Wakacyjny planer

Szymon Twardosz, Juliusz Wasieleski, Jakub Kasperski,
Michał Nożkiewicz, Przemysław Rola

Maj 2024

1 Opis problemu

Zadanie polega na wybraniu optymalnego terminu na wyjazd wakacyjny grupki znajomych. Niestety na wycieczkę może jechać ograniczona liczba osób w związku z dostępną liczbą miejsc w samochodzie. W związku z tym, każdy znajomy przyporządkowuje priorytety wszystkim dostępnym dniom. Oprócz tego dni posiadają również ceny. Są to kwoty, jakie należy wydać na atrakcje przeznaczone na dany dzień. Celem zadania jest znalezienie takiej grupy znajomych, aby byli oni jak najbardziej zadowoleni z wyjazdu przy jak najniższej cenie.

2 Model matematyczny

2.1 Dane

1. F - zbiór znajomych
2. Pmax - maksymalny priorytet, jaki użytkownik może przyporządkować pojedynczemu dniowi
3. H - zbiór wszystkich dni, wraz z przyporządkowanymi cenami
4. P - funkcja F x H -> N, która parze (użytkownik, dzień) przyporządkowuje priorytet
5. D - minimalna liczba dni na jakie znajomi jadą na wakacje (minimalna liczba dni pod rząd)
6. C - maksymalna liczba znajomych, którzy zmieszczą się do samochodu

2.2 Wyjście

Wyjściem problemu jest para liczb O1, O2, oznaczające odpowiednio start i koniec wycieczki oraz F_{res}, zbiór znajomych, którzy pojadą na wycieczkę. Warto zaznaczyć, że O2 - O1 > D (znajomi pojadą na więcej niż D dni) oraz $|F_{res}| <= |C|$

2.3 Funkcja kosztu

$$k(\pi) = -\sum_{f \in F} \sum_{d=O_1}^{O_2} P_{f,d} + (\sum_{d=O_1}^{O_2} H_d) * \alpha + (|C| - |F|) * 1000$$

Pierwszy składnik odpowiada za priorytety dni wybrane przez znajomych. Drugi bierze pod uwagę cenę wyjazdu. Dzięki temu, jeżeli dwa terminy wakacji posiadają taki sam priorytet, lepszy okaże się wyjazd tańszy. Ostatni składnik sumy penalizuję pustę miejsca w aucie. Oznacza to, że lepsze jest dla nas rozwiązanie, gdy jedzie więcej znajomych.

3 Opis algorytmów

3.1 Reprezentacja rozwiązania

W rozwiązaniu przyjaciele jadący na wakacje reprezentowani są jako lista odpowiadających im indeksów. Ponadto przechowywany jest także dzień początku oraz dzień końca wyjazdu. W każdym z algorytmów przyjęliśmy ten sam sposób kodowania rozwiązania.

3.2 Algorytm typu brute-force

3.2.1 Adaptacja

Brute-force jest prostym przechodzeniem po wszystkich możliwościach w postaci algorytmu dynamicznego. Tworzy po kolei każdą możliwą kombinację przyjaciół, zaczynając od C licznej, kończąc na 1. Dla każdej sumuje priorytety dla przyjaciół i dostosowuje do warunków polecenia, tj. dodaje do funkcji kosztu cenę wyjazdu oraz ewentualną karę za puste miejsca.

3.2.2 Wygenerowanie rozwiązania początowego

Początkowe rozwiązanie dla każdej iteracji to pierwsze n indeksów, gdzie n to ilość branych przyjaciół. Przy każdym pierwsza iteracje zaczyna się na pierwszym dniu i kończy się po minimalnym czasie wyjazdu.

3.2.3 Opis procedur

- $find\_friend\_interval\_priority(friend, interval)$ - funkcja obliczająca sumaryczny priorytet dla przyjaciela ‘friend‘ poprzez czas ‘interval‘
- $generate\_next\_idxs(friends\_mask)$ - jest to funkcja, która generuje kolejne kombinacje znajomych spośród wszystkich możliwych o zadanej ilości.

