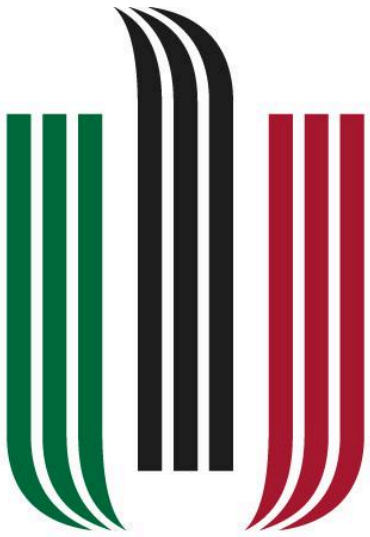


AGH

AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA
W KRAKOWIE

Badania operacyjne
Projekt

Jakub Czajka

Michał Kobiera

Patryk Klatka

Maciej Pięta

Martyna Sokołowska

Spis treści

Spis treści **2**
Wstęp **3**
Opis zagadnienia **3**
Sformułowanie problemu 3
Zastosowanie 3
Model matematyczny **4**
Dane 4
Szukane 4
Ograniczenia 4
Minimalizowana funkcja 4
Opis algorytmów **5**
Algorytm pszczeli 5
Adaptacja algorytmu dla przedstawionego problemu 6
Pseudokod algorytmu 6
Generacja losowego rozwiązania 6
Warunek stopu 6
Generacja sąsiedniego rozwiązania 7
Parametry algorytmu 7
Algorytm genetyczny 8
Adaptacja algorytmu dla przedstawionego problemu 9
Pseudokod algorytmu 9
Generacja losowego rozwiązania 9
Warunek stopu 9
Krzyżowanie 9
Mutacja 9
Parametry algorytmu 9
Aplikacja **10**
Definiowanie problemu do rozwiązania 10
Uruchamianie algorytmu pszczółkowego 12
Uruchamianie algorytmu genetycznego 12
Testy **13**
Porównanie wyników algorytmu genetycznego z algorytmem pszczółkowym 13
Badanie wpływu parametrów na wyniki algorytmu 15
Badanie wpływu funkcji generowania sąsiednich rozwiązań dla algorytmu pszczółkowego na wynik 17
Podsumowanie **18**
Literatura **19**

Wstęp

Celem projektu było znalezienie najlepszego rozplanowania pokoi piętra hotelu, tak, aby przy ustalonym budżecie budynek przynosił jak największe zyski roczne. Pod uwagę były brane takie parametry jak powierzchnia piętra, typy pokoi, zyski z danego typu pokoju, cena budowy, koszty utrzymania. W celu znalezienia optymalnego rozwiązania problemu posłużono się technikami badań operacyjnych. Rozwiązania zostały wygenerowane za pomocą algorytmu pszczelego oraz genetycznego.

Opis zagadnienia

Sformułowanie problemu

Za pomocą z góry ustalonej listy pokojów o danej powierzchni, cenie za dzień, potencjalnych kosztach utrzymywania pokoju, ceny budowy pokoju należy wyznaczyć taki zbiór pokoi, który będzie generował największe zyski roczne z utrzymywania hotelu.

Zastosowanie

Rozwiązanie powyższego problemu można oczywiście wykorzystać przy budowie hotelu - biorąc pod uwagę coraz mniejsze możliwości pod względem wielkości działek taki program pozwoli na maksymalizację zysku na niekoniecznie dużej działce np. w centrum miasta. Oczywiście rozwiązanie tego problemu niekoniecznie będzie przydatne tylko dla hoteli - również dla deweloperów budujących mieszkania takie rozwiązanie będzie przydatne. Dzięki takiemu programowi będzie można maksymalizować zyski ze sprzedaży lekko zmieniając oraz inaczej interpretując parametry programu.

Model matematyczny

Dane

$nr \in R_+$ - metraż piętra

$kr \in R_+$ - metraż korytarza

$p \in P$ - zbiór pokoi, każdy pokój to szóstka: (t, m, u, k, c, d):

$t \in N$ - typ pokoju

$m \in R_+$ - metraż pokoju

$u \in N$ - liczba dni użytkowania pokoju w danym roku

$k \in N$ - koszty utrzymywania pokoju rocznie - usługi sprzątania, naprawy

$c \in N$ - cena budowy pokoju oraz jego umeblowania

$d \in N$ - cena pokoju za dzień użytkowania (bez względu na liczbę gości)

$bz \in R_+$ - budżet budowy piętra hotelu

$w_1, w_2, \ldots, w_n \in N$ - minimalna liczba pokoi danego typu na danym piętrze

Szukane

$s \in P$ - zbiór pokoi dające największy zysk roczny.

