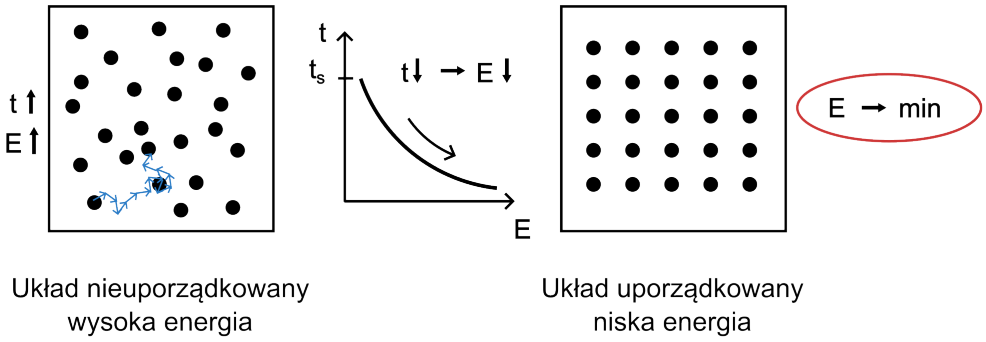
222800-1358 Nieklasyczne Metody Optymalizacji **Letni 2022/2023**

Ćwiczenia 3: Metoda Symulowanego Wyżarzania

Daniel Kaszyński *21-28 Marca 2023 r.*

3.1 Metoda Symulowanego Wyżarzania (Simulated Annealing)

Przedstawiona w latach '80 metoda Symulowanego Wyżarzania jest inspiracją ze świata fizyki nawiązującą do prawdziwego procesu metalurgicznego wyżarzania stali, patrz [KGV83]. Przyjmijmy że zadany jest układ cząsteczek mający wysoką temperaturę (sama koncepcja temperatury jest istotna ze względu na jego nazewnictwo w algorytmie); taki układ jest obdarzony wysoką energią, a zatem cząsteczki poruszają się w nim w sposób losowy z dużą prędkością. Jakbyśmy przyjrzeli się pojedynczej cząsteczce w tym układzie to poruszałaby się ona w sposób losowy po pewnej trajektorii oznaczonej na Rysunku 3.1 kolorem niebieskim; stan takiego układu jest nieuporządkowany. Następnie pozwólmy takiemu układowi się schłodzić, wraz ze spadkiem temperatury i energii układ przestanie się zachowywać w chaotyczny sposób i zacznie dążyć do stanu uporządkowanego, symetrycznego, stanu o minimalnej energii. Przejście takiego układu zobrazowano na rysunku 3.1, a uporządkowany układ często określany jest mianem modelu Metropolisa (od fizyka Nicholasa Metropolis).

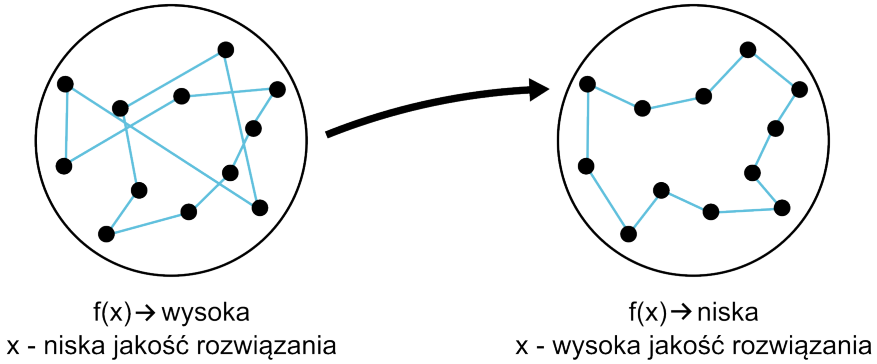


Rysunek 3.1: Przejście z chaotycznego układu do modelu Metropolisa

Zespół Kirkpatricka zajmował się problemami komiwojażera. Koncepcja wyżarzania miałaby pozwolić na przejście z układu chaotycznego (mało efektywne połączenia między miastami) do uporządkowanego (efektywne połączenia) jak ukazano na Rysunku 3.2. Wysoką energię układu można rozumieć jako wysoką wartość funkcji celu co cechuje się rozwiązaniem niskiej jakości, nastomiast minimalna energia oznacza niską wartość funkcji celu dające rozwiązanie wysokiej jakości. Algorytm S.A. jest algorytmem iteracyjnym definiującym przejście z punktu x_k do punktu x_{k+1} w przestrzeni rozwiązań $\mathcal{X}$ (ciągłej lub dyskretnej stąd niekoniecznie R^n).

