Metody Obliczeniowe w Nauce i Technice

Laboratorium 4

Wojciech Łącki

SPIS TREŚCI

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
```

a) 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.

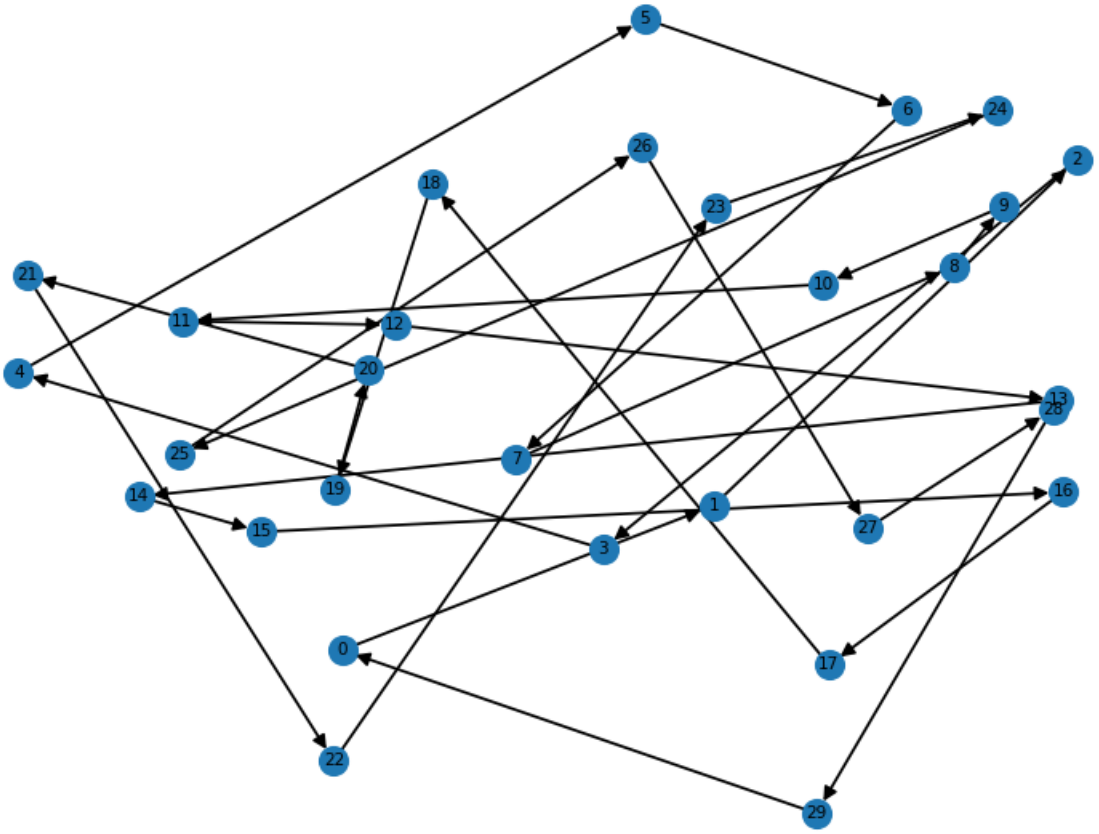
```
def current_solution_distance(points):
  n = len(points)
  sum = 0
  for i in range(n):
    sum += distance(points[i - 1], points[i])
  return sum


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”).

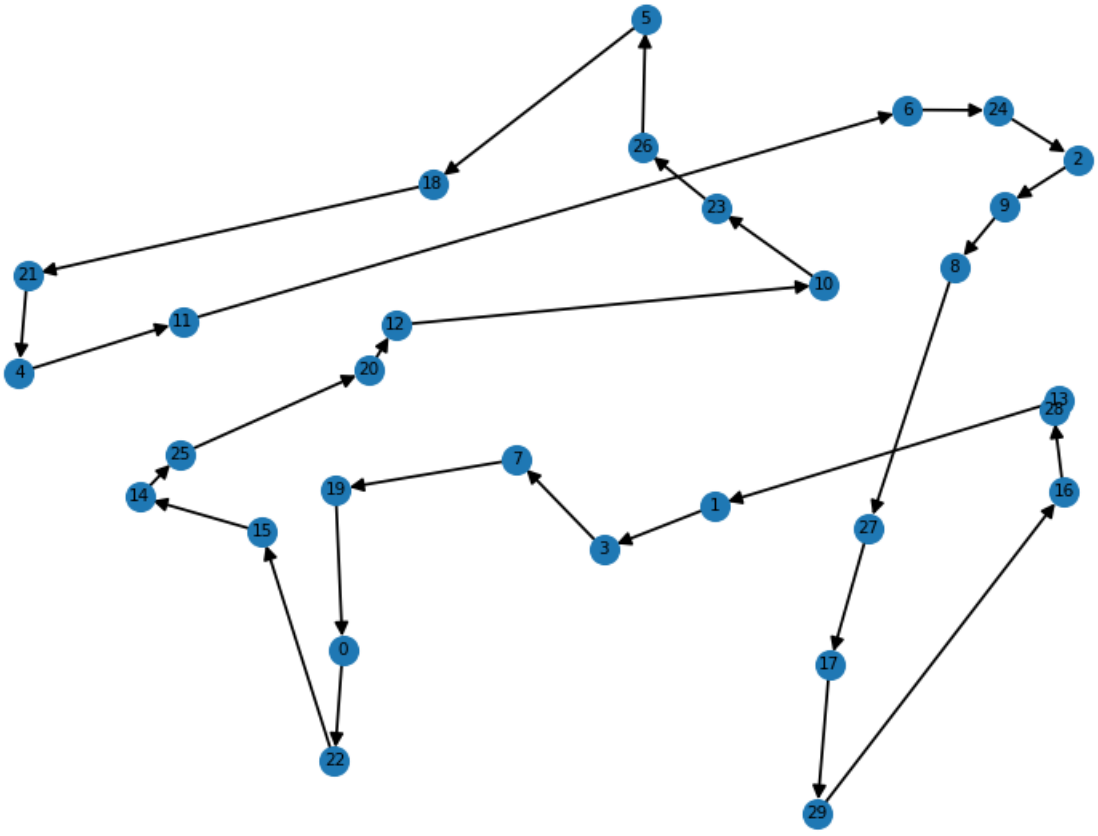
```
if __name__ == "__main__":
  n = 40
  temp_start = 5230
  temp_end = 0.1
  temp_iter = 20
  temp_rate = 0.99
  points = generate_random(n)
  # points = generate_uniform(n)
  # loc=2
  # scale = 4.3
  # points = generate_normal(n,loc,scale)
  # k = 9
  # points = generate_groups(n, k)
  graph = create_directed_graph(points)
  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)
```

I. Rozład jednostajny, n = 30

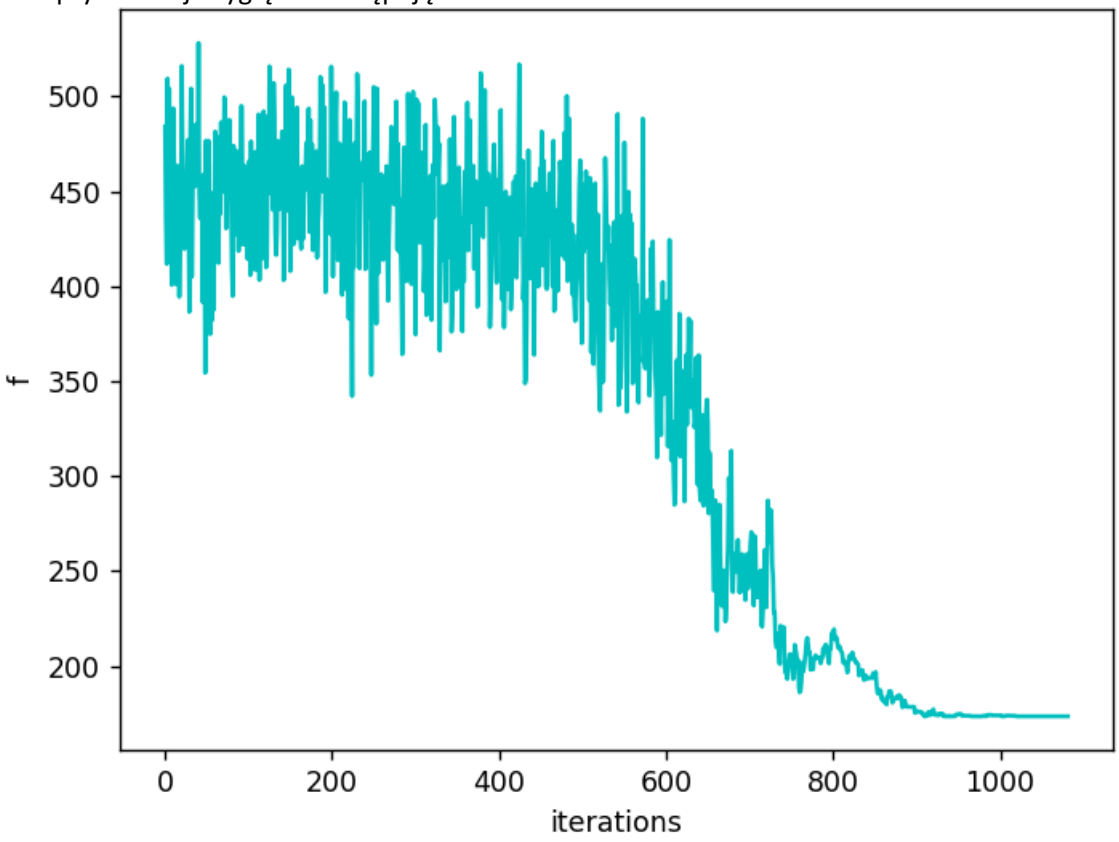
Wygenerowana trasa:



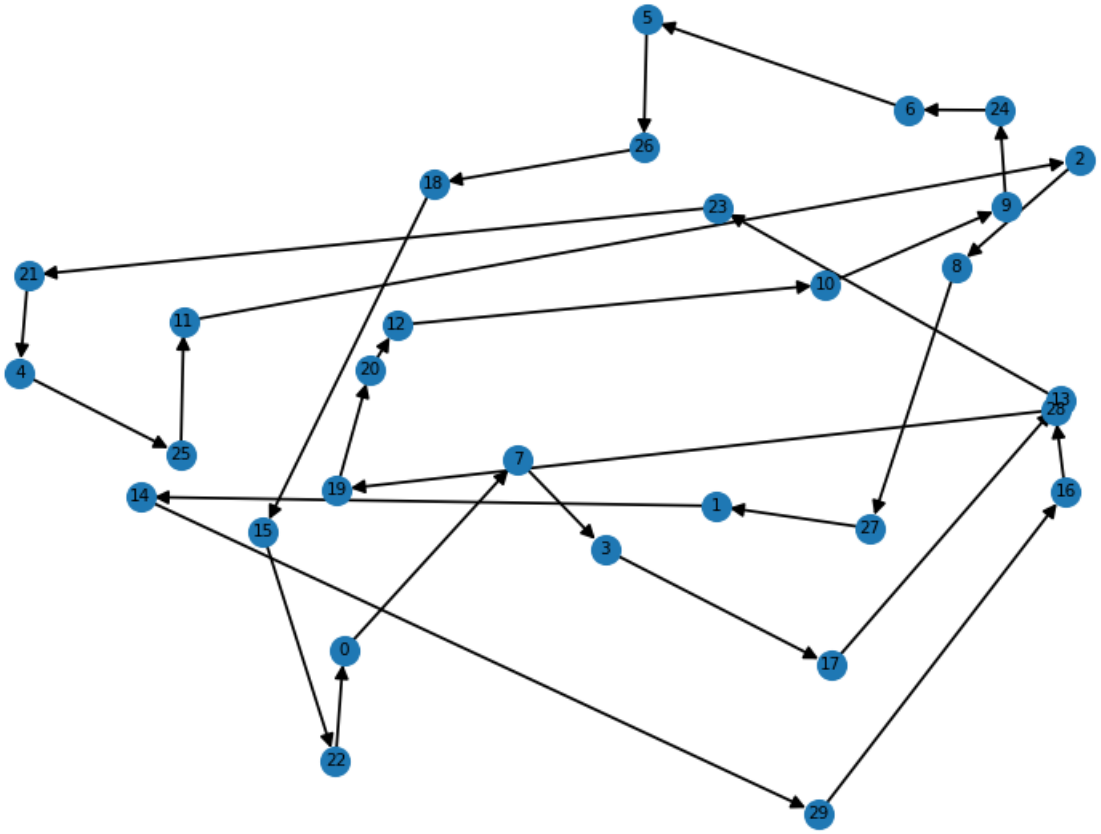
Uzyskana trasa przy losowej zamianie:



