Metody Obliczeniowe w Nauce i Technice

Labolatorium 5

Symulowane wyżarzanie

Mateusz Praski

Informatyka Rok 2
AGH WIET
4 V 2020

Spis treści

1 Algorytm symulowanego wyżarzania

W zadaniach 1 oraz 2 została wykorzystana własna napisana implementacja algorytmu wyżarzania napisana w pliku *anneal.py*. Klasa *SimulatedAnneal* nie posiada 3 funkcji: *cost, init, get_neighbour*, które są implementowane przez klasy dziedziczące po niej w celu zdefiniowania działania algorytmu dla konkretnego problemu.

Skrócony algorytm wyżarzania prezentuje się następująco:

Result: Para w postaci najlepszego rozwiązania oraz jego kosztu
T = T_0
current_sol, current_score = init()
best_sol, best_score = current_sol, current_score epochs = 0
stagnated_epochs = 0
while $T > \epsilon$ **do**
  if *epochs > max_iteration* **then**
    break
  if *stagnated_epochs > max_stagnated* **then**
    break
  if *best_score* $\leq$ *break_point* **then**
    break
  new_sol, new_score = get_neighbour(current_sol)
  if *accept(new_score, current_score, T)* **then**
    current_sol, current_score = new_sol, new_score
  if *current_score < best_score* **then**
    best_sol, best_score = current_sol, current_score
    stagnated_epochs = 0
  else
    stagnated_epochs++
  T = temp(T, α)
  epochs++
return *best_sol, best_score*

Gdzie funkcja *accept* jest wyrażona w następujący sposób:

$$accept(X_{new}, X_{old}, T) = e^{-\frac{X_{new}-X_{old}}{T}} > random(0, 1)$$

W algorytmie zostały zaimplementowane dwa schematy spadku temperatury:

- Wykładniczy (domyślny): $T_n = \alpha T_{n-1}$
- Liniowy: $T_n = T_{n-1} - \alpha$

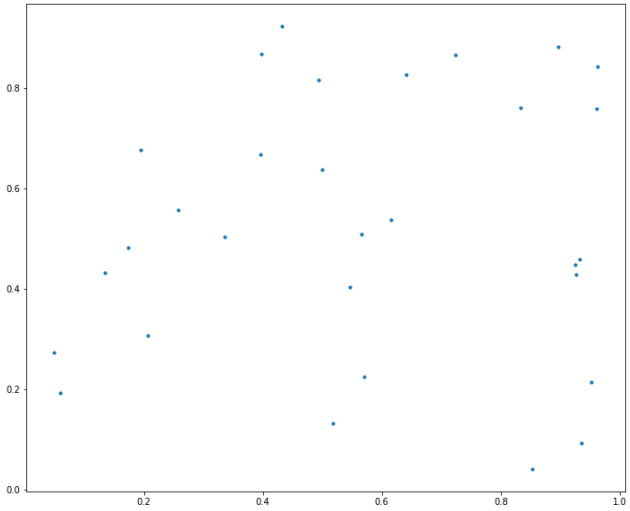
Algorytm również ma zaimplementowane zapisywanie historii przeszukiwanych stanów oraz ich wyników w celu późniejszej wizualizacji.

2 Problem TSP

Implementacja rozwiązaniu problemu przy pomocy symulowanego wyżarzania znajduje się w pliku *tsp.ipynb*.

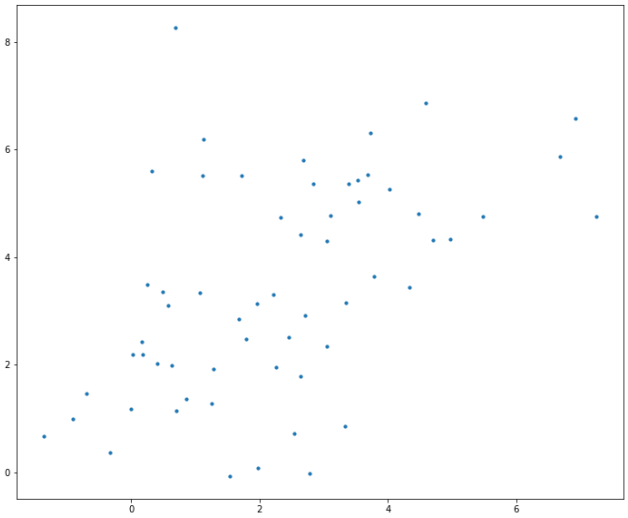
2.1 Generowanie zbiorów testowych

W rozwiązaniu zostały zaimplementowane trzy sposoby generowania zbioru punktów. Pierwszym z nich jest rozkład jednostajny wewnątrz kwadratu jednostkowego. Generator jako parametr przyjmuje jedynie liczbę miast.



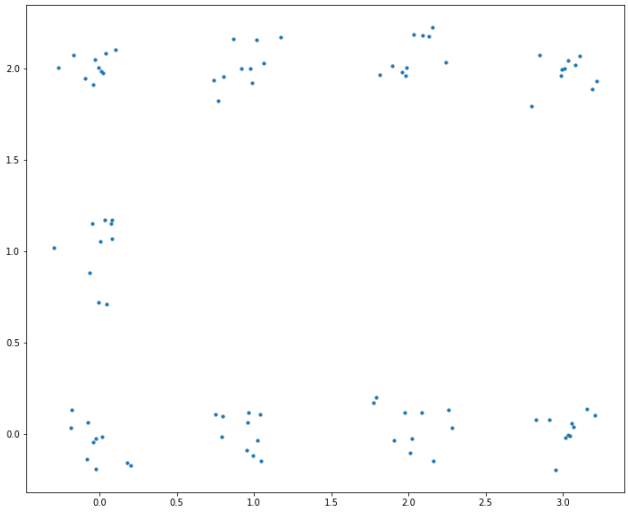
Rysunek 1: Chmura punktów z jednostajnego rozkładu wewnątrz kwadratu jednostkowego

Drugim generatorem jest suma 4 rozkładów normalnych. Metoda generuje $4k$ punktów gdzie k jest parametrem. Poza liczbą punktów jako parametry przyjmuje parametry 4 rozkładów normalnych w postaci średnich oraz macierzy kowariancji.