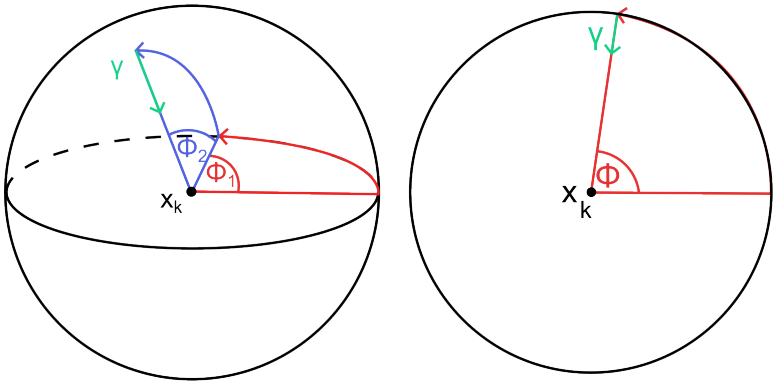
3.1.1 Przestrzeń ciągła

W przestrzeni ciągłej punkt x_{k+1} wybieramy z sąsiedztwa punktu x_k losując go z rozkładu jednostajnego i nazywamy go x^c_{k+1}. W przestrzeni ciągłej R^n do wyznaczenia sąsiedztwa możemy wykorzystać kulę opisaną



Rysunek 3.2: Rozwiązanie problemu komiwojażera jako zmiana energii w układzie

równaniem $N_F(x_k) = B(x_k, \delta)$ z środkiem w punkcie x_k i promieniu δ. Do określenia pozycji losowanego punktu w takiej kuli możemy wykorzystać dwa parametry ϕ_1 i ϕ_2, które będą opisywać obrót po sferze oraz parametr $\gamma \in [0, \delta]$ opisujący odległość od krawędzi sfery. Losując te trzy parametry, możemy określić losowanie każdego punktu w sąsiedztwie tego punktu, a wizualizację skutków przesunięcia tymi parametrami do punktu końcowego ukazano na Rysunku 3.3 (a). Oczywiście jest to przypadek w przestrzeni trójwymiarowej. W przypadku dwuwymiarowym zmniejsza się liczba potrzebnych parametrów do wyłącznie dwóch parametrów opisujących obrót i długości odcinka na okręgu, co ukazano na Rysunku 3.3 (b). W przypadku większej liczby wymiarów traktujemy sąsiedztwo jako hipersferę (co skutkuje dodaniem kolejnych parametrów obrotu), a w przypadku jednowymiarowym jako odcinek.

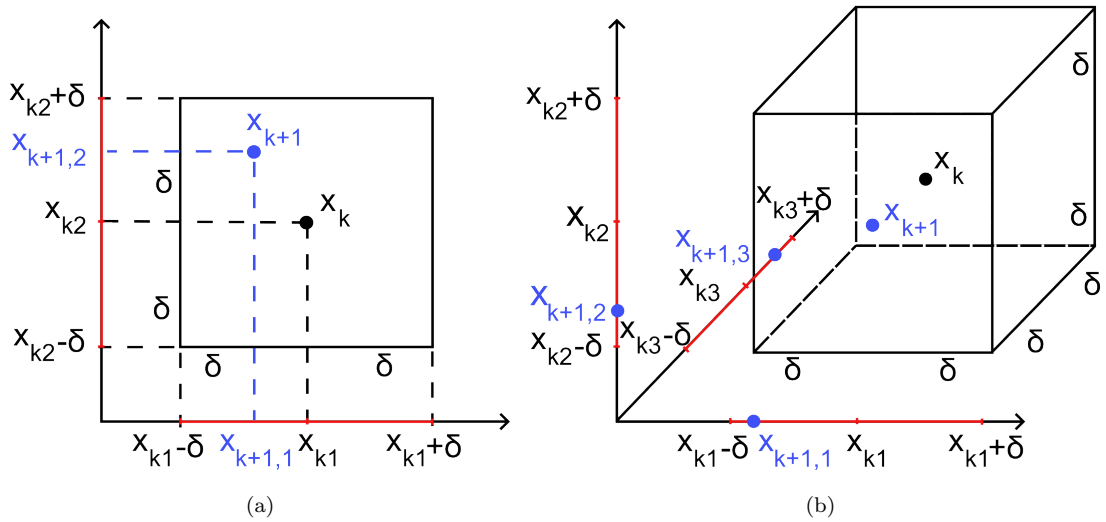


(a) (b)