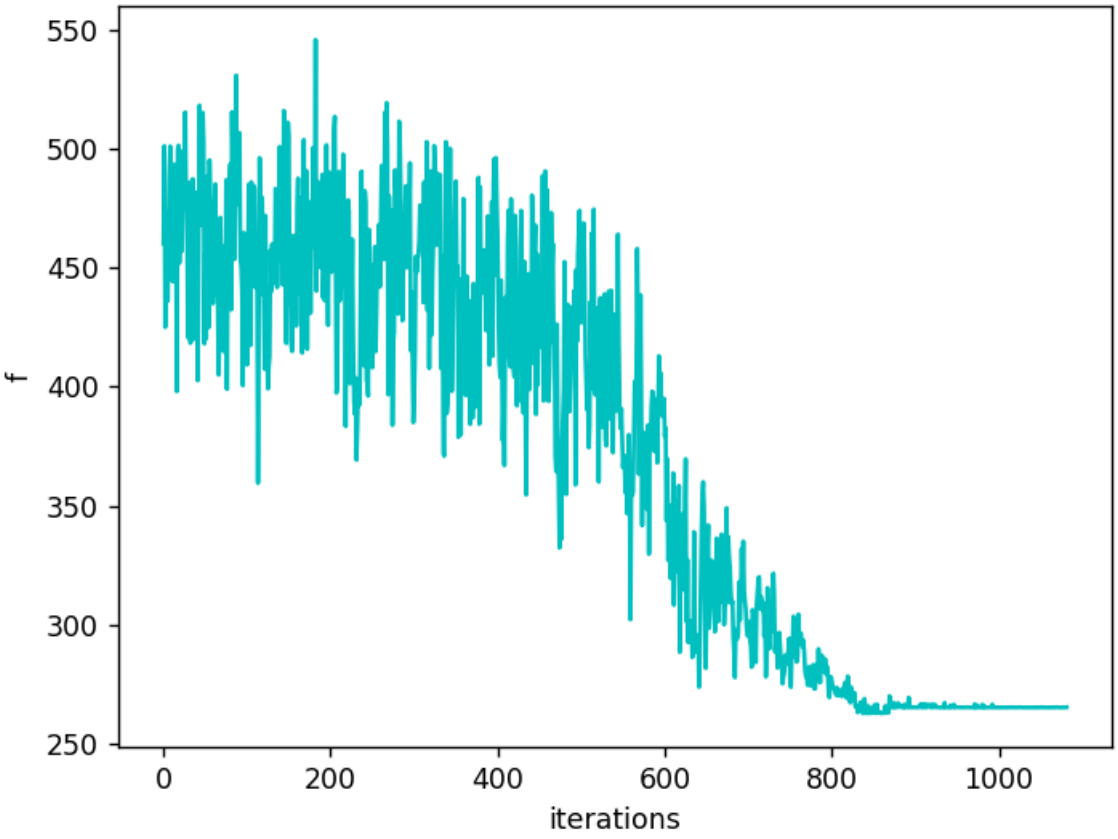
Wykres optymalizacji wyglądał następująco:



Natomiast przy zamianie jedynie sąsiadów:



A jego wykres optymalizacji:

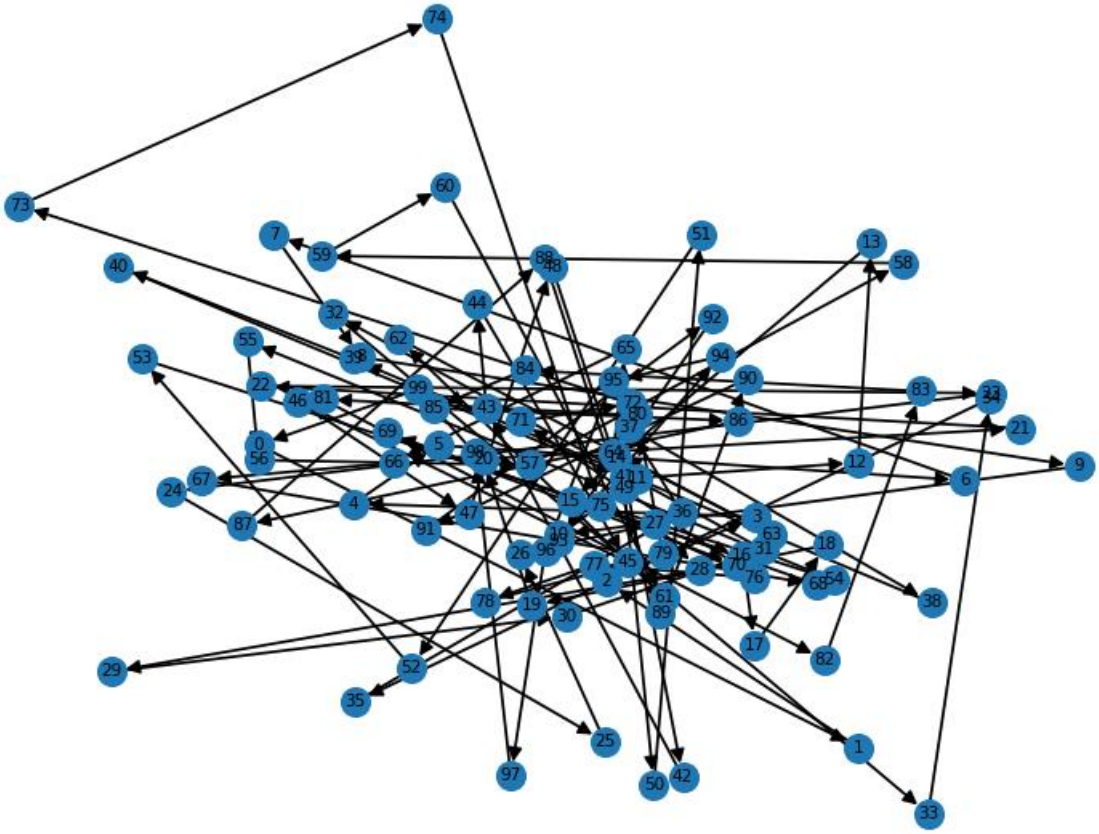


Ogólny wynik jest przedstawiony poniżej. Lepsza wyszła zamiana losowych punktów.

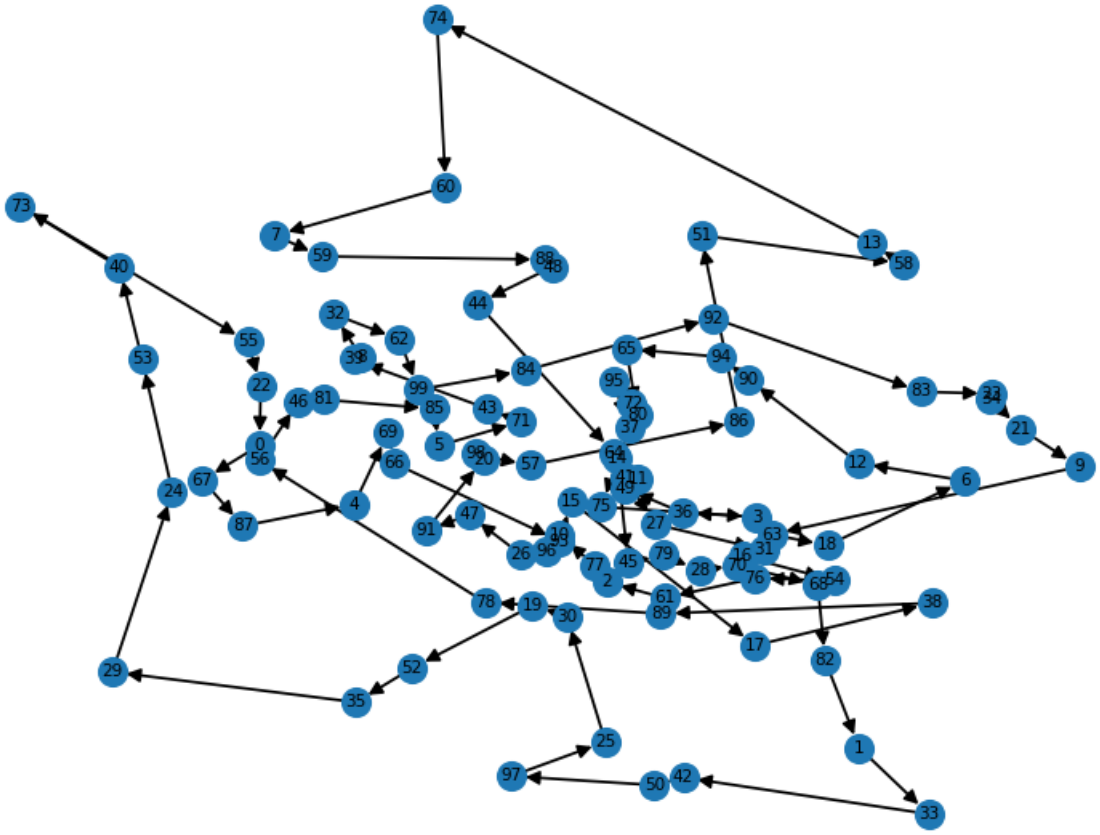
```
ARBITRARY
START DIST: 420.98994434411424
END DIST: 173.5993475635339
[(10, (22.99763256757222, 20.28778408002431))
CONSECUTIVE
START DIST: 420.98994434411424
END DIST: 265.62455718661965
[(1, (20.033858828261717, 12.266696473582535)
```

II. Rozkład normalny, n = 100

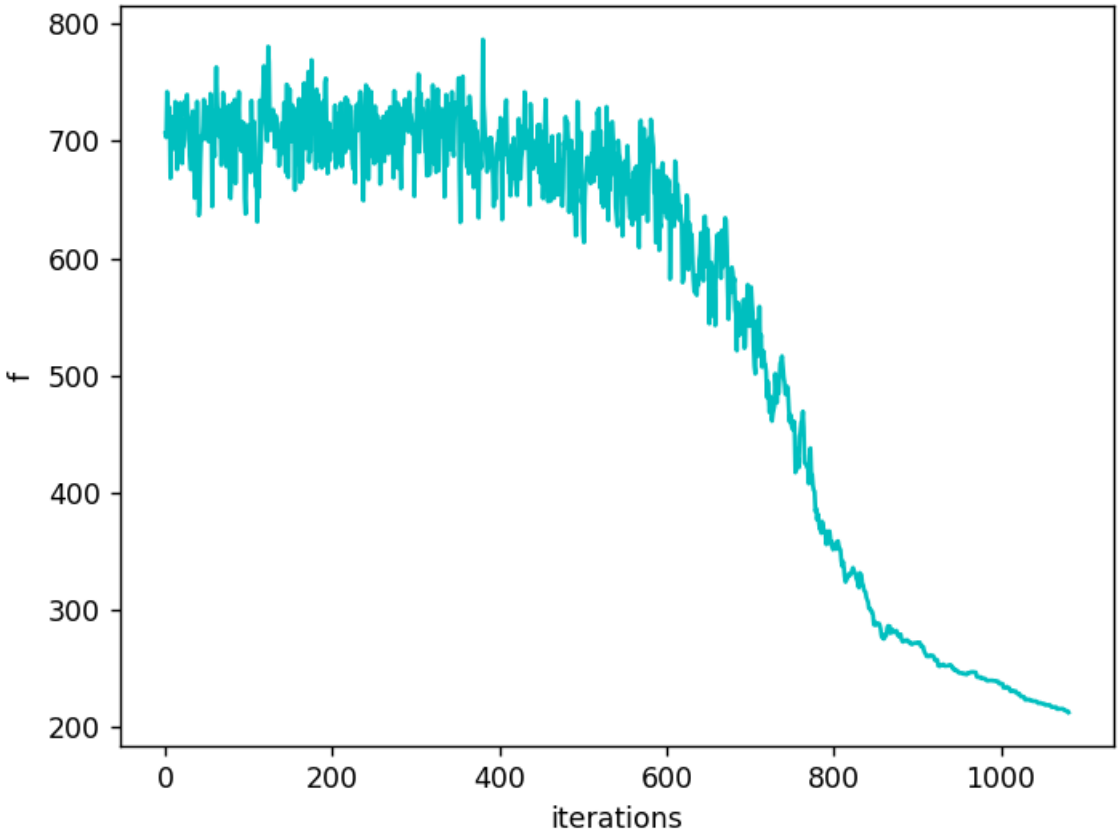
Zaczęło się od:



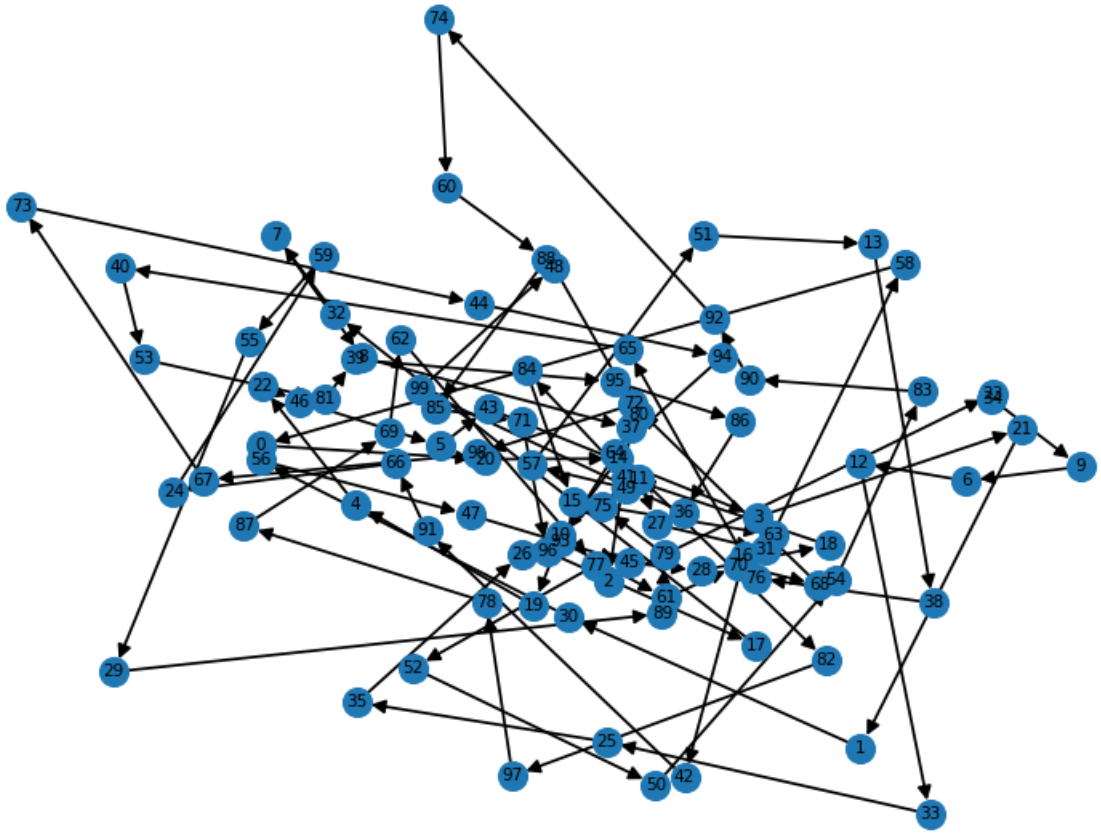
W wyniku zamiany losowej punktów otrzymaliśmy:



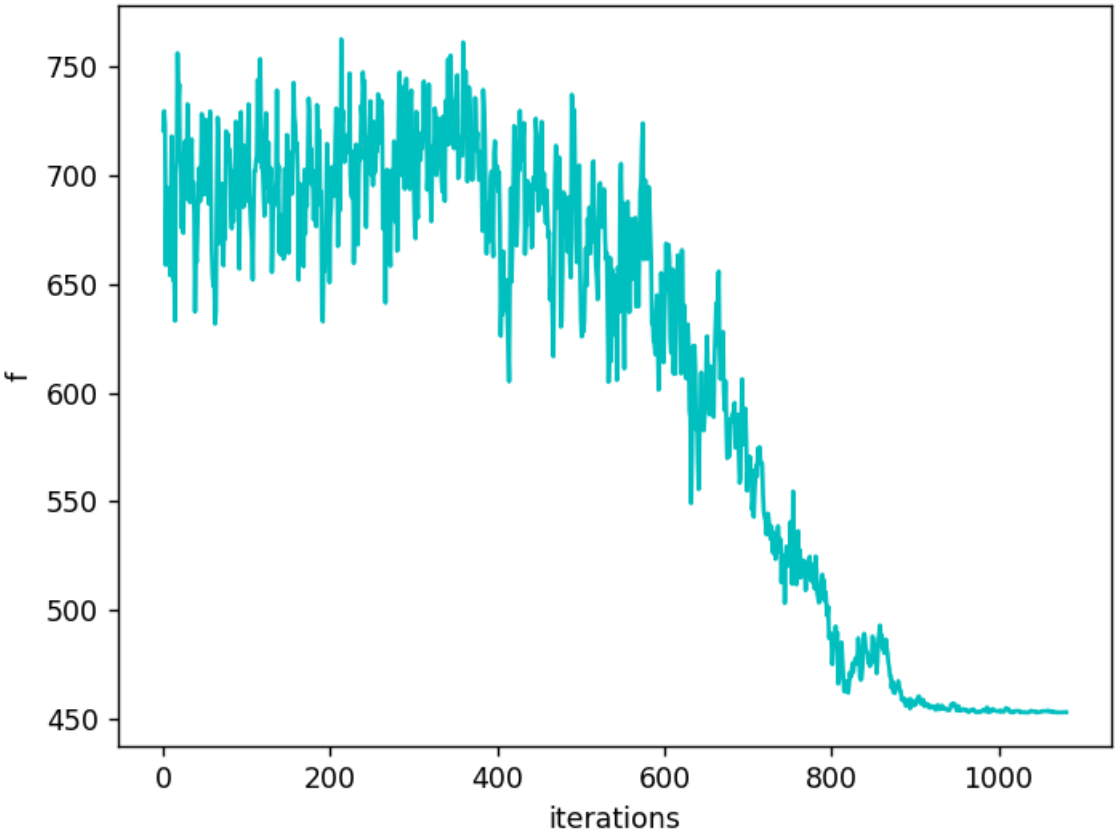
Wykres wartości funkcji dystansu od iteracji:



A zamieniając sąsiadów:



Z wykresem:

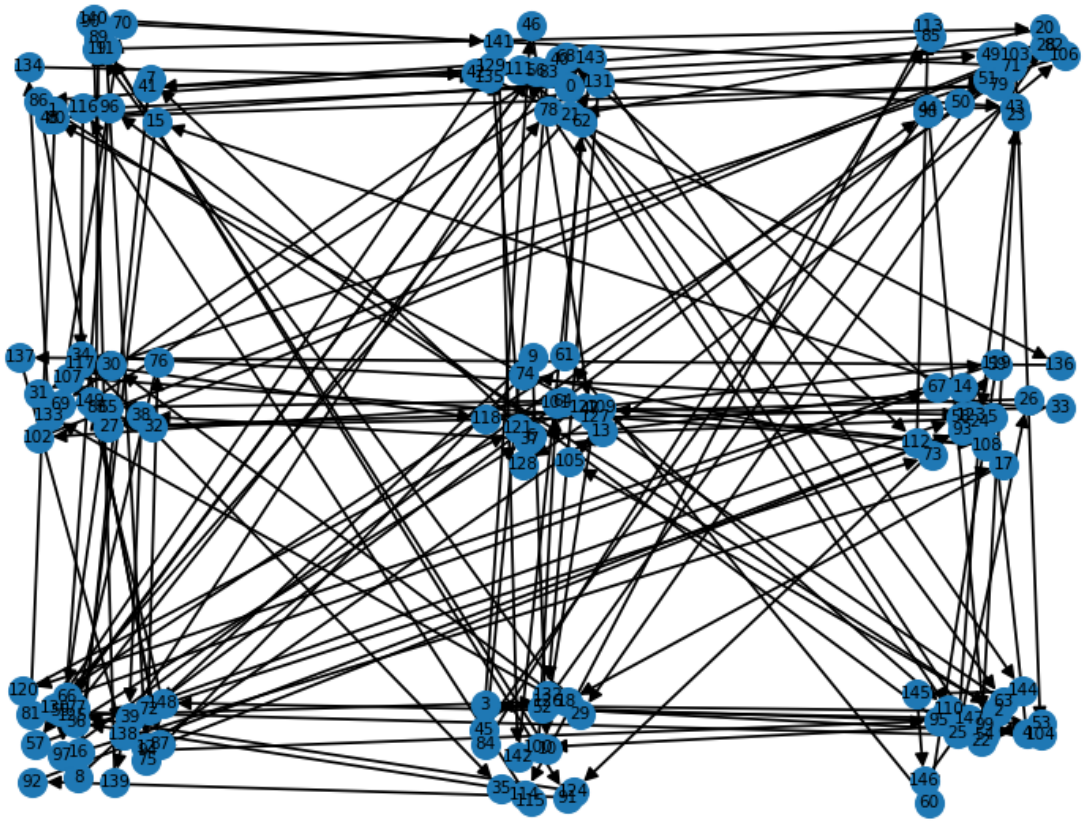


Rezultat:

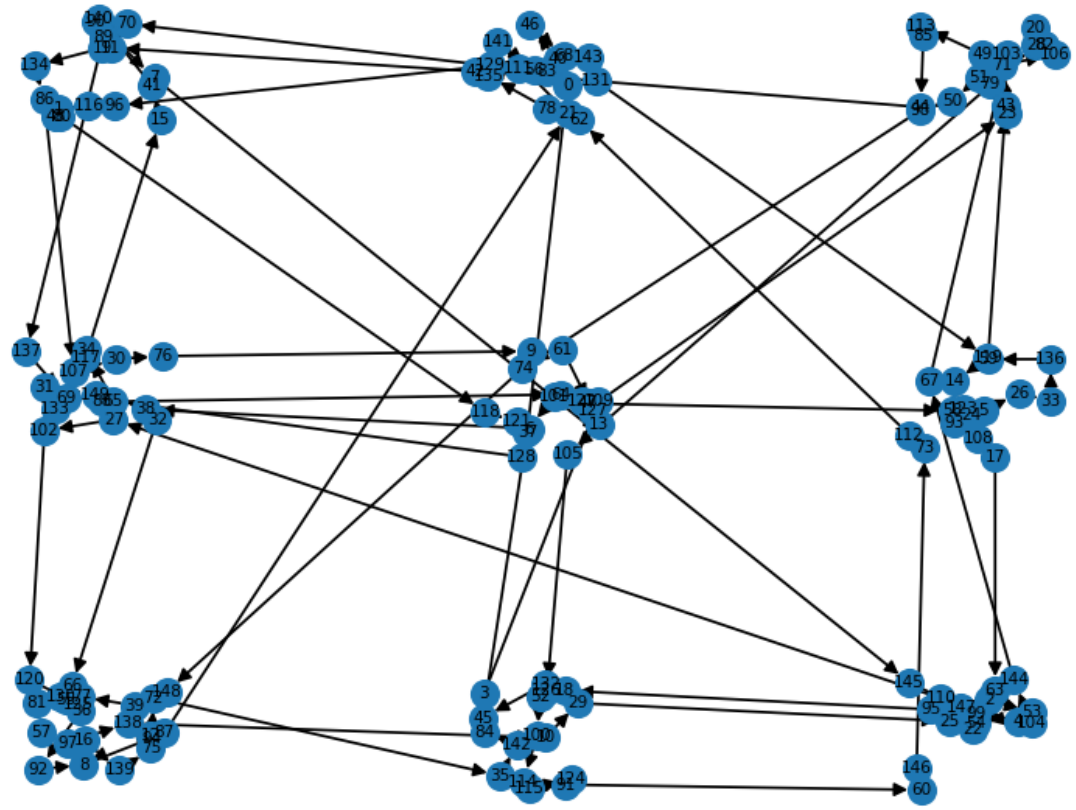
```
ARBITRARY
START DIST: 734.0481584926987
END DIST: 212.3012029824647
[(93, (49.79461721696447, 48.1
CONSECUTIVE
START DIST: 734.0481584926987
END DIST: 453.1236672110403
[(17, (53.06957755166154, 45.6
```

Ponownie lepsza okazała się zamiana losowych punktów.

