Metody Obliczeniowe w Nauce i Technice

Laboratorium 4

Wojciech Łącki

Spis treści

[zadanie 1 – tsp 2](#_Toc100666626)

[zadanie 2 – obraz binarny 20](#_Toc100666627)

[zadanie 3 – sudoku 27](#_Toc100666628)

# zadanie 1 – tsp

Polecenie: Wygeneruj chmurę n losowych punktów w 2D, a następnie zastosuj algorytm symulowanego wyżarzania do przybliżonego rozwiązania problemu komiwojażera dla tych punktów.

Potrzebne import.

import math  
import numpy as np  
import networkx as nx  
import matplotlib.pyplot as plt  
import os  
import glob  
from random import randint, random, shuffle  
from PIL import Image

1. Przedstaw wizualizację otrzymanego rozwiązania dla 3 różnych wartości n oraz 3 różnych układów punktów w 2D (rozkład jednostajny, rozkład normalny z czterema różnymi grupami parametrów, dziewięć odseparowanych grup punktów).

Na początek kilka funkcji generujących punktu w układzie 2D.

def generate\_random(n):  
 points = []  
 while len(points) != n:  
 x = randint(0, n)  
 y = randint(0, n)  
 tup = (x, y)  
 if tup not in points:  
 points.append(tup)  
  
 points = [(i, point) for i, point in enumerate(points)]  
 return points  
  
  
def generate\_uniform(n):  
 points = []  
 while len(points) != n:  
 x = np.random.uniform(n, high=1)  
 y = np.random.uniform(n, high=1)  
 tup = (x, y)  
 if tup not in points:  
 points.append(tup)  
  
 points = [(i, point) for i, point in enumerate(points)]  
 return points  
  
  
def generate\_normal(n, loc, scale):  
 points = []  
 while len(points) != n:  
 x = np.random.normal(n // loc, scale)  
 y = np.random.normal(n // loc, scale)  
 tup = (x, y)  
 if tup not in points:  
 points.append(tup)  
  
 points = [(i, point) for i, point in enumerate(points)]  
 return points  
  
  
def generate\_groups(n, k):  
 points = []  
 addX = 0  
 addY = 0  
 factor = int(math.sqrt(k))  
 while len(points) != n:  
 x = (n // k) \* random() + addX \* factor \* (n // k)  
 y = (n // k) \* random() + addY \* factor \* (n // k)  
 tup = (x, y)  
 if tup not in points:  
 points.append(tup)  
 if len(points) % (n // k) == 0:  
 addX += 1  
 if addX % (k // factor) == 0:  
 addX = 0  
 addY += 1  
 if addX + factor \* addY == k:  
 addY = 0  
  
 shuffle(points)  
 points = [(i, point) for i, point in enumerate(points)]  
 return points

Opis:

* generate\_random – generuje całkowicie losowe punkty
* generate\_uniform – generuje punkty w rozkładzie jednostajnym
* generate\_normal – generuje punkty w rozkładzie normalnym
* generate\_groups – generuje punkty w kilku grupach

Funkcja tworząca graf skierowany na podstawie punktów.

def create\_directed\_graph(points):  
 graph = nx.DiGraph()  
 n = len(points)  
 for i in range(n - 1):  
 graph.add\_edge(i, i + 1)  
 graph.add\_edge(n - 1, 0)  
 return graph

Funkcja rysująca graf (numery wierzchołków zawsze są te same).

def draw\_graph(graph, points):  
 pos = {i: point[1] for i, point in enumerate(points)}  
 node\_numbers = {i: point[0] for i, point in enumerate(points)}  
 nx.draw(graph, pos, labels=node\_numbers, font\_size=6, node\_size=100)  
 plt.show()

Aby można było zobaczyć, jak algorytm zmieniał połączenia zostały stworzone dwie funkcje. Jedna zapisująca graf jako obraz, a druga tworząca z tych obrazów gif’a.

def save\_graph(graph, points, index):  
 pos = {i: point[1] for i, point in enumerate(points)}  
 node\_numbers = {i: point[0] for i, point in enumerate(points)}  
 nx.draw(graph, pos, labels=node\_numbers, font\_size=6, node\_size=100)  
 plt.title = str(index)  
 plt.savefig(str(index) + ".png", format="PNG")  
 plt.clf()  
  
  
def create\_gif(text):  
 frames = []  
 imgs = glob.glob("\*.png")  
 imgs\_array = [(int(img[:len(img) - 4]), img) for img in imgs]  
 imgs\_array.sort(key=lambda img: img)  
 for \_, file\_name in imgs\_array:  
 new\_frame = Image.open(file\_name)  
 frames.append(new\_frame)  
 frames[0].save("tsp\_" + text + ".gif", format="GIF",  
 append\_images=frames[1:],  
 save\_all=True,  
 duration=500, loop=0)  
 for \_, file\_name in imgs\_array:  
 os.remove(file\_name)

Główną funkcją realizującą przedstawiony problem jest funkcja „solution”.

def solution(points, temp\_start, temp\_end, temp\_iter, temp\_rate, arbitrary\_swap):  
 n = len(points)  
 text = "ARBITRARY" if arbitrary\_swap else "CONSECUTIVE"  
 print(text)  
 best = current\_solution\_distance(points)  
 print("START DIST:", best)  
 iterations = 0  
 x = []  
 y = []  
 save\_graph(graph, points, iterations)  
 while temp\_start > temp\_end:  
 for i in range(temp\_iter):  
 p1 = randint(0, n - 2)  
 p2 = randint(p1 + 1, n - 1) if arbitrary\_swap else p1 + 1  
 points[p1], points[p2] = points[p2], points[p1]  
 possible = current\_solution\_distance(points)  
 if possible < best:  
 best = possible  
 else:  
 prob = math.e \*\* ((best - possible) / temp\_start)  
 check\_number = random()  
 if check\_number < prob:  
 best = possible  
 else:  
 points[p1], points[p2] = points[p2], points[p1]  
 x.append(iterations)  
 y.append(best)  
 # for i in range(temp\_iter):  
 # temp\_start \*= temp\_rate  
 temp\_start \*= temp\_rate  
 iterations += 1  
 if iterations % 100 == 0:  
 save\_graph(graph, points, iterations)  
 print("END DIST:", best)  
 save\_graph(graph, points, iterations)  
 plt.plot(x, y, "c-")  
 plt.show()  
 create\_gif(text)

Przyjmuje ona następujące parametry:

* points – aktualna trasa (kolejne punkty są połączone)
* temp\_start – temperatura początkowa
* temp\_end – temperatura końcowa
* temp\_iter – ilość iteracji przy jednej temperaturze
* temp\_rate – współczynnik o jaki maleje temperatura
* arbitrary\_swap – czy losowe dwa punkty są zamieniane

Funkcja kosztu jest sumarycznym dystansem, jaki należy przebyć. Więc chcemy wartość tej funkcji zminimalizować.

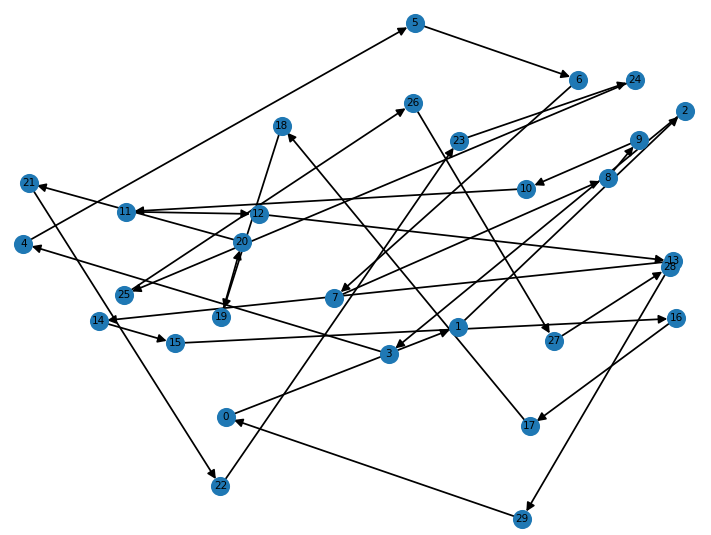
def current\_solution\_distance(points):  
 n = len(points)  
 sum = 0  
 for i in range(n):  
 sum += distance(points[i - 1], points[i])  
 return sum  
  
  
def distance(p1, p2):  
 i1, pos1 = p1  
 i2, pos2 = p2  
 x1, y1 = pos1  
 x2, y2 = pos2  
 return math.sqrt(math.pow(x1 - x2, 2) + math.pow(y1 - y2, 2))

Program został uruchamiany z parametrami temperatury takimi jak poniżej. Zmianie ulegały liczby punktów („n”) i sposób ich generacji („points”).

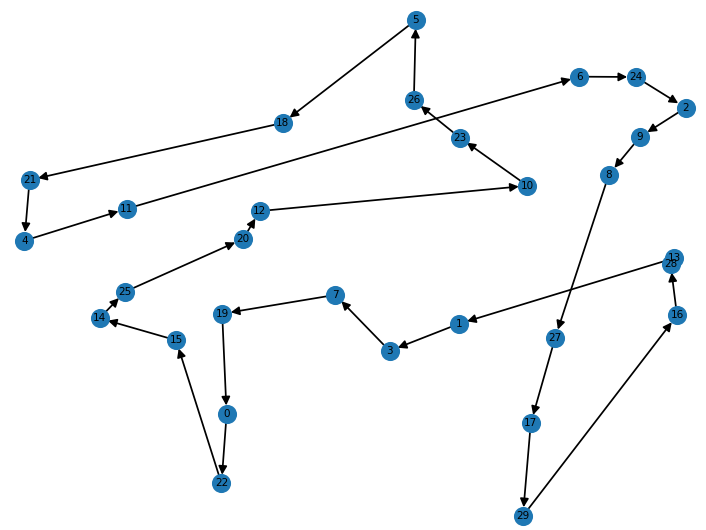
if \_\_name\_\_ == "\_\_main\_\_":  
 n = 40  
 temp\_start = 5230  
 temp\_end = 0.1  
 temp\_iter = 20  
 temp\_rate = 0.99  
 points = generate\_random(n)  
 # points = generate\_uniform(n)  
 # loc=2  
 # scale = 4.3  
 # points = generate\_normal(n,loc,scale)  
 # k = 9  
 # points = generate\_groups(n, k)  
 graph = create\_directed\_graph(points)  
 draw\_graph(graph, points)  
 arbitrary\_swap = True  
 solution(points, temp\_start, temp\_end, temp\_iter, temp\_rate,  
 arbitrary\_swap)  
 print(points)  
 draw\_graph(graph, points)  
 points.sort(key=lambda tup: tup[0])  
 arbitrary\_swap = False  
 solution(points, temp\_start, temp\_end, temp\_iter, temp\_rate,  
 arbitrary\_swap)  
 print(points)  
 draw\_graph(graph, points)

