

sprawozdanie

December 15, 2019

1 SPRAWOZDANIE 1

1.1 Projekt wykonali:

- Katarzyna Jędrocha,
- Szymon Bednarek,
- Michał Baran,
- Bartłomiej Kalata.

1.2 Projekt został napisany w języku Python.

1.3 Algorytm wspinaczki

1.3.1 Wstęp

Algorytm wspinaczkowy polega na wybraniu losowego X dla którego jest przypisana wartość Y . W pierwszej kolejności sprawdzane jest najbliższe sąsiedztwo X . Jeżeli wartość rozwiązania jest lepsza niż wartość rozwiązania początkowego wtedy sąsiadujący X jest brany jako punkt początkowy następnej iteracji algorytmu. W ten sposób możemy osiągnąć ekstremum lokalne funkcji minimalne lub maksymalne w zależności od założeń. Problem tego algorytmu polega na tym że znajdując się w wierzchołku funkcji kończymy działanie algorytmu. Jednakże nie zawsze osiągnięty wierzchołek jest ekstremum globalnym, tylko lokalnym. Aby zniwelować ten błąd należy uruchomić algorytm losując X , tyle razy, aby prawdopodobieństwo że otrzymany wynik jest ekstremum lokalnym było jak najmniejsze.

1.3.2 Opis kodu

W pierwszej kolejności zdefiniowana została zmienna 'KR' która to jest wskaźnikiem jakości ułożenia zadań. Wartość 'KR' jest optymalizowana. Wskaźnik ten ukazuje jak dużo pozostaje "wolnego" czasu między zadaniami. Oznacza to że im mniejsza wartość 'KR' tym lepsze jest rozwiązanie problemu.

Do przeszukiwania listy sąsiadów dla wylosowanego X , została stworzona funkcja `get_neighbours()`. Jej argumentami są :

idx - Jest to indeks dla którego szukamy sąsiadów.

neighbours_count - liczba sąsiadów którą chcemy wyszukać

df - wprowadzone dane

Funkcja działa w następujący sposób:

Dla zadanych wartości argumentów algorytm ustawia się w pozycji startowej:

| Numer indeksu | ... | n-4 | n-3 | n-2 | n-1 | n | n+1 | n+2 | n+3 | n+4 | ... |
|---------------|-----|-----|-----|-----|-----|---|-----|-----|-----|-----|-----|
| Sąsiedztwo | - | - | 0 | 0 | 0 | X | 0 | 0 | 0 | - | - |

W powyższym przykładzie możemy zauważyć że dla zadanych:

- `idx = n`;
- `neighbours_count = 6`;

Algorytm ustawił się w pozycji `n` i wyszukał 6 sąsiadów.

Dla przypadków końca danych z lewej lub z prawej strony zostały stworzone wyjątki. Wprowadzone zostały dodatkowe zmienne:

- `L` - Jest to długość danych minus wylosowana pozycja
- `idx_prev = idx - 1` - Jest to indeks dla którego szukamy sąsiadów
- `N_half` - Połowa wartości szukanych sąsiadów

Zobrazowanie sytuacji w której występuje wyjątek:

| Numer indeksu | 0 | 1 | n=2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | ... |
|---------------|---|---|-----|---|---|---|---|---|---|---|-----|
| Sąsiedztwo | 0 | 0 | X | 0 | 0 | 0 | 0 | - | - | - | - |

W powyższym przykładzie możemy zauważyć że dla zadanych:

- `idx = n = 2` gdzie `n < neighbours_count/2`;
- `neighbours_count = 6`;

W takim przypadku możemy znaleźć tylko dwa obiekty z lewej strony, dlatego w tym wyjątku dodajemy dodatkowy obiekt z prawej strony. Sytuacja jest analogiczna dla obserwacji na obu końcach zbioru danych. Do rozwiązania tych wyjątków wykorzystany został następujący kod:

```
if idx < N_half:
    idx = N_half

elif idx > L - N_half:
    idx = L - N_half
output = list(range(idx - N_half, idx + N_half + 1))
```

Jako ostateczny wynik funkcja zwraca listę sąsiadów, która zostanie przeszukana w celu uzyskania lepszego wyniku niż startowy `X`.

Ostateczna funkcja algorytmu to **`hill_climbing_algorithm()`**. Jednak do jej działania użyte zostały poszczególne pomniejsze funkcje:

- `calculate_improvements()`
- `search_for_the_best_neighbour()`

- `swap()`
 - `move()`
 - `plot_optimalization()`
1. Funkcja **`calculate_improvements()`** jest odpowiedzialna za sprawdzenie jak zmieni się wartość **KR** w przypadku gdy zamienimy miejscami **idx** z jakimś sąsiadem.
 2. Funkcja **`search_for_the_best_neighbour()`** jest odpowiedzialna za wyszukanie najlepszego sąsiada i przypisanie do zmiennej `idx_min` indeksu najmniejszego elementu.
 3. Funkcja **`swap()`** jest odpowiedzialna za zamianę miejscami dwóch wierszy “in place” - dzięki temu nie tworzymy nowej ramki danych. tylko pracujemy na tym samym zbiorze danych co optymalizuje szybkość wykonania algorytmu.
 4. Funkcja **`move()`** jest odpowiedzialna za zmianę startowego X od którego zaczniemy kolejną iterację algorytmu.
 5. Funkcja **`plot_optimalization()`** jest odpowiedzialna za wykonanie wykresu służącego do sprawdzenia czy algorytm poprawnie optymalizuje szukaną wartość.