3.2.4 Pseudokod

Algorithm 1: Solve

```
priority_per_friend ← 3D array of size [len(friends)][D2 - D][D2]
  filled with find_friend_interval_priority(friend_id, (d1, d2);
friends_taken ← fmax;
empty_seat_penalty ← 0;
while friends_taken > 0 do
    friends_idx ← [0, 1, ..., len(friends_taken)-1];
    repeat
        for start ← D1 to D2 - D do
            for end ← start + D to D2 do
                cost ← sum(prices[start:end + 1]) * alpha;
                for idx in friends_idx do
                    cost ← cost - priority_per_friend[idx][start][end];
                end
                cost ← cost + empty_seat_penalty;
                if cost < best_cost then
                    best_result ← Result(start, end, friends_idx.copy());
                    best_cost ← cost;
                end
            end
        end
    until generate_next_idxs(friends_idx);
    friends_taken ← friends_taken - 1;
    empty_seat_penalty ← empty_seat_penalty + 1000;
end
return best_result, best_cost;
```

3.3 Algorytm genetyczny

3.3.1 Wygenerowanie rozwiązania początkowego

W początkowym rozwiązaniu lista indeksów przyjaciół jest losowym podzbiorem zbioru wszystkich indeksów. Długość tej listy jest równa liczbie C, czyli maksymalnej ilości znajomych, którzy mogą wsiąść do samochodu. Początkowy dzień jest losowany z rozkładu jednostajnego z przedziału $[H_{min}, H_{max} - D]$. Mając już wylosowany dzień początkowy, dzień końcowy jest losowany z rozkładu Half-Cauchy'ego, o parametrach $loc = starting_day + D$ oraz $scale = 5$. Zmienna *starting_day* to dzień początkowy a D to minimalny czas wyjazdu. W tym rozkładzie jedynie wartości powyżej wartości *loc* mają niezerową gęstość prawdopodobieństwa.

3.3.2 Opis procedur

Wariant I

- **Mutacja** - Istnieją dwa rodzaje mutacji: zbioru znajomych oraz początku i końca rozwiązania.

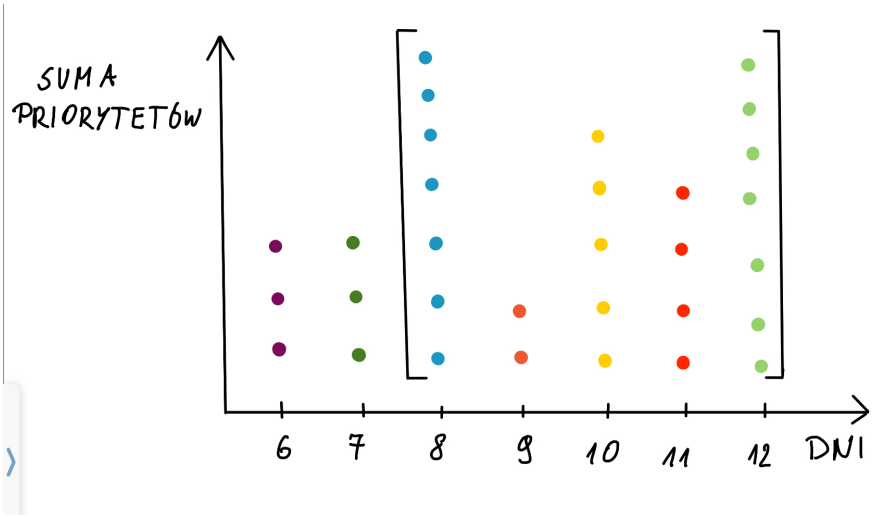
  Mutacja zbioru znajomych polega na wymianie jednego znajomego przez losowo wybraną osobę. Wskutek tej operacji liczba ludzi w aucie nie zmienia się. Dzięki temu rozwiązanie różni się od pierwotnego, ale nadal pozostaje poprawne.

  Mutacja początku i końca wyjazdu polega na przesunięciu jednej z dat wyjazdu o kilka dni. Liczba jednostek, o które przesuwane są daty, jak i kierunek, w którym są przesuwane (zmniejszenie / powiększenie przedziału) są wybierane losowo. Warto zaznaczyć, iż liczba dni wybierana jest w taki sposób, aby z rozwiązania poprawnego otrzymać ponownie rozwiązanie dopuszczalne.

- **Krzyżowanie** - Istnieją dwa rodzaje krzyżowań. Wybór jednego z nich dokonywany jest losowo przez algorytm. W szczególności są to krzyżowania: zbioru przyjaciół oraz granic przedziału wyjazdu.

  Krzyżowanie przyjaciół polega na "wymieszaniu"zbiorów przyjaciół z dwóch dopuszczalnych rozwiązań. Liczba osób w wynikowym rozwiązaniu może różnić się od tych w wejściowych rozwiązaniach. Jednak nadal pozostaje ona w przedziale $[1, C]$.