Rysunek 2: Chmura punktów z czterech rozkładów normalnych
$\mu = [(1,1),(3,5),(5,5),(2,2)]$, $\Sigma = [2 * \mathbf{1}, 2 * \mathbf{1}, \mathbf{1}, \mathbf{1}]$

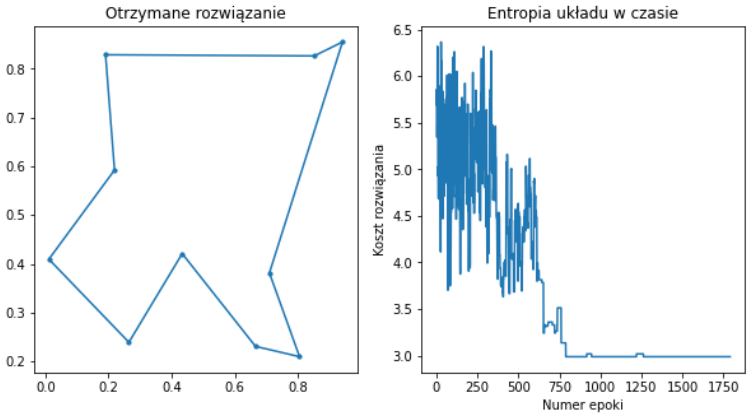
Trzeci generator tworzy 9 chmur punktów zlokalizowanych wewnątrz kół o środkach i promieniach podanych jako parametry funkcji. Rozkład punktów wewnątrz każdego z okręgów jest jednostajny.



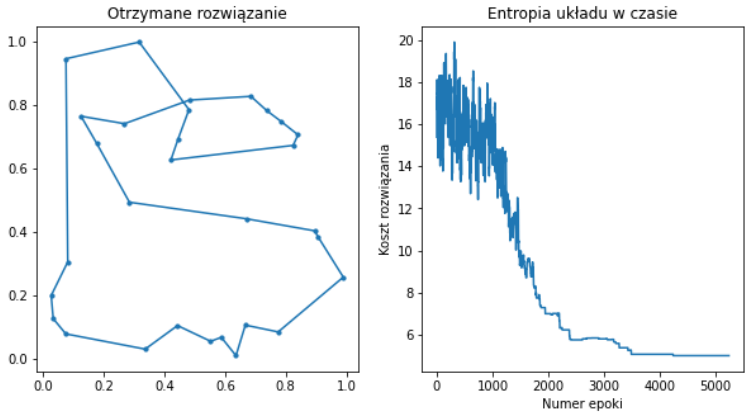
Rysunek 3: Chmura punktów 9 grup położonych na obwodzie prostokąta (2,0), (0, 3)