1.3.3 Algorytm

Do wykonania algorytmu wspinaczkowego dla wylosowanego X, została stworzona funkcja `hill_climbing_algorithm()`. Jej argumentami są:

- `number_of_iterations` - ustalona liczba iteracji wykonania algorytmu
- `break_counter` - zmienna która zatrzymuje działanie algorytmu, tzn. gdy przez np. 10 iteracji wynik się nie poprawia algorytm zostaje zastopowany.
- `neighbours_count` - liczba sąsiadów którą chcemy wyszukać

Wewnątrz funkcji zdefiniowane są także parametry umożliwiające poprawne działanie algorytmu:

- `idx` - w pierwszej iteracji jest wybierany losowo jako jeden z indeksów zbioru danych i stanowi punkt startowy naszego algorytmu.
- `improvements` - zmienna ustalona jako pusty szereg numeryczny (wypełniony wartościami 0).
- `optimalization` - przypisywana jest nazwa kolumny w zbiorze danych, którą posiłkuje się do wykonania wykresu.

Główne ciało funkcji to pętla **`while`** której ograniczeniami są **`number_of_iterations`** oraz **`break_counter`**. Na starcie pobierana jest początkowa wartość kombinacji, oraz pierwsza lista sąsiadów dla X. Następnie sprawdzane są przypadki co będzie gdy zamienimy **idx** z danym sąsiadem. Kolejnym krokiem jest wybór tego sąsiada który wykazał się najlepszą możliwą optymalizacją rozwiązania w danej iteracji. Jeżeli została zdefiniowana nowa wartość dla **`best_neighbour`** to wtedy ustawiamy ją jako aktualny punkt początkowy algorytmu. W przeciwnym wypadku od **`break_counter`** odejmowana jest wartość 1. W celu sprawdzenia poprawności algorytmu tworzymy listę aktualnych wartości ‘KR’ z każdej kolejnej iteracji. Zostanie ona wykorzystana do ukazania zależności efektywności algorytmu od ilości wykonanych iteracji.

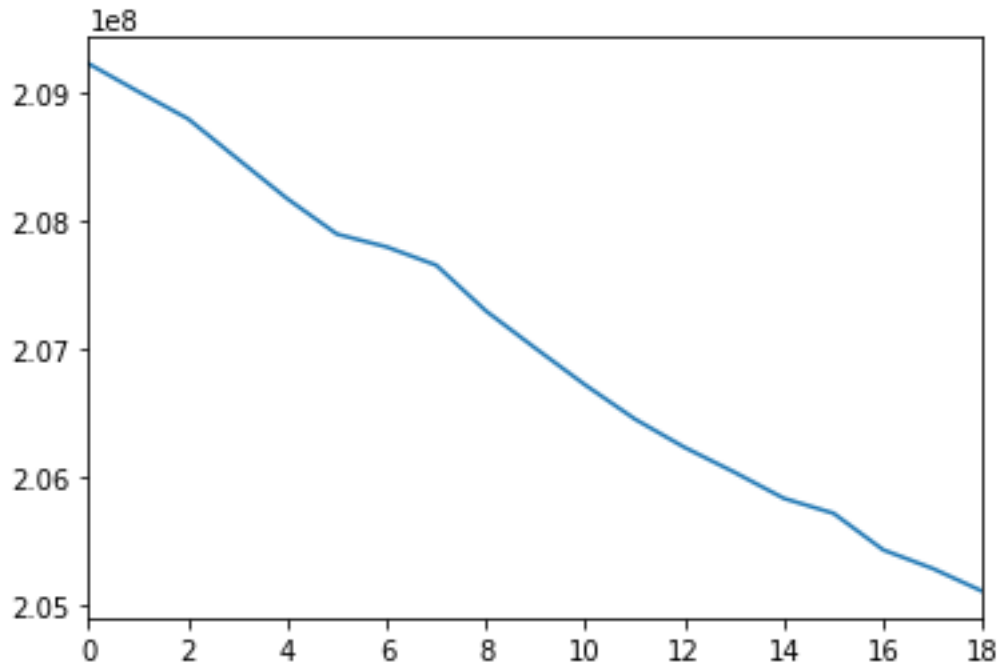
Na podstawie wykresu możemy stwierdzić że algorytm działa i wyszukuje coraz mniejszą wartość ‘**KR**’ wraz ze wzrostem liczby iteracji.

1.3.4 Wpływ parametrów na wyniki

Dla ustalonych wartości argumentów:

- `number_of_iterations = 500`
- `break_counter = 20`
- `neighbours_count = 20`

Wykres optymalizacji prezentuje się w następujący sposób:



Gdzie na osi X występują liczby iteracji, a na osi Y wartości 'KR'.

Ostateczny wynik dla 'KR' to 205111018.

Argument **number_of_iterations** ulepsza algorytm ponieważ im mniejsza jest ta wartość tym mniejsze jest prawdopodobieństwo że algorytm osiągnął ekstremum. Zwiększenie tego argumentu wydłuża czas wykonania się algorytmu lecz również zwiększy jego efektywność.

```
new_order, new_optimalization = hill_climbing_algorithm(  
    number_of_iterations=10000,  
    break_counter=20,  
    neighbours_count=20)
```

Ostateczny wynik dla 'KR' to 205111018.

