Optymalizacja kosztów podróży z wykorzystaniem algorytmu symulowanego wyżarzania

Automatyka i robotyka
2022/2023

Dawid Woźniak
Dominik Tomalczyk

Spis treści

Model zagadnienia

Opis słowny

Problem jaki będziemy rozważać to wybór stacji i ilości paliwa tankowanego na nich aby uzyskać jak najmniejszy koszt podróży. Problem optymalizujemy jedynie pod względem ceny, a nie czasu. Staramy się jednak przyjąć stacje oddalone o racjonalne odległości. Poza tym stosujemy kilka uproszczeń:

- Odległość trasa - stacja i stacja - trasa są równe
- Cały czas jednakowe spalanie

Podczas rozwiązania zakładamy, że mamy już wyznaczoną trasę, stacje obok niej oraz ceny paliw. Poza stacjami jest nam potrzebne również średnie spalanie samochodu, stan początkowy paliwa i pojemność baku. Model po nałożeniu odpowiednich warunków spełnia rzeczywiste realia takie jak:

- Nie tankujemy więcej niż pełny bak
- Dobieramy tak stacje aby do nich dojechać
- Staramy się tankować dopiero gdy mamy mniej niż połowę baku
- Uwzględniamy w kosztach cenę dojazdu do stacji i powrotu na trasę
- Uwzględniamy różnicę w paliwie początkowym i końcowym

Chcemy uniknąć zbyt częstego zjeżdżania z trasy na tankowanie (wtedy gdy mamy ponad połowę baku), więc gdy taka sytuacja występuje wprowadzamy funkcję kary która w oparciu o to ile paliwa mamy zwiększa nam funkcję celu.

Model uwzględnia dowolność jeśli chodzi o maksymalne dopuszczalne zjechanie z trasy do stacji.

Model matematyczny

W celu zdefiniowania modelu matematycznego wprowadzamy następujące zmienne, z których będziemy korzystać:

P - Pojemność baku samochodu.
P_{start} - Początkowa ilość paliwa w baku.
P_{end} - Końcowa ilość paliwa w baku
P_{curr} - Aktualna ilość paliwa w baku.
C_i - Cena paliwa na i-tej stacji.
K_i - Ilość paliwa zatankowanego i-tej stacji.
G(i) - Wartość funkcji kary dla i-tej stacji.
$\widehat{x}$ - Średnia ważona cen paliw z wszystkich wybranych stacji

Funkcja celu przyjmuje wtedy postać:

$$F(x) = \sum_{i=0}^{n} C_i * K_i + G(i) - \ \widehat{x} * (P_{end} - P_{start}) \rightarrow min$$

Wyrażenie $\widehat{x} * (P_{end} - P_{start})$ odpowiada za uwzględnienie różnic w ilości paliwa na starcie i na końcu.

Funkcję kary definiujemy następująco:

$$G(i) = \begin{cases} \frac{P}{2} - \frac{K_i}{2} & \text{dla } K_i < \frac{P}{2} \\ 0 & \text{dla } K_i \geq \frac{P}{2} \end{cases}$$

Karamy w ten sposób tankowanie małej ilości paliwa, które powoduje częstsze postoje na trasie.

Algorytm

Algorytm inspirowany zjawiskami metalurgicznymi który może być wykorzystywany w optymalizacji. Cechą charakterystyczną tej metody jest występowanie parametru sterującego, który maleje w trakcie wykonywania

algorytmu. Im wyższą wartość ma ten parametr, tym bardziej chaotyczne mogą być zmiany.

Ogólna zasada działania:

1. Losowy wybór punktu startowego ω. Przyjęcie parametru sterowania T = Tmax
2. Wyznaczenie wartości funkcji F(ω) w punkcie ω
3. Wyznaczenie ω' = ω + Δω
4. Wyznaczenie wartości funkcji F(ω') w nowym punkcie
5. Podstawienie wartości ω' do ω z prawdopodobieństwem danym rozkładem Boltzmanna b(E(ω') - E(w), T)
6. Zmniejszenie parametru sterowania T = αT, gdzie α jest najczęściej z przedziału (0.8, 0.99)
7. Spełnienie kryterium stopu lub powrót do kroku 3

Pseudokod

While not stop criterion **do**
    For number of new solution
        Select a new solution: ω + Δω
            If F(ω + Δω) < F(ω) then
                F_{new} = F(ω + Δω), ω = ω + Δω
            Else
                ΔF = F(ω + Δω) - F(ω)
                random r(0,1)
                If r > exp(-ΔF/T) **then**
                    Fnew = F(ω + Δω), ω = ω + Δω
                Else
                    Fnew = F(ω)
                End if
            End if

        F = Fnew
        T = αT
    End for
End while

Opis elementów opracowanych

Pierwszy kluczowy element w algorytmie to rozwiązanie inicjujące. W naszym przypadku rozwiązanie to a dokładniej stacje dobierane są w pełni losowo, a ilość paliwa jaką tankujemy na każdej stacji to taka aby mieć pełen bak. Algorytm jest opatrzony w odpowiednie walidatory które zapewniają nam spójność rozwiązania. Mamy pewność pomimo losowego wybierania, że na pewno dojedziemy do każdej stacji i że stację znajdują się w maksymalnym dopuszczalnym dystansie do zjechania.

