

Algorytmy i struktury danych

Symulowane wyżarzanie dla poszukiwania minimum globalnego funkcji Easoma

Mariusz Mańka

Informatyka III semestr, niestacjonarne

Wydział Matematyki Stosowanej

30 grudnia 2020

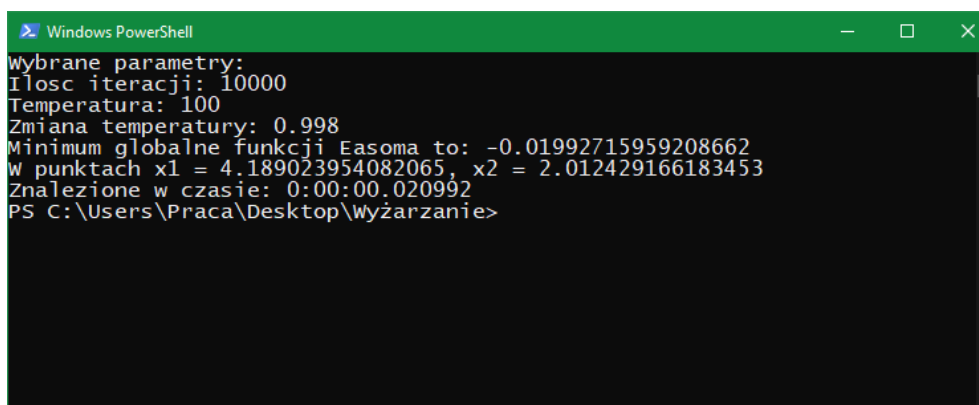
Część I

Opis programu

Program ma za zadanie znalezienie za pomocą algorytmu symulowanego wyżarzania **minimum globalne funkcji kryterialnej Easoma**. Dodatkowo za jego pomocą możemy przeanalizować wpływ różnych czynników, takich jak temperatura lub ilość iteracji na wynik działania funkcji. Podczas kolejnych uruchomień programu możemy dobrać takie parametry jak: **ilość iteracji**, **temperatura początkowa** oraz **wartość o jaką będziemy zmniejszać** temperaturę początkową w każdej kolejnej iteracji. Pozwala to na dobranie optymalnych parametrów oraz na dokładniejszą analizę wpływu danych parametrów inicjalnych na ostateczny wynik funkcji.

Instrukcja obsługi

Aby uruchomić program należy przejść za pomocą konsoli cmd do miejsca w którym mamy skrypt z zadaniem. Następnie wpisać polecenie `python nazwa_programu.py`. Gdy to zrobimy program uruchomi się nam w konsoli. Następnie zostaniemy poproszeni o podanie wcześniej wspomnianych parametrów inicjalnych. Podajemy więc **ilość iteracji** (im więcej iteracji tym dłuższy czas wykonywania się programu), **temperaturę początkową** oraz **ułamek o który ma zmienić się temperatura początkowa** po każdej z iteracji. Ważne jest aby pamiętać że ostatni parametr musi być **ułamkiem większym niż 0 oraz mniejszym niż 1**. Należy także pamiętać o tym że liczby ułamkowe należy wprowadzać nie ze znakiem przecinka, ale ze znakiem kropki, jak to zostało zaprezentowane na zrzucie poniżej. Gdy wprowadzimy już wymagane parametry program obliczy wartość minimum funkcji Easoma za pomocą zaimplementowanych metod symulowanego wyżarzania.



```
Windows PowerShell
wybrane parametry:
Ilosc iteracji: 10000
Temperatura: 100
Zmiana temperatury: 0.998
Minimum globalne funkcji Easoma to: -0.01992715959208662
w punktach x1 = 4.189023954082065, x2 = 2.012429166183453
Znalezione w czasie: 0:00:00.020992
PS C:\Users\Praca\Desktop\Wyżarzanie>
```

Rysunek 1: Przykładowe działanie programu

Dodatkowe informacje

Do prawidłowego działania tego programu potrzebujemy mieć zainstalowane środowisko uruchomieniowe pythona. Środowisko to możemy pobrać z [oficjalnej strony pythona](#). Zalecane jest również uruchamianie programu w środowisku Windows, gdyż bez tego niektóre funkcjonalności mogą przestać działać poprawnie.