2.2 Wizualizacje rozwiązań wygenerowanych przez algorytm symulowanego wyżarzania

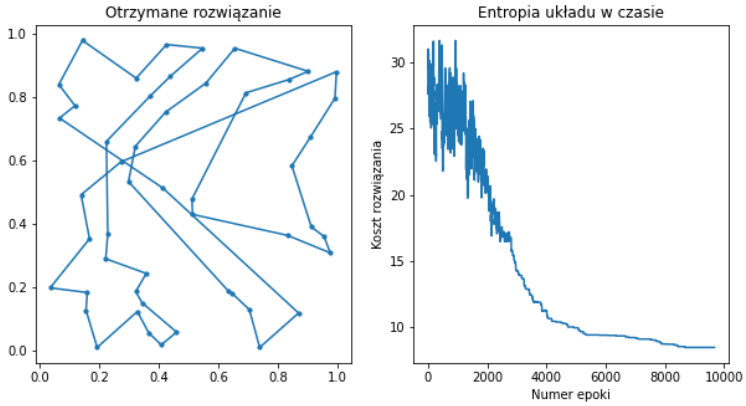
2.2.1 Rozkład jednostajny



Rysunek 4: Rozwiązanie dla N=10

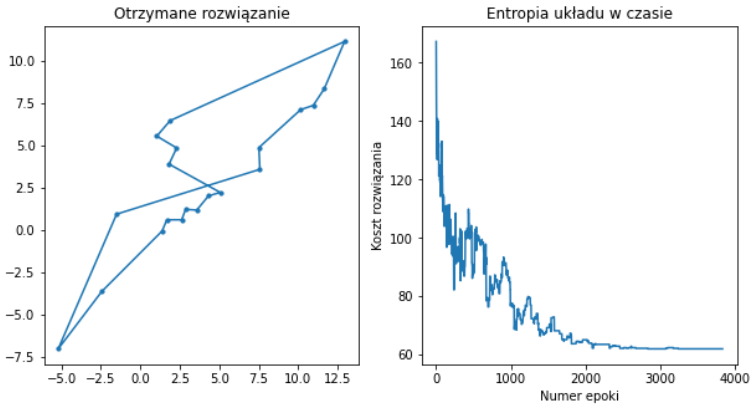


Rysunek 5: Rozwiązanie dla N=30

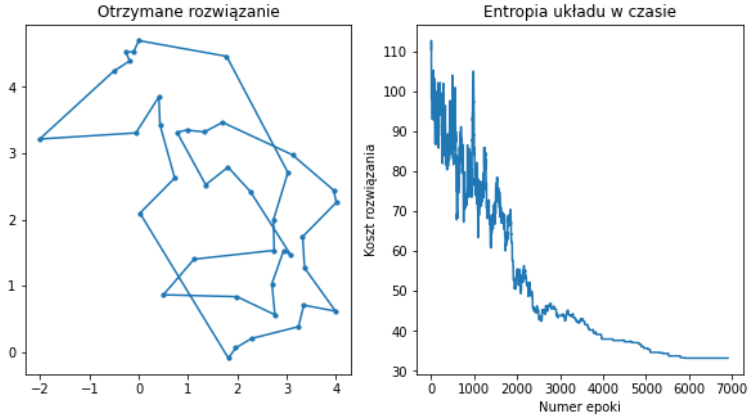


Rysunek 6: Rozwiązanie dla N=50

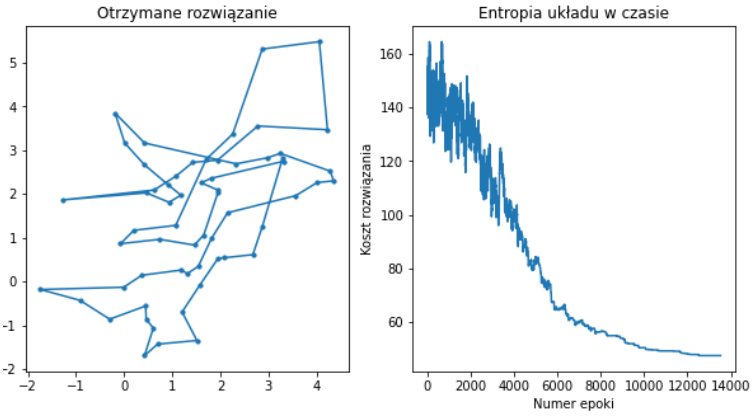
2.2.2 Rokzład normalny



Rysunek 7: Rozwiązanie dla N=20, losowe średnia, losowa kowariancja

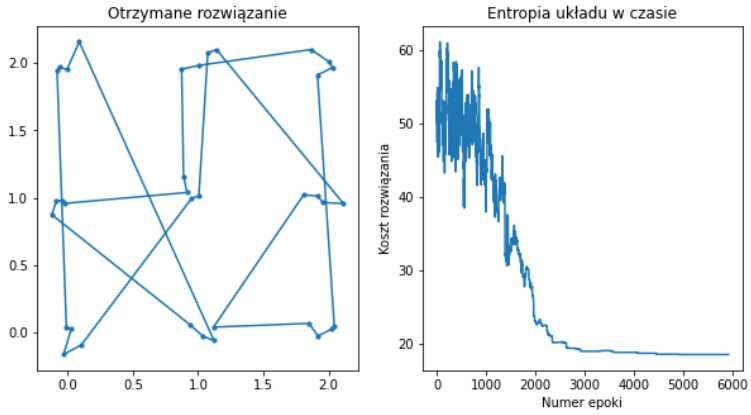


Rysunek 8: Rozwiązanie dla N=40, losowa średnia, kowariancja równa 1

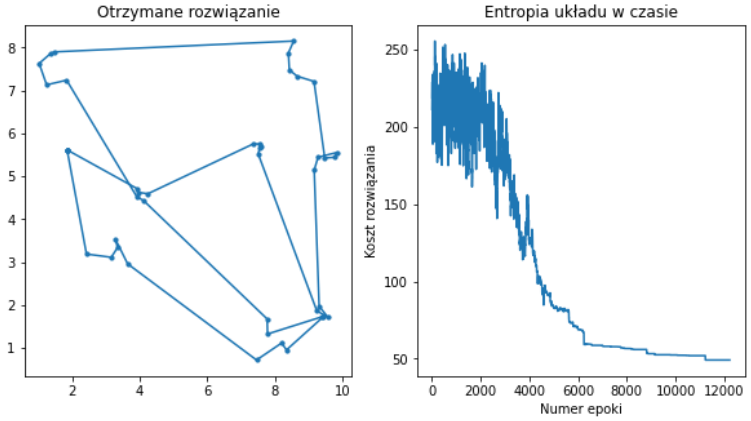


Rysunek 9: Rozwiązanie dla N=60, kolejne średnie w punktach (i, i), kowariancja równa 1

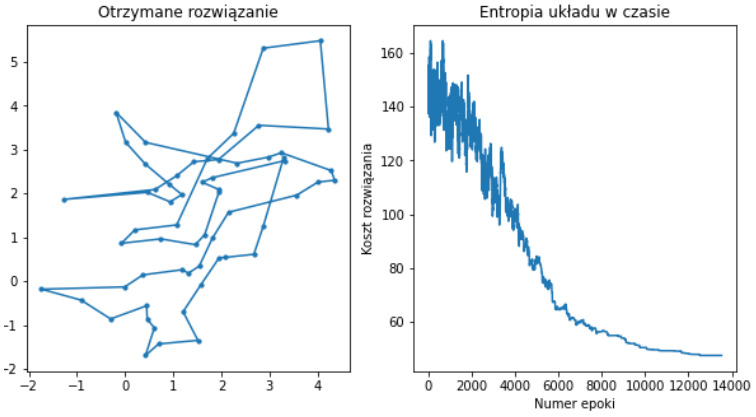
2.2.3 Rozkład 9 chmur punktów



Rysunek 10: Rozwiązanie dla N=36, położenia grup na punktach kratowych



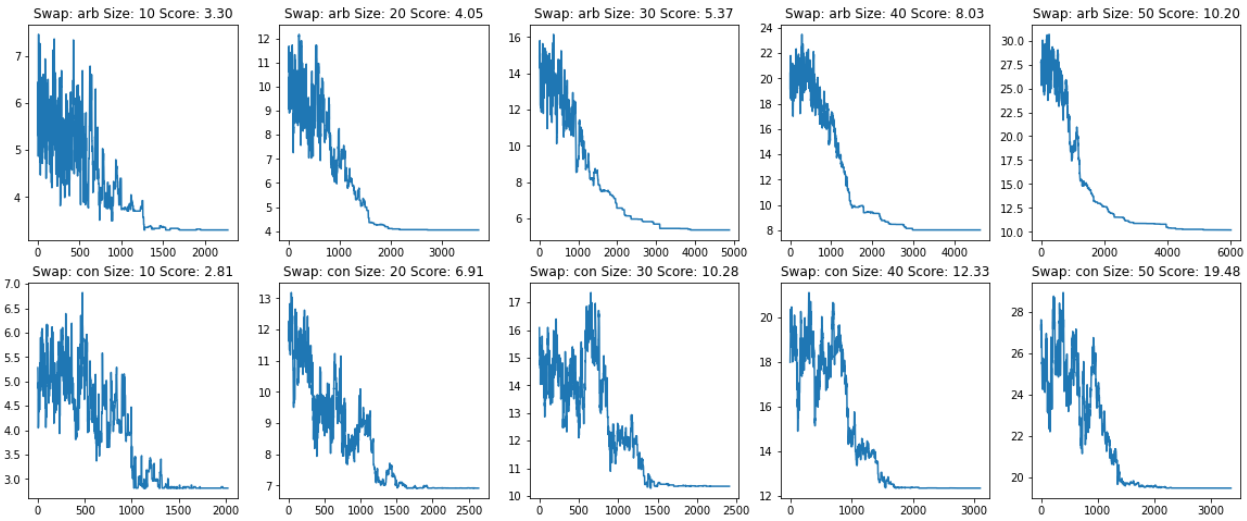
Rysunek 11: Rozwiązanie dla N=45, losowe środki, losowe promienie



Rysunek 12: Rozwiązanie dla N=90, losowe środki, losowe promienie

2.3 Zależność sposobu generowania sąsiedztwa na wynik

W ramach dostrajania symulowanego wyżarzania testowane były dwa sposoby generowania sąsiedztwa. *Consecutive swap*, podczas którego wybierany jest jeden punkt i jest zamieniany z sąsiadem oraz *arbitrary swap* podczas którego zamieniane są dwa losowe punkty w rozwiązaniu.