Argument **break_counter** jest to licznik, który po wyzerowaniu kończy algorytm. Jego zwiększenie powoduje że algorytm dłużej będzie "stał w miejscu" i sprawdzał czy nie ma lepszej opcji.

```
new_order, new_optimalization = hill_climbing_algorithm(  
    number_of_iterations=500,
```

```
break_counter=100,  
neighbours_count=20)
```

Ostateczny wynik dla 'KR' to 205111018.

Argument **neighbours_count** ulepsza algorytm ponieważ im mniejsza jest wartość tego argumentu tym mniejszy jest zakres sprawdzanych sąsiadujących obiektów. Sprowadza się to wydłużenia pracy algorytmu, oraz mniejszego prawdopodobieństwa że przy zadanych argumentach przy zadanym zakresie algorytm osiągnie ekstremum globalne.

```
new_order, new_optimalization = hill_climbing_algorithm(  
    number_of_iterations=500,  
    break_counter=20,  
    neighbours_count=6)
```

Ostateczny wynik dla 'KR' to 208786181.

```
new_order, new_optimalization = hill_climbing_algorithm(  
    number_of_iterations=1000000,  
    break_counter=10000,  
    neighbours_count=100  
)
```

Ostateczny wynik dla 'KR' to 186185180.

Wszystkie wyniki przedstawione dla poszczególnych wartości argumentów zależą także od czynnika losowego jakim jest początkowe zadanie obrane za punkt startowy algorytmu. Dlatego przedstawione powyżej zależności nie będą prawdziwe w każdym przypadku uruchomienia algorytmu.

1.4 Algorytm wyżarzania

1.4.1 Wstęp

Algorytm wyżarzania opiera się na badaniu sąsiedztwa z założeniem pewnego poziomu prawdopodobieństwa z którym jesteśmy skłonni przyjąć gorszy od obecnego wynik. Z każdą wykonaną iteracją prawdopodobieństwo przyjęcia wyniku gorszego niż poprzedni zostaje obniżone poprzez zastosowanie odpowiednio zdefiniowanej funkcji. Algorytm ten posiada dużo argumentów, które wpływają na jakość wyniku.

1.4.2 Opis kodu

Metody początkowe, odpowiedzialne za wyszukanie listy sąsiadów zostały już opisane przy omawianiu algorytmu wspinaczki.

Nowymi funkcjami stworzonymi na potrzeby tego algorytmu jest funkcja **reduce temperature()**, która jest pośrednio odpowiedzialna za obniżanie poziomu prawdopodobieństwa z którym przyjmujemy nowy gorszy wynik.

1.4.3 Algorytm

Funkcja `simulated_anniling_algorithm()` jest ostateczną funkcją algorytmu. wykorzystuje wszystkie wcześniej opisane metody oraz przyjmuje poniższe argumenty za dane

- `T` - Maksymalna temperatura przyjmowana na początku algorytmu.
- `Tmin` - Minimalna temperatura przyjmowana na początku algorytmu.
- `neighbours_count` - liczba sąsiadów którą chcemy wyszukać
- `number_of_generations` - maksymalna liczba iteracji w przypadku gdy nie jest ona określona przez malejącą temperaturę.

Główne ciało funkcji zajmują pętle `while` których ograniczeniami są temperatura nie mogąca spaść poniżej minimalnej temperatury oraz liczba iteracji.

Jako pierwszy krok losowany jest punkt startowy do którego następnie znajdowane jest sąsiedztwo. Ze zbioru zadań należących do sąsiedztwa wybierane jest losowo jedno z nich. Następnie jeżeli zamiana wybranego zadania z punktem startowym daje wynik lepszy od poprzedniego, algorytm dokonuje zamiany tych dwóch zadań. Jeżeli wynik wychodzi gorszy to wtedy może on zostać przyjęty jako rozwiązanie wraz z z prawdopodobieństwem, które z każdą kolejną iteracją zostaje obniżone.

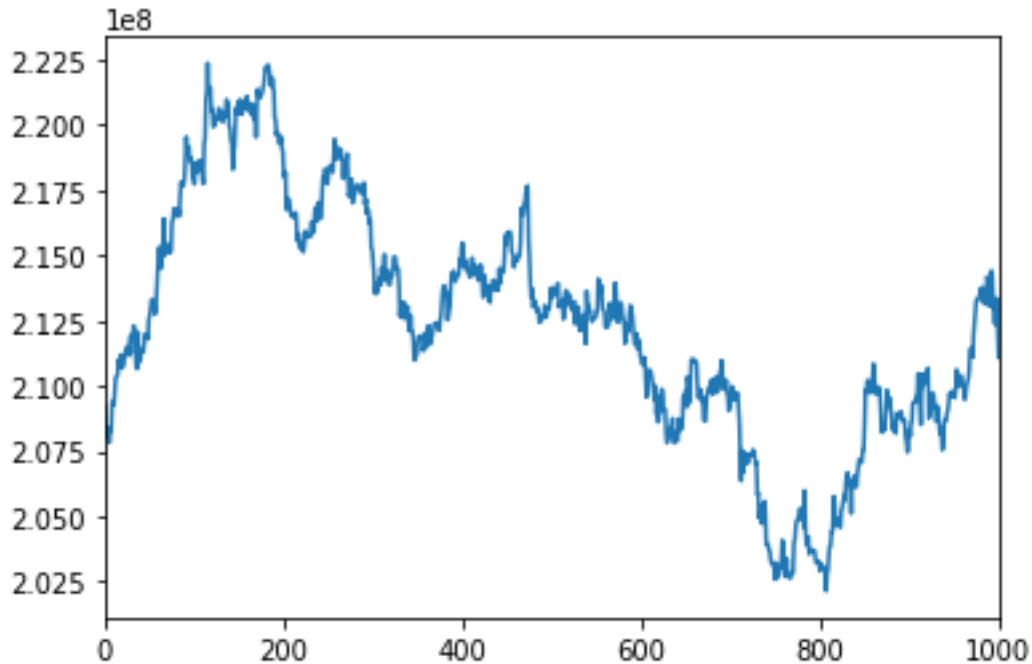
Algorytm kończy pracę gdy temperatura osiągnie stan minimalny, lub gdy zadany limit iteracji zostanie wykorzystany.