Część II

Opis działania

Program implementuje jeden z bardziej znanych algorytmów heurystycznych a mianowicie **algorytm symulowanego wyżarzania**. Sama **heurystyka** w informatyce jest metodą znajdowania rozwiązań dla której nie ma gwarancji znalezienia rozwiązania optymalnego, a często nawet prawidłowego. Pozwala ona znaleźć rozwiązanie przybliżone, a jedynie w szczególnym przypadku dokładne. Algorytmy heurystyczne pomimo tego że często zwracają jedynie przybliżone rozwiązanie, są wybierane wszędzie tam gdzie algorytmy dokładne nie są w stanie zwrócić żadnego rozwiązania w sensownym czasie, ponieważ koszt operacji które musiałyby wykonać jest zbyt wielki. Metody te służą także do znajdowania rozwiązań przybliżonych, na podstawie których później wylicza się ostateczny rezultat pełnym algorytmem.

Przykładem algorytmu heurystycznego jest użyty w moim programie **Algorytm Symulowanego Wyżarzania** (ang. **Simulated Annealing**). Jest to algorytm, który zaczerpnął idee działania ze zjawiska wyżarzania w procesach metalurgicznych, które polega na ogrzewaniu metalu do wysokiej temperatury, utrzymywaniu go w danych warunkach, a następnie powolnym jego ochładzaniu. Stąd jego cechą charakterystyczną jest występowanie parametru sterującego zwanego **temperaturą**, który maleje w trakcie wykonywania algorytmu. **Im wyższą wartość ma ten parametr, tym bardziej chaotyczne mogą być zmiany.** Dokładniejszy opis algorytmu można znaleźć poniżej w sekcji **Algorytm**

Algorytm

Algorytm Symulowanego Wyżarzania to algorytm iteracyjny. Na początku losowane jest pewne rozwiązanie, w przypadku mojej implementacji tego algorytmu jest to **zestaw dwóch punktów - x_1 oraz x_2** . z zakresu od **-100 do 100**. Następnie jest ono w kolejnych krokach modyfikowane. Jeśli w danym kroku uzyskamy rozwiązanie lepsze, wybieramy je. Istotną cechą symulowanego wyżarzania jest jednak to, że z pewnym prawdopodobieństwem **możliwe jest zaakceptowanie gorszego rozwiązania** (ma to na celu umożliwienie wyjścia z minimum lokalnego). Prawdopodobieństwo przyjęcia gorszego rozwiązania wyrażone jest wzorem: $e^{-\frac{\delta}{T}}$. δ jest równa: $\delta = f(X) - f(X')$ gdzie X jest poprzednim rozwiązaniem, X' nowym rozwiązaniem, a $f()$ jest jedną z **funkcji kryterialnych**. **Im wyższa wartość $f(X)$, tym lepsze rozwiązanie.** We wzorze można zauważyć, że prawdopodobieństwo przyjęcia gorszego rozwiązania spada wraz ze spadkiem temperatury i wzrostem różnicy jakości obu rozwiązań. W samej implementacji tego algorytmu można się również spotkać z różnymi **funkcjami kryterialnymi** (testowymi). Do najpopularniejszych należą: **Funkcja De Jonga, Rastragina, Rosenbrock** oraz użyta w moim algorytmie **funkcja Easoma**. Funkcja ta wyrażona jest wzorem: $Easoma(x_1, x_2) = -\cos x_1 * \cos x_2 * e^{-(x_1 - \pi)^2 - (x_2 - \pi)^2}$. Wyżej opisane operacje zazwyczaj wykonywane są w pętli zadana ilość razy. Każda iteracja pętli nazywana jest **epoką**. W każdej **epoce** losowane jest nowe rozwiązanie, **liczona jest wartość funkcji kryterialnej** dla obecnego oraz nowego rozwiązania po czym na podstawie tych wyników program liczy **prawdopodobieństwo przyjęcia nowego rozwiązania**. Na samym końcu każdej iteracji zmniejszamy początkową temperaturę o zadana wcześniej wartość i przechodzimy do następnej epoki. Gdy algorytm wykona zadana liczbę kroków, zatrzyma się a my otrzymamy nasz wynik.