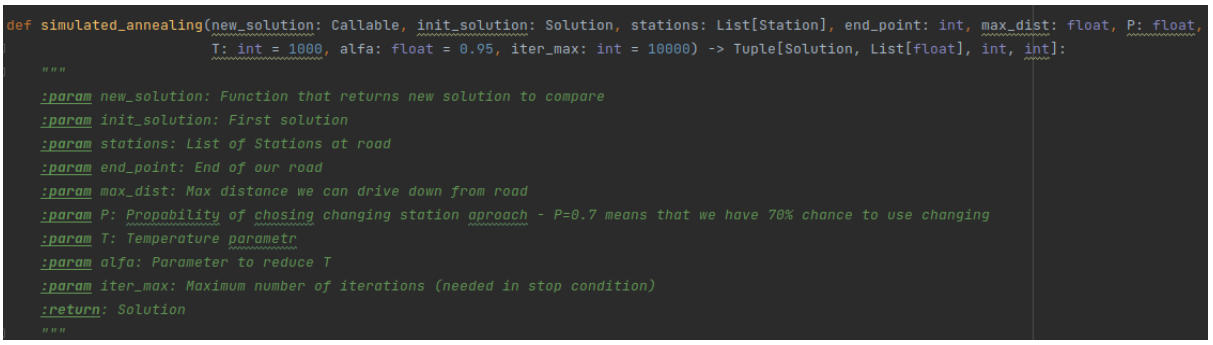
Z punktu widzenia algorytmu symulowanego wyżarzania kluczowe jest zdefiniowanie sąsiedztwa rozwiązań oznaczanego w pseudokodzie jako $\omega + \Delta\omega$. W naszym przypadku problem ten rozwiązaliśmy, wykorzystując dwa sąsiedztwa i możliwość ‘połączenia’ ich. Zaimplementowaliśmy to za pomocą specjalnego parametru który mówi o prawdopodobieństwie przyjęcia określonego sąsiedztwa. Nasze sąsiedztwa to:

- Zmiana ilości tankowanego paliwa na stacjach, zaimplementowane jest tutaj również to aby dobierać ilości paliwa tak aby nigdy na trasie nam go nie brakło
- Zmiana stacji i zmiana ilości tankowanego paliwa, tutaj również algorytm jest bezpieczny, dobieramy taką stację aby na pewno była ona w zasięgu i na trasie nigdy nie zabrakło nam paliwa

Rozwiązanie sąsiednie przyjmujemy w każdej iteracji rozwiązania.
Ustawiając odpowiednią wartość parametru P [0, 1] ustawiamy prawdopodobieństwo tego, z jaką szansą zmieniamy stację.
W przypadku P = 1, mamy pewność że zostanie zamieniona stacja
W przypadku P = 0, zamieniamy jedynie ilość tankowanego paliwa
W przypadku np P = 0.7, mamy 70% szans na zmianę stacji (zawsze zamieniamy ilość paliwa).

Parametry Algorytmu

Główna funkcja która wywołuje cały algorytm przyjmuje kilka parametrów.



```
def simulated_annealing(new_solution: Callable, init_solution: Solution, stations: List[Station], end_point: int, max_dist: float, P: float,
                        T: int = 1000, alfa: float = 0.95, iter_max: int = 10000) -> Tuple[Solution, List[float], int, int]:
    """
    :param new_solution: Function that returns new solution to compare
    :param init_solution: First solution
    :param stations: List of Stations at road
    :param end_point: End of our road
    :param max_dist: Max distance we can drive down from road
    :param P: Propability of chosing changing station aproach - P=0.7 means that we have 70% chance to use changing
    :param T: Temperature parametr
    :param alfa: Parameter to reduce T
    :param iter_max: Maximum number of iterations (needed in stop condition)
    :return: Solution
    """
```

Rys 1. Funkcja główna wraz z parametrami.

I tak kolejne parametry wraz z ich znaczeniem:

- New_solution - Funkcja która ma zwrócić nowe rozwiązanie (ze sąsiedztwa)
- Init_solution - Rozwiązanie inicjujące, w pełni losowe
- Stations - Wektor instancji stacji, instancje zawierają informacje o kilometrze trasy na którym jest stacja, drodze do stacji oraz cenie
- End_point - Oznacza ostatni punkt trasy (jej długość)
- Max_dist - Parametr który oznacza jak daleko chcemy zjeżdżać z trasy do stacji aby zatankować
- P - Prawdopodobieństwo z jakim poza zmianą ilości paliwa na stacjach zamieniamy też stację w sąsiednim rozwiązaniu
- T - Zwany także parametrem sterującym, jest to odpowiednik temperatury w literaturze, jest zmniejszany podczas trwania algorytmu, ma wpływ na to z jakim prawdopodobieństwem przyjmujemy gorsze rozwiązanie. Wraz ze spadkiem T, maluje wykładnik w eksponencie a co za tym idzie mamy mniejszą szansę na przyjęcie gorszego rozwiązania
- Alfa - Współczynnik przez który mnożymy T, aby je zmniejszać w każdej iteracji, zalecane jest stosowanie alfy z przedziału [0.8, 0.99]
- Iter_max - maksymalna ilość dopuszczalnych iteracji w algorytmie