1.4.4 Wpływ parametrów na wyniki

Algorytm wyżarzania z bazowymi argumentami dla:

```
new_order, new_optimalization = simulated_anniling_algorithm(  
    T = 2500,  
    Tmin = 100,  
    neighbours_count=50,  
    number_of_generations = 1000)
```

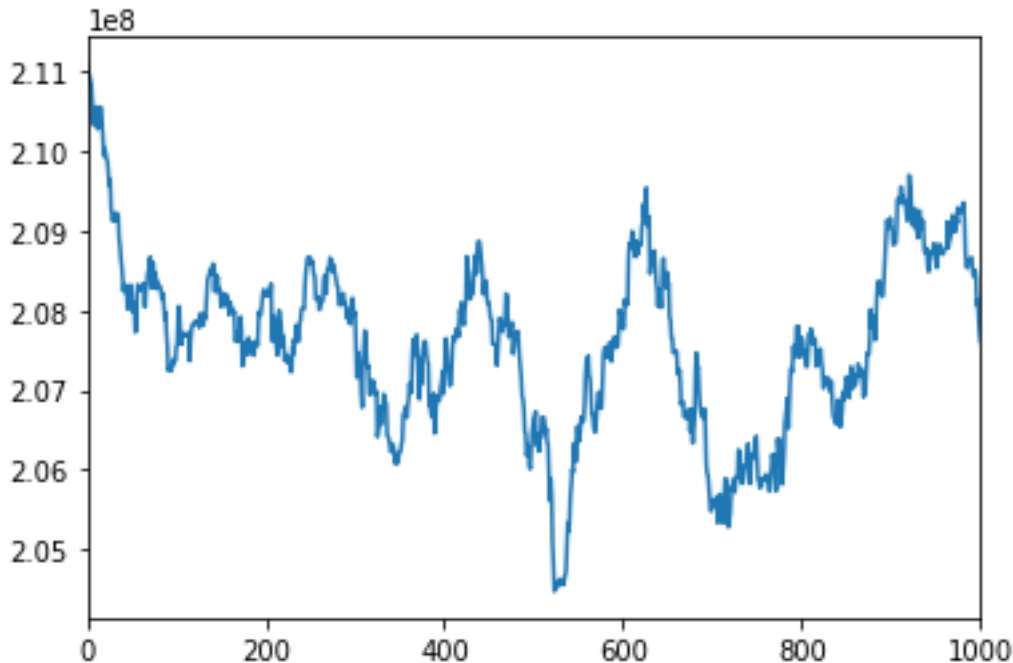
Ostateczna wartość 'KR' to 211094417.



Argument **T** wpływa na algorytm ponieważ może określać on maksymalną liczbę iteracji w przypadku gdy nie jest ustalony argument **number_of_generations**. Wpływa on również na prawdopodobieństwo przyjęcia gorszego niż poprzedniego wyniku. Im większa temperatura tym większe prawdopodobieństwo i tym samym większe wahania pomiędzy wynikami między poszczególnymi iteracjami. Tą zależność zauważyć można także na wykresie przedstawionym powyżej.

```
new_order, new_optimalization = simulated_annealing_algorithm(
    T = 5000,
    Tmin = 100,
    neighbours_count=20,
    number_of_generations = 1000)
```

Ostateczna wartość 'KR' to 207602394.



Argument **number_of_generations** ulepsza algorytm ponieważ im mniejsza jest ta wartość tym mniejsze jest prawdopodobieństwo że algorytm osiągnął optymalny wynik. Zwiększenie tego argumentu wydłuża czas wykonania się algorytmu lecz również zwiększy jego efektywność co zaobserwować możemy na powyższym wykresie.

```
new_order, new_optimalization = simulated_annealing_algorithm(
    T = 2500,
    Tmin = 100,
    neighbours_count=20,
    number_of_generations = 500)
```

Ostateczna wartość 'KR' to 694751. 208173780

Argument **neighbours_count** ulepsza algorytm ponieważ im mniejsza jest wartość tego argumentu tym mniejszy jest zakres sprawdzanych sąsiadujących obiektów. Sprowadza się to wydłużenia pracy algorytmu, oraz mniejszego prawdopodobieństwa że przy zadanych argumentach przy danym zakresie algorytm osiągnie ekstremum globalne.

```
new_order, new_optimalization = simulated_annealing_algorithm(
    T = 2500,
    Tmin = 100,
    neighbours_count=6,
    number_of_generations = 1000)
```

Ostateczna wartość 'KR' to 692948. 210143183

Wszystkie wyniki przedstawione dla poszczególnych wartości argumentów zależą także od czynnika losowego jakim jest początkowe zadanie obrane za punkt startowy algorytmu oraz losowanie z danym prawdopodobieństwem przyjęcia gorszego wyniku

jako rozwiązanie. Dlatego przedstawione powyżej zależności nie będą prawdziwe w każdym przypadku uruchomienia algorytmu.