Data: Dane wejściowe temperatura początkowa - T_{max}

Data: Dane wejściowe O ile zmniejszyć temperaturę w każdym kolejnym kroku - α

Data: Dane wejściowe Ilość iteracji - n

Data: Dane wejściowe Funkcja kryterialna - $Easom$

Result: Minimum globalne funkcji Easoma

inicjalizacja n

inicjalizacja T_{max}

inicjalizacja α

while $i < n$ **do**

 losuj punkty x_1, x_2

while $T_{max} > 0$ **do**

 losuj nowe punkty x_{new1}, x_{new2}

 licz $\Delta := Easom(x_1, x_2) - Easom(x_{new1}, x_{new2})$

$rand :=$ losowa liczba z przedziału $[0, 1]$

if $rand < \exp(-\Delta/T)$ **then**

 | Przyjmij nowe rozwiązanie

else

 | kontynuuj

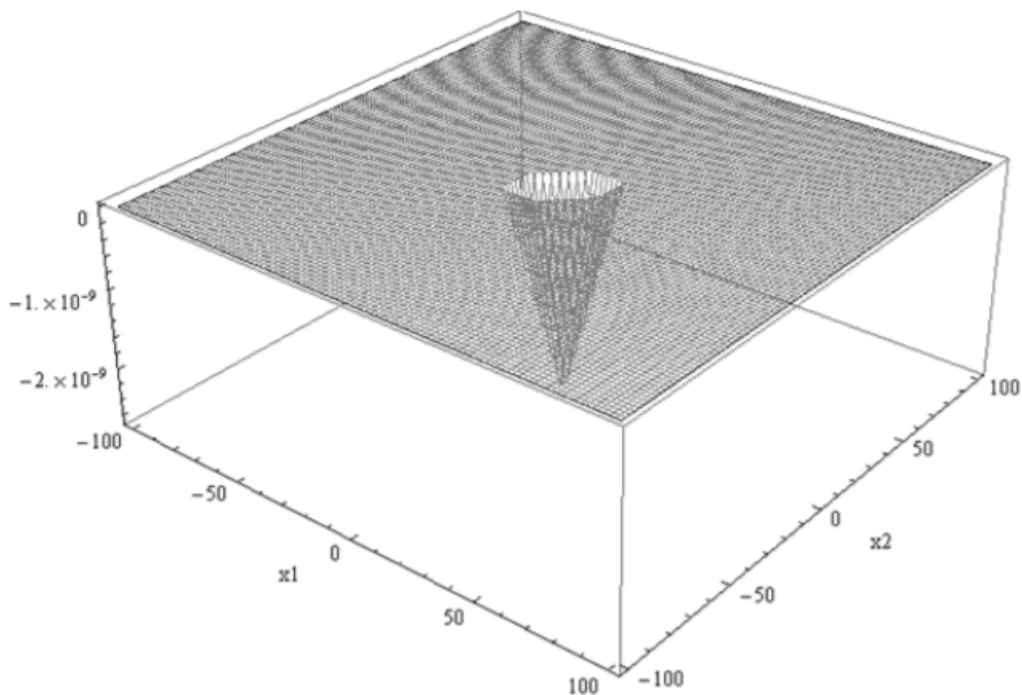
end

 Obniż temperaturę T_{max} o α

end

end

Algorithm 1: Algorytm Symulowanego Wyżarzania.



Rysunek 2: Wykres przedstawiający funkcję Easoma (zdjęcie zaczerpnięte z wykładu)

Implementacja

Działanie programu jest oparte w pełni na funkcji sterującej *main()* oraz klasie sterującej **SA**. Zadanie funkcji **main()** jest proste. Z racji tego że jest to funkcja sterująca jej zadaniem jest 'kontakt' z użytkownikiem. Pobiera ona od niego parametry wejściowe takie jak: **żądana ilość iteracji**, **temperatura początkowa** oraz **wartość o którą temperatura będzie zmniejszana** w każdym kolejnym kroku iteracji. Gdy funkcja pobierze oraz poprawnie zwaliduje dane, tworzy nową instancję klasy sterującej **SA** podając pobrane wcześniej parametry jako argumenty. Następnie uruchamia na stworzonej wcześniej instancji metodę **get_solution()** z której pobiera rozwiązanie oraz czas w jakim funkcja się wykonała. Na samym końcu wszystkie niezbędne dane drukowane są na ekran konsoli w formie przyjaznej użytkownikowi. Kod funkcji sterującej **main()** można zobaczyć poniżej:

```
1 def main():
2     try:
3         iterations = int(input("Podaj ilosc iteracji: "))
4         temp = int(input("Podaj temperature poczatkowa: "))
5         alpha = float(
6             input("Podaj zmiane temperatury (ulamek z zakresu od 0.8 do
              0.99): "))
7         if alpha <= 0.7 or alpha >= 1:
8             raise ValueError(
9                 "Blad zakresu! Podana liczba musi byc wieksza od 0 i
              mniejsza od 1")
10    except ValueError:
11        print("Podano niepoprawna wartosc!")
12
13    sa = SA(iterations, temp, alpha)
14    solution, duration = sa.get_solution()
15    system("cls")
16    print('Wybrane parametry:')
17    print(f'Ilosc iteracji: {iterations}')
18    print(f'Temperatura: {temp}')
19    print(f'Zmiana temperatury: {alpha}')
20    print(f'Minimum globalne funkcji Easoma to: {solution[0]}')
21    print(f'W punktach x1 = {solution[1]}, x2 = {solution[2]}')
22    print(f'Znalezione w czasie: {duration}')
23
24
25
26 if __name__ == "__main__":
27     main()
```