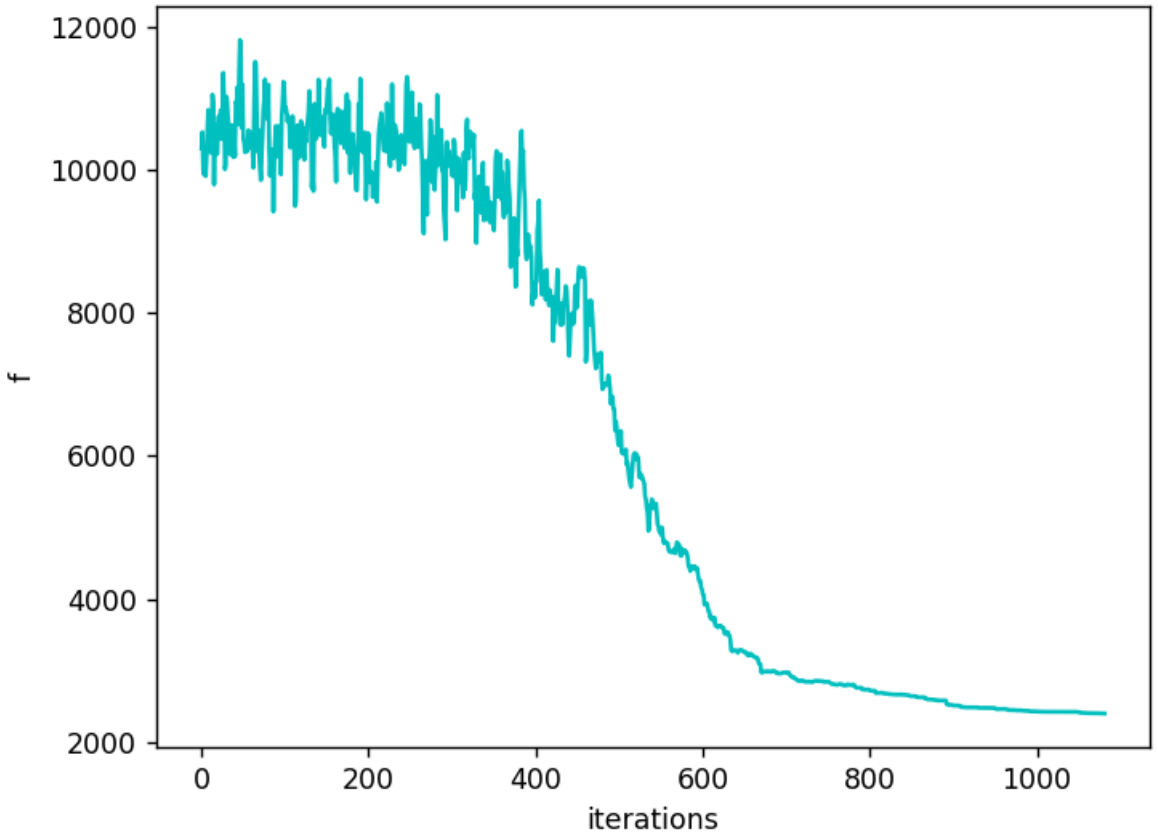
III. Grupy punktów, n = 150



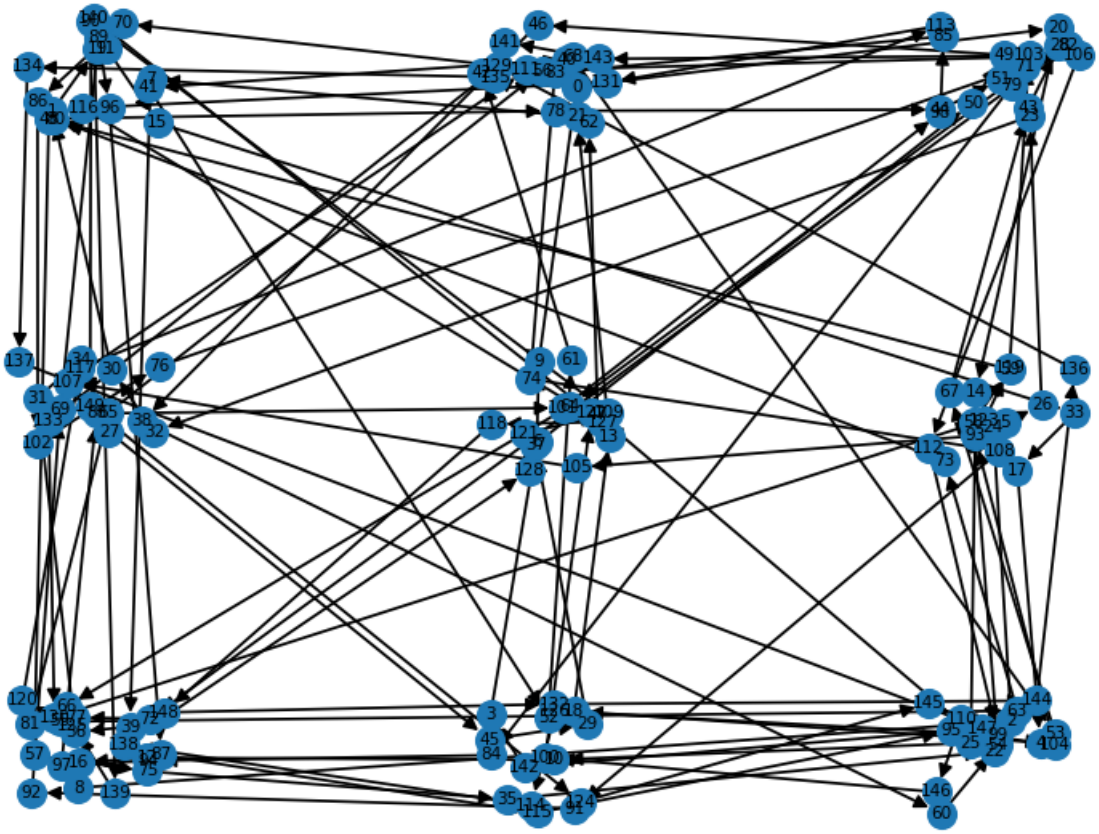
Punkty zostały ładnie wygenerowane.
Teraz uruchamiając algorytm symulowanego wyżarzania zamieniając losowe dwa punkty otrzymaliśmy:



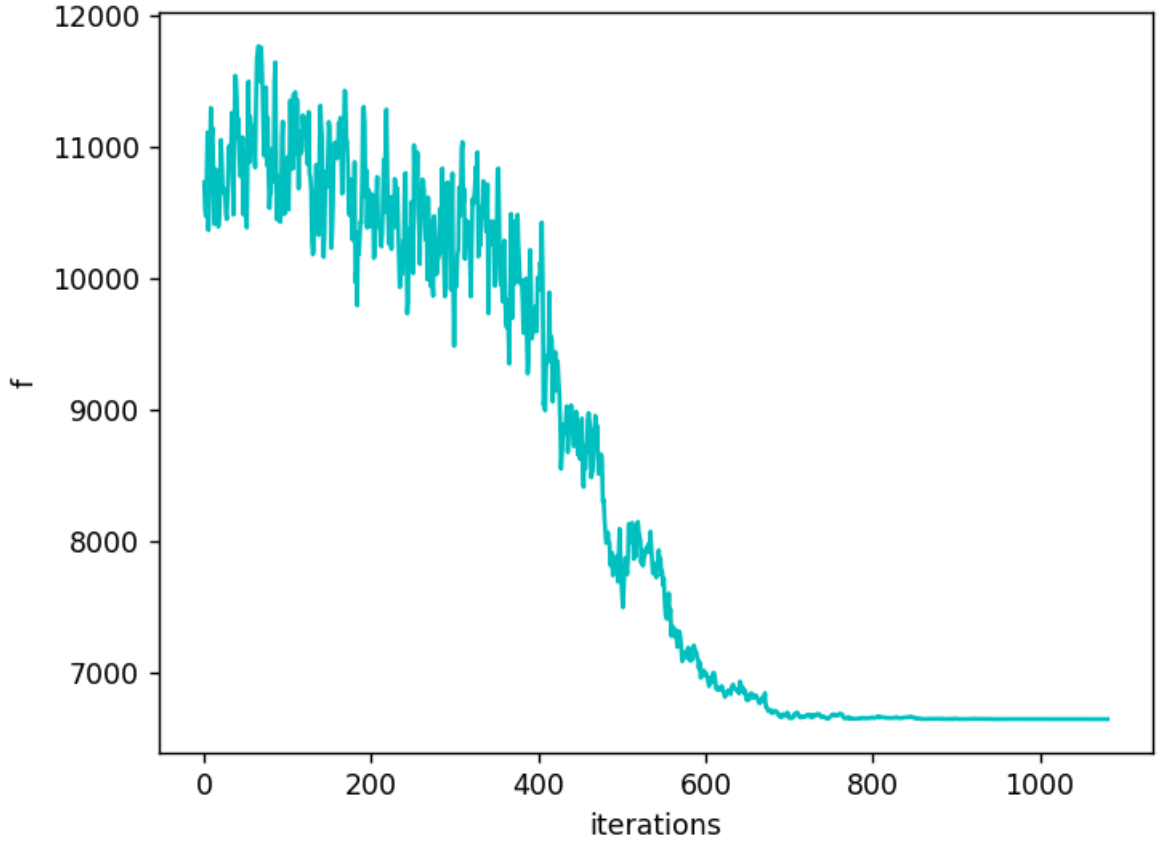
A funkcja zmieniała się tak:



Dla kontrastu zamiana jedynie sąsiadów:



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

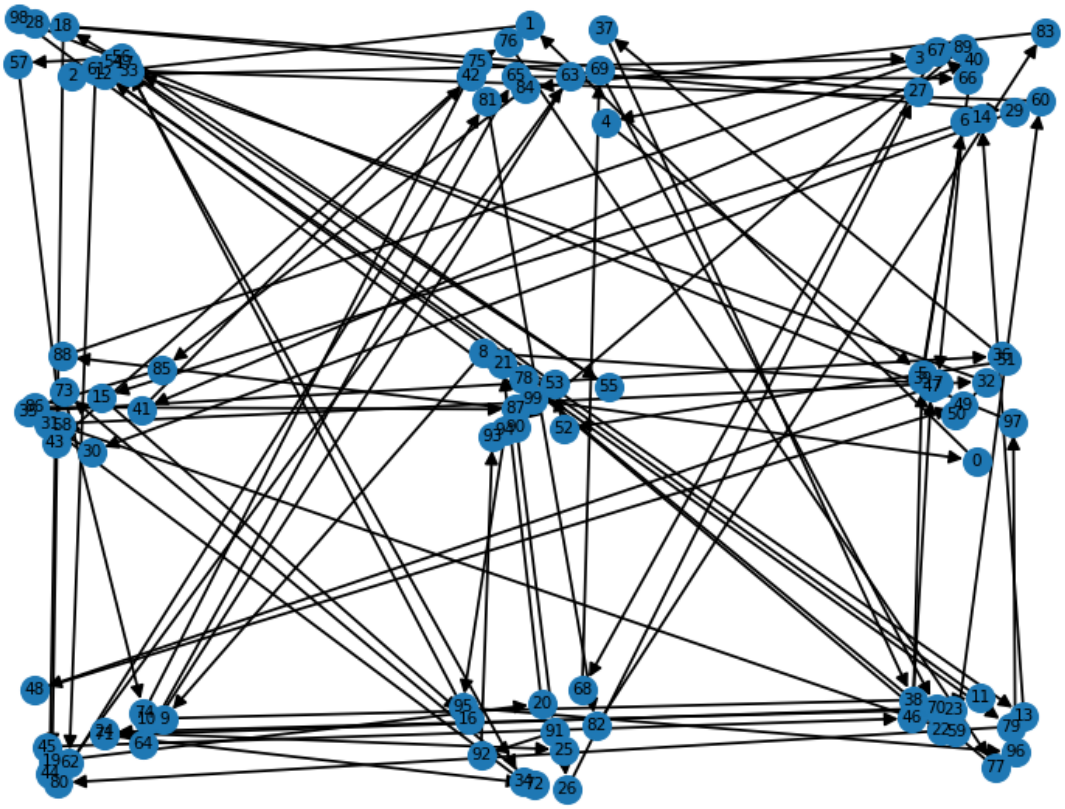


Rezultaty są następujące:

```
ARBITRARY
START DIST: 10601.148316887353
END DIST: 2403.1796394642106
[(74, (54.40985668786705, 61.09
CONSECUTIVE
START DIST: 10601.148316887353
END DIST: 6644.314261984024
[(3, (50.21317274017003, 13.911
```

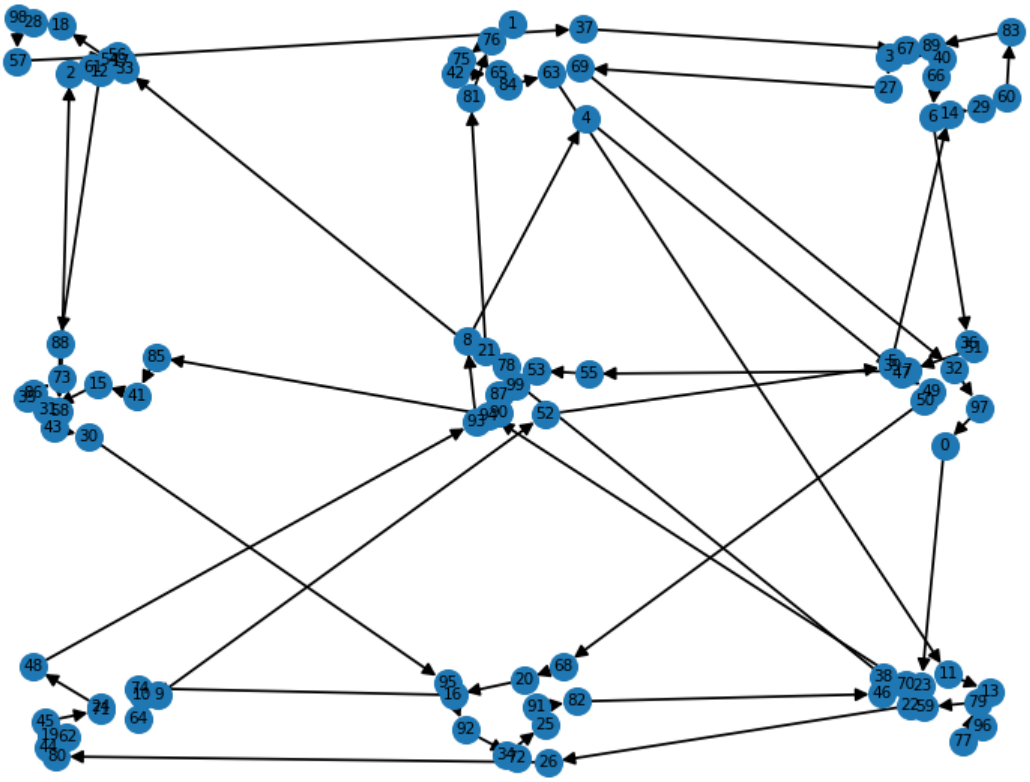
Znacznie lepsze okazało się zamienianie losowych punktów.