Ograniczenia

Budżet nie może zostać przekroczony:

$$bz - \sum_{p \varepsilon P} p.c > 0$$

Powierzchnia pokoi nie może przekroczyć powierzchni danego piętra:

$$nr - kr - \sum_{p \varepsilon P} p.m > 0$$

W hotelu musi być conajmniej x pokoi danego typu:

$$\forall i \in N: \sum_{p \varepsilon P,\, p.t = i} 1 = w_i$$

Minimalizowana funkcja

Suma po zyskach z pokoi w trakcie roku

$$f(P) = \sum_{p \varepsilon P} [p.d * (p.u - p.k)]$$

Opis algorytmów

Algorytm pszczeli

Pierwszym algorytmem, z którego skorzystaliśmy był algorytm pszczeli. Algorytm ten opiera się na pewnej procedurze, która rozpoczyna się od wylosowania początkowej puli rozwiązań. Kolejnym krokiem jest ocena jakości każdego z tych rozwiązań, na podstawie której wybierane są rozwiązania dobre oraz elitarne, które będą dalej brane pod uwagę.

Za każdym razem, kiedy wybrane zostanie rozwiązanie dobre lub elitarne, generowane są kolejne rozwiązania z jego bliskiego otoczenia. Ilość generowanych rozwiązań zależy od dwóch parametrów, oddzielnych dla rozwiązań dobrych i elitarnych. Konieczne jest, jednak, zdefiniowanie, co to znaczy "bliskie otoczenie" dla danego problemu.

Następnie spośród wygenerowanych rozwiązań, wybierane jest najlepsze rozwiązanie dobre i najlepsze rozwiązanie elitarne. Proces nie kończy się na tym - do tych rozwiązań dodawane są kolejne losowe rozwiązania.

Cała wyżej wymieniona procedura jest powtarzana iteracyjnie. Kryterium zakończenia to osiągnięcie kryterium stopu lub wykonanie ustalonej liczby iteracji.

```
1 for i=1,…,ns
    i  scout[i]=Initialise_scout()
    ii flower_patch[i]=Initialise_flower_patch(scout[i])
2 do until stopping_condition=TRUE
    i   Recruitment()
    ii  for i =1,...,na
          1 flower_patch[i]=Local_search(flower_patch[i])
          2 flower_patch[i]=Site_abandonment(flower_patch[i])
          3 flower_patch[i]=Neighbourhood_shrinking(flower_patch[i])
    iii for i = nb,...,ns
          1 flower_patch[i]=Global_search(flower_patch[i])}
```

Rysunek 1 - Pseudokod algorytmu pszczelego

Adaptacja algorytmu dla przedstawionego problemu

Pseudokod algorytmu

Nasz algorytm niewiele różni się od oryginalnej wersji algorytmu pszczółkowego. Poniższy pseudokod opisuje strukturę głównej funkcji algorytmu:

1. Wygeneruj losową populację rozmiaru **NUM_ROOM_TYPES**.
2. Przez **NUM_ITERATIONS** iteracji wykonuj:
    a. Posortuj populację rosnąco.
    b. Wygeneruj losową populację o rozmiarze **BEES_RANDOM** w celu uniknięcia lokalnych ekstremów.
    c. Dla pierwszych **NUM_STRICTLY_BEST_BEES**:
        i. Wykonaj funkcję *send_close_to_bee* **BEES_FOR_TOP_BEST** razy, która szuka lepszego rozwiązania w pobliżu pszczoły - usuwa pokoje danego typu jeżeli to możliwe i wybiera nowe, które spełniają warunki zadania.
        ii. Dodaj wyniki do nowej populacji.
    d. Dla ostatnich **NUM_STRICTLY_BEST_BEES** pszczół:
        i. Wykonaj funkcję *send_close_to_bee* **BEES_FOR_DOWN_BEST** razy.
        ii. Dodaj wyniki do nowej populacji.

Generacja losowego rozwiązania

Losowe rozwiązanie jest tworzone przez metodę *build_batch* w klasie **FloorFactory**. Poniżej przedstawiony jest pseudokod działania w/w funkcji:

1. Dla podanych typów pokoi oraz minimalnej liczby pokoi danego typu uaktualnij wartości powierzchni piętra oraz koszt budowy piętra.
2. Dopóki koszt mieści się w budżecie oraz powierzchnia piętra nie przekracza minimum:
    a. Losowo wyszukaj nowy pokój spośród dostępnych typów pokoi.
    b. Zaktualizuj wartość powierzchni piętra oraz koszt budowy piętra.

Warunek stopu

Algorytm szuka najlepszego rozwiązania **NUM_ITERATIONS** razy.