1. Rozład jednostajny, n = 30

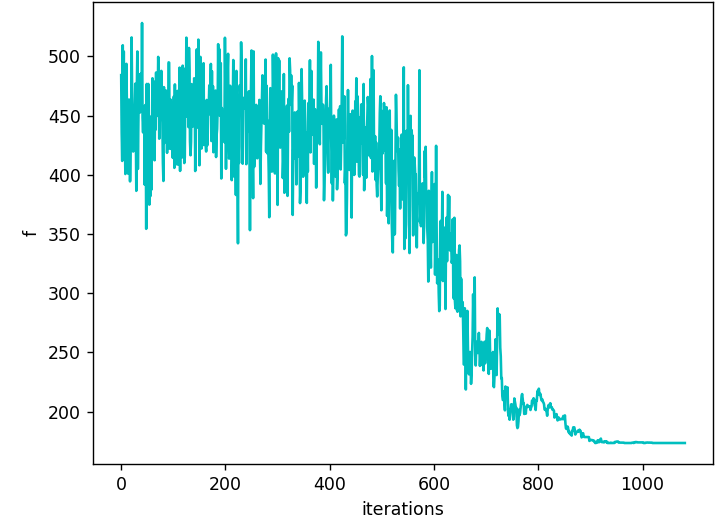
Wygenerowana trasa:



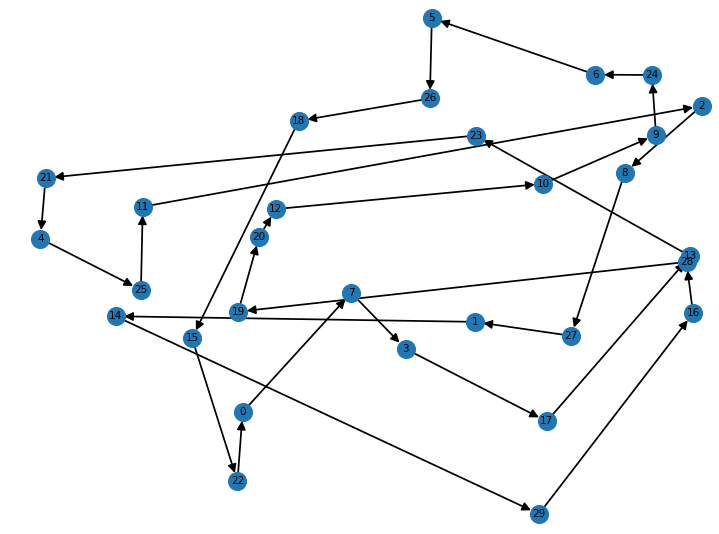
Uzyskana trasa przy losowej zamianie:



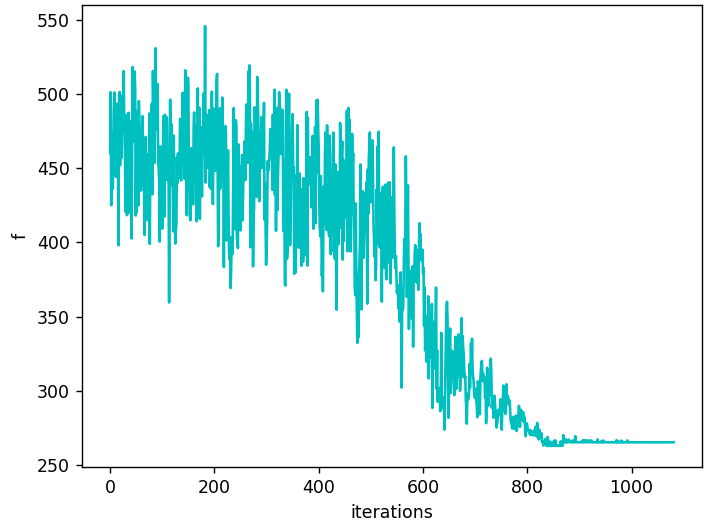
Wykres optymalizacji wyglądał następująco:



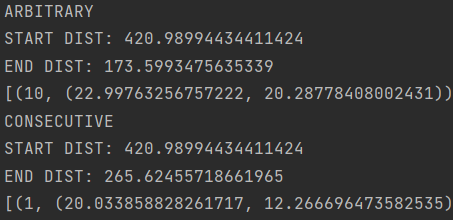
Natomiast przy zamianie jedynie sąsiadów:



A jego wykres optymalizacji:

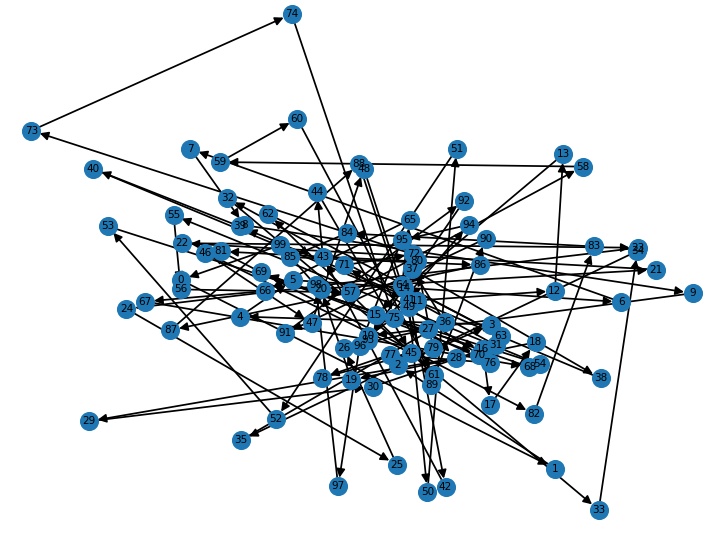


Ogólny wynik jest przedstawiony poniżej. Lepsza wyszła zamiana losowych punktów.

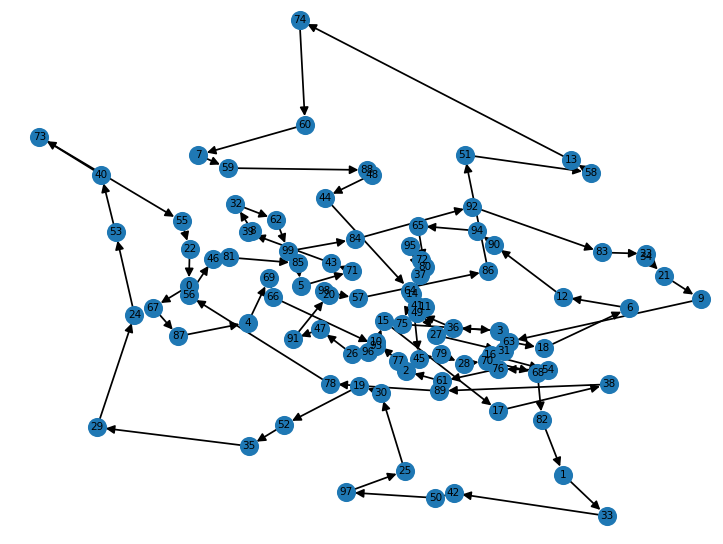


1. Rozkład normalny, n = 100

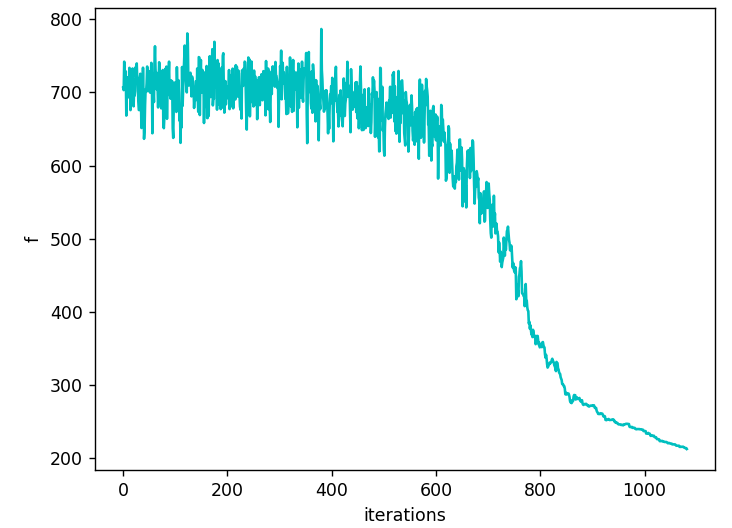
Zaczęło się od:



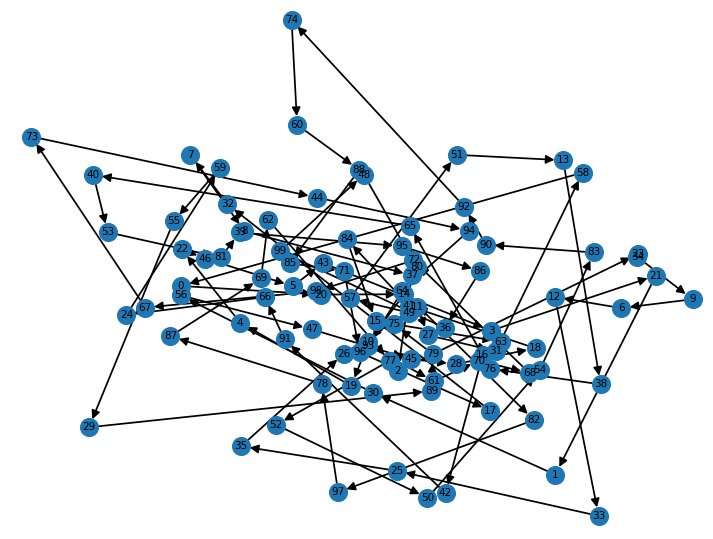
W wyniku zamiany losowej punktów otrzymaliśmy:



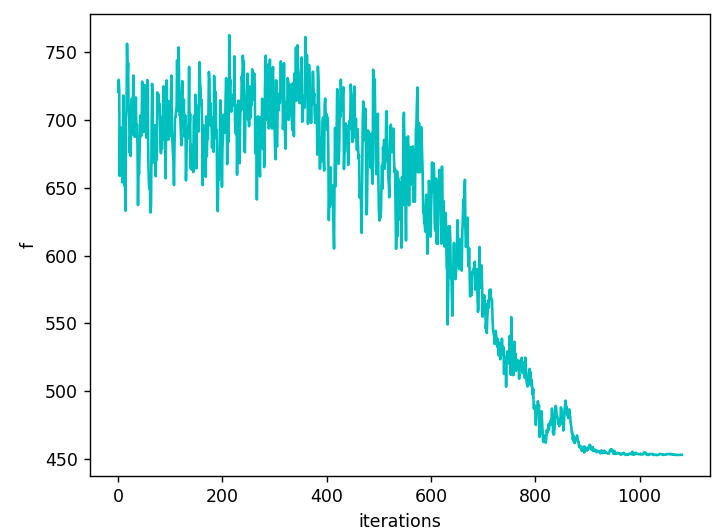
Wykres wartości funkcji dystansu od iteracji:



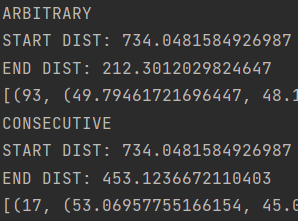
A zamieniając sąsiadów:



Z wykresem:

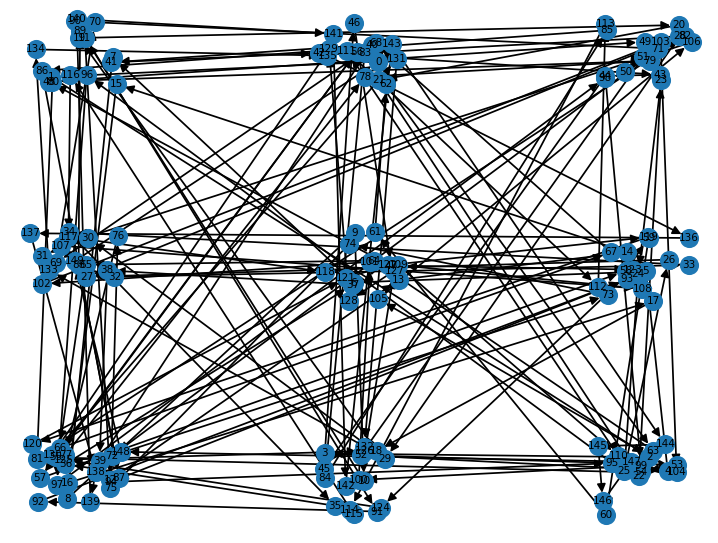


Rezultat:



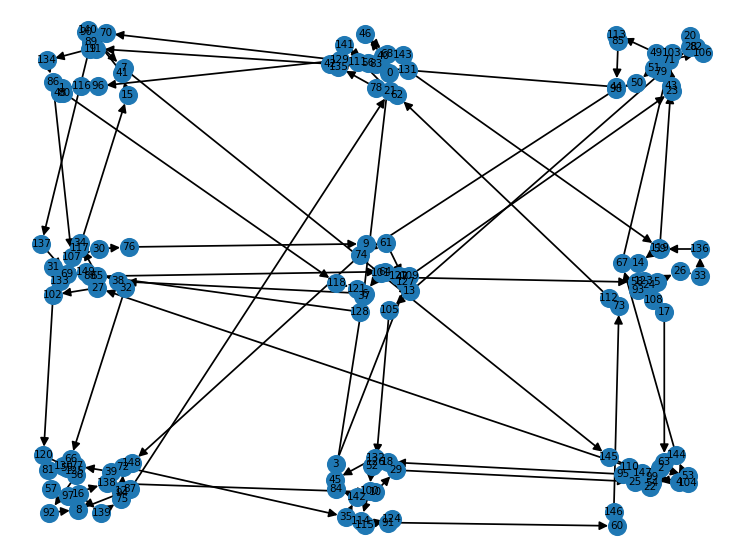
Ponownie lepsza okazała się zamiana losowych punktów.

1. Grupy punktów, n = 150

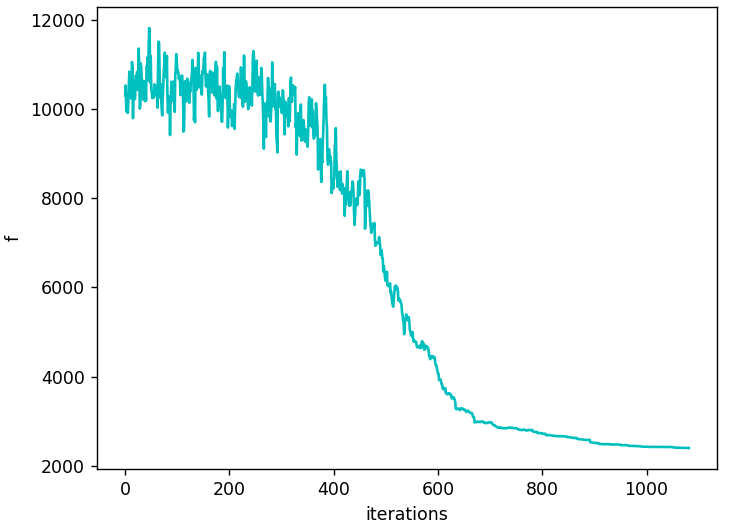


Punkty zostały ładnie wygenerowane.

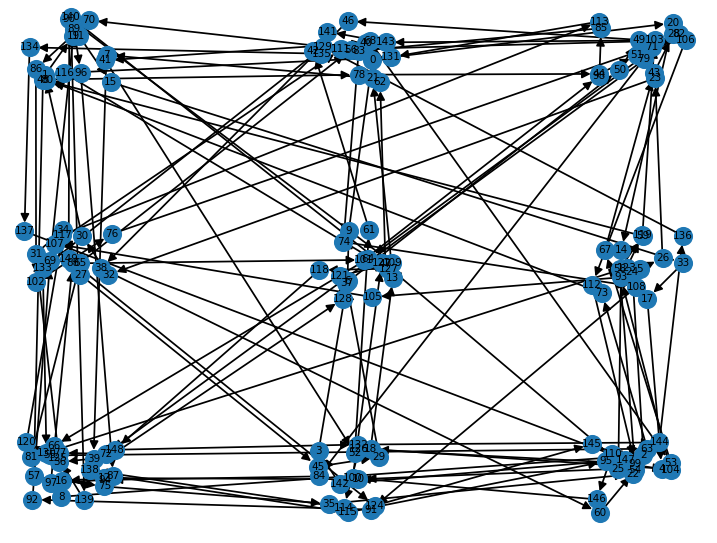
Teraz uruchamiając algorytm symulowanego wyżarzania zamieniając losowe dwa punkty otrzymaliśmy:



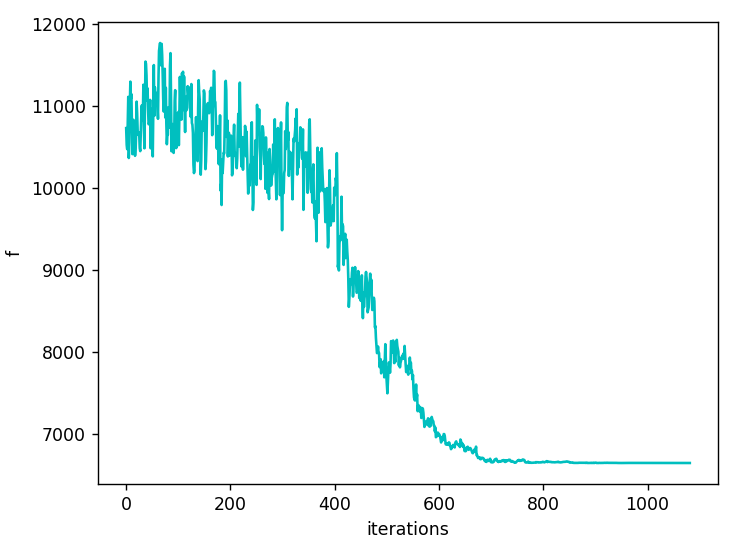
A funkcja zmieniała się tak:



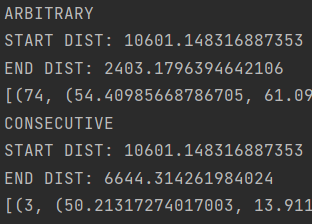
Dla kontrastu zamiana jedynie sąsiadów:



Dała znacznie gorszy wynik. Optymalizowana funkcja zmieniała się w taki sposób:

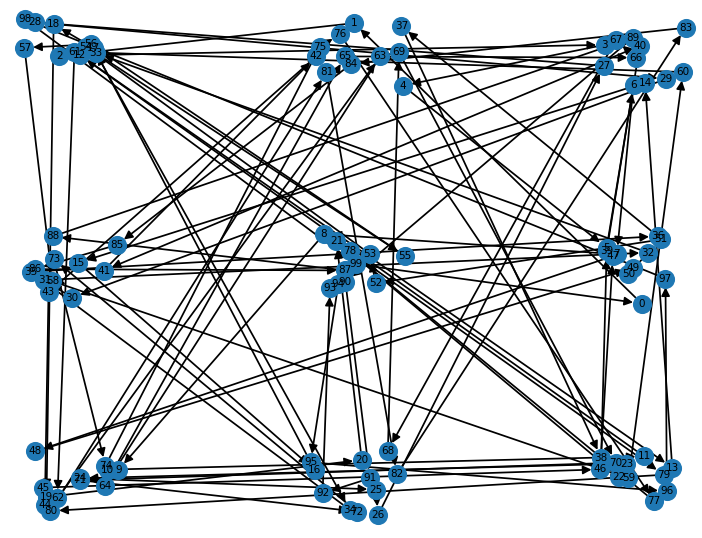


Rezultaty są następujące:



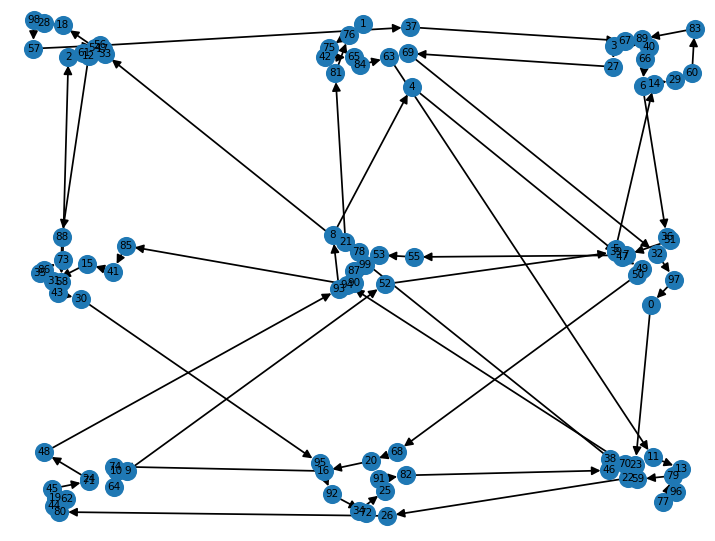
Znacznie lepsze okazało się zamienianie losowych punktów.