Aplikacja

Implementacja całego projektu została wykonana w języku Python, wykorzystując jedne z najbardziej popularnych Pythonowych bibliotek których opis znajduję się poniżej.
Nasza aplikacja to prosta konsolowa aplikacja która wymaga od użytkownika wskazania ścieżki do pliku zawierającego wszystkie dane. W rezultacie

otrzymujemy listę stacji (w odpowiedniej kolejności) na jakie musimy zjechać, cenę paliwa i ilość jaką tankujemy. Cały projekt jednak był dedykowany do używania w narzędziu pythona jupyter-notebook. Uznaliśmy to za dużo wygodniejsze narzędzie, ponieważ umożliwia segmentowe uruchamianie kodu po załadowaniu już części danych. Możemy mieć wpływ i dowolność co do parametrów, wielkości rozmiaru. Z doświadczenia praca z danymi jest wygodniejsza w jupyterze niż w zwykłych skryptach pythona, stąd też nasza decyzja.

Podczas pracy z danymi często zachodzi potrzeba zwizualizowania osiągniętych efektów przez co zdefiniowaliśmy kilka funkcji mających za zadanie wizualizację wyników funkcji. Funkcje te umieściliśmy w pliku visualization.py. Jupyter-notebook jest dostosowany do pracy z wykresami znacznie bardziej niż standardowe IDE.

Wymagania odnośnie uruchomienia

Do użytkowania aplikacji niezbędne jest środowisko Pythona wraz z odpowiednimi bibliotekami. Dokładny wykaz bibliotek i ich wersji znajduje się w pliku requirements.txt, umożliwia to nam zainstalowanie niezbędnych pakietów za pomocą jednej instrukcji.

Główne biblioteki jakie wykorzystaliśmy:

- Matplotlib - najpopularniejsza pythonowa biblioteka do rysowania wykresów, zastosowaliśmy ją w części wizualizacji.
- Jupyter-Notebook - Pakiet umożliwiający min. Segmentowe uruchamianie kodu, zalecany w pracy z danymi.

Zalecanym systemem operacyjnym jest Linux, projekt został stworzony na wersji Ubuntu (22.0.0).

Format danych / wyników

Format danych jaki aktualnie obsługuje aplikacja to pliki tekstowe z rozszerzeniem .txt. Plik taki musi zawierać wszystkie informacje niezbędne do wykonywania obliczeń.

Wymagane informacje o samochodzie:

- Średnie spalanie
- Pojemność baku
- Początkowy stan paliwa

Długość trasy, oraz listę stacji, gdzie każda stacja musi mieć informację o:

- Nazwie (identyfikator stacji)
- Cenie paliwa
- Drodze jaką musimy zjechać z trasy
- Kilometr na trasie

Przykładowy plik .txt umożliwiający rozpoczęcie obliczeń:

```
50 10 0 20

300

A 2 10 50
B 3 30 60
C 4 15 70
```

Rys 2. Przykładowy plik .txt z danymi.

W pliku ważna jest kolejność linijek i danych i tak kolejno:
Linijka 1 - informacje o samochodzie w kolejności
<pojemność baku> <średnie spalanie> <pozycja początkowa> <stan paliwa>

Linijka 2 i 4 - puste

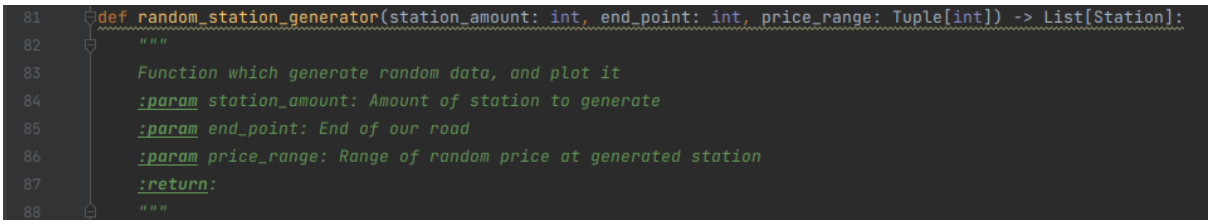
Linijka 3 - informacje o trasie
<długość trasy>

Linijki od 5 w dół - informacje o stacjach
<identyfikator> <cena paliwa> <dystans do zjechania> <pozycja na trasie>

Przed wczytaniem danych do programu plik tekstowy jest walidowany przez odpowiednia funkcję która sprawdza poprawność danych. W razie znalezienia niespójności informuje użytkownika o linijce w której pojawił się błąd.

Funkcjonalność

Program oferuje dużą dowolność co do algorytmu, wizualizacji jak i danych. Rozpoczynając od początku możemy wprowadzić własny zestaw danych w pliku .txt który ponadto będzie odpowiednio zwalidowany. Możemy również skorzystać z funkcji generowania losowych stacji:



Rys 3. Funkcja do generowania losowych stacji.