Rysunek 13: Wyniki różnych metod wybierania sąsiedztwa dla różnych rozmiarów problemu

Metody były testowane przy tych samych hiperpametrach:

- $\alpha = 0.997$
- $T_0 = \sqrt{N}$
- $\epsilon = 1e - 7$
- $max\_stagnation = 1000$

Na powyższych wykresach możemy zauważyć, że dla większości przypadków *consecutive swap* osiąga znacząco gorsze wyniki. Ponadto na wykresach gdzie został wykorzystany consecutive swap (con) widać bardziej chaotyczne zachowanie spadku, a także faza przeszukiwaniu obszaru z przyjmowaniem gorszych wyników trwa dłużej.

2.4 Zależność parametrów temperatury układu na wynik

Jednym z najważniejszych elementów tworzenia modelu symulowane wyżarzania jest dostrajanie hiper parametrów. Od nich zależy czas trwania działania algorytmu oraz długość poszczególnych faz. Parametry były dopierane na podstawie wykresów energii układu w czasie.

Wśród temperatur początkowych były testowane następujące możliwości:

- $T_0 = 1$
- $T_0 = k * |N|$
- $T_0 = |N| * var(N)$
- $T_0 = \sqrt{|N|}$
- $T_0 = \sqrt{|N|} * var(N)$
- $T_0 = \log |N|$
- $T_0 = \log |N| * var(N)$

Wśród parametrów spadków temperatur dla funkcji wykładniczej były testowane następujące wartości:

$$\alpha \in (0.995, 0.996, 0.997, 0.998, 0.999)$$

Zależność logarytmiczna okazała się dawać zbyt niską temperaturę początkową układowi, przez co algorytm od samego początku schodził do najbliższego minimum. Wykorzystanie wariancji poprawiło zachowanie algorytmu dla zbiorów punktów znajdujących się poza kwadratem jednostkowym.

Wśród parametrów spadków temperatur dla funkcji liniowej testowane były nastepujące wartości:

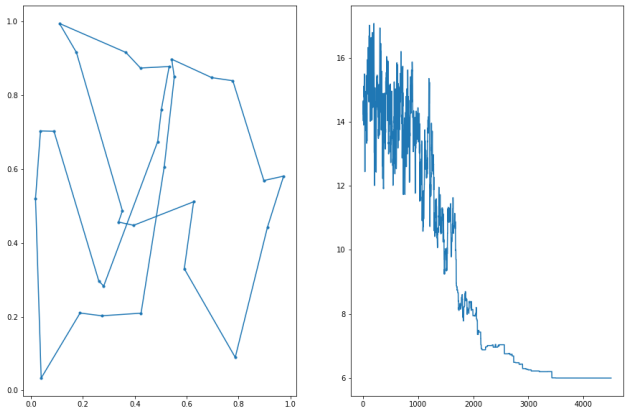
$$\alpha \in (0.00001, 0.00005, 0.0001, 0.0005, 0.001, 0.005, 0.01)$$

Liniowy model spadku temperatury okazał gorszy od wykładniczego. Zbyt powolny spadek temperatury powodował, że algorytm przez większość czasu podążał losowo i dopiero w ostatnich epokach zaczynał schodzić do optimum. Próby dostrojenia parametru α by wydłużyć czas spadku do optimum kończyły się zbyt małą liczbą iteracji (wyżarzanie zostawało zakończone gdy temperatura spadała do 0 po kilkuset iteracjach).

Dla punktów losowanych z kwadratu jednostkowego zostały wybrane parametry $T_0 = \sqrt{|N|}, \alpha = 0.9888$ przy modelu wykładniczym. Dla punktów w większej odległości lepiej sprawdziły się parametry gdzie $T_0 = \sqrt{|N|} * var(N)$.

2.5 Wizualizacja procesu działania algorytmu

W notatniku *tsp.ipynb* została dołączona funkcja *save_to_file* która zapisuje wizualizację wyżarzania do pliku video. Na wizualizacji znajdują się dwie plansze, po lewej stronie aktualne rozważane rozwiązania, a po prawej wykres energii układu w czasie. Metoda ta przyjmuje jako parametr obiekt klasy *SimulatedAnnealing* po wykonaniu metody *fit*.



Rysunek 14: Przykładowa klatka z wizualizacji

3 Generowanie binarnego obrazka

3.1 Generowanie sąsiedztwa

Pierwszym pomysłem na generowanie nowego sąsiedztwa było wybranie pewnej liczby pikseli (np. 10% obrazka) po czym dokonanie operacji permutacji na wybranym podzbiorze. Algorytm ten okazał się jednak nieefektywny dla obrazków, gdzie była znacząca różnica pomiędzy liczbą czarnych a białych pikseli. W takiej sytuacji algorytm często losował podzbiór pikseli tylko jednego koloru, finalnie nie zmieniając obrazka.

Rozwiązaniem tego problemu był alternatywny generator sąsiedztwa. Wybiera on losowo k (parametr generatora) białych oraz czarnych punktów, po czym zamienia je miejscami. Dzięki takiemu algorytmowi jest pewność, że nowy obraz będzie się różnił od poprzedniego.

3.2 Funkcja kosztu obrazka

W kodzie znajdują się dwie podstawowe funkcje kosztu *black_penalty* której wartość jest równa sumie czarnych pikseli w otoczeniu każdego czarnego piksela oraz *white_penalty* której wartość jest równa sumie białych pikseli w podanej masce dla każdego czarnego piksela.

Ostateczna implementacja funkcji kosztu *ImageAnneal.cost* przyjmuje zbiór różnych masek (sąsiedztw) wraz z odpowiadającymi im współczynnikami. Dla każdego dodatniego współczynnika funkcja kosztu przemnaża wynik *white_penalty* przez współczynnik, w przypadku ujemnych współczynników funkcja kosztu przemnaża wynik *black_penalty* przez wartość bezwzględną współczynnika. W ten sposób możemy składać dowolną kombinację sąsiedztw z różnymi priorytetami.