  Krzyżowanie granic przedziału to algorytm, który stara się dopasować jak najlepszy przedział zawierający się w rozwiązaniu I do priorytetów przyjaciół z rozwiązania II. Sumuję on priorytety osób na podanym przedziale i znajduję okno o największej sumie. W ten sposób dla podanej grupy przyjaciół pozwala on znaleźć przedział, który najbardziej im odpowiada.



Rysunek 1: Wybór najlepszego okna dla liczby dni = 5

- **Selekcja** - po przebiegu każdej generacji, z powstałej populacji wybierane są najlepsze próbki. Nowa populacja składa się z 90 % najlepszych próbek z poprzedniej generacji. Pozostałe 10 % to nowo powstałe (dolosowane) próbki.

Wariant II

- **Mutacja** - Istnieje algorytm mutacji zarówno dla listy przyjaciół jak i przedziału reprezentującego przedział.

  Mutacja listy znajomych - każdy znajomy znajdujący się w rozwiązaniu z prawdopodobieństwem p może zostać wymieniony na inną losową osobę, nieznajdująca się obecnie w liście znajomych. Liczba p jest zazwyczaj rzędu $0,1$.

  Mutacja listy znajomych - przedział jest przesuwany (tj. ta sama liczba jest dodawana do początku i do końca) o pewną wartość wylosowaną z rozkładu normalnego o zerowej wartości oczekiwanej. Odchylenie standardowe tego rozkładu miało zazwyczaj wartość rzędu 10.

- **Krzyżowanie** — tak samo jak przy mutacji istnieją dwa rodzaje krzyżowań zarówno dla listy indeksów jak i dla przedziałów.

  Krzyżowanie list znajomych polega na konkatenacji połówek z dwóch list. Nie ma znaczenia czy dany rodzic daje lewą, czy prawą połówkę dla

osobnika potomnego, gdyż w algorytmie para rozwiązań rodzicielskich zawsze daje dwóch potomków, więc rozważane są obie opcje. Ponadto może zdarzyć się tak, że w obu połówkach znajdzie się ta sama osoba. Nie jest to w żaden sposób rozwiązywane, w funkcji kosztu ta osoba i tak będzie policzona tylko raz i odpowiada to sytuacji, że w drodze na wakacje będzie jedno puste miejsce.

Krzyżowanie przedziałów początek nowego przedziału jest średnią arytmetyczną początków dwóch przedziałów rodzicielskich, a koniec średnią arytmetyczną ich końców.

- **Selekcja** — Początkowa populacja jest w całości losowana. W kolejnych iteracjach po obliczeniu funkcji kosztu dla każdego osobnika wybierane jest 25% procent najlepszych osobników rodzicielskich, które są potem między sobą krzyżowane. Te 25% osobników trafia do kolejnej populacji, 50% nowej populacji stanowią osobniki potomne powstałe wskutek krzyżowania losowych par osobników wybranych z populacji rodzicielskiej, a 25% nowych osobników jest dolosowywane.

3.3.3 Pseudokod

Algorithm 2: Genetic Algorithm

Input: Population size N, Mutation rate p_m, Crossover rate p_c, Generation Number
Output: Optimal solution

```
/* Initialize population                                    */
Initialize(population);
while GenerationNumber > 0 do
    /* Crossover                                            */
    population ← Crossover(parents, p_c);

    /* Mutation                                             */
    Mutation(population, p_m);

    /* Selection                                            */
    population ← Selection(population);
    GenerationNumber = GenerationNumber − 1;
return Best individual found in population;
```

3.4 Algorytm pszczeli

3.4.1 Wygenerowanie rozwiązania początkowego

Rozwiązanie początkowe jest generowane identycznie jak w przypadku algorytmów generycznych.

3.4.2 Opis procedur

- **Tworzenie losowych pszczół** - Każda pszczoła to losowe rozwiązanie. Z całej przestrzeni rozwiązań wybieranych jest losowych C znajomych, oraz losowy przedział.
- **Aktualizowanie pszczół zwiadowców** - Z całej populacji sprawdzane są rozwiązania funkcją kosztu. 3 najlepsze rozwiązania (pszczoły) zostają zwiadowcami.
- **Lot pszczół zwiadowców** - Każdy ze zwiadowców rekrutuje pszczoły do lotu. Jest to odpowiednio 40%, 20%, 20% całości populacji. Każda nowo zrekrutowana pszczoła bada najbliższe otoczenie danego zwiadowcy. Jeśli znalazła lepszy obszar, to zastępuje swojego zwiadowcę. Jeśli dany zwiadowca po x lotach nie znalazł lepszego rozwiązania, jego złoże uznaje się za wyczerpane, a sam zwiadowca leci zwiedzać nowy, losowy obszar.