Funkcja wprowadza dowolność co do zakresu cen paliwa i ilości stacji. Musimy jedynie podać długość trasy (end_point).
Kolejna funkcjonalność to możliwość dopasowania wszystkich parametrów opisanych w rozdziale *Parametry Algorytmu* zgodnie z preferencjami użytkownika.
Użytkownik może również rozpocząć działanie algorytmu od rozwiązania początkowego zdefiniowanego przez siebie, może to być np. Rozwiązanie optymalne w celu obserwacji zachowania funkcji celu.
Jako że projekt jest dedykowany do pracy w narzędziu jupyter-notebook, posiada również kilka funkcji umożliwiających wyświetlanie wykresów. I tak możemy wyrysowywać wykresy:

- Przebiegu wartości funkcji celu
- Rozmieszczenie losowo wygenerowanych stacji
- Podgląd rozwiązania, na którym kilometrze musimy zjechać do stacji i jak daleko

Dodatkowo w klasach zostały odpowiednio nadpisane metody magiczne *__str__ i __repr__* umożliwiające czytelny podgląd obiektów w dowolnym momencie.

Klasy reprezentujące obiekty samochodu i rozwiązania zostały rozszerzone o metody umożliwiające następujące wartości:

Klasa Car:

- get_fuel_level_at_step - umożliwia podgląd paliwa w określonym kroku rozwiązania.

Klasa Solution:

- get_solution - umożliwia podgląd wektora rozwiązania
- get_stations - zwrócenie jedynie stacji które są w rozwiązaniu, w odpowiedniej kolejności
- get_station - zwrócenie stacji na odpowiedniej pozycji
- get_station_position - zwrócenie pozycji na trasie danej stacji
- get_cost_of_solution - koszt paliwa w rozwiązaniu, nie uwzględniając funkcji kary
- get_penalty - wartość funkcji kary
- solution_value - Wartość funkcji celu wraz z funkcją kary

Zostały również zaimplementowane specjalne wyjątki czytelne dla użytkownika, rzucane w momencie gdy program napotka problem.

To czego użytkownik nie widzi, to walidatory w które został wzbogacony projekt, mające zapewnić spójność rozwiązania i sprawdzanie poprawności plików.

Testy

Wykaz scenariuszy

Testowanie naszej pracy wprowadziliśmy już na etapie implementacji. Wykorzystaliśmy testy jednostkowe aby mieć pewność, że wprowadzane nowe funkcjonalności działają w sposób zamierzony oraz są kompatybilne z dotychczasową pracą. W tym celu wykorzystywaliśmy wbudowaną bibliotekę Pythona - unittest. Każdą funkcję (która jest deterministyczna) opatrzyliśmy odpowiednimi testami, podobnie z każdą klasą i jej metodami. Przykładowy test jednostkowy w projekcie:

```python
class TestAny_station_too_far(unittest.TestCase):
    def test_station_too_far(self):
        c = ds.Car(0, 0, 0, 0)
        s1 = ds.Station("A", 10, 30, 0)
        s2 = ds.Station("B", 20, 40, 0)
        solution = ds.Solution(c)

        solution.add_station((s1, 10))
        solution.add_station((s2, 20))
        self.assertTrue(any_station_too_far(25, solution))
```

Rys 4. Przykładowy test jednostkowy

Łącznie napisane zostało 39 testów które pomagały nam sprawdzać na bieżąco czy kod jest spójny a funkcje zwracają zamierzone wyniki.
Wynik uruchomienia testów jednostkowych:

```
(venv) dominik@dt-linux:~/Dokumenty/studia/BO2/Simulated-annealing$ python -m unittest
.....................................
----------------------------------------------------------------------
Ran 39 tests in 0.003s

OK
```

Rys 5. Wynik testów jednostkowych

Po zakończeniu prac implementacyjnych testowaliśmy nasz algorytm pod kątem następujących aspektów:

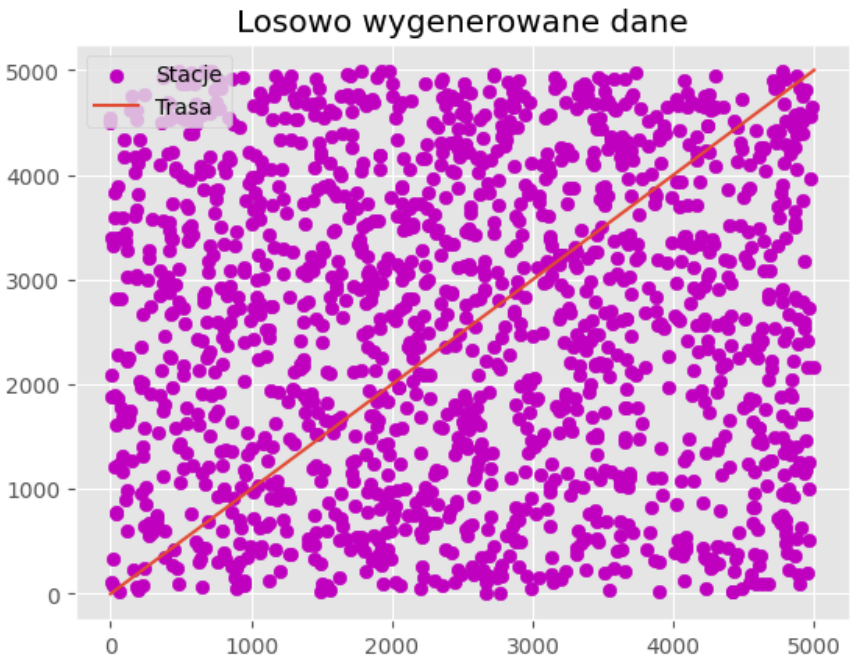
- Dobór najlepszej wartości parametru P (prawdopodobieństwo zmiany stacji w rozwiązaniu sąsiednim)
- Wpływ parametrów T oraz α na działanie algorytmu
- Użycie otrzymanego wyniku jako rozwiązania początkowego w kolejnym wywołaniu algorytmu.
- Czas wykonania w zależności od wielkości problemu
- Ilość rozwiązań początkowych

