wizualizacja wyników z rys. <poprzedni numer>.

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*.

A blue line on a white background

Description automatically generated with medium confidence

Rys. <numer>. Poprawa rozwiązania podczas działania algorytmu genetycznego.

# 6 Podsumowanie

# 7 Literatura

# 8 Podział pracy