  Znajdowanie otoczenia zwiadowcy - Każda pszczoła różni się od zwiadowcy, albo wybranymi osobami, albo przedziałem czasowym. Losowane jest to z prawdopodobieństwem p. Wybór jednej nowej osoby następuje przez usunięcie kogoś z samochodu i wylosowanie dowolnej innej osoby. Zmiana przedziału następuje poprzez wylosowanie przesunięcia oryginalnego przedziału z rozkładu normalnego.
- **Aktualizacja najlepszego rozwiązania** — Sprawdzane są wszystkie pszczoły z populacji. Jeśli któraś znalazła lepszy obszar niż aktualny najlepszy wynik, to jest on aktualizowany.

3.4.3 Pseudokod

Algorithm 3: Bee Algorithm

Input: Number of generations G, Population size N
Output: Optimal solution

$population$ = `CreateRandomBees(`N`)`;
$bestScore = +\infty$;
$GenerationNumber = 0$;
while $GenerationNumber <= G$ **do**
    $scouts \leftarrow$ `UpdateScouts(`$population$`)`;
    `ScoutsFly(`$scouts$`)`;
    $randomBees \leftarrow$ `CreateRandomBees(`$N \div 5$`)`;
    $population = scouts + randomBees$;
    $bestScore$ = `Min(UpdateScore(`$population$`),` $bestScore$`)`;
    $GenerationNumber = GenerationNumber + 1$;
return $bestScore$;

3.5 Algorytm symulowanego wyżarzania

3.5.1 Wygenerowanie rozwiązania początkowego

Początkowe rozwiązanie jest generowane tak samo jak w algorytmie genetycznym.

3.5.2 Opis procedur

- **Funkcja temperatury** - Temperatura określa, jak szybko w trakcie algorytmu spada temperatura, czyli czynnik wpływający na prawdopodobieństwo przechodzenia do stanów o gorszej wartości funkcji celu. W tym przypadku była to funkcja wykładnicza, a hiperparametrami były początkowa wartość temperatury (zazwyczaj około 100) oraz czynnik, przez który temperatura była mnożona w kolejnych iteracjach (rzędu $0.99 - 0.999$).
- **Funkcja akceptacji** - Jest to funkcja, która określa prawdopodobieństwo przejścia do kolejnego stanu. Prawdopodobieństwo wynosi 1 jeśli przejście powoduje poprawienie funkcji kosztu, w przeciwnym razie jest ono dane wzorem $e^{\frac{-\Delta f}{T}}$, gdzie Δf to różnica w funkcji kosztu w obu stanach, a T to temperatura w danej iteracji.
- **Generowanie sąsiadów** - w funkcji generującej sąsiadów zmianie mógł ulegać zbiór rozważanych przyjaciół albo sam przedział, który określa dni wyjazdu. Przy generowaniu pojedynczego sąsiada była rozważana tylko jedna opcja naraz, a każda mogła być wybrana z tym samym prawdopodobieństwem.

Generowanie sąsiednich przedziałów - przedział przesuwa się o jeden w lewo lub w prawo.

Generowanie list indeksów - losowy indeks z aktualnego rozwiązania jest podmieniany na inny. Podobnie jak w algorytmie genetycznym może zdarzyć się, że jakiś indeks się powtórzy, lecz jest to ignorowane.