Opis metodyki badań

Testowanie działania algorytmu oraz własności naszego rozwiązania wykonaliśmy w jupyter notebook. Zdefiniowaliśmy dla naszego algorytmu problem średniej wielkości (trasa o długości 5000km) i dla niego badaliśmy własności rozwiązania. Dane testowe w postaci stacji, zapewniła stworzona przez nas funkcja random_station_generator. Zakres cen na stacjach wyniósł 7-10 PLN. Jako samochód, którym będziemy się poruszać przyjęliśmy popularny van Volskwagen Transporter T4. Parametry charakterystyczne dla samochodu i niezbędne do możliwości wykonania algorytmu czyli maksymalna pojemność baku oraz średnie spalanie na 100km zaczerpnęliśmy z noty technicznej samochodu.

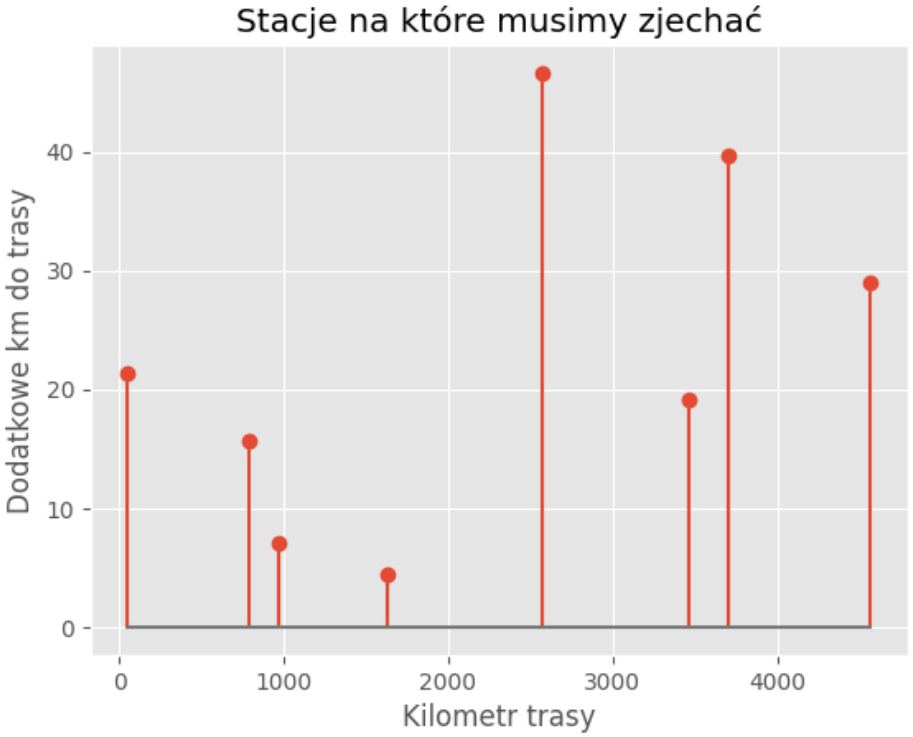
Zdefiniowane zadania testowe

Zainicjowaliśmy obiekt klasy Car, zgodnie z wartościami z noty technicznej. Początkowa ilość paliwa w baku wynosiła 30 litrów. Maksymalna odległość na jaką mogliśmy zboczyć z trasy to 50km. Wygenerowaliśmy losowo stacje wzdłuż naszej trasy.