Generacja sąsiedniego rozwiązania

Zostały zaimplementowane dwa algorytmy generujące sąsiednie rozwiązanie: *send_close_to_bee oraz send_close_to_bee2.*

Poniżej został przedstawiony pseudokod funkcji *send_close_to_bee*:

1. Znajdź typy pokoi, z których możemy usunąć pokój (zgodnie z założeniem na minimalną liczbę pokoi danego typu).
2. Wylosuj jeden typ pokoju i usuwamy jeden pokój tego typu.
3. Dopóki prawda:
    a. Dla danych typów pokoi wyszukaj kandydatów na nowy pokój - każdy z nich musi spełnić warunki powierzchni jak i budżetu (dodanie pokoju danego typu nie może ich przekroczyć).
    b. Gdy brak kandydatów, zakończ program zwracając nową pszczołę.
    c. Wylosuj jeden pokój z listy kandydatów i dodaj go do rozwiązania.

Pseudokod *send_close_to_bee2* wygląda następująco:

1. Znajdź typy pokoi, z których możemy usunąć pokój (zgodnie z założeniem na minimalną liczbę pokoi danego typu).
2. Wylosuj jeden typ pokoju i usuwamy jeden pokój tego typu.
3. Dopóki prawda:
    a. Dla danych typów pokoi wyszukaj kandydatów na nowy pokój - każdy z nich musi spełnić warunki powierzchni jak i budżetu (dodanie pokoju danego typu nie może ich przekroczyć).
    b. Gdy brak kandydatów, zakończ program zwracając nową pszczołę.
    c. Posortuj listę kandydatów po rocznym zysku pokoju.
    d. Dodaj ten pokój z listy kandydatów do rozwiązania, który ma największy zysk.

Parametry algorytmu

Poniżej zostały przedstawione parametry, z których korzysta nasza implementacja algorytmu pszczółkowego:

- **NUM_ITERATIONS** - liczba iteracji wyszukiwania najlepszego rozwiązania
- **NUM_BEES** - liczba pszczół
- **NUM_STRICTLY_BEST_BEES** - rozmiar przeszukiwanego sąsiedztwa dla rozwiązania elitarnego oraz normalnego
- **NUM_BEST_BEES** - liczba pszczół, które mają najlepsze wyniki

Algorytm genetyczny

Drugą metodą, z której skorzystaliśmy, był algorytm genetyczny. Ten algorytm zaczyna się od generowania początkowej populacji rozwiązań, które są potem poddawane ocenie. Każde rozwiązanie jest oceniane na podstawie funkcji przystosowania, która pozwala na identyfikację najlepszych osobników w populacji.

Główna procedura algorytmu genetycznego opiera się na iteracyjnym procesie selekcji, krzyżowania i mutacji. W trakcie selekcji, najlepsze rozwiązania z aktualnej populacji są wybierane do tworzenia kolejnej generacji. Następnie, za pomocą operacji krzyżowania, tworzone są nowe rozwiązania, które łączą cechy swoich "rodziców". Operacja mutacji wprowadza małe, losowe zmiany w niektórych nowo utworzonych rozwiązaniach, zapewniając różnorodność genetyczną.

Procedura jest powtarzana do momentu, gdy zostanie spełnione kryterium zakończenia - na przykład po osiągnięciu określonej liczby generacji lub jeżeli jakość najlepszego rozwiązania nie poprawia się przez określoną liczbę iteracji.

```
Genetic_Algorithm() {
                Initialize random population;
                Evaluate the population;
                Generation = 0;
     While termination criterion is not
     satisfied {
            Generation = Generation + 1;
            Select good chromosomes by
            reproduction procedure;
            Perform crossover with probability
            crossover (Pc);
            Perform mutation with probability
            of mutation (Pm);
            Evaluate the population;
     }
}
```

Rysunek 2 - Pseudokod algorytmu genetycznego

Adaptacja algorytmu dla przedstawionego problemu

Pseudokod algorytmu

Implementacja naszego algorytmu również nie odbiega od wersji oryginalnej:

1. Dopóki otrzymujemy lepsze wyniki:
    a. Wybierz rodziców poprzez losowy wybór **PARENTS_SIZE** genomów z danej populacji (wybór przykładowych piętr) - zostaną oni wykorzystani w procesie krzyżowania
    b. Wykonaj krzyżowanie:
        i. Utwórz kolejne pary rodziców.
        ii. Wybierz losowo typy pokoi.
        iii. Zamień liczbę pokoi, jeżeli mieszczą się one w wymaganiach.
    c. Wykonaj mutacje:
        i. Wylosuj dwa typy pokoi
        ii. Dla jednego typu usuń jeden pokój, a dla drugiego dodaj nowy pokój
    d. Oblicz totalny zysk roczny, i jeżeli jest on lepszy niż dotychczasowy, podmień najlepszy genom.

Generacja losowego rozwiązania

Procedura jest identyczna jak dla algorytmu pszczelego.

Warunek stopu

W przeciwieństwie do algorytmu pszczelego tutaj algorytm działa dopóki nie zmieni najlepszego wyniku przez **NO_IMPROVEMENT_TIME** iteracji.

Krzyżowanie

Tworzymy kolejne pary rodziców idąc po kolei w liście wybranych rodziców. Następnie wykonujemy krzyżowanie poprzez losowy wybór typów pokoi, a następnie zamieniamy liczbę pokoi, jeżeli mieszczą się one w wymaganiach.