Kolejną najważniejszą funkcją z punktu widzenia naszego programu jest metoda klasy **SA** - **get_solution()**. Jest to główna metoda w naszej klasie która wykorzystując inne metody pomocnicze liczy minimum globalne funkcji Easoma dla wylosowanych punktów przy zadanych parametrach - czyli w praktyce zwraca nam rozwiązanie oraz czas w jakim funkcja się wykonała. Na samym początku wykorzystując zaimportowaną wcześniej funkcję **datetime** oraz dostępne w niej metody pobiera aktualny czas. Następnie za pomocą funkcji **get_random_solution()** generowane jest aktualne rozwiązanie - czyli **losowy punkt z przedziału od -100 do 100.**, oraz pobierana jest za pomocą gettera **get_temperature** początkowa temperatura ustawiona przez użytkownika w funkcji *main()* oraz przekazana jako parametr do klasy. Gdy inicjalizacja tych zmiennych dobiegła już końca, zaczynamy naszą główną pętlę która wykona się zadaną przez użytkownika w funkcji *main()* ilość razy. Każda iteracja tej pętli to jedna **epoka** dla naszego algorytmu. Wewnątrz tej pętli losowane jest za pomocą metody **shift()** nowe rozwiązanie. Następnie jeśli wartość funkcji **Easom** dla nowego rozwiązania będzie większa niż wartość tej funkcji dla rozwiązania obecnego, zawsze nadpisujemy obecne rozwiązanie nowym. Jeżeli zdarzy się że wartość nowego rozwiązania będzie mniejsza (co za tym idzie mamy do czynienia z rozwiązaniem gorszym) to za pomocą funkcji **get_probability** decydujemy które rozwiązanie należy zostawić. Na samym końcu pętli zmieniamy naszą aktualną temperaturę mnożąc ją przez podany w funkcji *main()* parametr i przechodzimy do następnej **epoki** (iteracji). Gdy wszystkie iteracje dobiegną końca do zmiennej **duration** przypisujemy czas w jakim wykonał się nasz algorytm. Jest on w istocie różnicą między czasem pobranym na samym początku działania funkcji a czasem pobranym w momencie zakończenia jej działania. Obie wartości są zwracane a funkcja kończy swoje działanie. Kod funkcji poniżej:

```
1 import datetime
2
3     def get_solution(self):
4         start = datetime.datetime.now()
5         randomPoint = self.get_random_solution() # x0
6         temp = self.get_temperature()
7
8         for i in range(0, self.iterations):
9             randomNew = self.shift(randomPoint[1], randomPoint[2]) # x1
10
11             if randomPoint[0] > randomNew[0]:
12                 randomPoint = randomNew
13
14             if randomPoint[0] < randomNew[0]:
15                 randomPoint = self.get_probability(randomPoint,
16                                                     randomNew)
17
18             temp = temp * self.alpha
19
20         duration = datetime.datetime.now() - start
21         return randomPoint, duration
```