1.5 Algorytm tabu

1.5.1 Wstęp

Algorytm przeszukiwania tabu opiera się na wymianie zadań.

Algorytm sprawdza, która zamiana zadań w największym stopniu zmniejszy czas realizacji projektu. Następnie dane zadania zamienia miejscami i zapisuje ten ruch na liście tabu. Owa lista przechowuje zbiór ruchów, których przez następne x iteracji algorytmu nie będziemy mogli użyć ponownie. Mechanizm ten pozwala na uniknięcie sytuacji, w której cyklicznie wymieniamy ze sobą dwa zadania. Działanie algorytmu zostaje zakończone w momencie ukończenia określonej liczby iteracji bądź w momencie gdy y razy algorytm stwierdzi, że czas realizacji nie może zostać poprawiony.

1.5.2 Opis kodu

W pierwszej kolejności została zdefiniowana zmienna 'KR', która przedstawia nam czas realizacji projektu. To właśnie tę wartość będziemy optymalizować. Do stworzenia 'KR' została użyta funkcja **get_KR()**.

Następnie tworzymy macierz z wartościami poszczególnych ruchów oraz tablicę możliwych ruchów. Wyszukujemy najlepszą możliwą wymianę (taką, która najbardziej obniży wartość KR) i zamieniamy ze sobą miejscami wybrane zadania. W następnych iteracjach powtarzamy powyższy proces oraz sprawdzamy czy dany ruch nie znajduje się na liście tabu. W tym celu zostały stworzone następujące funkcje:

- **gen_Z()** - generuje tablice możliwych ruchów,
- **search_for_the_best_move()** - wyszukuje najlepszy możliwy do wykonania ruch z tablicy,
- **move()** - zamienia ze sobą kolejność dwóch zadań, dodaje wybraną kombinację na listę tabu oraz aktualizuje listę tabu,
- **check_in_tabu()** - sprawdza czy dana kombinacja znajduje się na liście tabu,
- **update_tabu()** - aktualizuje listę tabu i usuwa z niej ruchy, których pozostały czas oczekiwania wykonał 0.

1.5.3 Algorytm

Ostateczna funkcja algorytmu to **tabu_search_algorithm()**, która wykorzystuje wcześniej opisane funkcje.

Algorytm przyjmuje następujące argumenty:

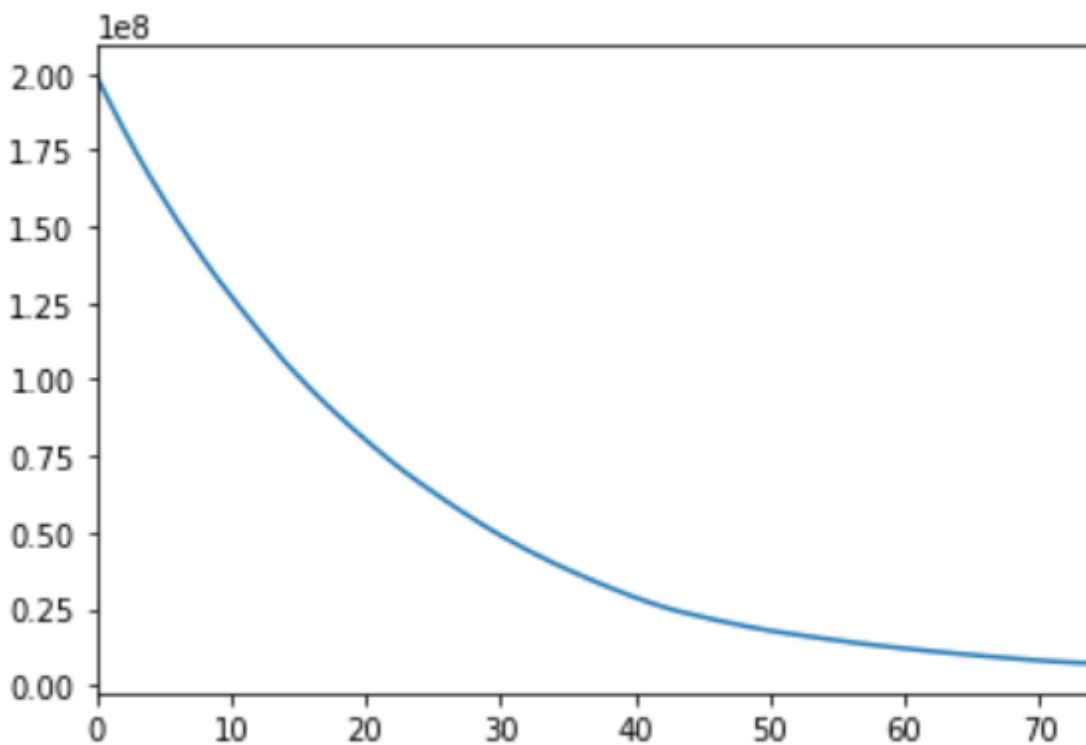
- **inactive time** - wartość, którą nadajemy zadaniu na liście tabu i zmniejszamy po każdej iteracji. Gdy wartość osiągnie 0, dana kombinacja może zostać ponownie użyta. Zwiększenie tej wartości spowoduje wydłużenie czasu wykonywania algorytmu, ponieważ będziemy musieli zbadać więcej kombinacji przed wybraniem tej odpowiedniej.