Mutacja

Dla wszystkich genomów losujemy losowe dwa typy pokoi i dla jednego typu usuwamy jeden pokój, a dla drugiego dodajemy nowy pokój.

Parametry algorytmu

NO_IMPROVEMENT_TIME - liczba iteracji w których wynik nie został poprawiony.

Aplikacja

Aplikacja została napisana w języku Python ze wsparciem wersji 3.10. Dodatkowo, zostały wykorzystane biblioteki:

- attrs
- prettyprinter
- factory-boy
- seaborn
- matplotlib

Wszystkie zewnętrzne biblioteki zostały zapisane w pliku requirements.txt w celu wygodnej instalacji odpowiednich wersji.

Definiowanie problemu do rozwiązania

Aby zdefiniować problem, należy na początku zdefiniować typy pokoi, które chcemy aby znalazły się na piętrze hotelu. W tym celu możemy posłużyć się metodą **RoomTypeFactory.build_batch**, która losowo definiuje typy pokoi o parametrach:

- int size - wielkość pokoju
- int frequency_of_use - liczba dni użytkowania pokoju w danym roku
- float cost_of_maintenance - koszty utrzymywania pokoju
- float cost_of_building - koszt budowy pokoju
- float cost_per_day - koszt wynajmu pokoju na jeden dzień

Pokoje można również zdefiniować ręcznie, tworząc listę obiektów **RoomType**.

```
from src.model_factory import RoomTypeFactory, FloorFactory, RoomType

# Random data
rooms = RoomTypeFactory.build_batch(size=NUM_ROOM_TYPES)

# Hardcoded data
rooms = [
    RoomType(type=0, size=10, frequency_of_use=10, cost_of_maintenance=10, cost_of_building=10, cost_per_day=10),
    RoomType(type=1, size=20, frequency_of_use=20, cost_of_maintenance=20, cost_of_building=20, cost_per_day=20),
    RoomType(type=2, size=30, frequency_of_use=30, cost_of_maintenance=30, cost_of_building=30, cost_per_day=30),
    RoomType(type=3, size=40, frequency_of_use=40, cost_of_maintenance=40, cost_of_building=40, cost_per_day=40),
    RoomType(type=4, size=50, frequency_of_use=50, cost_of_maintenance=50, cost_of_building=50, cost_per_day=50),
    RoomType(type=5, size=60, frequency_of_use=60, cost_of_maintenance=60, cost_of_building=60, cost_per_day=60),
    RoomType(type=6, size=70, frequency_of_use=70, cost_of_maintenance=70, cost_of_building=70, cost_per_day=70),
    RoomType(type=7, size=80, frequency_of_use=80, cost_of_maintenance=80, cost_of_building=80, cost_per_day=80),
    RoomType(type=8, size=90, frequency_of_use=90, cost_of_maintenance=90, cost_of_building=90, cost_per_day=90),
    RoomType(type=9, size=100, frequency_of_use=100, cost_of_maintenance=100, cost_of_building=100, cost_per_day=100)
]
```

Rysunek 3 - Definicja typów pokoi

Następnie należy utworzyć piętro za pomocą metody **FloorFactory.build** - ta metoda pozwala na utworzenie pierwszego rozwiązania dla naszego zbioru pokoi. W celu utworzenia piętra należy podać:

- float capacity - powierzchnia piętra
- float corridor_capcity - powierzchnia korytarza
- float budget - budżet budowy piętra
- list[int] min_room_num - minimalna liczba pokoi danego typu
- list[RoomType] room_types - lista z typami pokojów

```
floor = FloorFactory.build(
    capacity=CAPACITY,
    corridor_capacity=CORRIDOR_CAPACITY,
    budget=BUDGET,
    min_room_num=[1 for _ in range(len(rooms))],
    room_types=rooms
)
```

Rysunek 4 - Utworzenie piętra na podstawie dobranych parametrów

Uruchamianie algorytmu pszczółkowego

Po zdefiniowaniu typów pokoi wystarczy uruchomić funkcję **bee_algorithm**, która przyjmuje parametry:

- int num_bees - liczba pszczół
- int num_iterations - liczba iteracji szukania rozwiązania
- list[RoomType] rooms - lista z typami pokojów
- float capacity - powierzchnia piętra
- float corridor_capacity - powierzchnia korytarza
- float budget - budżet budowy piętra

Funkcja ta zwraca dwie wartości - najlepsze rozwiązanie oraz historię polepszania wyniku. W celu uzyskania wartości najlepszego rozwiązania można odwołać się do własności *fitness*. Uzyskanie optymalnego rozłożenia jest możliwe poprzez własność *position*. Funkcję **bee_algorithm** należy zaimportować z modułu PSO.

```
rooms = RoomTypeFactory.build_batch(size=NUM_ROOM_TYPES)
best_solution, fitness_history = bee_algorithm(NUM_BEES, NUM_ITERATIONS, rooms, CAPACITY, CORRIDOR_CAPACITY, BUDGET)
print("Najlepsze rozwiązanie:", best_solution)
print(f"Wartość najlepszego rozwiązania: {best_solution.fitness:.2e}")
```