Następną wartą uwagi funkcją jest funkcja **shift**(x_1, x_2). Przyjmuje on dwa parametry - x_1 oraz x_2 , na podstawie których ma za zadanie wyznaczyć nowe rozwiązanie. Parametry te są punktami x_1 oraz x_2 aktualnego rozwiązania, przekazywanymi do funkcji wraz z jej wywołaniem w metodzie **get_solution()**. Z początku podobnie jak w funkcji **get_solution()** pobierana do zmiennej **temp** jest aktualna temperatura. W kolejnym kroku liczona jest zmienna **delta** poprzez pomnożenie aktualnej temperatury przez losowo wygenerowany punkt z przedziału od -1 do 1. Tak policzona **delta** dodawana jest następnie do naszych punktów x_1 oraz x_2 . Operacje te wykonywane są w celu znalezienia punktu który będzie "w sąsiedztwie" naszego aktualnego rozwiązania z którego punktów jako parametrów korzystamy. Duży wpływ na wyliczanie nowych punktów ma aktualna temperatura - im jest **niższa** tym "bliżej" aktualnego rozwiązania są nowe punkty. Dzięki metodzie **easom()**, **liczona jest wartość funkcji Easoma dla nowych punktów**. Na samym końcu zwracamy nowe rozwiązanie **w postaci tablicy 3 elementów** gdzie pierwszym elementem jest wartość funkcji Easoma dla danych dwóch punktów, a kolejnymi dwoma elementami są nowe punkty powstałe w wyniku działania tej funkcji. Pełen kod funkcji poniżej:

```

1 import random
2
3     def shift(self, x1, x2):
4         temp = self.get_temperature()
5         delta1 = temp * random.uniform(-1, 1)
6         delta2 = temp * random.uniform(-1, 1)
7         xnew = x1 + delta1
8         xprev = x2 + delta2
9         easom_value = self.easom(xnew, xprev)
10        return [easom_value, xnew, xprev]
```

Dalsze metody naszej klasy sterującej są dużo prostsze w swoim działaniu. Metoda **get_probability(point_1, point_2)** określa przy pomocy **uproszczonego równania Boltzmanna** prawdopodobieństwo z którym przyjęte zostanie nowe rozwiązanie. Przyjmuje ona dwa argumenty **point_1** oraz **[point_2]** które reprezentują aktualne rozwiązanie oraz te nowo wygenerowane za pomocą funkcji **shift()**. Standardowo na samym początku pobierana jest do zmiennej **temp** za pomocą metody **get_temperature** aktualna temperatura. Następnie do zmiennej **prob** przypisywana jest **losowa liczba z zakresu od 0 do 1** potrzebna przy liczeniu prawdopodobieństwa. Do zmiennej **delta** za pomocą funkcji **get_difference** przypisywana jest różnica wartości funkcji Easoma dla tych rozwiązań którą możemy znaleźć w pierwszych 'szufladkach' naszych tablic *point_1* oraz *point_2* reprezentujących aktualne i nowe rozwiązanie. Gdy funkcja zwróci nam już potrzebną różnicę sprawdzamy za pomocą uproszczonego równania Boltzmanna ($e^{-\frac{\delta}{T}}$) gdzie δ to nasza zmienna **delta** a **T** to nasza temperatura - **temp**, sprawdzamy czy jego wynik (równania Boltzmanna) jest mniejszy od wcześniej wygenerowanego losowo punktu **prob**. Jeśli tak to zwracane jest aktualne rozwiązanie - **point_1** - oznacza to że nie dokonujemy zamiany i 'zostawiamy' aktualne rozwiązanie. Jeżeli zaś zmienna **prob** jest większa od wyniku, wtedy przyjmujemy nowe rozwiązanie = **point_2** w miejsce starego. Gdy to się stanie metoda kończy działanie. Kod metody na następnej stronie:

```

1
2 import random
3 import math
4
5     def get_probability(self, point1, point2):
6         temp = self.get_temperature()
7         prob = random.uniform(0, 1)
8         delta = self.get_difference(point1[0], point2[0])
9
10        if prob < math.exp(- delta/temp):
11            return point1
12        else:
13            return point2

```

Zadaniem metody `get_random_solution()` jest zwrócenie losowego rozwiązania. Metoda to nie przyjmuje żadnego argumentu, i jest używana przed rozpoczęciem pętli w metodzie `get_solution` do wygenerowania losowego rozwiązania z przedziału od -100 do 100. W swoim ciele funkcja za pomocą metody `randint` z biblioteki `random` zwraca dwa losowe punkty z zadanego przedziału - `x1` oraz `x2` - a następnie wywołuje na nich metodę `easom(x1, x2)` licząc tym samym wartość funkcji Easoma dla tych dwóch punktów. Na samym końcu zwraca rozwiązanie w postaci tablicy trzech elementów - Pierwszy to wartość funkcji Easoma a dwa kolejne to losowo wygenerowane wcześniej punkty. Kod metody poniżej:

```

1 import random
2
3     def get_random_solution(self):
4         x1 = random.randint(-100, 100)
5         x2 = random.randint(-100, 100)
6         easom_value = self.easom(x1, x2)
7         return [easom_value, x1, x2]