- **break_couter** - wartość, która zmniejsza się po iteracji w której algorytm nie znalazł lepszego rozwiązania. Gdy osiągnie 0 działanie algorytmu zostaje zakończone. Zwiększenie tej wartości spowoduje wydłużenie czasu wykonywania algorytmu, ponieważ program będzie oczekiwał dłużej na przejście wybranej liczby iteracji w których wynik końcowy nie ulega polepszeniu.
- **number_of_generations** - liczba iteracji. Zwiększenie tej wartości spowoduje wydłużenie czasu wykonywania algorytmu jednak pozwoli na dokładniejsze zbadanie problemu dzięki czemu otrzymane wyniki będą bardziej wiarygodne. Dzieje się tak ponieważ więcej razy badamy czy nie istnieją zadania, których zamiana spowoduje zmniejszenie czasu realizacji.

1.5.4 Wpływ parametrów na wyniki

Algorytm Tabu z bazowymi argumentami dla

```
new_order, new_opt = tabu_search_algorithm(
    inactive_time = 5,
    break_counter = 10,
    number_of_generations = 75)
```



Ostateczna wartość 'KR' to 7183470.

Argument `number_of_generations` ulepsza algorytm ponieważ im mniejsza jest ta wartość tym mniejsze jest prawdopodobieństwo że algorytm osiągnął optymalny wynik. Zwiększenie tego argumentu wydłuża czas wykonania się algorytmu lecz również zwiększy jego efektywność co zaobserwować możemy na powyższym wykresie.

Argument `break_counter` jest to licznik, który po wyzerowaniu kończy algorytm. Jego zwiększenie powoduje że algorytm dłużej będzie “stał w miejscu” i sprawdzał czy nie ma lepszej opcji.

```
new_order, new_opt = tabu_search_algorithm(  
    inactive_time = 5,  
    break_counter = 3,  
    number_of_generations = 75)
```

Ostateczny wynik dla ‘KR’ to 7183470.

Argument `inactive_time` opisuje ilość kolejek których dana zamiana dwóch zadań jest zablokowana. Trudnym jest ustalenie jak zmiana tego argumentu wpływa na wartość rozwiązania gdyż z jednej strony zbyt mała liczba naraża algorytm na cykliczne powtarzanie wciąż tych samych zamian, a z drugiej zbyt wielka wartość uniemożliwia optymalizację rozwiązania stosunkowo małej liczbie iteracji.

```
new_order, new_opt = tabu_search_algorithm(  
    inactive_time = 3,  
    break_counter = 10,  
    number_of_generations = 75)
```

Ostateczny wynik dla ‘KR’ to 2758246.

1.6 Porównanie zestawionych wyników z rozwiązaniem osiągniętym za pomocą pakietu Solver

| Nazwa Metody | Najlepsze Optymalne Rozwiązanie | Czas trwania (w sek) |
|---------------------|---------------------------------|------------------------|
| Tabu Search | 4114636 | 3020 |
| Hill Climbing | 186185180 | 1 |
| Simulated Annealing | 207602394 | 4 |
| Solver | 4743888 | 180 |

Z wyżej przedstawionego zestawienia można wywnioskować że najbardziej wydajnym algorytmem pod względem szybkości osiągniętego rozwiązania jest HILL CLIMBING, które w stosunkowo krótkim czasie było w stanie znaleźć dobre rozwiązanie. Najmniej optymalne okazało się jednak być TABU SEARCH. Ostateczny wynik tego algorytmu wynosi 4114636 i jest bardzo zbliżony do wyniku wykonanego w pakiecie Solver.

Wykres optymalizacji uzyskanej przez ten algorytm po 100 iteracjach:

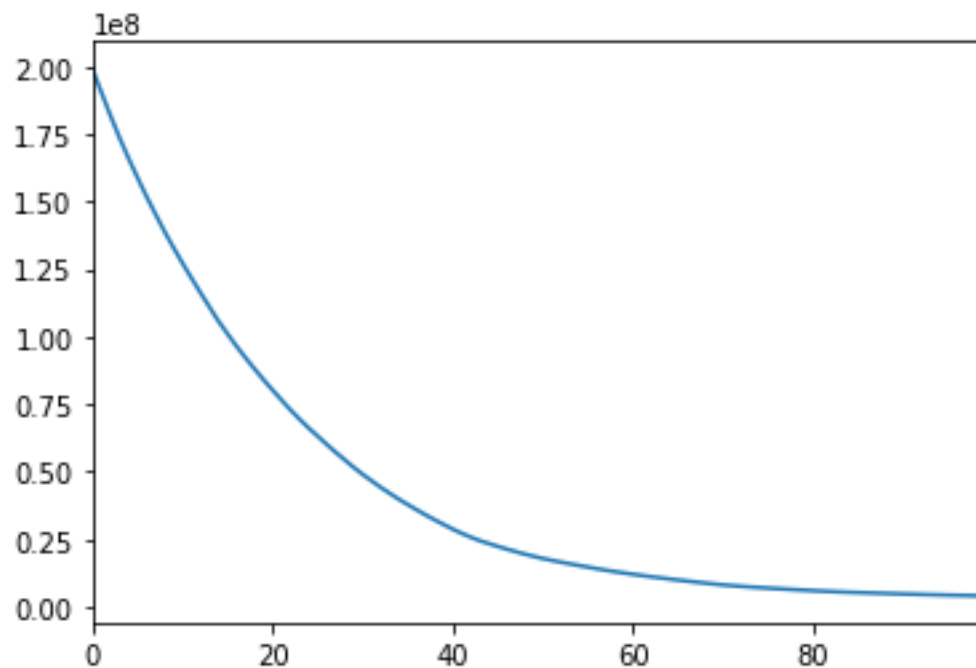


Tabela z zadaniami po wykonaniu tego algorytmu przedstawiona jest poniżej:

| Idx | Zadanie | Czas wykonania | Termin |
|-----|---------|----------------|--------|
| 0 | 98 | 2 | 43 |
| 1 | 195 | 19 | 30 |
| 2 | 17 | 7 | 8 |
| 3 | 83 | 16 | 10 |
| 4 | 184 | 1 | 68 |
| 5 | 35 | 16 | 241 |
| 6 | 7 | 16 | 196 |
| 7 | 56 | 15 | 26 |
| 8 | 54 | 4 | 44 |
| 9 | 38 | 11 | 119 |
| 10 | 130 | 16 | 192 |
| 11 | 12 | 1 | 153 |
| 12 | 122 | 14 | 123 |
| 13 | 133 | 10 | 22 |
| 14 | 115 | 19 | 76 |
| 15 | 16 | 15 | 77 |
| 16 | 121 | 12 | 234 |
| 17 | 172 | 14 | 106 |
| 18 | 84 | 18 | 247 |
| 19 | 147 | 16 | 327 |
| 20 | 200 | 10 | 380 |
| 21 | 157 | 5 | 329 |
| 22 | 55 | 17 | 218 |

| Idx | Zadanie | Czas wykonania | Termin |
|-----|---------|----------------|--------|
| 23 | 170 | 12 | 277 |
| 24 | 67 | 20 | 202 |
| 25 | 110 | 4 | 253 |
| 26 | 192 | 3 | 218 |
| 27 | 180 | 17 | 263 |
| 28 | 191 | 16 | 494 |
| 29 | 150 | 14 | 335 |
| 30 | 179 | 14 | 392 |
| 31 | 62 | 8 | 496 |
| 32 | 5 | 6 | 380 |
| 33 | 105 | 10 | 350 |
| 34 | 171 | 7 | 372 |
| 35 | 142 | 5 | 318 |
| 36 | 37 | 12 | 427 |
| 37 | 137 | 20 | 460 |
| 38 | 190 | 7 | 377 |
| 39 | 40 | 18 | 294 |
| 40 | 28 | 20 | 551 |
| 41 | 88 | 6 | 523 |
| 42 | 19 | 8 | 581 |
| 43 | 132 | 17 | 461 |
| 44 | 24 | 2 | 501 |
| 45 | 34 | 8 | 645 |
| 46 | 148 | 10 | 503 |
| 47 | 48 | 8 | 435 |
| 48 | 187 | 9 | 551 |
| 49 | 50 | 10 | 468 |
| 50 | 51 | 14 | 602 |
| 51 | 52 | 15 | 702 |
| 52 | 53 | 14 | 529 |
| 53 | 9 | 2 | 619 |
| 54 | 164 | 13 | 510 |
| 55 | 8 | 19 | 575 |
| 56 | 159 | 11 | 650 |
| 57 | 140 | 5 | 551 |
| 58 | 99 | 14 | 510 |
| 59 | 100 | 14 | 706 |
| 60 | 85 | 7 | 546 |
| 61 | 32 | 11 | 628 |
| 62 | 196 | 9 | 738 |
| 63 | 103 | 11 | 787 |
| 64 | 161 | 8 | 739 |
| 65 | 181 | 16 | 670 |
| 66 | 25 | 15 | 880 |
| 67 | 143 | 4 | 646 |
| 68 | 23 | 19 | 781 |