Rysunek 5 - Przykładowy kod przedstawiający uruchomienie algorytmu pszczółkowego

Uruchamianie algorytmu genetycznego

Zasada korzystania z implementacji algorytmu genetycznego jest nieco inna - po zdefiniowaniu pierwszej populacji oraz odpowiednich parametrów dla naszego problemu, tworzymy instancję klasy **GeneticSearch.** Następnie, w celu uzyskania rozwiązania, należy wywołać metodę **run**, a następnie **calculate_fitness**, która zwróci wartość zysku rocznego, a **room_count** odpowiednią liczbę pokoi danego typu dla najlepszego rozwiązania.

```
initial_population = model_factory.FloorFactory.build_batch(size=50,
                                                             capacity=capacity,
                                                             corridor_capacity=corridor_capacity,
                                                             budget=budget,
                                                             min_room_num=min_room_num,
                                                             room_types=rooms)

search = GeneticSearch(initial_population)
result = search.run()
cpprint(result.calculate_fitness())
cpprint(result.room_count)
```

Rysunek 6 - Przykładowy kod przedstawiający uruchomienie algorytmu genetycznego

Testy

Zostały wykonane trzy typy testów - pierwszym z nich było porównanie który z algorytmów lepiej radzi sobie dla pewnego zdefiniowanego zadania - drugie natomiast polegały na zbadaniu wpływu parametrów na wynik algorytmu. W trzecim typie testów porównaliśmy metody odpowiedzialne za generowanie sąsiednich rozwiązań dla algorytmu pszczółkowego.

Porównanie wyników algorytmu genetycznego z algorytmem pszczółkowym

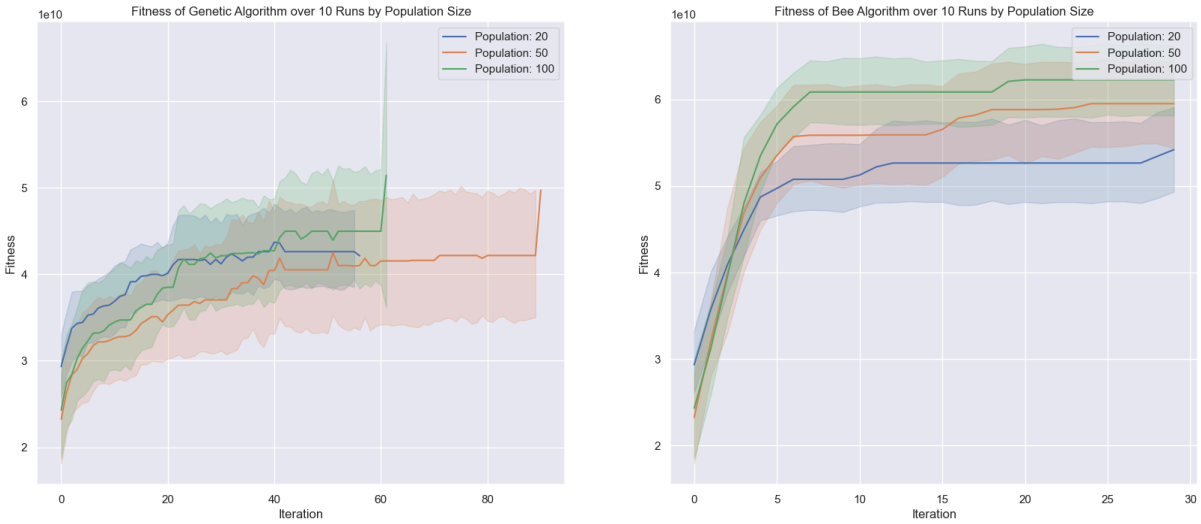
Wynik algorytmów został przetestowany na problemie opisanym poniżej. Każdy algorytm został uruchomiony dziesięciokrotnie na różnych rozmiarach populacji.