```

Metody `get_difference(new_solution, current_solution)`, `get_temperature()` oraz `easom(x1, x2)` mają za zadanie tylko zwracanie wartości, są to funkcje pomocnicze. Pierwsza z nich `get_difference` przyjmuje jako dwa argumenty dwie wartości funkcji easoma dla obecnego oraz nowego rozwiązania i zwraca ich różnicę. Używana jest we wcześniej opisanej funkcji `get_probability()`. Funkcja `get_temperature` jest typowym **getterem**. Zwraca prywatną wartość `_temp`. Natomiast funkcja `easom` implementuje niezbędną do poprawnego działania algorytmu funkcję kryterialną **Easoma**. Funkcja ta otrzymuje jako parametry dwa punkty x_1 oraz x_2 po czym zwraca wartość funkcji **Easoma** dla tych punktów. Funkcja **Easoma** wyrażona jest wzorem: $-\cos x_1 * \cos x_2 * e^{-(x_1-\pi)^2-(x_2-\pi)^2}$. Kod opisanych metod na następnej stronie:


```

1 import random
2 import math
3
4 ###
5
6     def get_difference(self, new_solution, current_solution):
7         return new_solution - current_solution
8
9 ###
10
11     def get_temperature(self):
12         return self._temp
13
14 ###
15
16     def easom(self, x1, x2):
17         easom = -math.cos(x1)*math.cos(x2) * \
18             math.exp(-(x1-math.pi)**2-(x2-math.pi)**2)
19         return easom

```

Testy / Eksperymenty

Kluczowym aspektem jeśli mowa o funkcjach heurystycznych jest obranie odpowiedniej, to znaczy optymalnej **strategii wyszukiwania**. Przy algorytmie symulowanego wyżarzania z którym mamy do czynienia mamy możliwość zmiany parametrów startowych takich jak **temperatura początkowa** (temperatura od której rozpoczynamy wyżarzanie), **ilość epok** (całkowita ilość iteracji algorytmu) oraz **funkcja zmiany temperatury** mówiąca o tym o ile stopni 'ochładzamy nasz układ' przy każdej kolejnej iteracji. Naszym zadaniem jest taki dobór parametrów aby w jak najkrótszym czasie funkcja zwróciła nam rozwiązanie jak najbliższe rozwiązaniu poprawnemu. Czytając więcej o naszej funkcji kryterialnej Easoma, możemy dowiedzieć się że posiada ona swoje **minimum globalne** (którego za pomocą tego algorytmu szukamy) w punkcie $f(x, y) = -1$, gdzie parametry x oraz y równe są wartości π . Formalny zapis tego zjawiska wygląda następująco: $Easom(\pi, \pi) = -1$ dla argumentów z przedziału $-100 \leq x, y \leq 100$. Więc zadaniem naszej funkcji będzie **zwrócenie wyniku jak najbliższego liczbie -1**. Przeanalizujmy więc który z naszych parametrów startowych ma największy wpływ na poprawność otrzymanego wyniku oraz jaka konfiguracja parametrów będzie konfiguracją optymalną. W moich testach badałem wpływ funkcji zmiany temperatury oraz ilości iteracji na nasz wynik. I tak w pierwszym podejściu przyjąłem ilość iteracji równą **1000**, i stopniowo zmieniałem funkcję zmiany temperatury podając coraz to większe ułamki, oto wynik tego badania:

Temperatura	Ilość iteracji	Zmiana temperatury	Wynik	Czas wykonania (s)
100	1000	0.6	-8,07892144765564E-09	00.002028
100	1000	0.8	-3,2076427967362E-13	00.001777
100	1000	0.85	-4,31426019796651E-26	00.018494
100	1000	0.90	-3,12649721627056E-35	00.006754
100	1000	0.95	-4,93640654100089E-07	00.004756
100	1000	0.995	-0,000410085	00.009965

Rysunek 3: Wykres przedstawiający pomiary dla ilości iteracji równej 1000 oraz różnymi wartościami funkcji zmiany temperatury

Jak możemy zauważyć na powyższym zrzucie ekranu wyniki otrzymaliśmy bardzo szybko, bo zaledwie w przeciągu średnio **3 ms**, jednak znacząco odbiega on od wyniku który chcielibyśmy otrzymać. Można więc założyć że funkcja zmiany temperatury pomaga, ale ilość iteracji musi zostać zdecydowanie zwiększona. Idąc za tym wnioskiem, w drugim podejściu zwiększyłem ilość iteracji z 1000 do **100000** przy takiej samej temperaturze początkowej i identycznej jak w poprzednim przykładzie funkcji zmiany temperatury. Oto wyniki:

Temperatura	Ilość iteracji	Zmiana temperatury	Wynik	Czas wykonania (s)
100	100000	0.6	-0.8708438764653831	00.21
100	100000	0.8	-0.9775573656013127	00.18
100	100000	0.85	-0.9821945505695575	00.22
100	100000	0.90	-0.9096894428343578	00.31
100	100000	0.95	-0.9997978353365331	00.38
100	100000	0.995	-0.8010144193520484	00.24

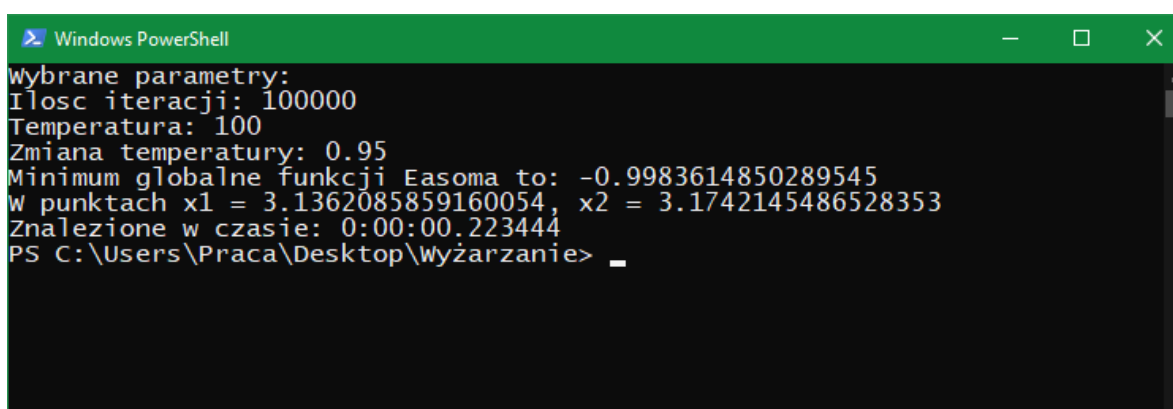
Rysunek 4: Wykres przedstawiający pomiary dla ilości iteracji równej 100000 oraz różnymi wartościami funkcji zmiany temperatury

Z powyższego zrzutu możemy jasno wywnioskować że podniesienie ilości iteracji do **100000** znacznie poprawiło skuteczność naszego algorytmu, nawet przy najniższej (w teorii najmniej wydajnej) funkcji zmiany temperatury na poziomie **0,6** otrzymany wynik jest bardzo bliski ideałowi. Im większa wartość zmiany temperatury, tym wyniki nieznacznie się poprawiają. Jednak jak możemy zaobserwować w ostatnim wierszu, gdzie zmiana temperatury jest najbliższa ideałowi, poprawa ta może wynikać jedynie z losowości algorytmu, a zmiana temperatury może nie mieć wielkiego wpływu. Czas w którym algorytm się wykonał również jest jak bardzo satysfakcjonujący, i nie przekracza **0,4s**. Idąc o krok dalej po raz kolejny zwiększyłem ilość iteracji tym razem aż do **10000000** nie zmieniając pozostałych założeń. Oto jaki wynik otrzymałem:

Temperatura	Ilość iteracji	Zmiana temperatury	Wynik	Czas wykonania (s)
100	10000000	0.6	-0.8708438764653831	23.77
100	10000000	0.85	-0.9993213642468429	21.54
100	10000000	0.95	-0.9957630733402537	25.11
100	10000000	0.995	-0.9973217152048691	25.98

Rysunek 5: Wykres przedstawiający pomiary dla ilości iteracji równej 10000000 oraz różnymi wartościami funkcji zmiany temperatury