Teraz zobaczymy jak spadek temperatury ma wpływ na rozwiązanie. Do testów użyliśmy generowania punktów w grupach dla n = 100. Temperatura spadała ze współczynnikami 0.99, 0.95 oraz 0.8. Pokazywane będą tylko wyniki dla losowej zamiany punktów. Na początku wylosowana została taka trasa:

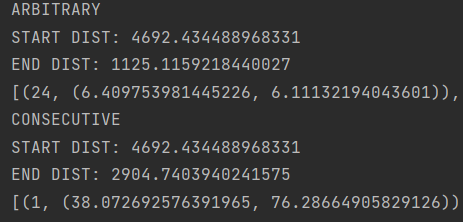


1. Temp\_rate = 0.99

Otrzymaliśmy następujący wynik:

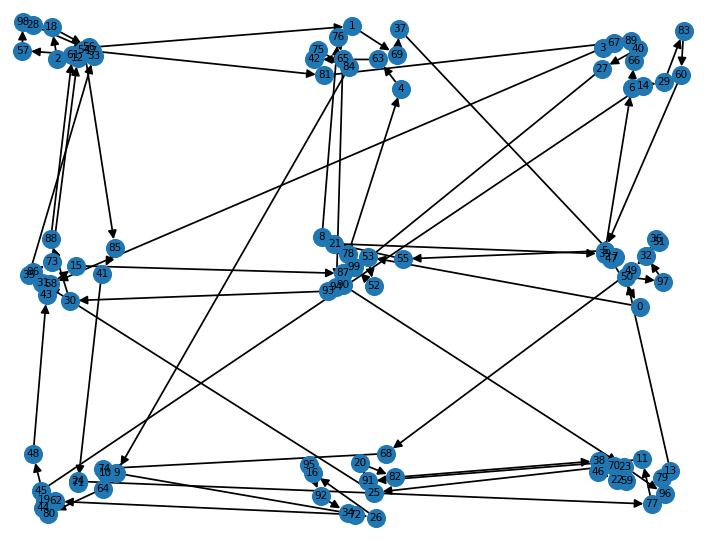


A tu jest zestawienie wyników:

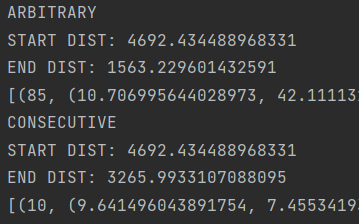


1. Temp\_rate = 0.95

Zamiana dwóch losowych punktów dała taki wynik:



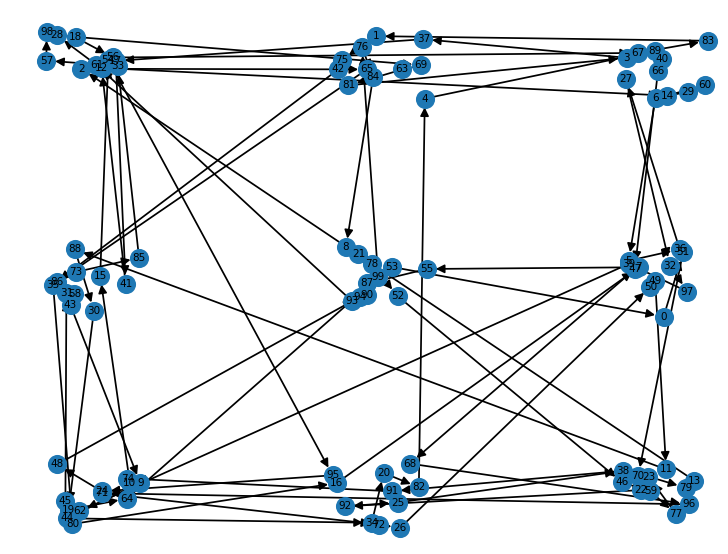
A ogólny wynik:



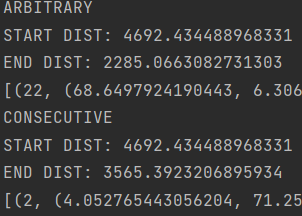
Jest to gorszy wynik niż wyżej.

1. Temp\_rate = 0.8

Wynik jest dość słaby.



Temperatura za szybko spadała i było za mało iteracji.



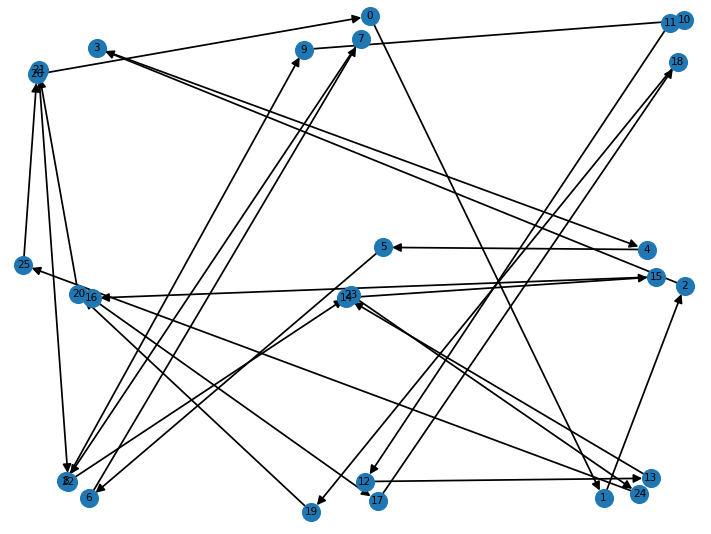
1. Zbadaj wpływ sposobu generacji sąsiedniego stanu (consecutive swap vs. arbitrary swap) oraz funkcji zmiany temperatury na zbieżność procesu optymalizacji.

Zamiana dwóch losowych punktów (arbitrary swap) daje dużo lepsze wyniki niż zamiana jedynie sąsiadów (consecutive swap). Może to być związane, że niektóre zamiany mogą znacznie zmniejszać sumaryczny dystans.

Natomiast zmiana funkcji zmiany temperatury również ma duży wpływ na uzyskiwany wynik. Czym wolniej temperatura maleje tym mamy większą szansę na znalezienie lepszego wyniku, gdyż będziemy wykonywać więcej skoków, które nam mogą psuć rezultat, ale dzięki takiemu zabiegowi mamy również szansę trafić na lepsze rozwiązanie.

1. Przedstaw wizualizację działania procedury minimalizującej funkcję celu.

Algorytm symulowanego wyżarzania został uruchomiony na 27 punktach wrzuconych do 9 grup.

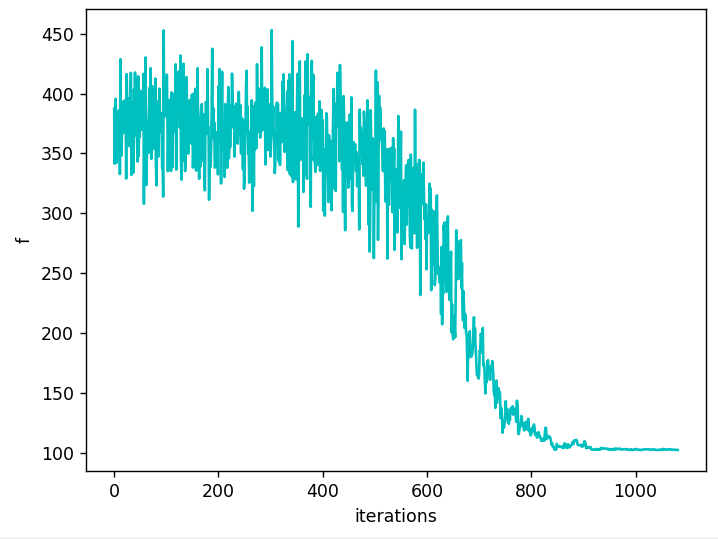


Rezultat jest bardzo zadowalający.