Atrybut	Wartość
number_of_room_types	5
capacity	2808
corridor_capacity	281
budget	75105702.3

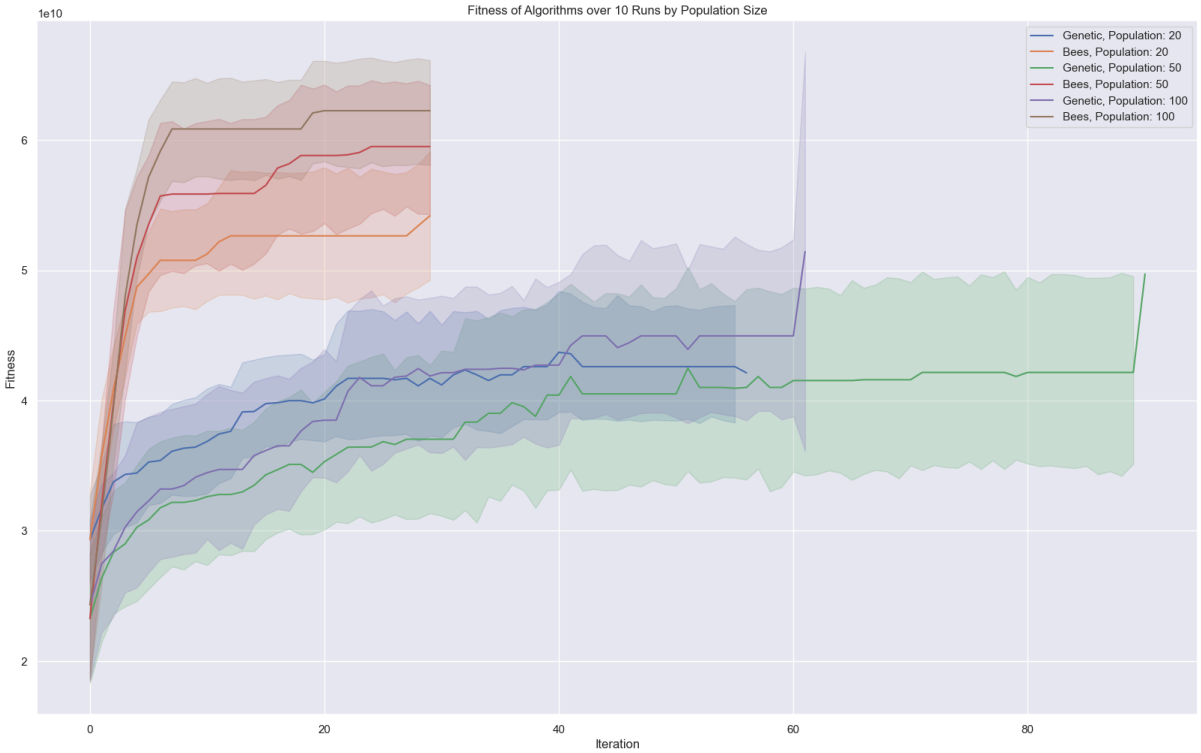
Tabela 1 - Wartości testowego problemu

id	size	frequency_of_use	cost_of_maintenance	cost_of_building	cost_per_day
0	58	20	11908.38	58754.89	1362.83
1	60	353	134094.40	115048.76	16146.82
2	92	129	162309.59	120133.41	24275.96
3	39	211	14658.98	187534.14	28380.27
4	56	361	128878.20	25394.70	23745.70

Tabela 2 - Typy pokoi dla testowego problemu



Rysunek 7.1 - Porównanie wartości zwracanych przez algorytmy dla różnych wartości wielkości populacji

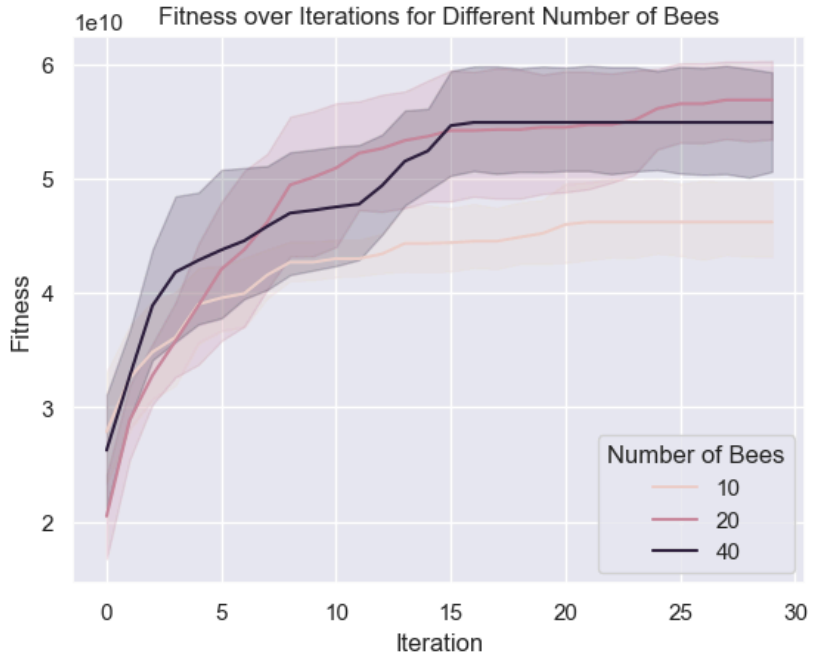


Rysunek 7.2 - Porównanie wartości zwracanych przez algorytmy dla różnych wartości wielkości populacji