Algorytm poprzez dołożenie kolejnych iteracji bardzo znacząco stracił na wydajności czasowej. Wyniki po raz kolejny były bardzo dobre, jednak czas w którym algorytm je zwrócił pozostawia wiele do życzenia. W najlepszym przypadku otrzymaliśmy wyniki w czasie **21,54 sekund** a w najgorszym w czasie prawie **26 sekund**. Odnotowując tym samym aż **100 krotny wzrost czasu**. Wyniki otrzymane przy tej liczbie iteracji są natomiast równie bliskie ideałowi jak wyniki otrzymane przy liczbie iteracji rzędu **100000**, co prowadzi do wniosku że najlepszą - dającą najlepsze wyniki w optymalnie niskim czasie - ilością iteracji dla tego algorytmu jest ilość **100000** przedstawiona na środkowym zrzucie ekranu. Powyżej tej wartości poprawność wyniku praktycznie nie ulega polepszeniu natomiast czas wykonania algorytmu drastycznie spada. Jak mogliśmy również zauważyć funkcja zmiany temperatury nie ma znaczącego wpływu w tym przypadku na poprawność otrzymanych danych. Dlatego jako optymalną wartość należy przyjąć wartość z przedziału **0,85** do **0,99**. Poniżej zrzut ekranu z wykonaniem programu z najbardziej optymalnymi parametrami (należy zauważyć że punkty x_1 oraz x_2 są bardzo bliskie wartości liczby $\pi - 3,14$):



```
Windows PowerShell
wybrane parametry:
Ilosc iteracji: 100000
Temperatura: 100
Zmiana temperatury: 0.95
Minimum globalne funkcji Easoma to: -0.9983614850289545
W punktach x1 = 3.1362085859160054, x2 = 3.1742145486528353
Znalezione w czasie: 0:00:00.223444
PS C:\Users\Praca\Desktop\Wyżarzanie>
```

Rysunek 6: Wykonanie programu przy **optymalnej strategii wyszukiwania**

Pełen kod aplikacji

```
1 import random
2 import math
3 import datetime
4
5
6 class SA:
7     def __init__(self, iterations, temp, alpha):
8         self.iterations = iterations
9         self._temp = temp
10        self.alpha = alpha
11
12    def easom(self, x1, x2):
13        easom = -math.cos(x1)*math.cos(x2) * \
14            math.exp(-(x1-math.pi)**2-(x2-math.pi)**2)
15        return easom
16
17    def shift(self, x1, x2):
18        temp = self.get_temperature()
19        delta1 = temp * random.uniform(-1, 1)
20        delta2 = temp * random.uniform(-1, 1)
21        xnew = x1 + delta1
22        xprev = x2 + delta2
23        easom_value = self.easom(xnew, xprev)
24        return [easom_value, xnew, xprev]
25
26 # Gettery
27
28    def get_temperature(self):
29        return self._temp
30
31    def get_difference(self, new_solution, current_solution):
32        return new_solution - current_solution
33
34    def get_random_solution(self):
35        x1 = random.randint(-100, 100)
36        x2 = random.randint(-100, 100)
37        easom_value = self.easom(x1, x2)
38        return [easom_value, x1, x2]
39
40    def get_probability(self, point1, point2):
41        temp = self.get_temperature()
42        prob = random.uniform(0, 1)
43        delta = self.get_difference(point1[0], point2[0])
44
45        if prob < math.exp(- delta/temp):
46            return point1
47        else:
48            return point2
49
50
51
52
```

```

53     def get_solution(self):
54         start = datetime.datetime.now()
55         randomPoint = self.get_random_solution() # x0
56         temp = self.get_temperature()
57
58         for i in range(0, self.iterations):
59             randomNew = self.shift(randomPoint[1], randomPoint[2]) # x1
60
61             if randomPoint[0] > randomNew[0]:
62                 randomPoint = randomNew
63
64             if randomPoint[0] < randomNew[0]:
65                 randomPoint = self.get_probability(randomPoint,
66                                                         randomNew)
67
68             temp = temp * self.alpha
69
70         duration = datetime.datetime.now() - start
71         return randomPoint, duration
72
73     def main():
74         try:
75             iterations = int(input("Podaj ilosc iteracji: "))
76             temp = int(input("Podaj temperature poczatkowa: "))
77             alpha = float(
78                 input("Podaj zmiane temperatury (ulamek z zakresu od 0.8 do
79                     0.99): "))
80             if alpha <= 0.7 or alpha >= 1:
81                 raise ValueError(
82                     "Blad zakresu! Podana liczba musi byc wieksza od 0 i
83                     mniejsza od 1")
84
85         except ValueError:
86             print("Podano niepoprawna wartosc!")
87
88         sa = SA(iterations, temp, alpha)
89         solution, duration = sa.get_solution()
90         system("cls")
91         print('Wybrane parametry:')
92         print(f'Ilosc iteracji: {iterations}')
93         print(f'Temperatura: {temp}')
94         print(f'Zmiana temperatury: {alpha}')
95         print(f'Minimum globalne funkcji Easoma to: {solution[0]}')
96         print(f'W punktach x1 = {solution[1]}, x2 = {solution[2]}')
97         print(f'Znalezione w czasie: {duration}')
98
99     if __name__ == "__main__":
100         main()

```