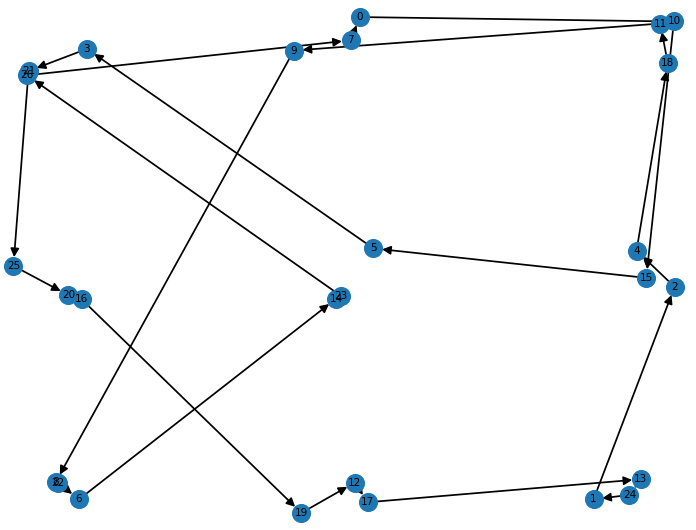
Obraz zawierający jazda na nartach, dzień

Opis wygenerowany automatycznie

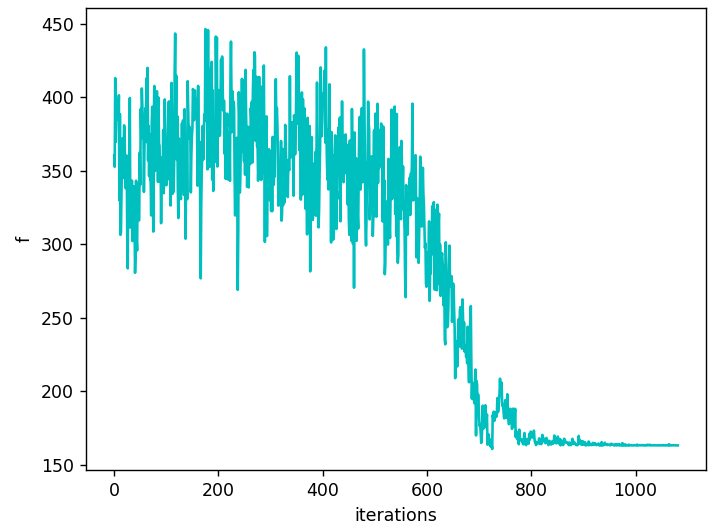
A wykres wartości funkcji optymalizowanej:



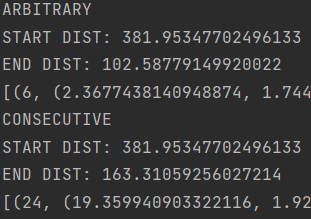
Znacznie gorszy rezultat otrzymaliśmy zamieniając jedynie połączone wierzchołki:



Z wykresem:



Tutaj jest zbiorcza informacja o wynikach:



# zadanie 2 – obraz binarny

Polecenie: Wygeneruj losowy obraz binarny o rozmiarze *n×n* i wybranej gęstości *\_* czarnych punktów

δ= 0*.*1*,* 0*.*3*,* 0*.*4. Korzystając z różnego typu sąsiedztwa (4-sasiadów, 8-sasiadów, 8-16-sasiadów) zaproponuj funkcje energii (np. w bliskiej odległości te same kolory przyciągają

się, a w dalszej odpychają się) i dokonaj jej minimalizacji za pomocą algorytmu symulowanego wyżarzania. W jaki sposób można generować stany sąsiednie? Jak różnią się uzyskane wyniki w zależności od rodzaju sąsiedztwa, wybranej funkcji energii i szybkości spadku temperatury?

Importujemy następujące biblioteki:

import math  
import matplotlib.pyplot as plt  
import numpy as np  
from random import randint, random  
from matplotlib import colors

Program jest uruchamiany poprzez poniższy kod.

if \_\_name\_\_ == "\_\_main\_\_":  
 n = 50  
 density = 0.1  
 neighbourhood = 3  
 image = generate\_binary\_image(n, density)  
 draw(image)  
  
 temp\_start = 5230  
 temp\_end = 0.1  
 temp\_iter = 20  
 temp\_rate = 0.99  
 solution(image, temp\_start, temp\_end, temp\_iter, temp\_rate,  
 neighbourhood)  
 draw(image)

Obraz generowany jest przez prostą funkcję.

def generate\_binary\_image(n, density):  
 return np.array([[random() < density for \_ in range(n)] for \_ in range(n)])

Obraz jest rysowany poprzez funkcję:

def draw(image):  
 n = len(image)  
 colormap = colors.ListedColormap(["white", "black"])  
 plt.figure(figsize=(7, 7))  
 plt.imshow(image, cmap=colormap)  
 plt.xlim([0, n - 1])  
 plt.ylim([0, n - 1])  
 plt.show()

Funkcja, która wprowadza naszą fizykę do życia.

def solution(points, temp\_start, temp\_end, temp\_iter, temp\_rate,  
 neighbourhood):  
 n = len(points)  
 best = f(points)  
 # best = f2(points, neighbourhood)  
 iterations = 0  
 x = []  
 y = []  
 while temp\_start > temp\_end:  
 for i in range(temp\_iter):  
 for \_ in range(n):  
 p1x = randint(neighbourhood, n - neighbourhood - 1)  
 p1y = randint(neighbourhood, n - neighbourhood - 1)  
 add\_x = randint(-neighbourhood, neighbourhood)  
 add\_y = randint(-neighbourhood, neighbourhood)  
 p2x = p1x + add\_x  
 p2y = p1y + add\_y  
 if (points[p1y, p1x] + points[p2y, p2x]) % 2 == 1:  
 points[p1y, p1x], points[p2y, p2x] = points[p2y, p2x],  
 points[p1y, p1x]  
 possible = f(points)  
 # possible = f2(points, neighbourhood)  
 if possible < best:  
 best = possible  
 else:  
 prob = math.e \*\* ((best - possible) / temp\_start)  
 check\_number = random()  
 if check\_number < prob:  
 best = possible  
 else:  
 points[p1y, p1x], points[p2y, p2x] = points[p2y, p2x],  
 points[p1y, p1x]  
  
 x.append(iterations)  
 y.append(best)  
 for i in range(temp\_iter):  
 temp\_start \*= temp\_rate  
 # temp\_start \*= temp\_rate  
 iterations += 1  
 print(temp\_start)  
  
 plt.plot(x, y, "c-")  
 plt.xlabel("iterations")  
 plt.ylabel("f")  
 plt.show()

Funkcja kosztu oblicza odległość wszystkich czarnych punktów od jakiejś prostej. Można też wybrać opcję liczącą odległość od konkretnego punktu.

def f(points):  
 energy = 0  
 n = len(points)  
 for i in range(n):  
 for j in range(n):  
 if points[i, j]:  
 energy += distance\_line(points, i, j)  
 # energy += distance\_point(points, i, j)  
 return energy

Poniżej funkcja oblicza odległość od zadanej prostej podanego punktu (zostało zawartych kilka przykładów funkcji). Druga natomiast odległość od konkretnego punktu.

def distance\_line(points, i, j):  
 n = len(points)  
 p1 = (0, 0)  
 p2 = (n - 1, n - 1)  
 p3 = (j, i)  
 # p1 = (23\*(n//33), 2\*n//20)  
 # p2 = (n//5, 4\*(n//5))  
 # p3 = (j, i)  
 energy = (abs((p2[1] - p1[1]) \* (p2[0] - p3[0]) - (p2[1] - p3[1]) \*  
 (p2[0] - p1[0])) / np.sqrt(  
 np.square(p2[1] - p1[1]) + np.square(p2[0] - p1[0])))  
 # energy = abs(np.cross(p2 - p1, p3 - p1) / np.linalg.norm(p2 - p1))  
  
 # energy = abs(2 \* (n // 3) - i)  
  
 # energy = abs(n // 5 - j)  
 return energy  
  
  
def distance\_point(points, i, j):  
 n = len(points)  
 p1 = [n // 2, n // 2]  
 p2 = [j, i]  
 energy = np.sqrt((p2[0] - p1[0]) \*\* 2 + (p2[1] - p1[1]) \*\* 2)  
 return energy

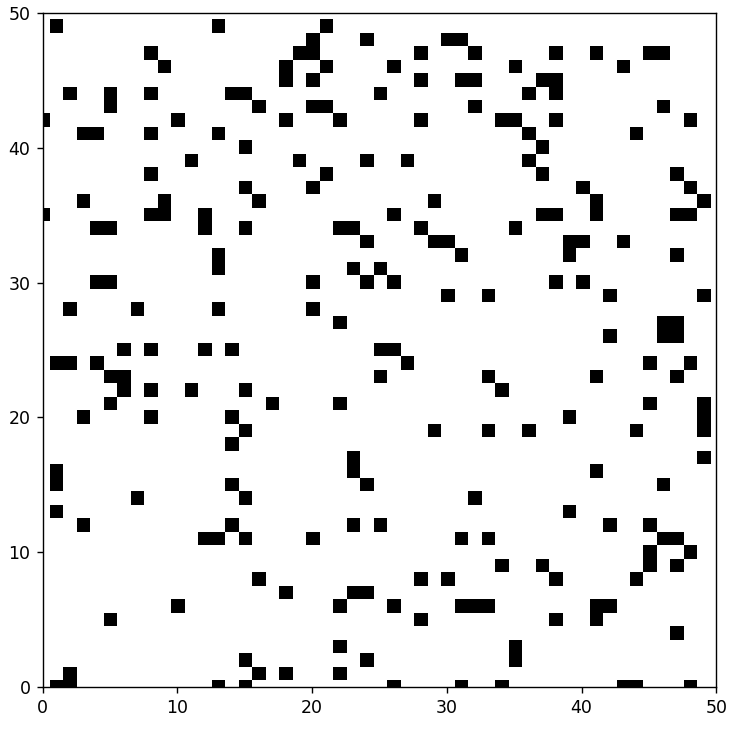
Teraz możemy przetestować naszą fizykę.