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:



I. Temp_rate = 0.99

Otrzymaliśmy następujący wynik:

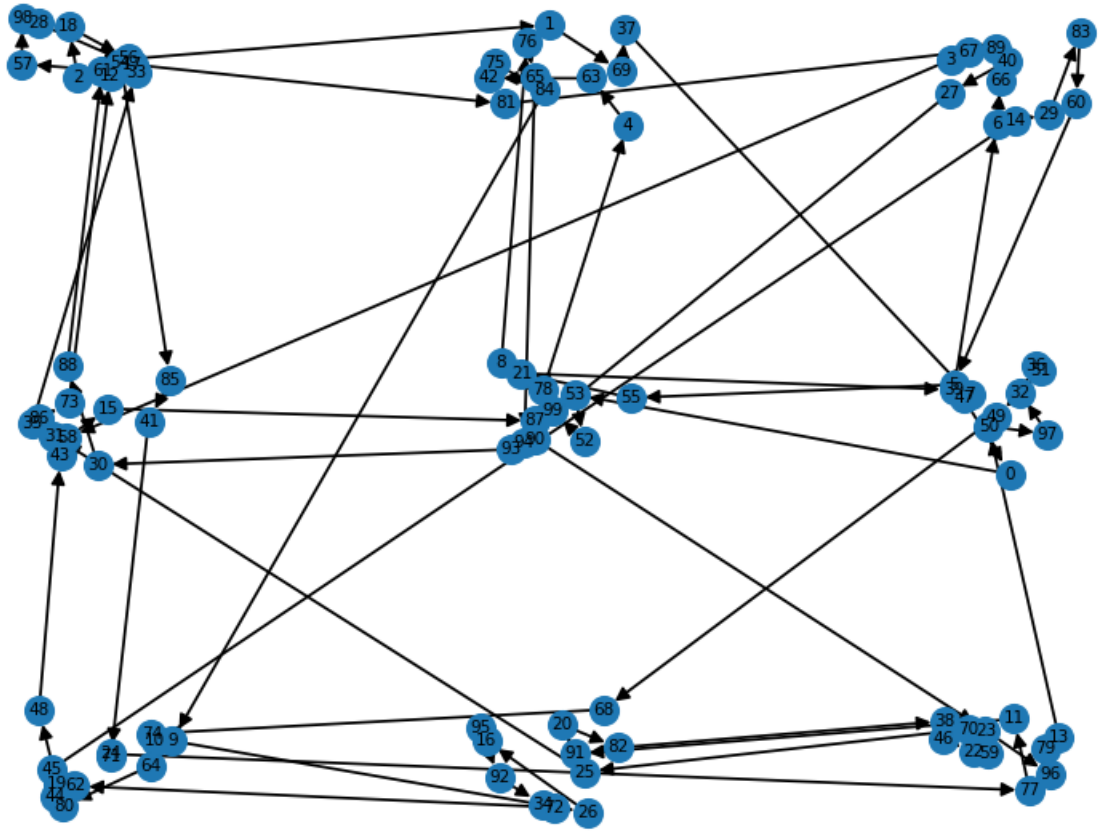


A tu jest zestawienie wyników:

```
ARBITRARY
START DIST: 4692.434488968331
END DIST: 1125.1159218440027
[(24, (6.409753981445226, 6.11132194043601)),
CONSECUTIVE
START DIST: 4692.434488968331
END DIST: 2904.7403940241575
[(1, (38.072692576391965, 76.28664905829126))
```

II. Temp_rate = 0.95

Zamiana dwóch losowych punktów dała taki wynik:



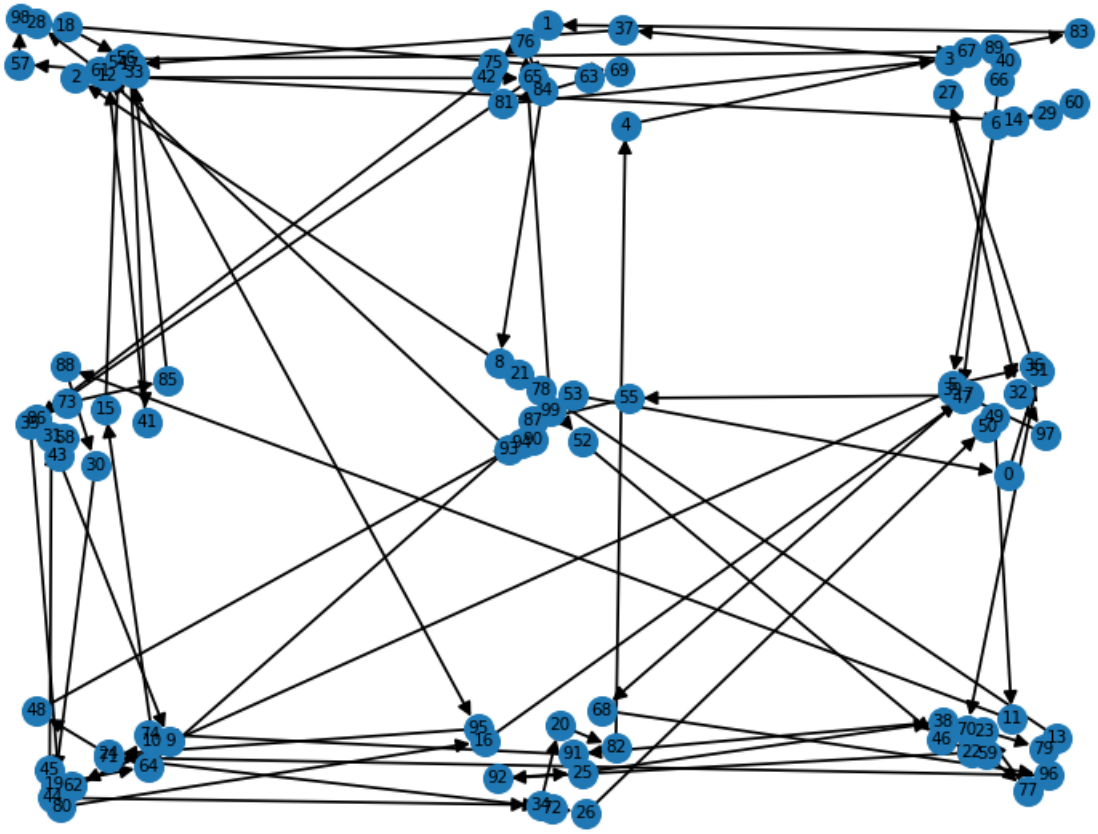
A ogólny wynik:

```
ARBITRARY
START DIST: 4692.434488968331
END DIST: 1563.229601432591
[(85, (10.706995644028973, 42.11113
CONSECUTIVE
START DIST: 4692.434488968331
END DIST: 3265.9933107088095
[(10, (9.641496043891754, 7.4553419
```

Jest to gorszy wynik niż wyżej.

III. Temp_rate = 0.8

Wynik jest dość słaby.



Temperatura za szybko spadała i było za mało iteracji.

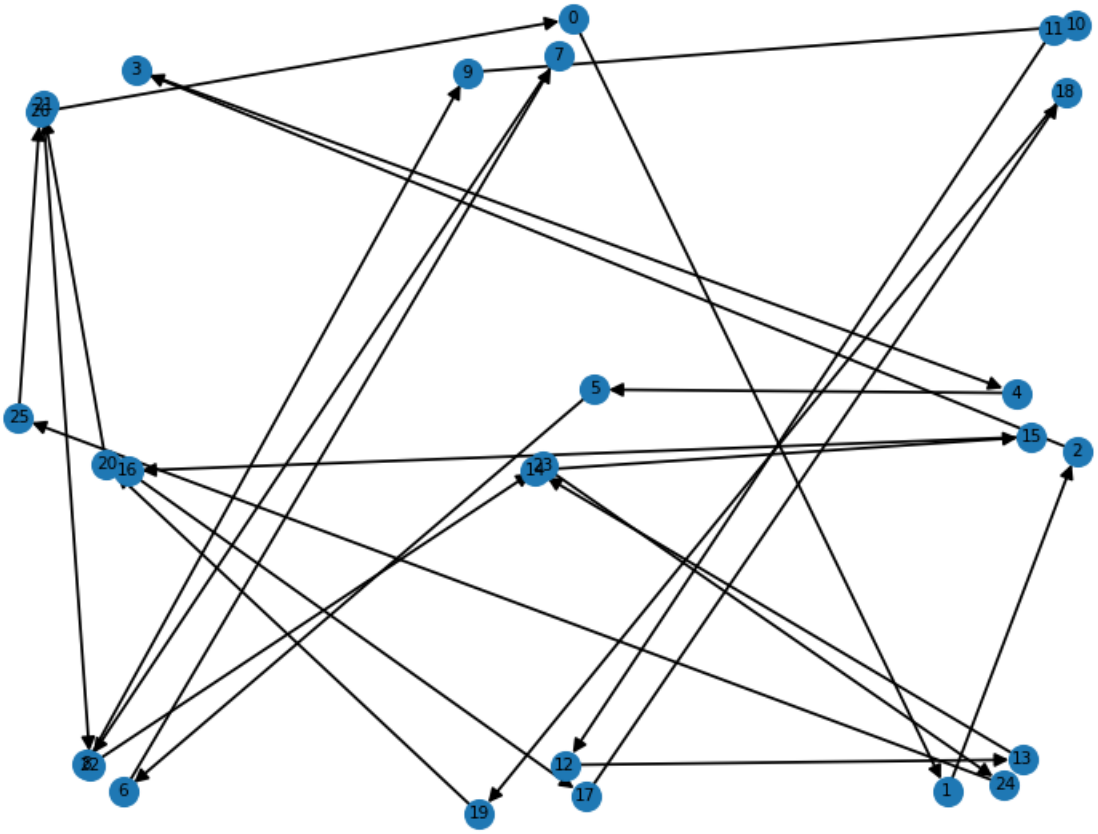
```
ARBITRARY
START DIST: 4692.434488968331
END DIST: 2285.0663082731303
[(22, (68.6497924190443, 6.306
CONSECUTIVE
START DIST: 4692.434488968331
END DIST: 3565.3923206895934
[(2, (4.052765443056204, 71.25
```

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

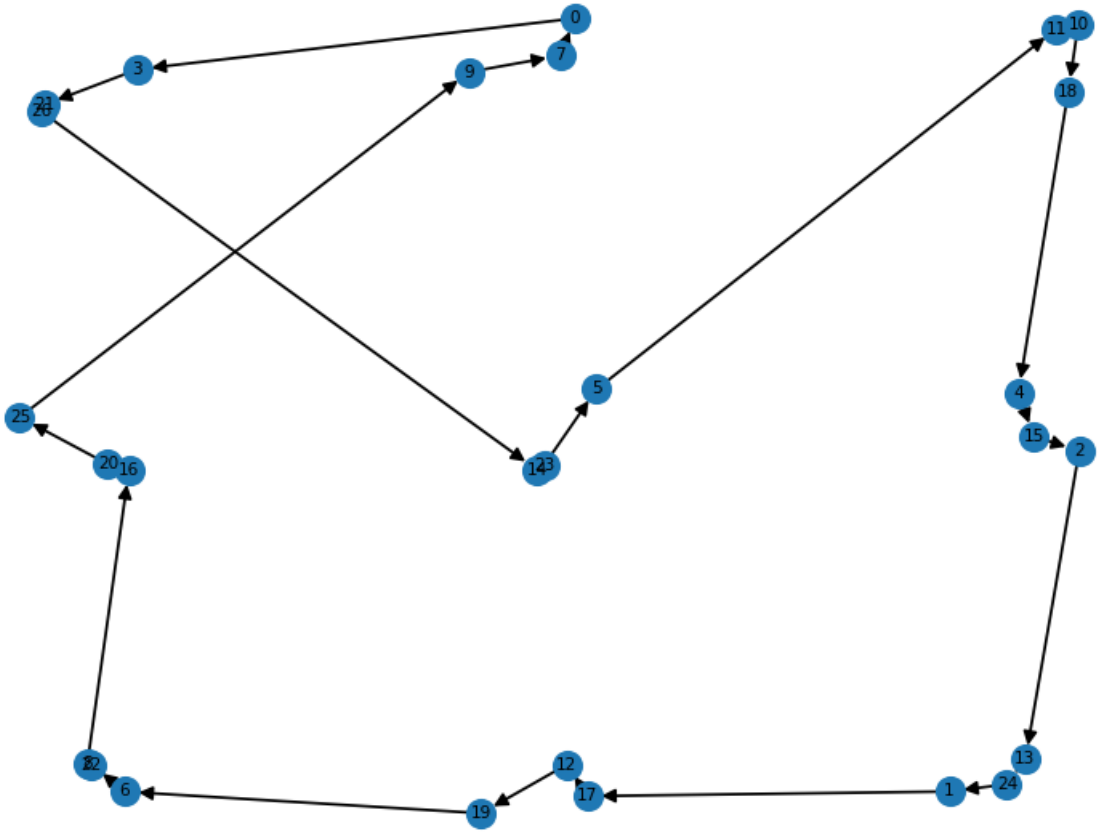
Zamiana dwóch losowych punktów (arbitrary swap) daje dużo lepsze wyniki niż zamiana jedynie sąsiadów (consecutive swap). Może to być związane, że niektóre zamiany mogą znacznie zmniejszać sumaryczny dystans.