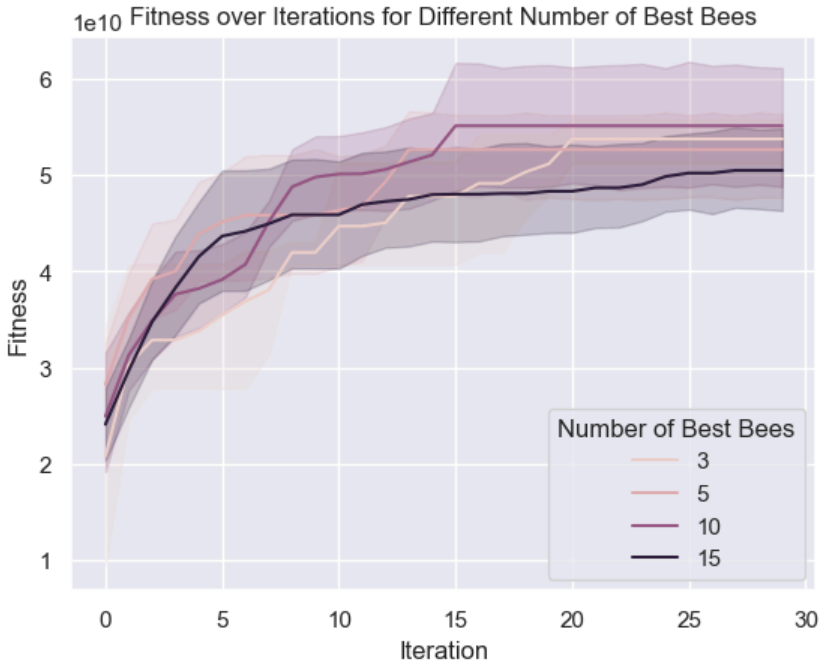
Możemy zauważyć, że algorytm pszczółkowy zdecydowanie daje lepsze wyniki. Algorytm genetyczny poprawia wyniki dopiero po większej liczbie iteracji niż algorytm pszczółkowy. Większy rozmiar populacji polepsza również wynik algorytmu.

Badanie wpływu parametrów na wyniki algorytmu

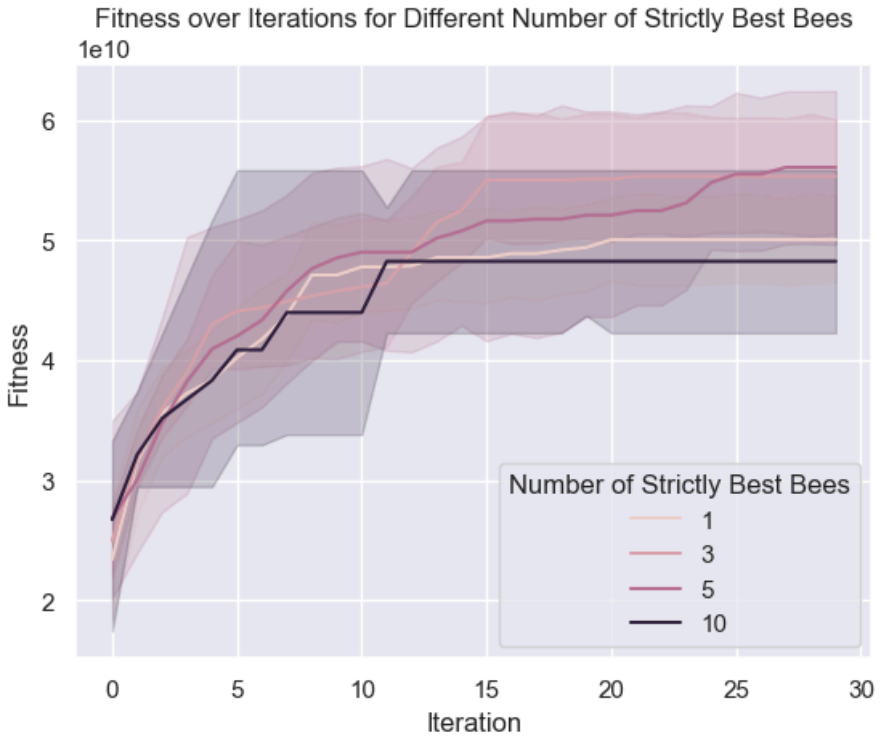
Głównym obiektem testowym był algorytm pszczółkowy - głównie przez łatwą modyfikację parametrów algorytmu. Algorytm genetyczny nie został przetestowany. Porównane zostały parametry **NUM_BEES**, **NUM_BEST_BEES**, **NUM_STRICTLY_BEST_BEES**.