Dla n=50, δ=0.1 oraz poszukiwanie sąsiadów do 3 kratek różnicy. Parametry, z jakimi na ten moment są uruchamiane programy to:

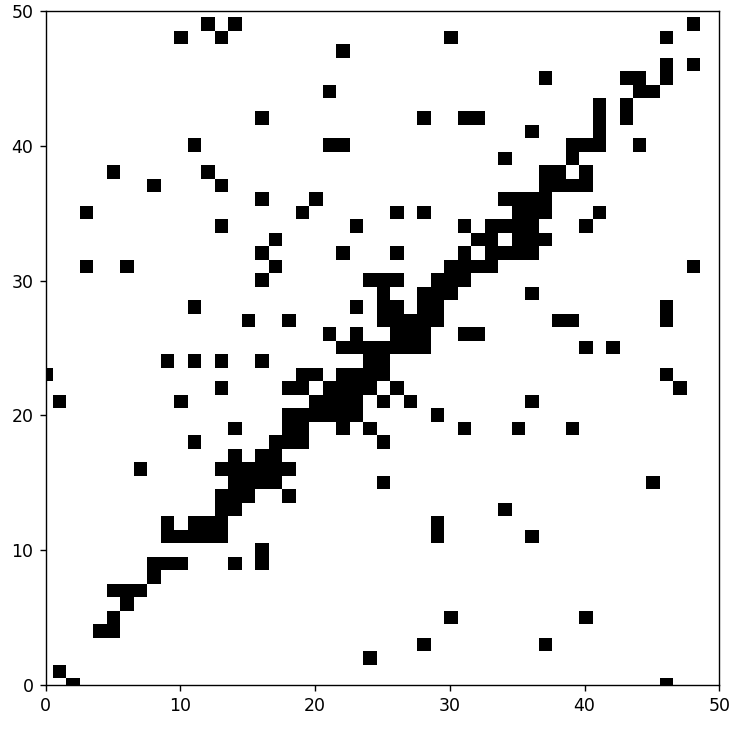
Obraz zawierający tekst

Opis wygenerowany automatycznie

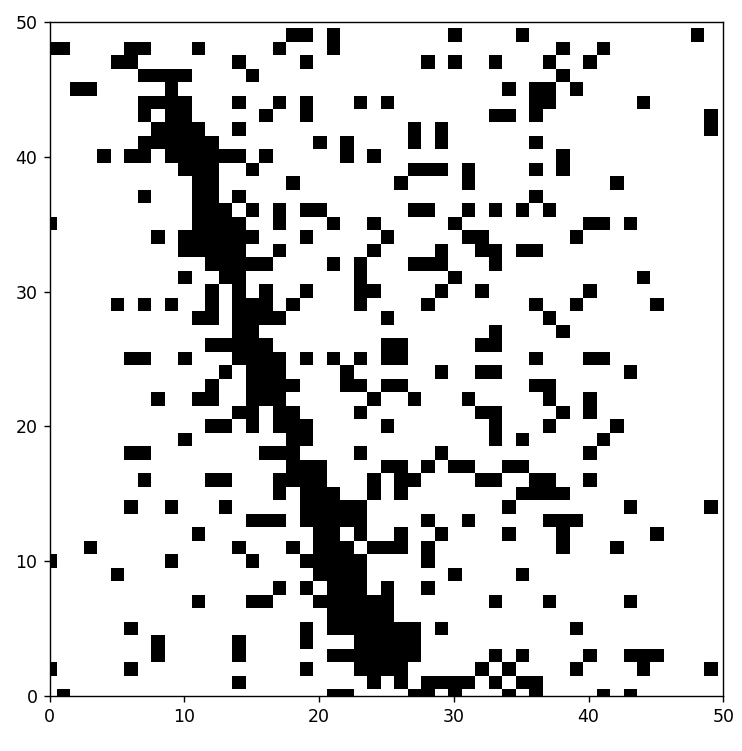
Poniżej został przedstawiony przykładowy rozkład punktów.



Po zakończeniu algorytmu otrzymaliśmy punkty bardzo blisko przekątnej, tak jak chcieliśmy.

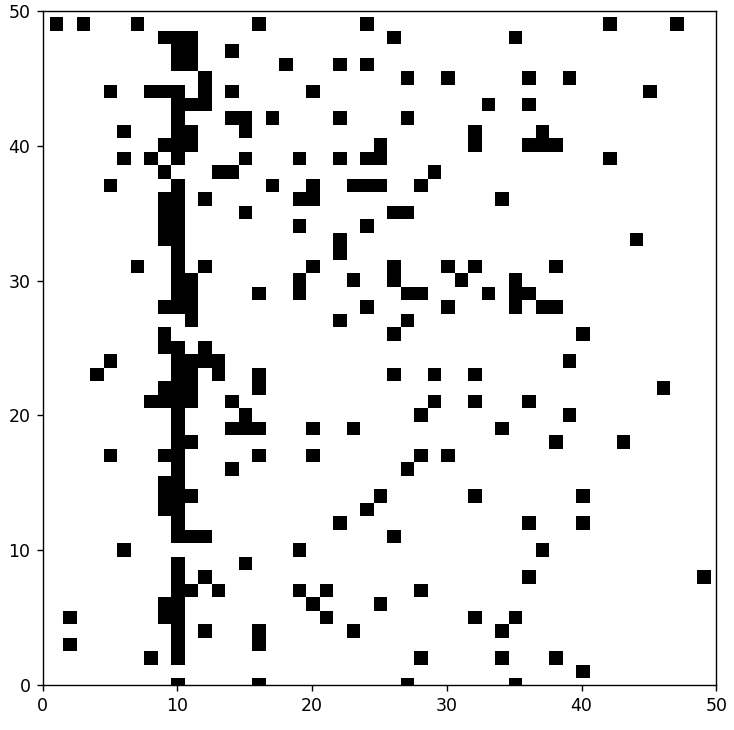


Teraz przybliżać będziemy do innej prostej. Bierzemy n=50, δ=0.2 oraz sąsiedztwo do 2 kratek.



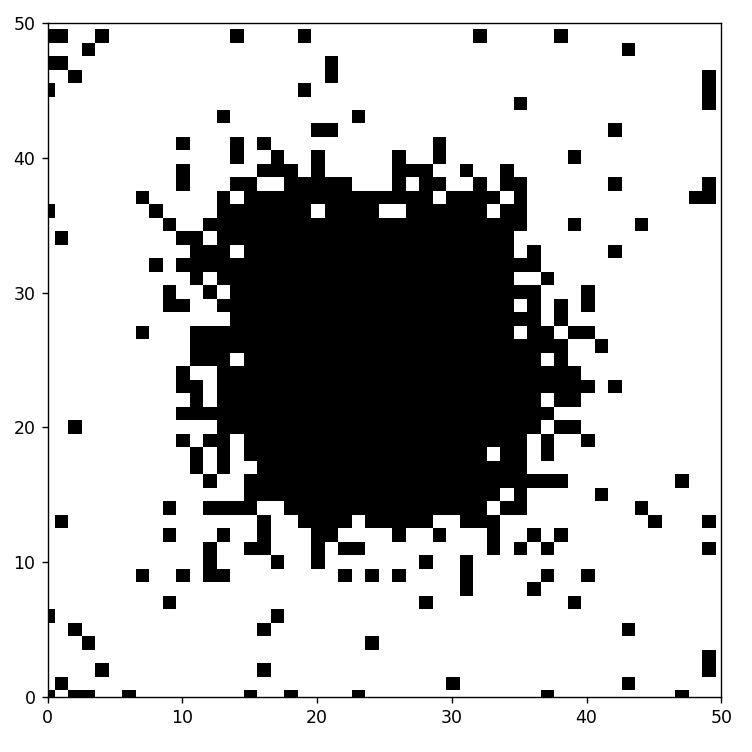
Funkcja ma ciekawy kształt i widzimy, że punkty koło niej się gromadzą.

Inny ciekawy przykład, gdy prosta przyciąga punkty.



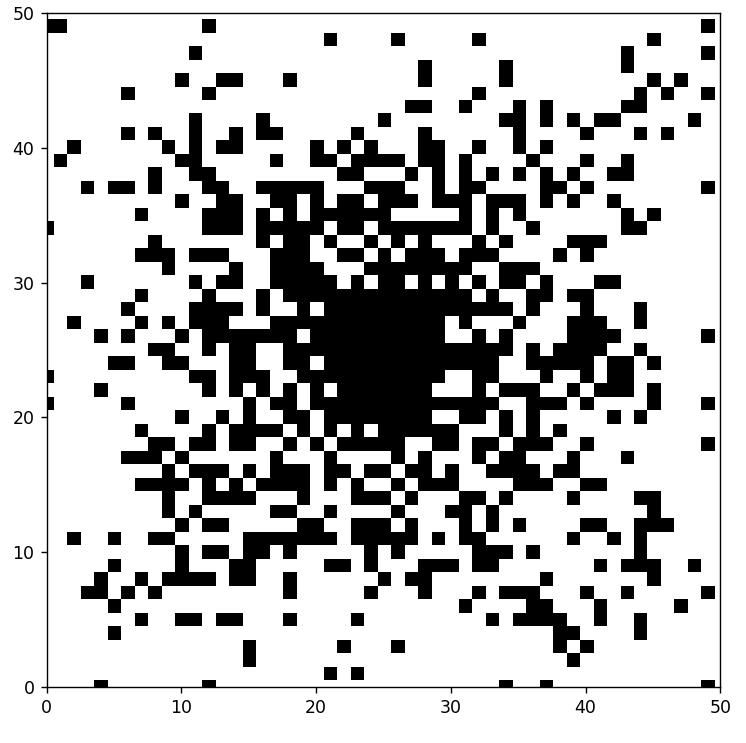
Teraz przetestujemy gromadzenie się wokół punktu.

Parametry: n=50, δ=0.3 oraz zasięg pięciu sąsiadów w każdą stronę.



Udało się ładnie zgromadzić punkty.

Teraz zmienimy możliwość przeskoku punktu z pięciu do dwóch pozycji.

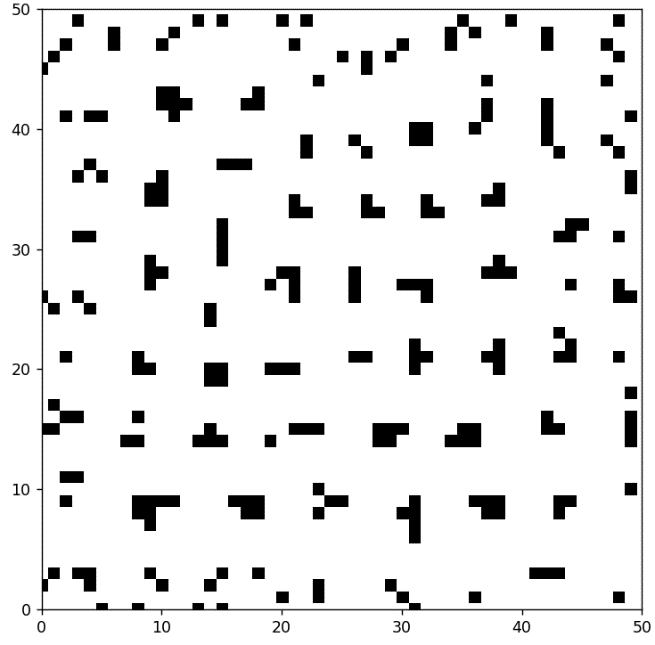


Uzyskany wynik jest znacznie gorszy.