3.5.3 Pseudokod

Algorithm 4: Simulated annealing

Input: Number of steps - N, Initial temperature - T, temperature scaling factor - α, Number of steps without changing the temperature - m

Output: Best found solution

```
/* Generate starting solution                                    */
currentState ← RandomSolution();
bestSolution ← currentState;

for i ← 1 to N do
    for j ← 1 to m do
        /* Generate neigbhor                                     */
        neighbor ← GenerateNeighbor(currentState);

        /* Decide wheter to move                                 */
        if Evaluate(neighbor) < Evaluate(currentState) then
            currentState ← neighbor;
        else
            Δf ← Evaluate(neighbor) - Evaluate(currentState);
            if exp Δf/T > random(0, 1) then
                currentState ← neighbor;

        /* Update Solution                                       */
        if Evaluate(currentState) < Evaluate(bestSolution) then
            bestSolution ← currentState;

    /* Update temperature                                        */
    T ← T * α

return bestSolution;
```

4 Opis aplikacji

Danymi wejściowymi do każdego algorytmu była instancja klasy Data. Posiadała ona pola:

- D1 - dzień startowy,
- D2 - dzień końcowy ,
- alpha - ważność ceny w funkcji kosztu, im alfa wyższa tym ceny ważniejsze,
- max_per_interval - maksymalny priorytet możliwy dla dnia,
- max_seats - maksymalna liczba osób, która może jechać,
- min_days - minimalna liczba dni wyjazdu
- number_of_people - liczba wszystkich osób,
- prices - lista cen wyjazdów dla wszystkich dni,
- F - słownik z priorytetami, gdzie każdej osobie jest przypisana lista priorytetów na cały przedział [D1, D2]

Do reprezentacji rozwiązania użyliśmy klasy Result, która miała pola:

- D1 - dzień startowy wyjazdu
- D2 - dzień końcowy wyjazdu
- friends - lista osób, które jadą w dniach D1 - D2

4.1 Korzystanie z aplikacji

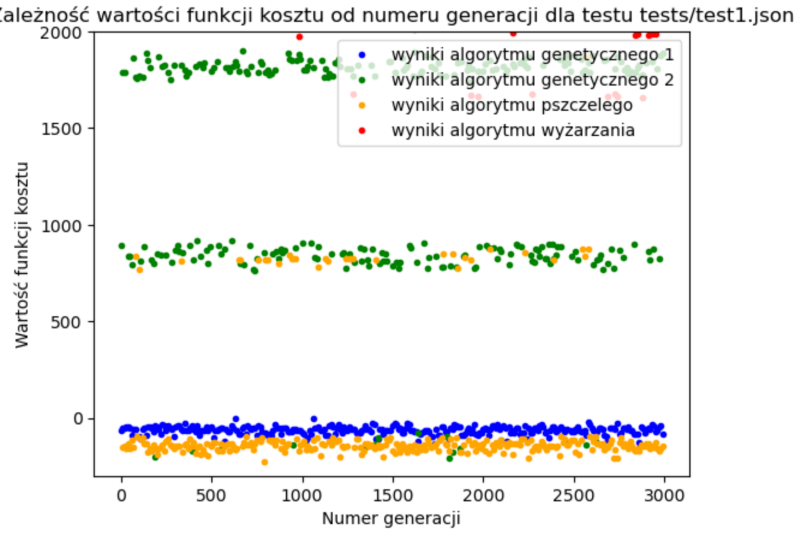
Aby uruchomić aplikację, należy przygotować swój plik wejściowy, którego format podobny jest do klasy Data oraz opisany jest w Readme.md. Następnie wystarczy w folderze głównym aplikacji wywołać komendę:

```
python3 main.py
```

Następnie należy postępować zgodnie z informacjami wypisywanymi na konsolę. Jest wtedy możliwość wielokrotnego wybrania pliku wejściowego, oraz algorytmu który rozwiąże problem zadany plikiem wejściowym. Dodatkowo, użytkownik zawsze może podać hiperparametry algorytmów, takie jak liczbę generacji czy rozmiar populacji.