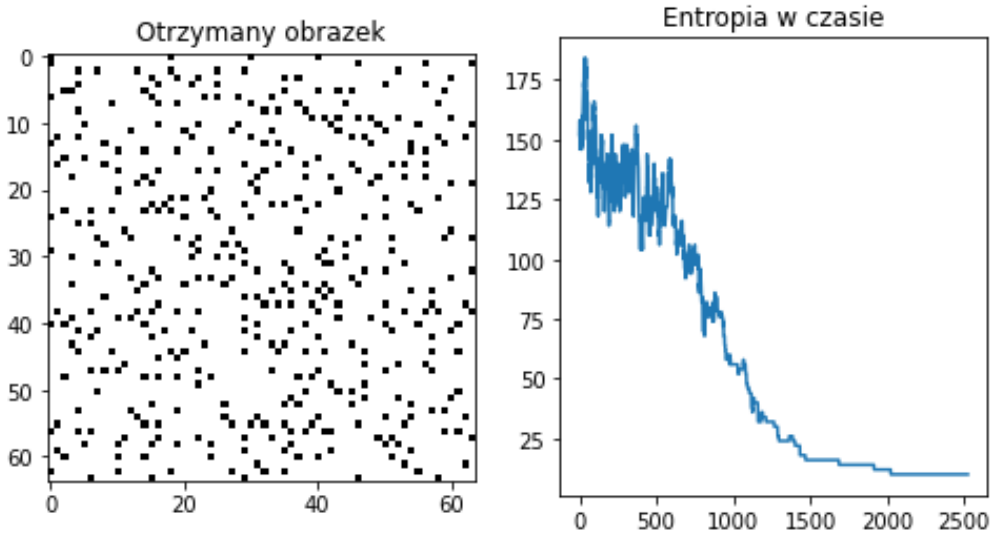
3.3 Wpływ wykorzystanych sąsiedztw oraz parametrów wyżarzania na efekt końcowy

Czynnikiem wpływającym bezpośrednio na zwartość obrazka jest funkcja energii. Od niej zależy do jakiego wzoru będzie dążył algorytm. Wprowadzenie wag do bardziej złożonych funkcji kosztu pozwala na priorytetyzację różnych rodzajów sąsiedztw.

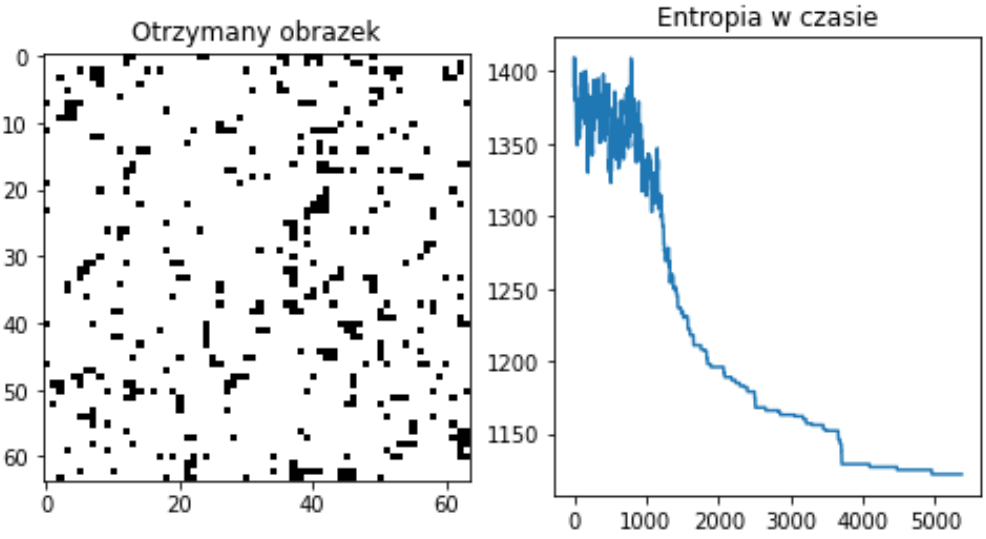
Parametry procesu wyżarzania mają głównie wpływ na jego proces, co odpowiada za jakość zbiegania. Wpływa to pośrednio na finalny obrazek, jednakże nie zauważono znaczących różnic we wzorach w zależności od wykorzystanych parametrów wyżarzania.

3.4 Przykładowe obrazki otrzymane w procesie wyżarzania

3.4.1 Sąsiedztwo 4

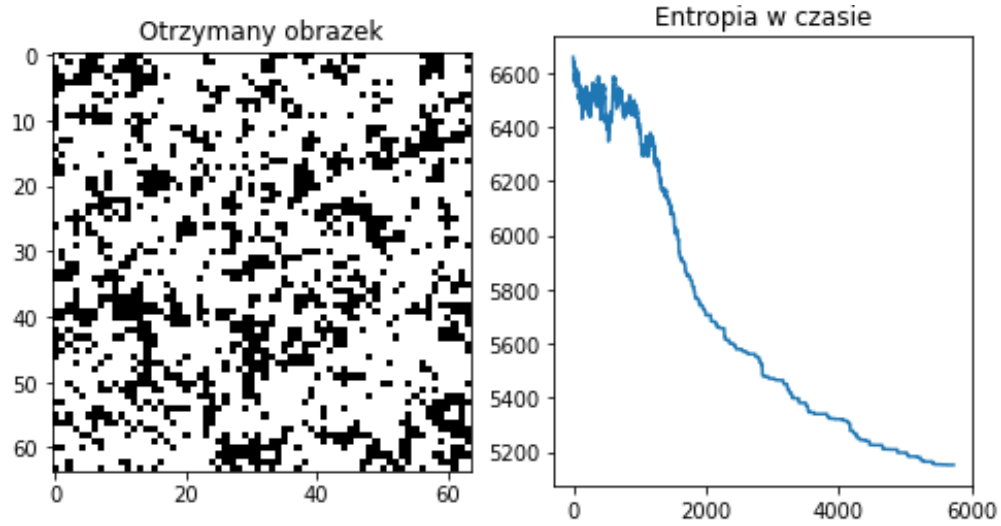


Rysunek 15: Wynik wyżarzania dla 4 -sąsiedztwa z karą za czarne piksele w otoczeniu, $\delta = 0.1$, spadek temperatury wykładniczy