Została dopisana funkcja, która oblicza energię punktu w zależności od odległości od sąsiadów.

def f2(points, neighbourhood):  
 energy = 0  
 n = len(points)  
 for i in range(n):  
 for j in range(n):  
 if points[i, j]:  
 energy += neighbourhood\_distance(points, i, j, neighbourhood)  
 return energy  
  
  
def neighbourhood\_distance(points, i, j, neighbourhood):  
 n = len(points)  
 energy = 0  
 p2 = [j, i]  
 for y in range(i - neighbourhood, i + neighbourhood):  
 if 0 <= y < n:  
 for x in range(j - neighbourhood, j + neighbourhood):  
 if 0 <= x < n and points[y, x]:  
 p1 = [x, y]  
 energy += np.sqrt((p2[0] - p1[0]) \*\* 2 + (p2[1] - p1[1]) \*\* 2)  
 return energy

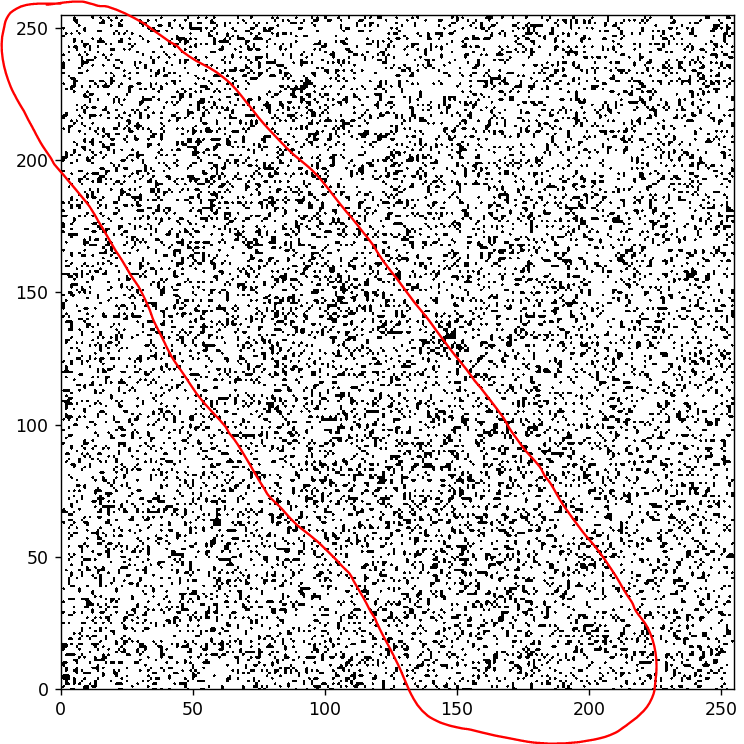
Oto dwa przykładowe wywołania

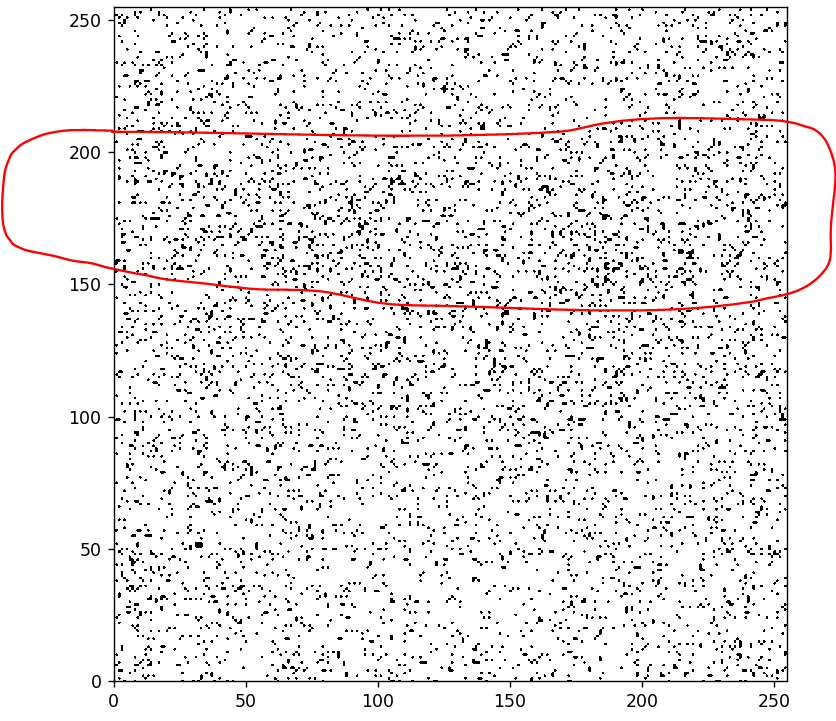


Można dostrzec formowanie się punktów w grupy.

Uzyskane wyniki są znacznie gorsze, jeśli temperatura spada za szybko (współczynnik temperatury jest niski). Tak samo jest, gdy weźmiemy stosunkowo niewielkie sąsiedztwo, w którym punkty mogą się zamieniać. Dlatego trzeba przemyśleć sprawę parametrów przed uruchomieniem algorytmu.

Został też napisany podobny program do tego co został przedstawiony powyżej, ale działa on dla większej ilości punktów, lecz wyniki nie są tam widoczne jak w opisanych wcześniej przypadkach. Nowe podejście nie sprawdza nowej wartości funkcji po każdej zamianie, lecz zamienia najpierw jakąś liczbę punktów, a później ewentualnie ją cofa. Sprawia to, że możemy szybciej obliczyć podany problem dla większego rozmiaru macierzy, lecz wynik nie jest lepszy od uzyskanego w przypadkach ukazanych wcześniej. Program zapisany w pliku 2\_2.py.





# zadanie 3 – sudoku

Polecenie: Napisz program poszukujący rozwiązania łamigłówki Sudoku za pomocą symulowanego

wyżarzania. Plansza 9 *×* 9 ma zostać wczytana z pliku tekstowego, w którym pola puste

zaznaczone są znakiem x. Jako funkcje kosztu przyjmij sumę powtórzeń cyfr występujących

w wierszach bloku 9 *×* 9, kolumnach bloku 9 *×* 9 oraz blokach 3 *×* 3. Zaproponuj

metodę generacji stanu sąsiedniego. Przedstaw zależność liczby iteracji algorytmu od

liczby pustych miejsc na planszy. Czy Twój program jest w stanie znaleźć poprawne rozwiązanie dla każdej z testowanych konfiguracji wejściowych?

Na początek przydatne importy.

import math  
from random import randint, random  
from matplotlib import pyplot as plt

Funkcja pobierająca plansze Sudoku do rozwiązania.

def getData(file\_name, separator):  
 board = []  
 f = open(file\_name, "r")  
 for line in f:  
 row = line.strip().split(separator)  
 for i in range(len(row)):  
 if row[i] != "x":  
 row[i] = int(row[i])  
 board.append(row)  
 f.close()  
 return board

Funkcja wypełniająca mniejsze plansze 3x3.

def fill\_empty\_fields\_3x3(board):  
 n = len(board)  
 for i in range(n // 3):  
 for j in range(n // 3):  
 for k in range(1, n + 1):  
 includes, y, x = includes\_3x3(board, i, j, k)  
 if not includes:  
 board[y][x] = k

Funkcja mówiąca czy dana liczba jest w planszy 3x3 i zwracająca jedno z pustych miejsc.

def includes\_3x3(board, i, j, number):  
 n = len(board)  
 emptyY, emptyX = None, None  
 found = False  
 test\_to\_pass = random()  
 for k in range(n):  
 y = i \* 3 + k // 3  
 x = j \* 3 + k % 3  
 if board[y][x] == number:  
 return True, emptyY, emptyX  
 elif board[y][x] == "x":  
 if found:  
 test = random()  
 if test > test\_to\_pass:  
 emptyY, emptyX = y, x  
 else:  
 emptyY, emptyX = y, x  
 found = True  
 return False, emptyY, emptyX

Teraz główna funkcja próbująca rozwiązać Sudoku.

def solution(original\_board, board, empty\_fields, temp\_start, temp\_end,  
 temp\_iter, temp\_rate):  
 n = len(board)  
 best = count\_repeats(board)  
 iterations = 0  
 x\_data = []  
 y\_data = []  
 solved = False  
 while temp\_start > temp\_end:  
 for i in range(temp\_iter):  
 y = randint(0, n // 3 - 1)  
 x = randint(0, n // 3 - 1)  
 possible\_fields = empty\_fields[y][x]  
 if possible\_fields >= 2:  
 p1, p2 = choose\_random\_fields(original\_board, y, x,  
 possible\_fields)  
 board[p1[0]][p1[1]], board[p2[0]][p2[1]] = board[p2[0]][p2[1]],  
 board[p1[0]][p1[1]]  
 possible = count\_repeats(board)  
 if possible < best:  
 best = possible  
 if best == 0:  
 solved = True  
 break  
 else:  
 prob = math.e \*\* ((best - possible) / temp\_start)  
 check\_number = random()  
 if check\_number < prob:  
 best = possible  
 else:  
 board[p1[0]][p1[1]], board[p2[0]][p2[1]] = board[p2[0]][p2[1]],  
 board[p1[0]][p1[1]]  
 x\_data.append(iterations)  
 y\_data.append(best)  
 temp\_start \*= temp\_rate  
 iterations += 1  
 if solved:  
 break  
 plt.plot(x\_data, y\_data, "c-")  
 plt.xlabel("iterations")  
 plt.ylabel("f")  
 plt.show()  
 return solved

Poniższa funkcja sprawdza, ile jest miejsc możliwych do zamiany w obrębie danej planszy 3x3, czyli ile znaków „x” było w niej wprowadzone.

def count\_possible\_fields(original\_board, i, j):  
 n = len(original\_board)  
 count = 0  
 for k in range(n):  
 y = i \* 3 + k // 3  
 x = j \* 3 + k % 3  
 if original\_board[y][x] == "x":  
 count += 1  
 return count

Funkcja „count\_repeats” jest funkcją kosztu, gdzie liczone są powtórzenia w wierszach i kolumnach. W planszach 3x3 nie trzeba liczyć powtórzeń, gdyż każda z nich zawiera cyfry bez powtórzeń.