| Idx | Zadanie | Czas wykonania | Termin |
|-----|---------|----------------|--------|
| 69 | 70 | 20 | 945 |
| 70 | 71 | 5 | 985 |
| 71 | 72 | 14 | 882 |
| 72 | 124 | 15 | 851 |
| 73 | 166 | 12 | 620 |
| 74 | 146 | 7 | 855 |
| 75 | 178 | 11 | 853 |
| 76 | 111 | 14 | 965 |
| 77 | 194 | 15 | 789 |
| 78 | 57 | 11 | 1027 |
| 79 | 112 | 20 | 911 |
| 80 | 152 | 19 | 800 |
| 81 | 82 | 8 | 812 |
| 82 | 4 | 17 | 1059 |
| 83 | 102 | 9 | 952 |
| 84 | 61 | 16 | 1120 |
| 85 | 141 | 10 | 1021 |
| 86 | 87 | 11 | 871 |
| 87 | 42 | 11 | 958 |
| 88 | 75 | 13 | 1077 |
| 89 | 90 | 6 | 1061 |
| 90 | 65 | 15 | 1117 |
| 91 | 92 | 17 | 1076 |
| 92 | 93 | 16 | 1118 |
| 93 | 128 | 19 | 1174 |
| 94 | 176 | 4 | 1174 |
| 95 | 131 | 3 | 1046 |
| 96 | 174 | 20 | 1090 |
| 97 | 20 | 7 | 1261 |
| 98 | 59 | 14 | 1394 |
| 99 | 60 | 10 | 1206 |
| 100 | 3 | 19 | 1339 |
| 101 | 43 | 13 | 1166 |
| 102 | 64 | 6 | 1233 |
| 103 | 104 | 17 | 1443 |
| 104 | 46 | 16 | 1306 |
| 105 | 10 | 16 | 1301 |
| 106 | 58 | 13 | 1452 |
| 107 | 135 | 13 | 1226 |
| 108 | 154 | 14 | 1526 |
| 109 | 26 | 16 | 1202 |
| 110 | 189 | 5 | 1348 |
| 111 | 80 | 13 | 1542 |
| 112 | 113 | 8 | 1473 |
| 113 | 114 | 14 | 1575 |
| 114 | 15 | 10 | 1555 |

| Idx | Zadanie | Czas wykonania | Termin |
|-----|---------|----------------|--------|
| 115 | 160 | 18 | 1511 |
| 116 | 163 | 14 | 1599 |
| 117 | 36 | 11 | 1584 |
| 118 | 119 | 18 | 1630 |
| 119 | 175 | 16 | 1535 |
| 120 | 101 | 11 | 1695 |
| 121 | 13 | 1 | 1336 |
| 122 | 198 | 18 | 1705 |
| 123 | 107 | 6 | 1529 |
| 124 | 186 | 14 | 1782 |
| 125 | 188 | 1 | 1613 |
| 126 | 127 | 10 | 1527 |
| 127 | 68 | 14 | 1669 |
| 128 | 168 | 7 | 1539 |
| 129 | 11 | 11 | 1631 |
| 130 | 96 | 9 | 1763 |
| 131 | 44 | 15 | 1586 |
| 132 | 22 | 12 | 1666 |
| 133 | 134 | 17 | 1659 |
| 134 | 108 | 16 | 1910 |
| 135 | 136 | 10 | 1739 |
| 136 | 106 | 16 | 1968 |
| 137 | 173 | 12 | 1684 |
| 138 | 139 | 12 | 1601 |
| 139 | 73 | 6 | 1777 |
| 140 | 86 | 10 | 1811 |
| 141 | 118 | 17 | 1858 |
| 142 | 94 | 7 | 1818 |
| 143 | 144 | 13 | 1874 |
| 144 | 145 | 14 | 2039 |
| 145 | 89 | 17 | 2089 |
| 146 | 1 | 10 | 1746 |
| 147 | 47 | 20 | 1806 |
| 148 | 177 | 13 | 1957 |
| 149 | 30 | 2 | 1875 |
| 150 | 6 | 8 | 1820 |
| 151 | 162 | 18 | 1916 |
| 152 | 153 | 9 | 2158 |
| 153 | 109 | 17 | 1892 |
| 154 | 155 | 5 | 2009 |
| 155 | 29 | 16 | 2223 |
| 156 | 14 | 2 | 1804 |
| 157 | 158 | 13 | 2251 |
| 158 | 183 | 8 | 1895 |
| 159 | 116 | 17 | 2315 |
| 160 | 199 | 18 | 2125 |

| Idx | Zadanie | Czas wykonania | Termin |
|-----|---------|----------------|--------|
| 161 | 81 | 5 | 2123 |
| 162 | 117 | 20 | 2249 |
| 163 | 69 | 6 | 2187 |
| 164 | 197 | 17 | 2136 |
| 165 | 74 | 12 | 2284 |
| 166 | 167 | 3 | 2135 |
| 167 | 129 | 11 | 2048 |
| 168 | 169 | 11 | 2301 |
| 169 | 2 | 10 | 2134 |
| 170 | 151 | 8 | 2233 |
| 171 | 18 | 5 | 2098 |
| 172 | 138 | 18 | 2381 |
| 173 | 77 | 1 | 2109 |
| 174 | 120 | 8 | 2353 |
| 175 | 95 | 11 | 2201 |
| 176 | 149 | 4 | 2143 |
| 177 | 76 | 15 | 2310 |
| 178 | 31 | 4 | 2104 |
| 179 | 41 | 20 | 2399 |
| 180 | 66 | 17 | 2252 |
| 181 | 182 | 17 | 2346 |
| 182 | 79 | 3 | 2312 |
| 183 | 33 | 1 | 2323 |
| 184 | 185 | 18 | 2370 |
| 185 | 125 | 4 | 2131 |
| 186 | 49 | 20 | 2310 |
| 187 | 126 | 19 | 2470 |
| 188 | 97 | 7 | 2298 |
| 189 | 39 | 19 | 2374 |
| 190 | 156 | 18 | 2423 |
| 191 | 27 | 15 | 2439 |
| 192 | 193 | 4 | 2195 |
| 193 | 78 | 20 | 2480 |
| 194 | 45 | 2 | 2385 |
| 195 | 63 | 11 | 2453 |
| 196 | 165 | 8 | 2489 |
| 197 | 123 | 3 | 2439 |
| 198 | 91 | 10 | 2460 |
| 199 | 21 | 19 | 2396 |