Rysunek 8.1 - Porównanie wartości zwracanych przez algorytm pszczółkowy dla różnych wartości parametru **NUM_BEES**



Rysunek 8.2 - Porównanie wartości zwracanych przez algorytm pszczółkowy dla różnych wartości parametru **NUM_BEST_BEES**

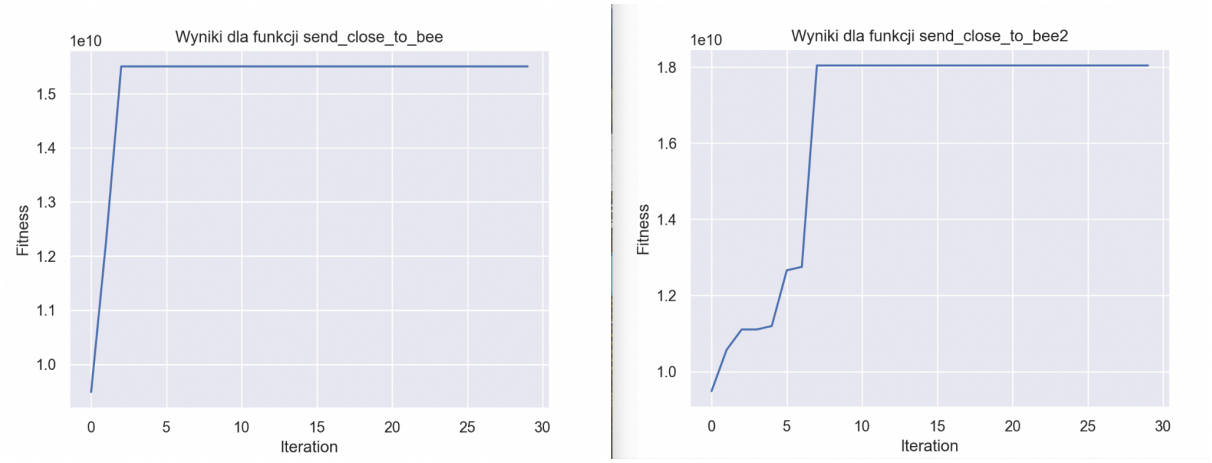


Rysunek 8.3 - Porównanie wartości zwracanych przez algorytm pszczółkowy dla różnych wartości parametru **NUM_STRICTLY_BEST_BEES**

W przypadku parametru **NUM_BEES** im więcej pszczół tym lepszy wynik - jest to spodziewane działanie algorytmu. W przypadku **NUM_BEST_BEES** oraz **NUM_STRICTLY_BEST_BEES** algorytm dawał lepsze wyniki dla mniejszych wartości.

Badanie wpływu funkcji generowania sąsiednich rozwiązań dla algorytmu pszczółkowego na wynik

Na danych z tabeli 1 oraz 2 przetestowaliśmy zdefiniowane przez nas metody generowania sąsiednich rozwiązań dla algorytmu pszczółkowego - *send_close_to_bee* oraz *send_close_to_bee2*:



Rysunek 9 - Porównanie funkcji generowania sąsiednich rozwiązań

Dla danych testowych, *send_close_to_bee2* dał nieco lepsze wyniki niż w przypadku podstawowej funkcji do generacji sąsiednich rozwiązań.

Podsumowanie

Po dogłębnych testach, jesteśmy w stanie stwierdzić, że algorytm pszczółkowy w większości przypadków zwraca lepsze wyniki. Algorytm genetyczny niestety nie radzi sobie tak dobrze jak pszczółkowy - wyniki dość często nie są poprawiane albo są minimalnie lepsze. Być może jest to spowodowane z metod krzyżowania czy mutacji - zaproponowanie innej metody prawdopodobnie polepszyłoby działanie algorytmu.

W trakcie pracy najwięcej problemów sprawił nam algorytm genetyczny - wyniki bardzo rzadko poprawiały się na lepsze. Początkowo sądziliśmy, że jest to wina metody generowania pierwszej populacji - natomiast postanowiliśmy lekko zmodyfikować metody odpowiedzialne za krzyżowanie oraz mutację i wyniki zaczęły się poprawiać, lecz nadal nie są one tak często poprawiane jak w przypadku algorytmu pszczółkowego.

Podsumowując projekt można uznać za udany - za pomocą obu algorytmów jesteśmy w stanie znaleźć lepsze rozwiązanie dla naszego zadanego problemu.

Literatura

- Bees algorithm, URL: https://en.wikipedia.org/wiki/Bees_algorithm (odwiedzony 03/06/2024)
- Genetic algorithms, URL: https://www.cs.ucc.ie/~dgb/courses/tai/notes/handout12.pdf (odwiedzony 03/06/2024)
- Genetic algorithms explained, URL: https://medium.com/@AnasBrital98/genetic-algorithm-explained-76dfbc5de85d (odwiedzony 03/06/2024)
- Pham, D. & Ghanbarzadeh, Afshin & Koç, Ebubekir & Otri, Sameh & Rahim, Sahra & Zaidi, Mb. (2005). The Bees Algorithm Technical Note. Manufacturing Engineering Centre, Cardiff University, UK. 1–57.
- Wykłady z przedmiotu Badania Operacyjne