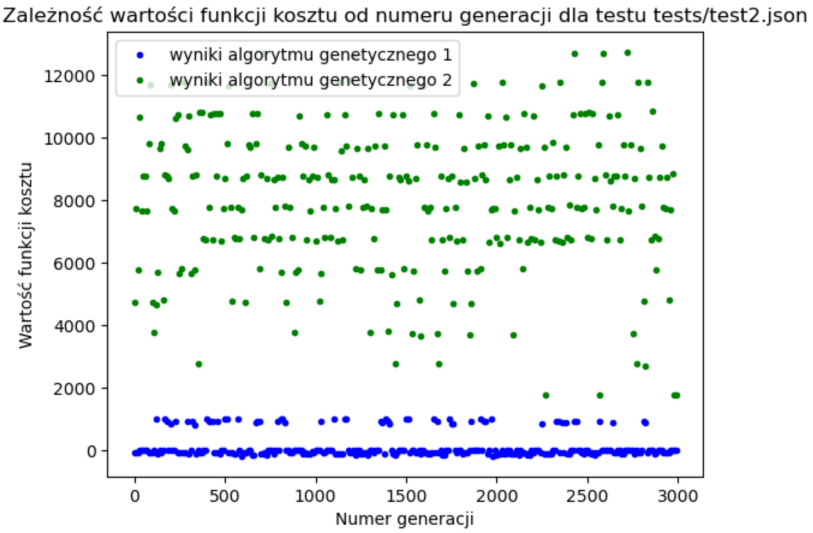
5 Eksperymenty

W celu porównania działania, wszystkie algorytmy zostały uruchomione na kilku testach, a wartości funkcji kosztu w zależności od numeru generacji przedstawione na wykresach.



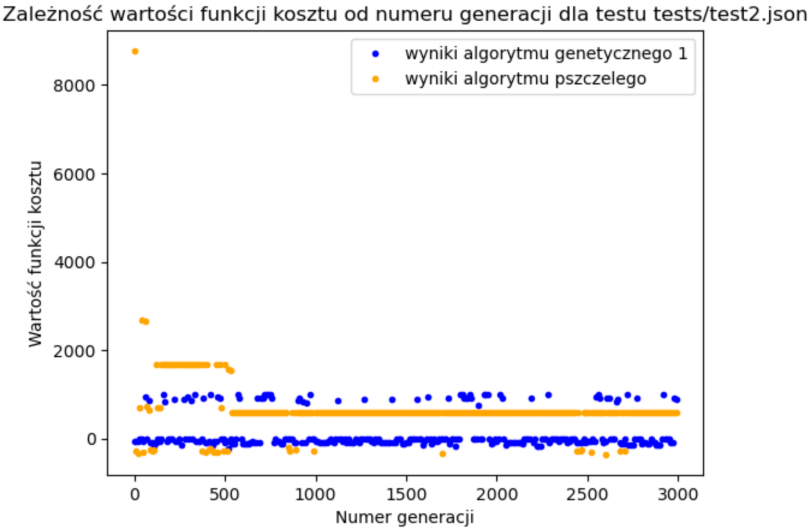
Wykr. 1 Wykres zależności wartości funkcji kosztu od numeru generacji dla testu 1

Na wykresie nr 1 jasno widać, że porównywalnie dobre wyniki (wartość funkcji kosztu około -300) dają trzy algorytmy: dwa genetyczne oraz pszczeli. Algorytm wyżarzania daje zdecydowanie gorsze wyniki, więc w dalszych eksperymentach został pominięty.



Wykr. 2 Wykres porównujący oga warianty algorymu genetycznego

Na wykresie nr 2 widać, że algorytm genetyczny nr 2 ma dużo większą skłonność do mutacji niż algorytm w wersji pierwszej. Warto też zauważyć że algorytm w wersji pierwszej już od początku ma dość dobre rozwiązanie, tzn. rozwiązanie początkowe ma dość małą funkcję kosztu i potem już niezbyt się zmienia. W porównaniu do niego, algorytm numer dwa utrzymuje się dookoła dużo większych wartości funkcji kosztu. Oba algorytmy nie wykazują zbyt dużej zbieżności, tzn. wartości funkcji kosztu się nie zmniejszają a jedynie oscylują dookoła pewnych wartości.



Wykr. 3 Wykres porównujący pierwszy wariant algorytmu genetycznego z

algorytmem pszczelim

Na wykresie nr 3 można zauważyć pewną charakterystykę algorytmu pszczelego, mianowicie po znalezieniu minimum lokalnego ciężko jest je opuścić. Z tego względu dla większości generacji funkcja kosztu nie była bliska najlepszym wartościom oraz nie zbliżała się do nich. Jednakże najmniejsze wartości funkcji kosztu osiągał jednak algorytm pszczeli, a nie genetyczny.