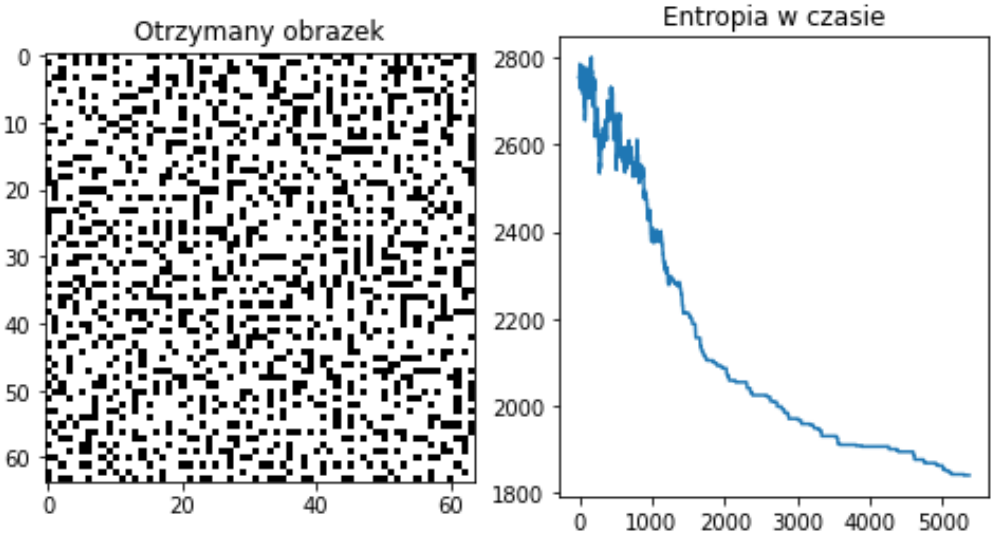


Rysunek 16: Wynik wyżarzania dla 4 -sąsiedztwa z karą za białe piksele w otoczeniu, $\delta = 0.1$, spadek temperatury wykładniczy

3.4.2 Sąsiedztwo 8

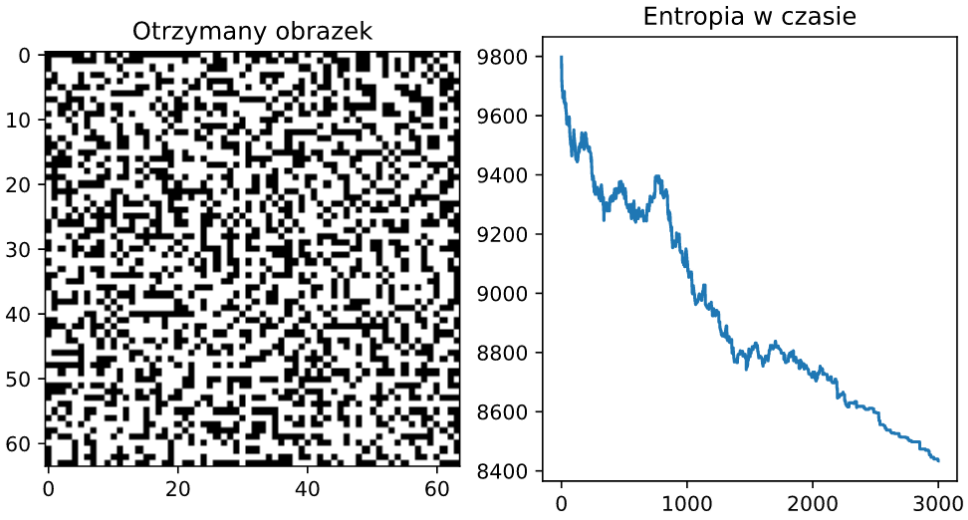


Rysunek 17: Wynik wyżarzania dla 8-sąsiedztwa z karą za białe piksele w otoczeniu, $\delta = 0.3$, spadek temperatury wykładniczy

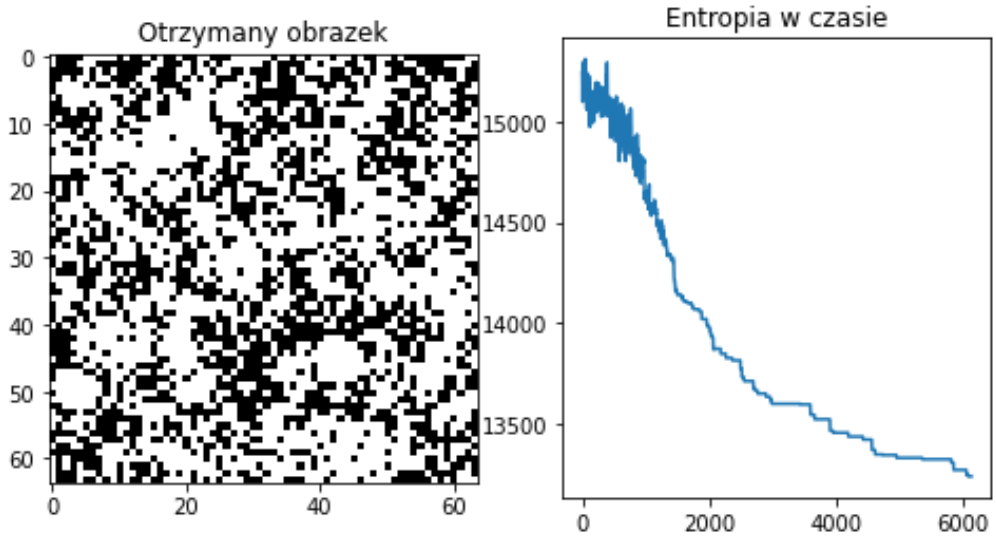


Rysunek 18: Wynik wyżarzania dla 8-sąsiedztwa z karą za czarne piksele w otoczeniu $\delta = 0.3$, spadek temperatury wykładniczy

3.4.3 Sąsiedztwo 16



Rysunek 19: Wynik wyżarzania dla 16-sąsiedztwa z karą za czarne piksele w otoczeniu, $\delta = 0.4$, spadek temperatury liniowy



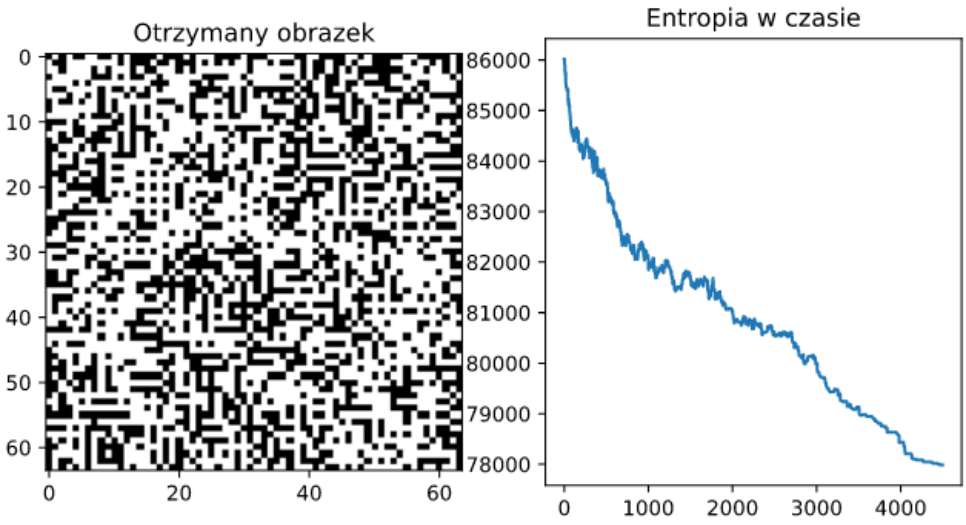
Rysunek 20: Wynik wyżarzania dla 16-sąsiedztwa z karą za białe piksele w otoczeniu, $\delta = 0.4$, spadek temperatury wykładniczy