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.

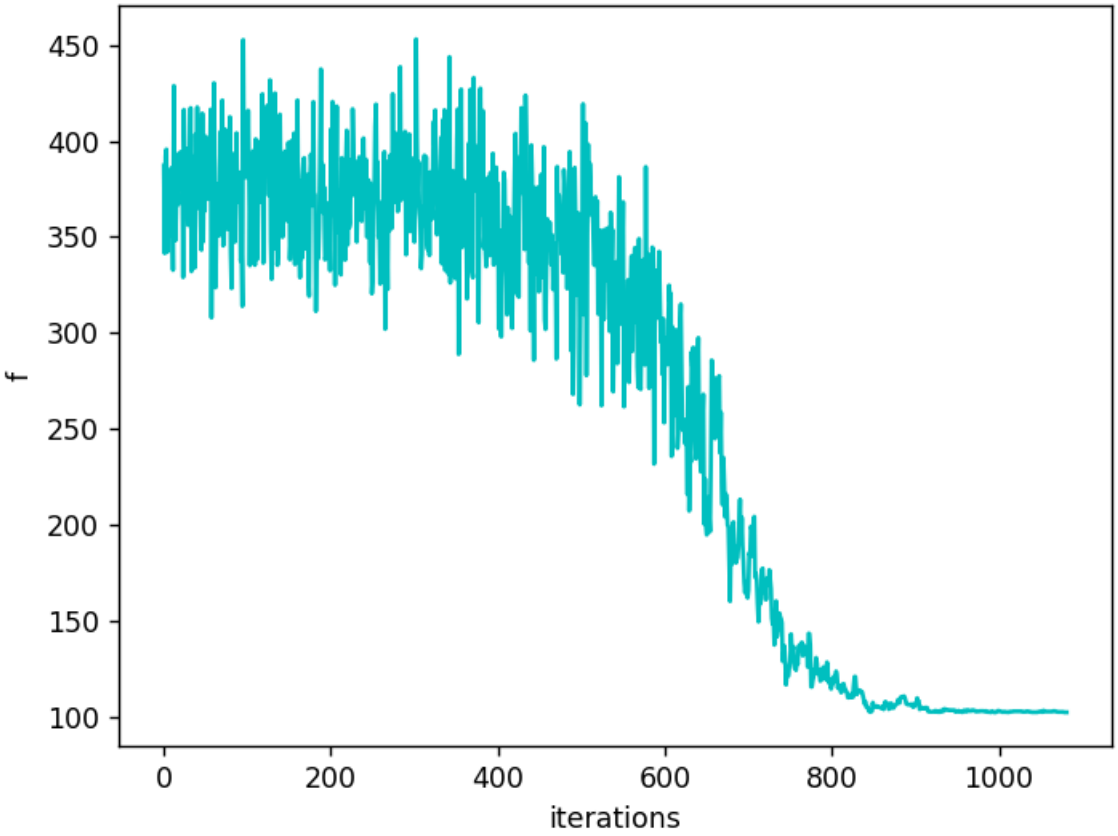
c) Przedstaw wizualizację działania procedury minimalizującej funkcję celu.
Algorytm symulowanego wyżarzania został uruchomiony na 27 punktach wrzuconych do 9 grup.



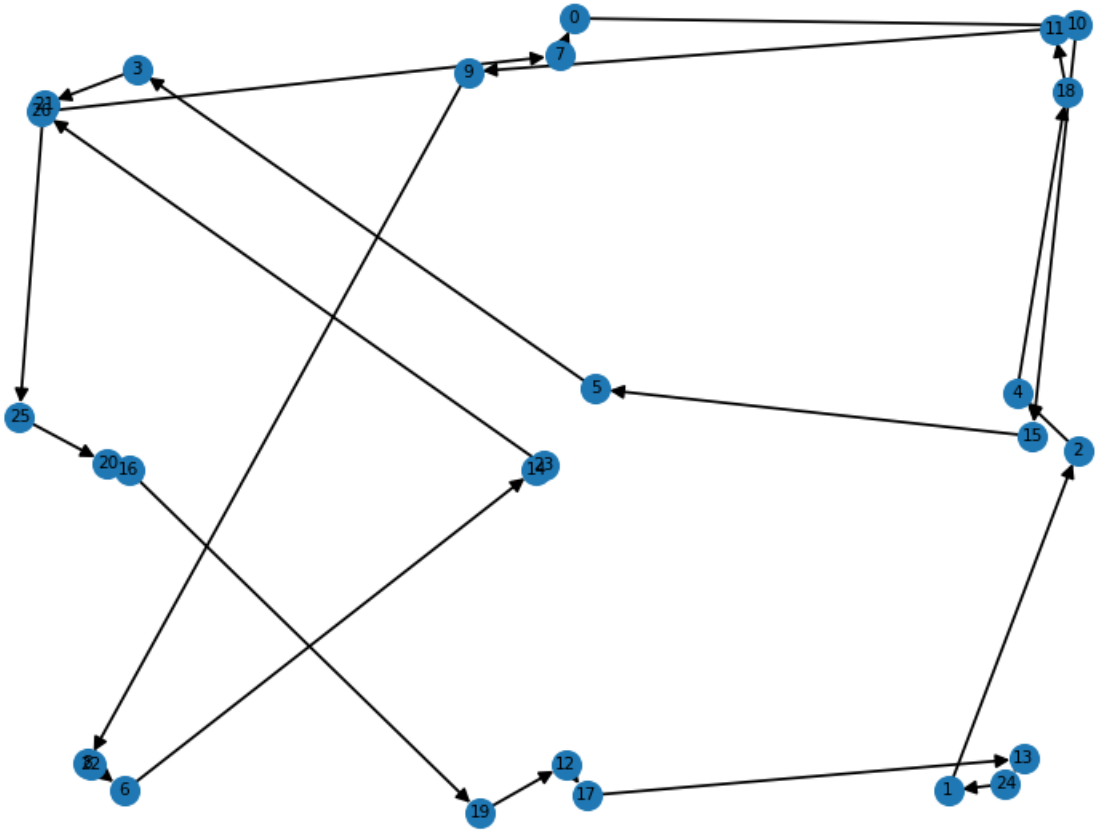
Rezultat jest bardzo zadowalający.



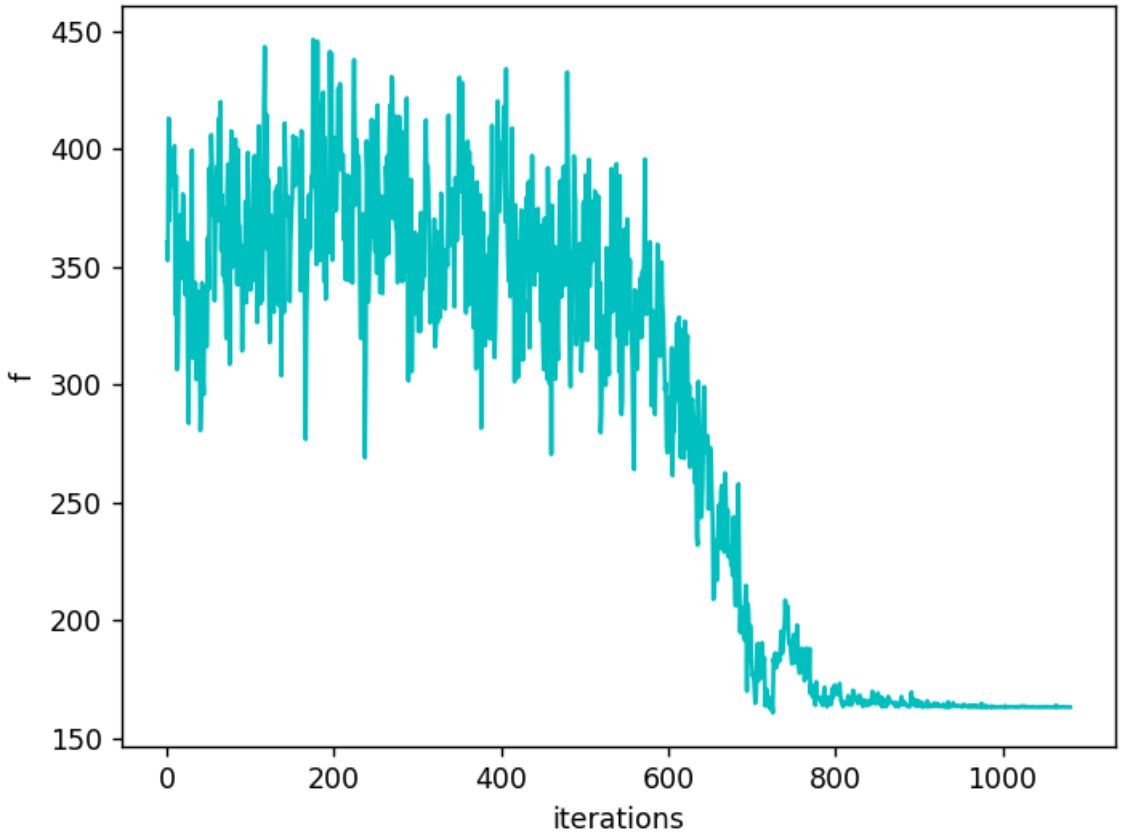
A wykres wartości funkcji optymalizowanej:



Znacznie gorszy rezultat otrzymaliśmy zamieniając jedynie połączone wierzchołki:



Z wykresem:



Tutaj jest zbiorcza informacja o wynikach:

```
ARBITRARY
START DIST: 381.95347702496133
END DIST: 102.58779149920022
[(6, (2.3677438140948874, 1.744
CONSECUTIVE
START DIST: 381.95347702496133
END DIST: 163.31059256027214
[(24, (19.359940903322116, 1.92
```

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
```

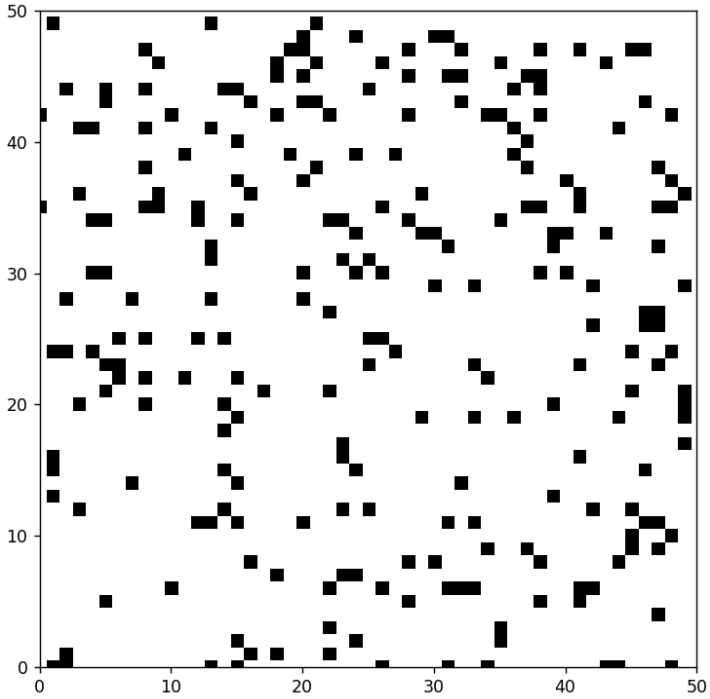
Poniżej funkcja oblicza odległość od zadanej prostej podanego punktu (zostało zawartych kilka przykładów funkcji). Druga natomiast odległość od konkretnego punktu.