Nazwa algorytmu	liczba generacji	rozmiar populacji	minimalna wartość funkcji kosztu	numer iteracji	maksymalna wartość funkcji kosztu	numer iteracji
Algorytm genetyczny 1	500	100	-139	437	1000	441
Algorytm genetyczny 1	1000	100	-232	711	1923	930
Algorytm genetyczny 1	2000	100	-316	1898	1000	213
Algorytm genetyczny 2	500	100	1685	197	14762	278
Algorytm genetyczny 2	1000	100	760	59	15754	317
Algorytm genetyczny 2	2000	100	753	971	14767	1165
Algorytm pszczeli	500	100	-398	50	4801	0
Algorytm pszczeli	1000	100	-401	777	5813	0
Algorytm pszczeli	2000	100	-398	325	4779	0

Tabela 1: Tabela przedstawiająca maksymalne i minimalne wartości funkcji kosztu dla różnyh laczby generacji i różnych algorytmów

Analizując tabelę nr 1 można dostrzec, że dla algorytmu genetycznego nr 1 zwiększanie liczby generacji poprawia najlepsze wartości funkcji kosztu, natomiast najgorsze wartości funkcji kosztu, niezależnie od liczby iteracji znajdują się poniżej 1000 generacji. Dla algorytmu nr 2, poprawę można zaobserwować jedynie podczas przejścia z 500 na 1000 generacji, potem zmiana jest już niewidoczna. Algorytm pszczeli bardzo szybko znajduje dość dobre rozwiązanie i

następnie w kolejnych generacjach niezbyt oddala się od tego minimum lokalnego, ponieważ najgorsze wyniki były osiągane w pierwszej generacji, a najlepsze były możliwe już około 50 generacji.

Nazwa algorytmu	liczba generacji	rozmiar populacji	minimalna wartość funkcji kosztu	numer iteracji	maksymalna wartość funkcji kosztu	numer iteracji
Algorytm genetyczny 1	2000	60	-207	58	2882	12
Algorytm genetyczny 1	2000	100	-232	711	1923	930
Algorytm genetyczny 1	2000	200	-226	1220	1000	633
Algorytm genetyczny 2	2000	60	737	791	16769	996
Algorytm genetyczny 2	2000	100	760	59	15754	317
Algorytm genetyczny 2	2000	200	669	386	12637	952
Algorytm pszczeli	2000	60	-361	318	12721	0
Algorytm pszczeli	2000	100	-401	777	5813	0
Algorytm pszczeli	2000	200	-382	492	7778	0

Tabela 2: Tabela przedstawiająca maksymalne i minimalne wartości funkcji kosztu dla różnej liczby generacji i różnych algorytmów

Na podstawie tabeli nr 2 można wysnuć wniosek, że dla relatywnie dużych liczb generacji, rozmiar populacji nie ma znaczenia. Jednakże, dla algorytmu genetycznego nr 1, im większa populacja tym najlepsza wartość funkcji kosztu będzie znaleziona w późniejszej iteracji. Ta zależność nie jest jednak widoczna ani dla drugiego algorytmu genetycznego, ani dla algorytmu pszczelego. Wszystkie algorytmy charakteryzują się tym, że im większa populacja tym mniejsza jest maksymalna wartość funkcji kosztu.

6 Podsumowanie

6.1 Ogólne

- W eksperymentach zdecydowanie najgorzej wypadał algorytm wyżarzania.
- Najlepsze wyniki uzyskiwane były przy użyciu algorytmu genetycznego w pierwszym wariancie i algorytmu pszczelego.
- Oba te algorytmy uzyskiwały dość dobre (w porównaniu do pozostalych algorytmów) wartości funkcji kosztu przy niewielkiej liczbie generacji
- Wariant pierwszy algorytmu generycznego wykazał się pewną zbieżnością, ponieważ po zwiększeniu liczby iteracji dawał widocznie lepsze wyniki
- Dla algorytmów: drugiego genetycznego oraz pszczelego nie zaobserwowano efektu zbieżności.
- Zwiększanie rozmiaru populacji powodowało zmniejszenie najgorszych wartości funkcji kosztu dla wszystkich algorytmów, natomiast wpływ na najlepsze wartości miało jedynie dla pierwszego wariantu algorytmu genetycznego.
- Dla pierwszego wariantu algorytmu genetycznego wraz ze zwiększaniem rozmiaru populacji, najlepszy wynik osiągany był w późniejszej generacji.
- drugi wariant algorytmu genetycznego miał zdecydowanie większą podatność na mutacje niż pierwszy i dawał gorsze wyniki.

6.2 Brute-force

- Złożoność obliczeniowa jest wykładnicza względem ilości osób, tak więc możliwości sprawdzenia optymalnego rozwiązania ograniczają się do małych przykładów.
- ciężko o heurystykę, optymalizację — algorytm musi przejść wszystkie możliwości.