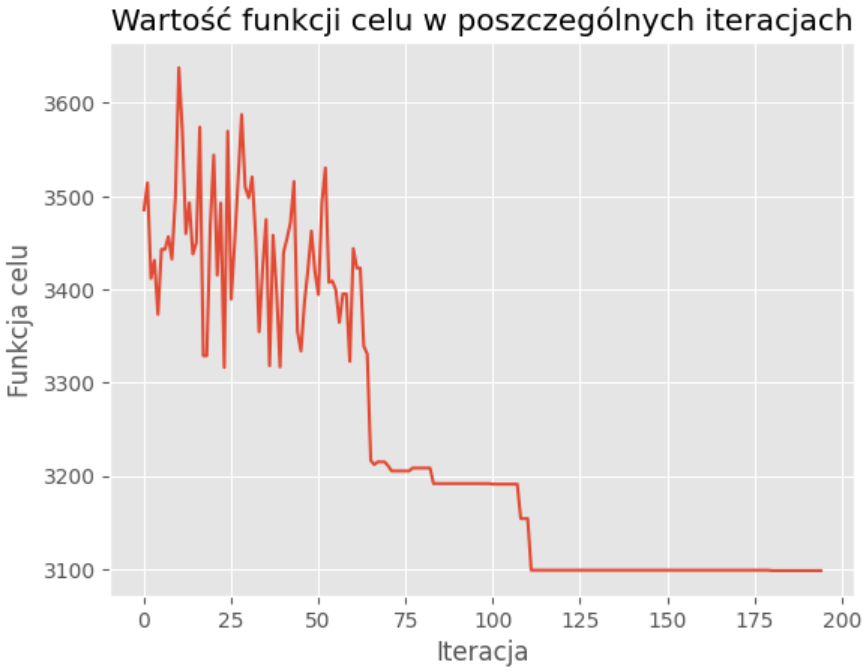
Wyk 1. Losowo wygenerowane dane

W wyznaczonym rozwiązaniu początkowym funkcja celu miała wartość 3547.38.



Wyk 2. Wizualizacja wybranych stacji w rozwiązaniu początkowym

Uruchomiliśmy algorytm z następującymi parametrami: T = 2000, α = 0.95, P = 0.7, iter_max = 10000. Przebieg funkcji celu w trakcie działania algorytmu wyglądał następująco:



Wyk 3. Wizualizacja przebiegu funkcji celu.

Analizując przebieg funkcji celu możemy stwierdzić że nasz algorytm działa poprawnie. Początkowo mamy duże wahania funkcji celu co wynika z dużej wartości parametru T na tym etapie przebiegu algorytmu. Większa wartość parametru T oznacza większe prawdopodobieństwo przyjęcia rozwiązania gorszego. Wraz z wzrostem iteracji wartość parametru T zmniejsza się poprzez mechanizm chłodzenia, a przebieg funkcji celu staje się mniej chaotyczny. Dodatkowe informacje, które możemy odczytać z przebiegu algorytmu oraz otrzymanego rozwiązania:

```
Najlepsze otrzymane rozwiązanie: 3098.59
Początkowe rozwiązanie: 3485.38
Liczba wykonanych iteracji: 194
Liczba zamian na gorsze rozwiązanie: 34
```

```
Wartość funkcji celu: 3098.59
Koszt zatankowanego paliwa: 2794.41
Wartość funkcji kary: 95.87
Bilans start/koniec paliwa -27.28
```

Rys 4. Szczegółowe informacje na temat przebiegu algorytmu oraz własności rozwiązania.

Następnie badaliśmy zachowanie w zależności od wartości parametru P. Oto wyniki.

Parametr P	0	0.1	0.3	0.5	0.7	0.9	1
Funkcja celu	3257.38	3206.27	3234.81	3170.56	3084.19	3123	3110.35

Tab 1. Porównanie wartości funkcji celu dla różnych parametrów P.

Najlepszy wynik został osiągnięty dla P = 0.7. Warte zauważenia jest to, że tym razem dla parametru P = 0.7 otrzymaliśmy inny wynik niż poprzednio. Wynika to z faktu iż algorytm symulacyjnego wyżarzania jak i dobór naszego zdefiniowanego sąsiedztwa, nie są deterministyczne.

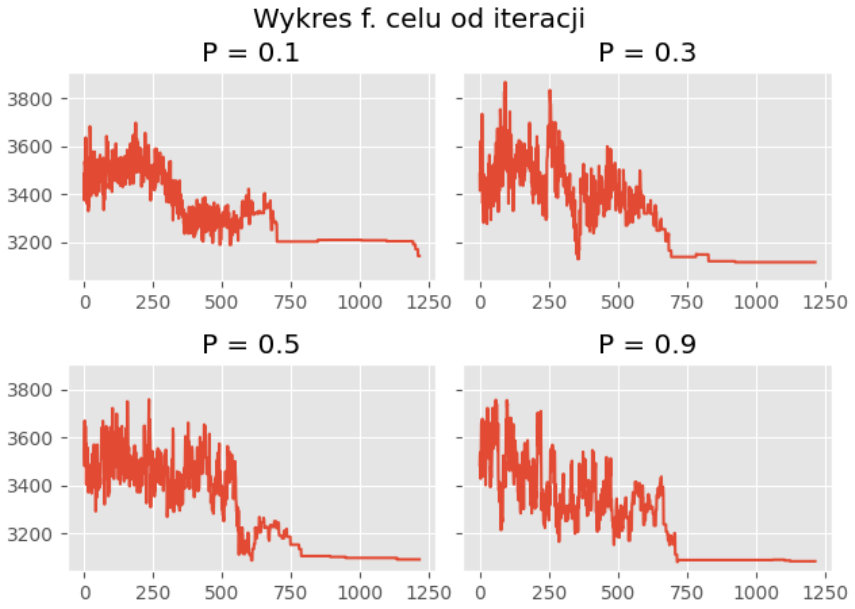
Kolejnym etapem było ustawienie wartości T = 20000 oraz α = 0.99. Parametr P pozostał równy 0.7. Spowodowało to wzrost liczby iteracji do wartości 1215.