```
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
```

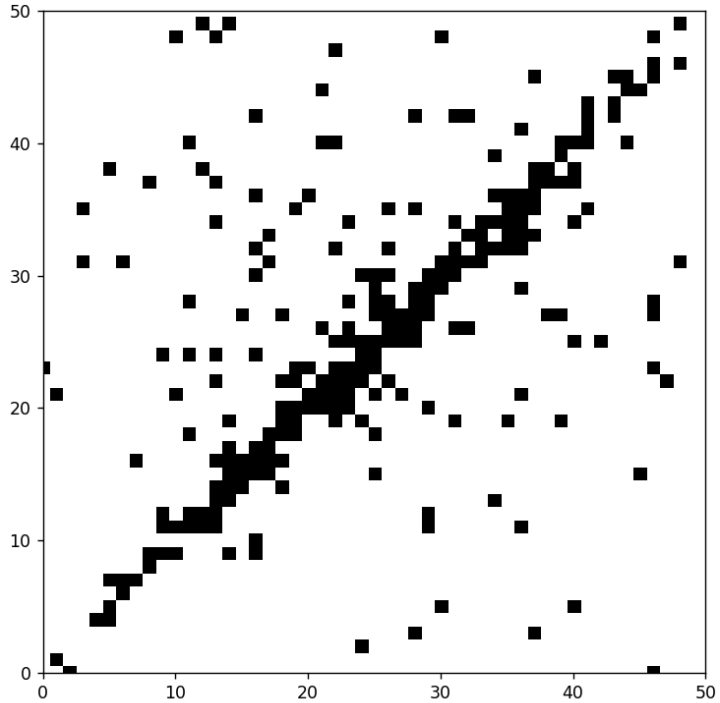
Teraz możemy przetestować naszą fizykę.
Dla n=50, δ=0.1 oraz poszukiwanie sąsiadów do 3 kratek różnicy. Parametry, z jakimi na ten moment są uruchamiane programy to:

```
temp_start = 5230
temp_end = 0.1
temp_iter = 20
temp_rate = 0.99
```

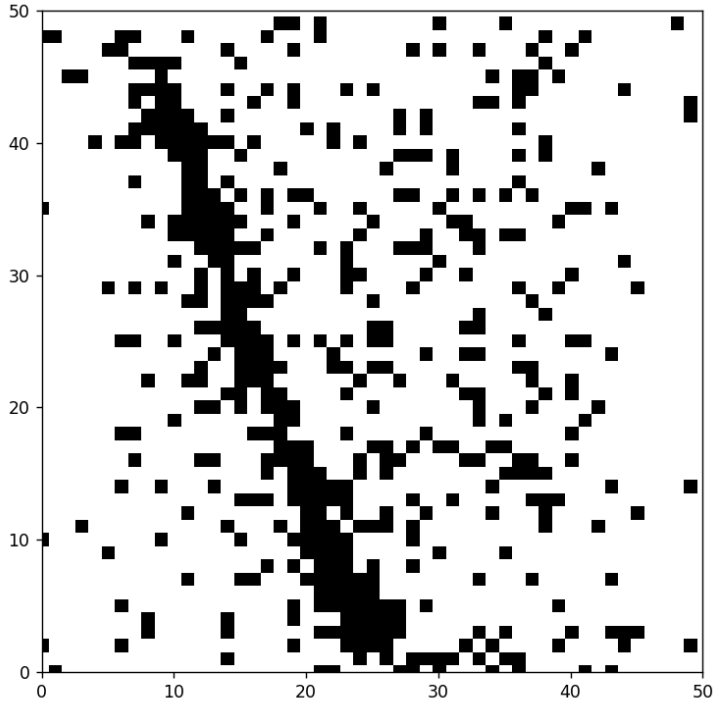
Poniżej został przedstawiony przykładowy rozkład punktów.



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

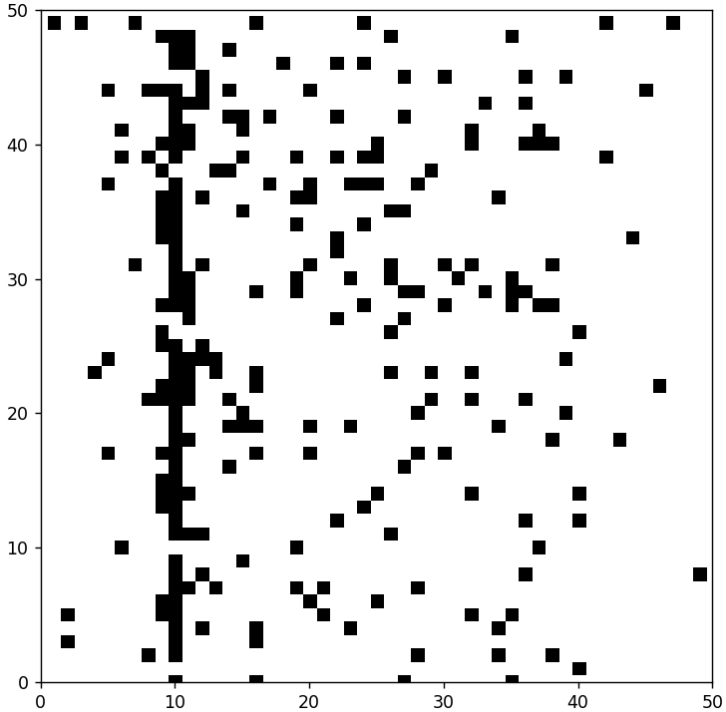


Teraz przybliżać będziemy do innej prostej. Bierzemy n=50, δ=0.2 oraz sąsiedztwo do 2 kratek.

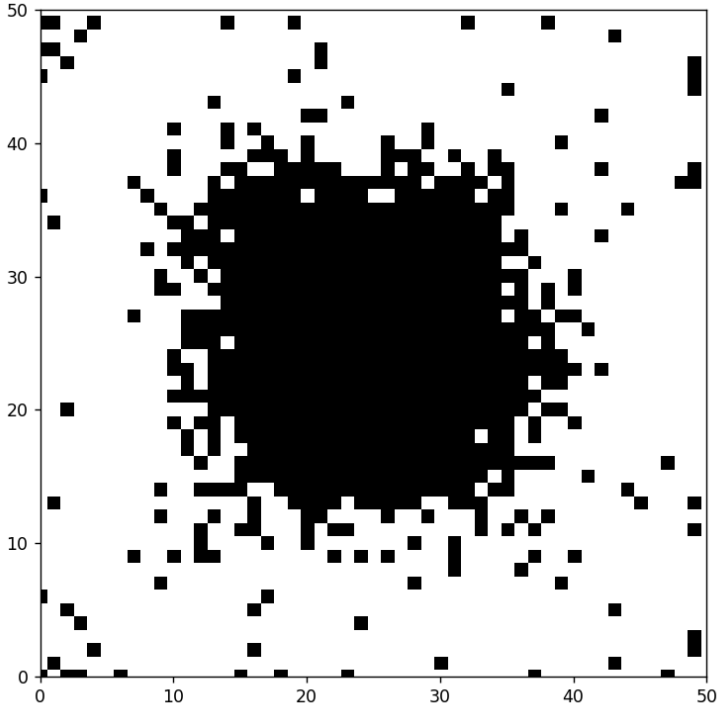


Funkcja ma ciekawy kształt i widzimy, że punkty koło niej się gromadzą.

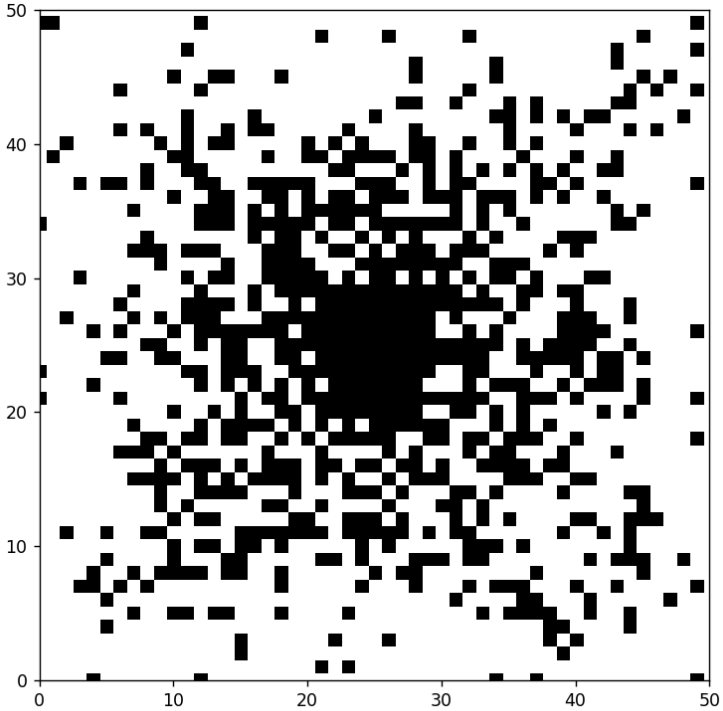
Inny ciekawy przykład, gdy prosta przyciąga punkty.



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.

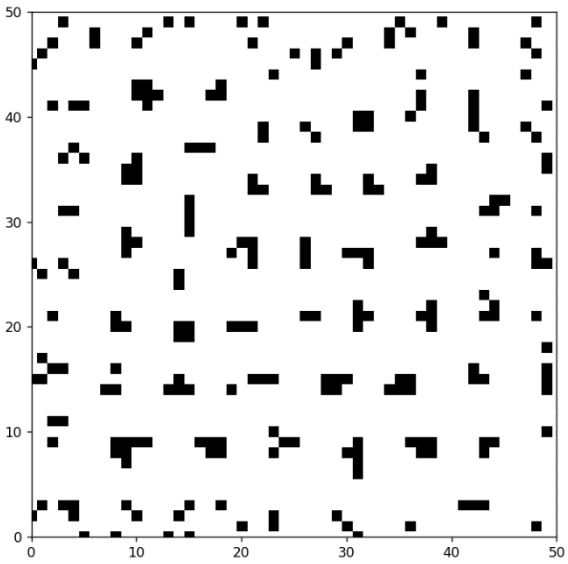


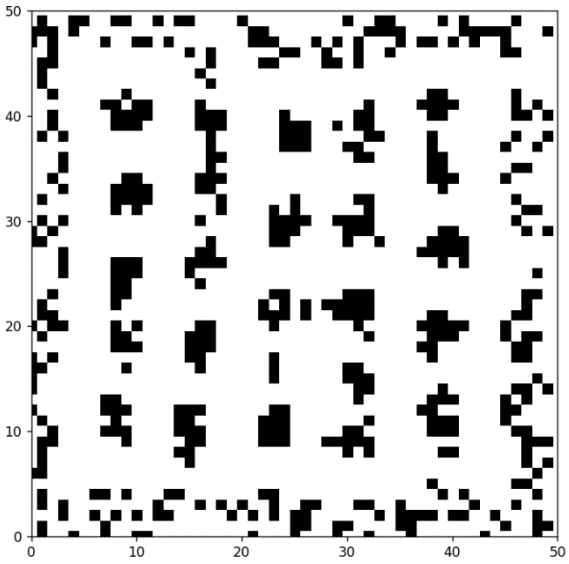
Uzyskany wynik jest znacznie gorszy.

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

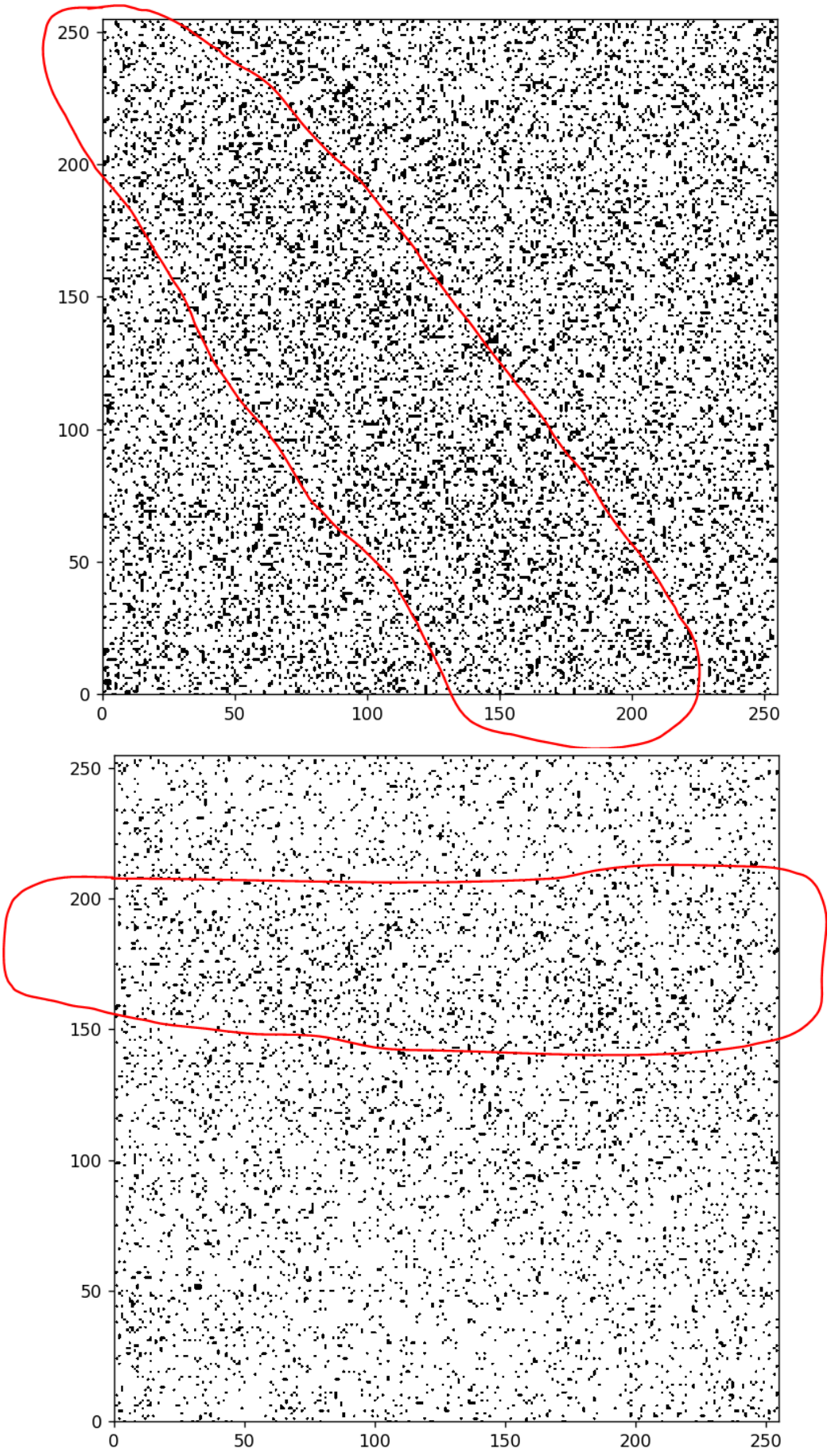




Można dostrzec formowanie się punktów w grupy.

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.


250
200
150
100
50
0
0
50
100
150
200
250
250
200
150
100
50
0
0
50
100
150
200
250

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:

```
temp_start = 5230
temp_end = 0.1
temp_iter = 100
temp_rate = 0.99
```

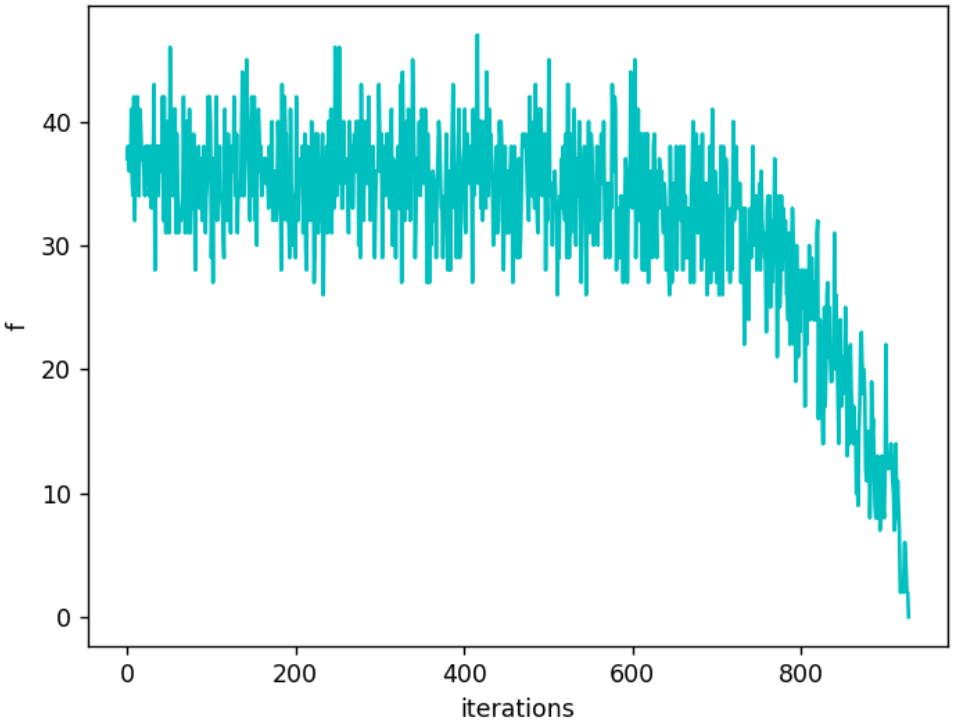
Nasza plansza została wypełniona w taki sposób:

```
FILLED
[6, 2, 5, 3, 9, 4, 7, 8, 1]
[9, 4, 7, 5, 8, 7, 2, 9, 6]
[1, 3, 8, 2, 6, 1, 5, 4, 3]
[4, 8, 2, 4, 5, 8, 9, 6, 7]
[9, 6, 5, 9, 3, 1, 4, 5, 2]
[3, 1, 7, 6, 7, 2, 3, 1, 8]
[9, 4, 5, 4, 2, 3, 8, 7, 5]
[2, 6, 7, 5, 8, 9, 3, 6, 4]
[8, 3, 1, 7, 6, 1, 1, 2, 9]
```

Udało się rozwiązać podany przykład.

```
SOLUTION
[6, 2, 7, 3, 9, 4, 5, 8, 1]
[5, 4, 3, 8, 1, 7, 2, 9, 6]
[1, 9, 8, 2, 6, 5, 7, 4, 3]
[4, 5, 2, 6, 3, 8, 9, 1, 7]
[7, 8, 6, 9, 5, 1, 4, 3, 2]
[3, 1, 9, 4, 7, 2, 6, 5, 8]
[9, 7, 4, 1, 2, 3, 8, 6, 5]
[2, 6, 1, 5, 8, 9, 3, 7, 4]
[8, 3, 5, 7, 4, 6, 1, 2, 9]
SOLVED: True
```

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
```

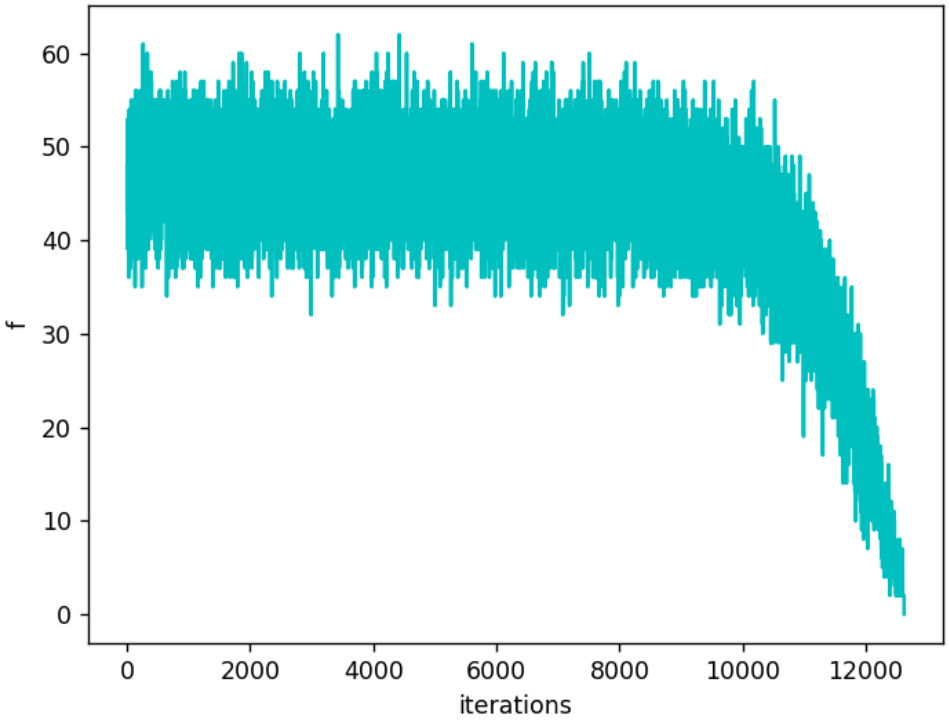
Parametry wywołania:

```
temp_start = 100000
temp_end = 0.1
temp_iter = 200
temp_rate = 0.999
```

Rozwiązanie:

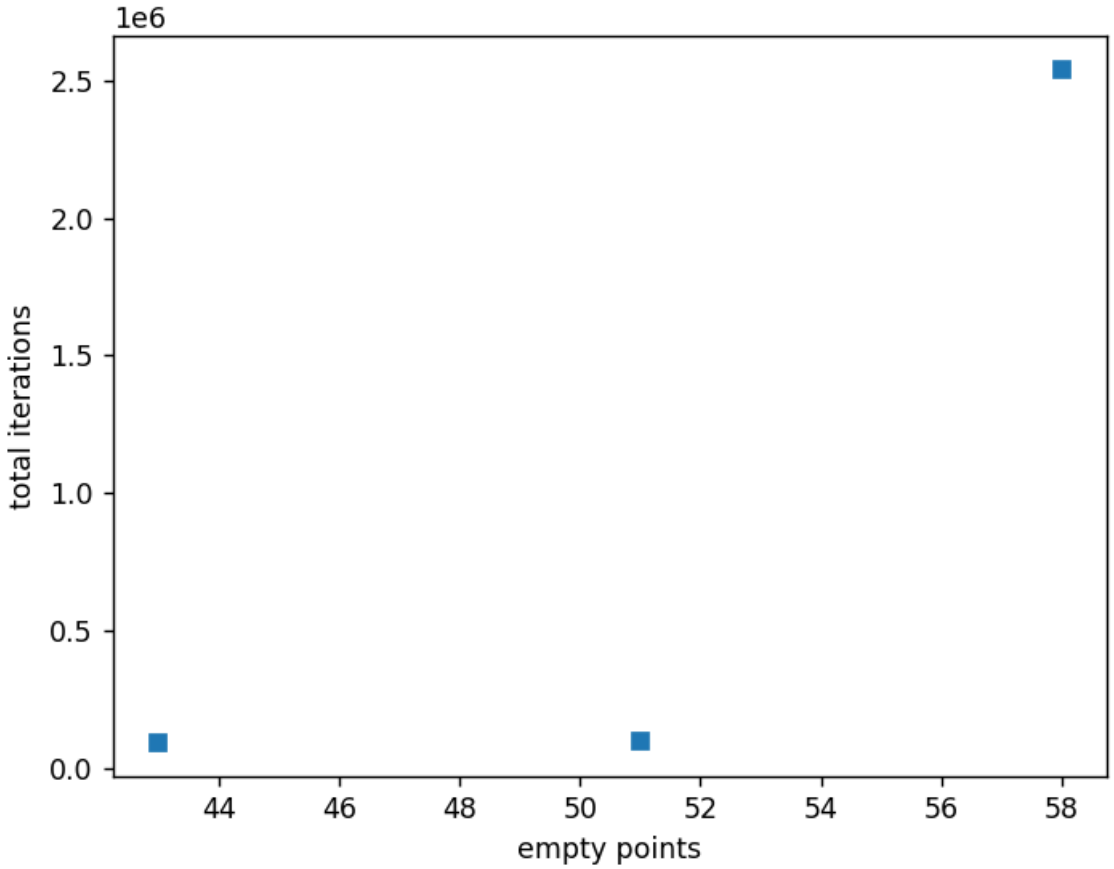
```
SOLUTION
[7, 6, 2, 9, 8, 4, 5, 3, 1]
[4, 5, 9, 6, 1, 3, 8, 2, 7]
[3, 8, 1, 2, 5, 7, 6, 4, 9]
[8, 2, 4, 1, 9, 6, 7, 5, 3]
[9, 3, 6, 7, 4, 5, 2, 1, 8]
[5, 1, 7, 8, 3, 2, 4, 9, 6]
[2, 9, 8, 5, 6, 1, 3, 7, 4]
[1, 4, 5, 3, 7, 8, 9, 6, 2]
[6, 7, 3, 4, 2, 9, 1, 8, 5]
SOLVED: True
```

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.