3.4.4 Próby otrzymania labiryntu

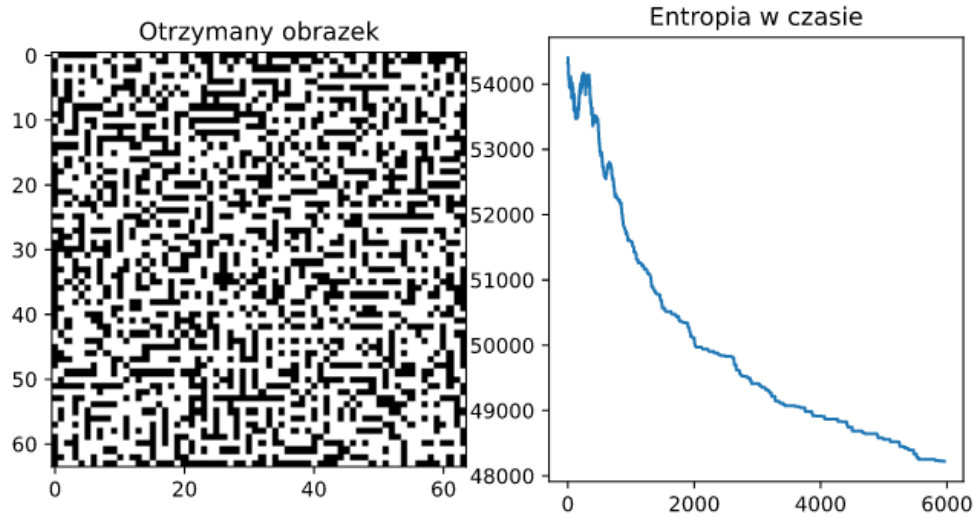
Szczególna uwaga została poświęcona próbom uzyskania generowania ‘labiryntów‘ przy pomocy symulowanego wyżarzania. Funkcja kosztu została określona jako kombinacja liniowa różnych sąsiedztw:

- 8-sąsiedztwo z karą za czarne piksele (w celu promowania korytarzy białych),
- sąsiedztwo kwadratowe w odległości 2 od piksela z karą za białe piksele (w celu promowania korytarzy o szerokości 1),
- 16-sąsiedztwo z karą za czarne piksele (z powodu efektów otrzymanych wcześniej),

Przykładowe wyniki otrzymane w ten sposób znajdują się poniżej. Notacja pod obrazkiem jest wyrażona w postaci: (8[A], S[B], 16[C]), gdzie $Y \in (4, 8)$, co oznacza: 8-sąsiedztwo z wagą A, sąsiedztwo kwadratowe z wagą B, 16-sąsiedztwo z wagą C:



Rysunek 21: (8[-1], S[4], 16[-2]), spadek temperatury liniowy $\delta = 0.4$



Rysunek 22: (8[-1], S[2], 16[-2]), spadek temperatury wykładniczy $\delta = 0.4$

4 Problem sudoku

4.1 Przestrzeń stanów sudoku

Plansza do sudoku składa się z 9 kwadratów, każde zawierające 9 pól. Rozwiązanie jest poprawne gdy w każdej kolumnie, rzędzie oraz w kwadracie cyfry są unikalne. W ten sposób otrzymujemy 81 pól do wypełnienia. Najprostszą formą przestrzeni stanów jest przedstawienie jej jako permutacji wszystkich cyfr rozwiązania (9 cyfr 9-krotnie powtórzonych). Możemy jednak zmniejszyć tą przestrzeń stanów pozbywając się równocześnie jednego z trzech warunków poprawności. Każdy z kwadratów sudoku będzie permutacją cyfr, a plansza do sudoku zbiorem dziewięciu takich kwadratów. Biorąc pod uwagę znajomość początkowego stanu nierozwiązanego sudoku X liczba stanów dla takiej planszy wynosi:

$$|S_X| = \Pi_{i=1}^{9}(9 - |B_i(X)|)!$$

Gdzie $B_i(X)$ jest liczbą znanych pól w i-tym kwadracie zagadki.

4.2 Generowanie stanu sąsiedniego

W związku z powyższym określeniem stanów sudoku, podczas zamiany trzeba zwrócić na zachowanie tej samej kolekcji cyfr w każdym z kwadratów. Z tego powodu algorytm tworzenia sąsiedztwa będzie wybierał losowo jeden z kwadratów (w którym można dokonać zamiany) a następnie zamieniał 2 losowe pola wewnątrz pola.