Wyk 4. Wizualizacja przebiegu funkcji celu

Uzyskana wartość funkcji celu wyniosła 3129.78. Dla tych parametrów również przetestowaliśmy zachowanie się algorytmu w zależności od parametru P.

Parametr P	0	0.1	0.3	0.5	0.7	0.9	1
Funkcja celu	3232.41	3147.53	3085.52	3133.56	3096.23	3070.96	3072.17

Tab 2. Porównanie wartości funkcji celu dla różnych parametrów P



Wyk 5. Przebieg funkcji celu w zależności od parametru P.

Tym razem najlepszą wartość funkcji celu uzyskaliśmy dla P = 0.9. W porównaniu z wartościami otrzymanymi dla poprzednich parametrów T i α tylko w jednym przypadku otrzymaliśmy gorszy wynik (dla wartości P, która zapewniła najlepszy wynik w poprzednim teście). Można stwierdzić, że zwiększenie parametrów T i α oraz równoczesne z tym zwiększenie liczby iteracji prowadzi do uzyskania lepszych wyników. Oczywiście zwiększanie tych parametrów w nieskończoność nie jest najlepszym pomysłem. W zależności od własności problemu, optymalna wartość tych parametrów może być inna. Co do parametru P, to w tym przypadku również nie możemy jednoznacznie stwierdzić jaka jego wartość będzie dawała najlepsze wyniki w poszczególnych problemach. Najlepszym sposobem będzie dobór tej wartości empirycznie.

Kolejnym aspektem, było sprawdzenie jak zachowa się algorytm gdy jako rozwiązanie początkowe użyjemy najlepsze rozwiązanie zwrócone podczas wcześniejszego uruchomienia algorytmu. Użyliśmy takich parametrów jak powyżej, lecz ograniczyliśmy liczbę iteracji do 1000. Otrzymaliśmy następujące wyniki:

Rozwiązanie otrzymane z losowego rozwiązania początkowego: wartość funkcji celu - 2891.44.
Rozwiązanie otrzymane z powyższego wyniku użytego jako rozwiązanie początkowe: wartość funkcji celu - 2884.76

Udało nam się znaleźć lepsze rozwiązanie. Nie możemy jednak być pewni, że zawsze tak będzie. Zagwarantowane mamy to, że nie otrzymamy rozwiązania gorszego. Jeżeli poprzednie rozwiązanie zostało wyznaczone w dużej ilości iteracji to program może nie dać rady go poprawić.

Kolejną rzeczą, którą testowaliśmy była złożoność czasowa naszego algorytmu. Badaliśmy ją dla 3 rzędów wielkości problemu:

- Mały - trasa o długości 1000 km
- Średni - trasa o długości 5000 km
- Duży - trasa o długości 10000 km

Wyniki można obserwować w poniższej tabeli.

	Wielkość problemu		
Parametr P	Mały	Średni	Duży
0	203 [ms]	502 [ms]	1170 [ms]
0.5	313 [ms]	856 [ms]	2070 [ms]
1	478 [ms]	1180 [ms]	2950 [ms]

Tab 3. Czas wykonania algorytmu dla różnych wielkości problemu i parametru P

Pomiary zostały przeprowadzone za pomocą biblioteki timeit. Wartości w komórkach to średni czas wykonania algorytmu z 5 obiegów testowych, gdzie w każdym z obiegów algorytm był uruchamiany dziesięciokrotnie.
Z racji, że wyliczenie sąsiedztw ma różne złożoności obliczeniowe, uwzględniliśmy to w naszych testach. Jak łatwo zauważyć większa wartość parametru P prowadzi do wydłużenia działania algorytmu. Ogółem czas wykonania algorytmu jest akceptowalny. Planując długą trasę jesteśmy w stanie poświęcić kilkadziesiąt sekund by dowiedzieć się jak zaoszczędzić pieniądze na tankowaniu. Kilkadziesiąt sekund dotyczy oczywiście dużego problemu wraz dobraniem odpowiednich wartości parametrów naszego algorytmu.

Ostatni testowany wariant to dobór ilości rozwiązań początkowych.
Dla każdego rozwiązania początkowego uruchomiliśmy główny algorytm w celu szukania rozwiązania. Testy były prowadzone dla następujących ilości rozwiązań początkowych: 1,3,5,10,15,20,25,30,40,50
Dla każdego rozwiązania początkowego algorytm symulowanego wyżarzania wykonywał ok 1000 iteracji po czym zwracał wynik. Dla każdej ilości rozwiązań początkowych zapisywaliśmy rozwiązania i spośród ich wybieraliśmy najlepsze (najmniejsze). Oto wykres który przedstawia najmniejsze wartości funkcji celu dla danej liczby rozwiązań początkowych:

Jak łatwo zauważyć wraz ze wzrostem początkowych rozwiązań wartość funkcji celu maleje. Jest to jednak bardzo czasochłonne zadanie ponieważ dla każdego rozwiązania początkowego uruchamiany jest algorytm. Jednak po wykresie można wnioskować że warto szukać rozwiązania po przynajmniej kilkunastu rozwiązaniach początkowych, zwiększając tym znacznie szanse na znalezienie lepszego rozwiązania.