6.3 Algorytm Genetyczny

- Algorytm krzyżowania — problem z wymyśleniem "poprawnego"algorytmu krzyżowania dwóch próbek. Z tego powodu przetestowane zostały dwa warianty tej funkcji (opisane powyżej)
- Algorytm mutacji — ze względu na specyfikę problemu, był łatwy do opracowania i wdrożenia. Znalezienie sposobu na generowanie rozwiązań "bliskich"do zadanego okazało się bardzo proste.

- Szybkość znajdowania optymalnego rozwiązania — funkcja kosztu dla obu rozważanych algorytmów szybko maleje w początkowej fazie poszukiwań. W miarę postępu programu poprawa wyników staje się coraz rzadsza.

- Zmiana kolejności mutacji i krzyżowania nie wpływa na algorytm w sposób znaczący. Wyniki zwracane przez algorytmy są bardzo podobne.

6.4 Algorytm pszczeli

- Znajdowanie otoczenia pszczół zwiadowców - jest to kluczowy element algorytmu, a nigdy nie wiadomo, czy nie istnieje lepszy sposób znajdowania otoczenia. Pomysły były różne. Zostawiliśmy algorytm, który najlepiej i najszybciej znajdował dobre rozwiązanie w naszych testach

- Powtarzalność pszczół — przy dużej ilości pszczół podczas scoutowej rekrutacji najprawdopodobniej wiele pszczół poleci w dokładnie to samo miejsce. Lot jednej pszczoły nie jest obliczeniowo kosztowny, dlatego uznaliśmy, że stosowanie mechanizmu wymuszającego różne miejsca lotu dla każdej pszczoły, byłyby bardziej kosztowne i spowolniły by znalezienie opzwiadowców — istnieje rozwiązania.

- Ilość pszczół zwiadowców - istnieje wiele możliwości i naprawdę każda ilość pszczół zwiadowców ma prawo działać. Zostaliśmy przy 3, ponieważ łatwo było dobrać jaką część populacji zabiera każdy zwiadowca na ekspedycje i ponieważ taka ilość dobrze radziła sobie w testach. Najprawdopodobniej istnieje lepsza liczba pszczół scoutów (lub optymalna wartość jest zależna od testu).

- Szybkość znajdowania optymalnego rozwiązania — funkcja kosztu przeważnie bardzo szybko spada i wpada w jakieś mocne minimum lokalne. Po jakimś czasie przeważnie udaje się z niego wyjść.

6.5 Algorytm wyżarzania

- Symulowane wyżarzanie praktycznie na wszystkich testach działało najgorzej. Dużo zależało od początkowego wylosowanego rozwiązania, szybko zbiegało do pewnego lokalnego minimum, z którego bardzo trudny już było się potem wydostać, szczególnie po kilku iteracjach, gdy temperatura była niska. Zmiana hiperparametrów na różne sposoby też nie zdawała się działać.

- Pomysłem na poprawę mogłaby być próba startowania od kilku rozwiązań początkowych i próba podążania tymi ścieżkami, które dają najlepsze wyniki. Jednak wtedy algorytm byłby ideowo bardzo podobny do algorytmu pszczółek i stwierdziliśmy, że na dłuższą metę stosowanie algorytmów z samotnego wędrowca nie ma w tym problemie sensu, gdyż przestrzeń rozwiązań jest zbyt nieregularna i posiada bardzo dużo minimów lokalnych, w

szczególności takich, które wartością znacząco odbiegają od sąsiadujących stanów.

7 Literatura

1. Python 3.11

8 Podział pracy

- Jakub Kasperski
  - Stworzenie generatora testów - 70 %
  - Przygotowanie testów – 40 %
  - Implementacja algorytmu pszczelego – 100 %
- Michał Nożkiewicz
  - Stworzenie generatora testów - 30 %
  - Przygotowanie algorytmu z wyżarzaniem - 100 %
  - Przygotowanie algorytmów genetycznych - 50 %
  - Przygotowanie testów - 20 %
  - Dokumentacja - 20 %
- Przemysław Rola
  - Przygotowanie algorytmu brute-force - 100 %
  - Dokumentacja.- 20%
- Szymon Twardosz
  - Przygotowanie modelu matematycznego - 70 %
  - Przygotowanie algorytmów genetycznych - 50 %
  - Przygotowanie testów - 60 %
  - Dokumentacja - 20 %
- Juliusz Wasieleski
  - Przygotowanie aplikacji - 100%
  - Przeprowadzenie eksperymentów, - 100%
  - Dokumentacja.- 20%