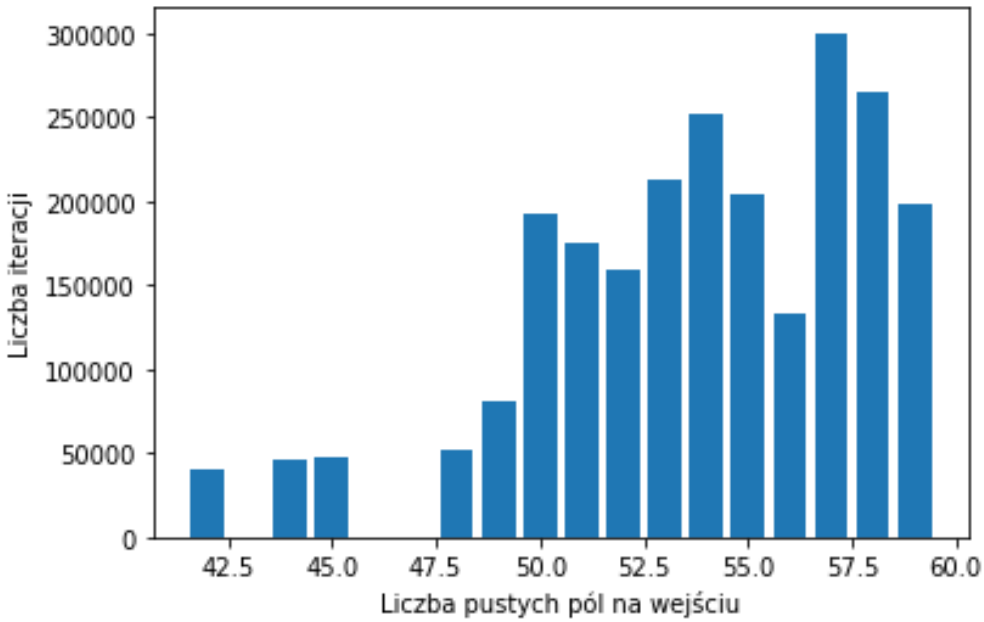
4.3 Modyfikacje algorytmu wyżarzania

W przypadku tego zadania, do algorytmu wyżarzania zostało zaimplementowane ponowne podgrzewanie układu. W przypadku gdy program trafi do lokalnego optimum a temperatura układu spadnie poniżej wartości ϵ algorytm zamiast przerywać działanie, ustawia temperaturę ponownie na T_0. Dzięki temu algorytm ma szansę wyjścia z niepoprawnego optimum i ponownego rozwiązania problemu.

4.4 Wyniki

Wykorzystana tutaj implementacja symulowanego wyżarzania z pakietu *simanneal* [1] wykorzystuje wykładniczy spadek temperatury oraz inny zestaw hiper parametrów. Model przyjmuje jako parametry temperaturę początkową, końcową oraz maksymalną liczbę iteracji. Współczynnik spadku temperatury jest wyliczany na podstawie powyższych parametrów.

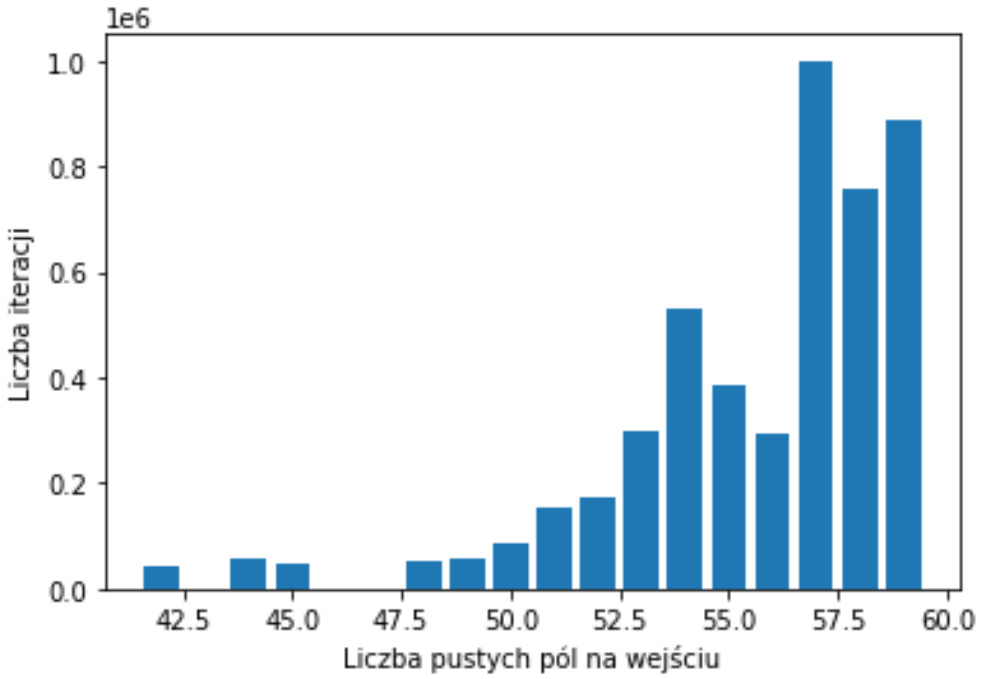
Po dostrojeniu modelu przy użyciu nastepujących hiper parametrów: $T_{max} = 10, T_{min} = 0.002$, $steps = 10^5$ oraz maksymalnej liczbie podgrzań równej 3, osiągnięto wskaźnik sukcesu na poziomie 72% na załączonym zestawie plansz [6]. W następujący sposób prezentuje się zależność pomiędzy liczbą iteracji a liczbą pustych miejsc na planszy:



Rysunek 23: Wykres słupkowy zależności pomiędzy liczbą pustych miejsc a liczbą iteracji, 3 ponowne podgrzania

Widoczna jest korelacja pomiędzy liczbą pustych pól a liczbą iteracji wyżarzania. Jednakże nie jest ona uwarunkowana tylko i wyłącznie od liczby pustych pól, ale także od rozłożenia ich na planszy.

Zwiększenie liczby podgrzań z 3 do 10 pozwoliło na wzrost wskaźnika sukcesu do 83%. Jednakże wydłużyło przy tym czas testowania dwukrotnie (z 13 minut 43 sekund do 26 minut 39 sekund), oraz zwiększyło liczbę iteracji dla plansz z liczbą pustych pól powyżej 50.



Rysunek 24: Wykres słupkowy zależności pomiędzy liczbą pustych miejsc a liczbą iteracji, 10 ponownych podgrzań

Literatura

[1] Matthew Perry *Python module for simulated annealing*
https://github.com/perrygeo/simanneal

[2] Rhyd Lewis. *Metaheuristics can Solve Sudoku Puzzles*
http://rhydlewis.eu/papers/META_CAN_SOLVE_SUDOKU.pdf

[3] Ai-Hua Zhou, Li-Peng Zhu, Bin Hu, Song Deng, Yan Song, Hongbin Qiu, Sen Pan *Traveling-Salesman-Problem Algorithm Based on Simulated Annealing and Gene-Expression Programming*
https://www.mdpi.com/2078-2489/10/1/7

[4] Wikipedia Simulated Annealing
https://en.wikipedia.org/wiki/Simulated_annealing

[5] Theodore W. Manikas, James T. Cain *Genetic Algorithms vs. Simulated Annealing: A Comparison of Approaches for Solving the Circuit Partitioning Problem*
https://scholar.smu.edu/engineering_compsci_research/1/

[6] Timo Mantere, Janne Koljonen *Sudoku research page*
http://lipas.uwasa.fi/~timan/sudoku/