Reasumując:

- Wartość parametru P ma wpływ na rozwiązanie. Nie ma uniwersalnej wartości optymalnej dla każdego problemu. Musimy ją dobrać w sposób empiryczny. Domyślnie proponujemy P = 0.7
- Zwiększenie parametrów T oraz α prowadzi do zwiększenia ilości iteracji oraz ogólnej poprawy jakości rozwiązania. Podobnie jak powyżej optymalne wartości tych parametrów nie są znane, musimy je wyznaczyć eksperymentalnie. Domyślnie proponujemy T = 20000 oraz α = 0.99
- Używając jako rozwiązania początkowego, rozwiązania otrzymanego w poprzednim wywołaniu algorytmu jesteśmy w stanie poprawić jakość naszego wyniku lecz nie mamy na to gwarancji.
- Złożoność czasowa algorytmu jest akceptowalna. Ze względu na ten aspekt byłby on użyteczny w zastosowaniu rzeczywistym.

- Zwiększenie ilości rozwiązań początkowych prowadzi do poprawy rozwiązania lecz zwiększa również czas poszukiwań.

Podsumowanie

Wnioski

Cały projekt okazał się dość ciekawym zadaniem, od samego początku wymagał on sporo planowania i połączenia odpowiednich elementów w jedną całość. Mogliśmy popracować z chociaż mocno przybliżonym realnym problemem. Próbowaliśmy zdefiniować jak najlepszy model matematyczny a następnie przenieść całą logikę do kodu. W trakcie pisania kodu, napotkaliśmy się na wiele problemów które wymagały dyskusji. Po wielokrotnych konsultacjach udało nam się ukończyć projekt i oddać działający program. Sam proces tworzenia oprogramowania był bardzo pouczający, doceniliśmy to jak ważna jest przysłowiowa ‘burza mózgów’ podczas projektowania aplikacji. Przećwiczyliśmy również podział obowiązków i dbałość o szczegóły. Kluczowe z perspektywy czasu okazało się również pisanie czytelnych dobrze nazwanych i komentowanych funkcji dzięki czemu nie mieliśmy problemów z powrotem do pracy czy zrozumieniem pracy partnera. Zrozumieliśmy też jak prawdziwe duże aplikacje których używamy powszednie mają rozbudowane strukturę i o ile szczegółów trzeba zadbać.
Co do samego zadania, to otrzymaliśmy oczekiwane rezultaty, przebieg funkcji celu dobrze obrazuje jak zmieniają się wartości w zależności od parametru T. Algorytm przechowuje najlepsze rozwiązanie i szuka innego, przez co mamy pewność że zwracamy najlepsze spośród wszystkich znalezionych rozwiązań.
Ponadto dzięki projektowi wzmocniliśmy nasze umiejętności programowania w Pythonie. Poszerzyliśmy umiejętności wykorzystywania narzędzia jupyter-notebook o tworzenie prezentacji. Zadbaliśmy o czytelność kodu i odpowiednie komentarze. Podczas całej pracy korzystaliśmy z narzędzi takich jak GIT czy Github co znacznie ułatwiało pracę.

Stwierdzone problemy

Problemów co do zagadnienia pojawiło się kilka, niektóre z nich są do rozwiązania niektóre jednak wynikają ze specyfikacji algorytmu

- Przede wszystkim co najważniejsze nie mamy pewności że znaleźliśmy rozwiązanie optymalne, wynika to wprost z własności algorytmu. Mamy za to pewność że zwrócone rozwiązanie przez funkcję główną jest najlepszym ze wszystkich jakie znaleźliśmy do tej pory
- Użycie programu ma jedynie sens dla dalekich tras gdzie będziemy kilkukrotnie tankować i przewidujemy możliwość na odbicie z trasy
- Optymalizujemy jedynie cenę, w praktyce może okazać się że nadrobimy dużo czasu aby zjechać do jakiejś stacji aby zaoszczędzić niewielką kwotę
- W obecnym stanie aplikacji użytkownik musi ręcznie wprowadzić wszystkie stacje ręcznie, aby mieć duży wybór rozwiązania należy wprowadzić duża liczbę stacji co za tym idzie jest to dość czasochłonne rozwiązanie

Kierunki dalszego rozwoju

Pomysłów do dalszego rozwoju podczas tworzenia oprogramowania pojawiło się sporo. Z tych najciekawszych są to między innymi:

- Wprowadzenie mapy, aby pogląd na trasę był lepszy oraz dokładnie oznaczone stacje
- Obliczanie dokładnej trasy droga-stacja i stacja-droga, a nie przyjmowanie takiej samej wartości.
- Można by się również pokusić o automatyzację całego procesu, z pewnością potrzebne byłoby wykorzystanie jakiegoś API w celu scrapowania danych odnośnie stacji, jej pozycji i ceny
- Sam model można by rozszerzyć o kilka dokładniejszych wartości, np. Obliczanie średniego spalania w zależności od załadowania samochodu, warunków pogodowych czy chociażby korków ulicznych
- Stworzenie dedykowanej aplikacji np. webowej by ułatwić korzystanie z naszego rozwiązania.

Kod źródłowy oraz cała struktura projektu znajdują się na platformie Github: https://github.com/13Dominik/Simulated-annealing