def count\_repeats(board):  
 repeated = 0  
 repeated += repeats\_in\_rows(board)  
 repeated += repeats\_in\_columns(board)  
 # repeated += repeats\_in\_3x3(board) # always 0 here because 3x3 board is filled with no repeats and numbers arent swapped between 3x3 boards  
 return repeated  
  
  
def repeats\_in\_rows(board):  
 n = len(board)  
 repeated = 0  
 count\_tab = [0 for \_ in range(n)]  
 for line in board:  
 for i in range(n):  
 count\_tab[i] = 0  
 for element in line:  
 if element != "x":  
 count\_tab[element - 1] += 1  
 for element in count\_tab:  
 if element > 1:  
 repeated += element - 1  
 return repeated  
  
  
def repeats\_in\_columns(board):  
 n = len(board)  
 repeated = 0  
 count\_tab = [0 for \_ in range(n)]  
 for i in range(n):  
 for j in range(n):  
 count\_tab[j] = 0  
 for j in range(n):  
 if board[j][i] != "x":  
 count\_tab[board[j][i] - 1] += 1  
 for element in count\_tab:  
 if element > 1:  
 repeated += element - 1  
 return repeated  
  
  
def repeats\_in\_3x3(board):  
 n = len(board)  
 repeated = 0  
 count\_tab = [0 for \_ in range(n)]  
 for i in range(n // 3):  
 for j in range(n // 3):  
 for k in range(n):  
 count\_tab[k] = 0  
 for k in range(n):  
 y = i \* 3 + k // 3  
 x = j \* 3 + k % 3  
 if board[y][x] != "x":  
 count\_tab[board[y][x] - 1] += 1  
 for element in count\_tab:  
 if element > 1:  
 repeated += element - 1  
 return repeated

Jeśli chcemy wybrać dwa losowe punkty do zamiany w obrębie jednej planszy 3x3, to użyjemy poniższej funkcji.

def choose\_random\_fields(original\_board, i, j, possible\_fields):  
 n = len(original\_board)  
 first = randint(0, possible\_fields - 1)  
 second = randint(0, possible\_fields - 1)  
 while first == second:  
 second = randint(0, possible\_fields - 1)  
 count = 0  
 if first > second:  
 first, second = second, first  
 p1Y, p1X = None, None  
 p2Y, p2X = None, None  
 for k in range(n):  
 y = i \* 3 + k // 3  
 x = j \* 3 + k % 3  
 if original\_board[y][x] == "x":  
 if count == first:  
 p1Y, p1X = y, x  
 elif count == second:  
 p2Y, p2X = y, x  
 count += 1  
 return (p1Y, p1X), (p2Y, p2X)

Oto jak wygląda przykładowe uruchomienie programu:

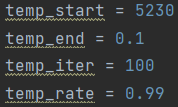
if \_\_name\_\_ == "\_\_main\_\_":  
 temp\_start = 5230  
 temp\_end = 0.1  
 temp\_iter = 100  
 temp\_rate = 0.99  
 file\_name = "sudoku\_easy.txt"  
 separator = ","  
 original\_board = getData(file\_name, separator)  
 empty\_fields = [[count\_possible\_fields(original\_board, i, j) for j in range(3)] for i in range(3)]  
 n = len(original\_board)  
 board = [[original\_board[i][j] for j in range(n)] for i in range(n)]  
 fill\_empty\_fields\_3x3(board)  
 for line in original\_board:  
 print(line)  
 print("FILLED")  
 for line in board:  
 print(line)  
 found\_solution = solution(original\_board, board, empty\_fields, temp\_start, temp\_end, temp\_iter, temp\_rate)  
 print("SOLUTION")  
 for line in board:  
 print(line)  
 print("SOLVED:", found\_solution)

W tym programie stan sąsiedni jest generowany przez zamianę dwóch losowych cyfr na obszarze jednej planszy 3x3. Można by zrobić podobny program generujący stan sąsiedni zamieniając elementy w obszarze kolumny czy wiersza.

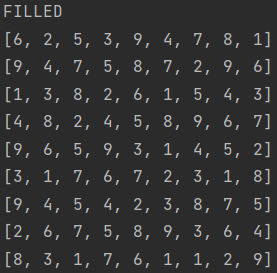
Dla łatwego Sudoku, rozwiązanie da się w łatwy sposób znaleźć. Na przykład wywołując powyższy program na planszy:

6,2,x,3,9,x,x,x,1  
x,4,x,x,x,x,2,9,x  
1,x,8,x,6,x,x,x,x  
4,x,2,x,x,8,9,x,x  
x,x,x,9,x,1,4,x,2  
3,1,x,x,7,x,x,x,8  
9,x,x,x,2,3,8,x,5  
2,6,x,5,8,x,3,x,x  
8,3,x,x,x,x,1,2,9

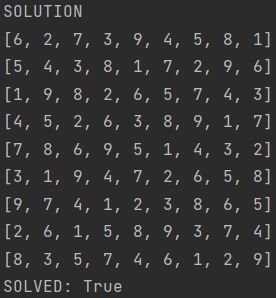
Z poniższymi parametrami:



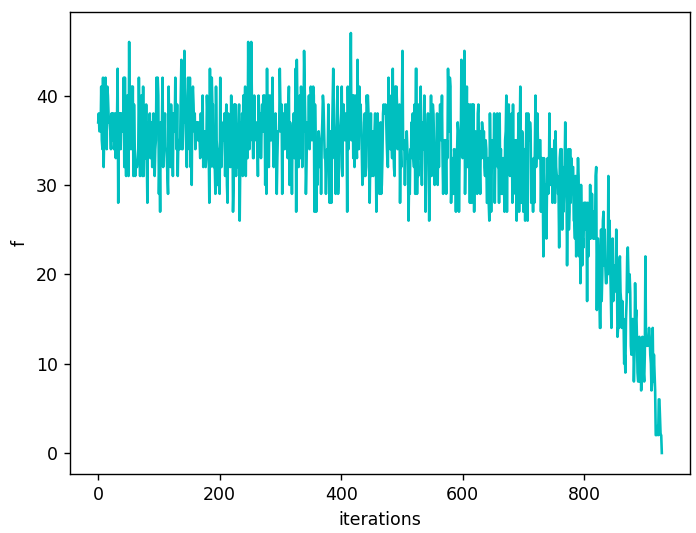
Nasza plansza została wypełniona w taki sposób:



Udało się rozwiązać podany przykład.



Wykres prezentuje się następująco.



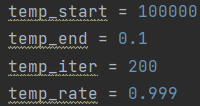
Otrzymamy następujący wynik.

Program został puszczony również dla poziomu ekspert i znalazł rozwiązanie (osobiście 35 min próbuję ułożyć). Choć trzeba było dać takie parametry, aby długo się wykonywał.

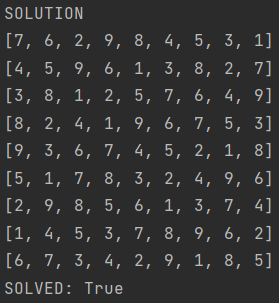
Plansza:

7,6,x,x,x,x,x,x,x  
x,5,x,x,x,x,8,2,x  
x,x,x,2,x,7,x,4,x  
x,x,x,x,9,6,7,x,x  
x,3,x,x,x,x,x,1,x  
x,x,x,8,x,x,x,x,x  
2,x,x,x,x,1,3,x,x  
1,x,x,x,x,x,9,6,x  
x,7,x,x,x,9,x,x,5

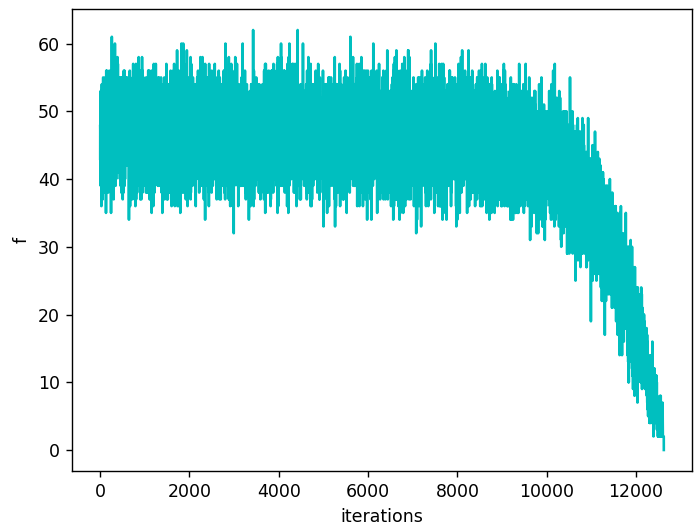
Parametry wywołania:



Rozwiązanie:

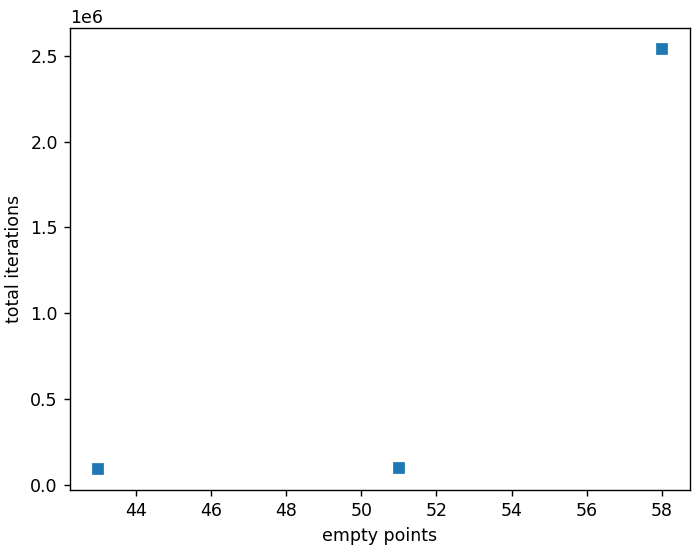


Wykres wartości funkcji od liczby iteracji:



Należy zaznaczyć, że dla trudnych planszy rozwiązanie nie zawsze zostaje znalezione.

Jeśli chodzi o zależność liczby iteracji od liczby pustych miejsc to zostały zebrane 3 informacje dla poziomów easy (43 -> 92328), medium (51 -> 99343), ekspert (58 -> 2540000).



Także przeskok jest dość spory, dlatego lepiej wiedzieć, ile miejsc jest wolnych, żeby można było dobrać odpowiednie wartości parametrów.