Rysunek 3.3: Sąsiedztwo punktu opisane na kuli

W praktyce o wiele lepszą bryłą do określenia sąsiedztwa jest sześcian o długości krawędzi równej 2δ. Zaczynając od przestrzeni dwuwymiarowej, środek punktu w takim kwadracie ma współrzędne (x_{k1}, x_{k2}), a jego graficzną reprezentację ukazano na Rysunku 3.4 (a). Aby wylosować nowy punkt x_{k+1} należy wylosować wartość współczynnika z zakresu $[-\delta, \delta]$ o którą chcemy przesunąć punkt na osi x, a następnie powtórzyć losowanie na osi y. Taka implementacja jest znacznie szybsza dla komputera dzięki pominięciu obliczeń wartości funkcji trygonometrycznych. Analogicznie w trzech wymiarach musimy po prostu powtórzyć losowanie dodatkowo dla osi z (Rysunek 3.4 (b)), więc takie rozwiązanie również dobrze przenosi się na wyższe wymiary. Implementacja takiego rozwiązania w języku programowania R wyglądałaby następująco:

$$x^c_{k+1} = x_k + runif(n, -\delta, \delta) \tag{3.1}$$



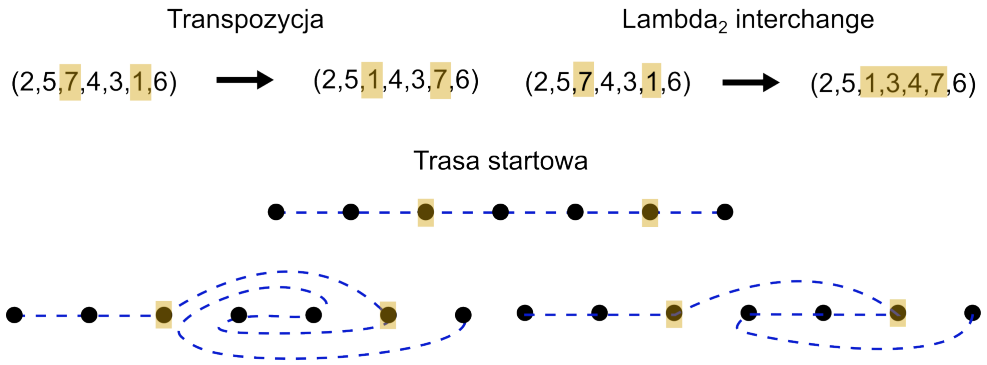
Rysunek 3.4: Sąsiedztwo punktu opisane na sześcianie

3.1.2 Przestrzeń dyskretna

W przypadku przestrzeni dyskretnej sytuacja zależy od tego jak wygląda przestrzeń rozwiązań. Dla przypadku komiwojażera, gdzie przestrzeń rozwiązań jest równa $x = (2, 5, 7, 4, 3, 1, 6)$ pojawia się problem z określeniem bliskości, gdyż nie mamy takiej dosłownej odległości jak w przestrzeni ciągłej. Stąd w przypadkach dyskretnych raczej mówi się o podobieństwie punktu niż o jego bliskości kojarzącej się z fizyczną odległością. Najczęściej w takim przypadku stosujemy pewien operator oznaczany przez $o(x_k)$; może to być na przykład inwersja polegająca na zamianie dwóch sąsiednich elementów. Równanie (3.2) przedstawia przykład wszystkich możliwych permutacji będących wynikiem inwersji na zbiorze składającym się z trzech miast.

$$x_k = (1, 3, 2) \quad o(x) = \{(3, 1, 2), (1, 2, 3), (3, 2, 1)\} \tag{3.2}$$

Innym sposobem na indukcję sąsiedztwa może być zastosowanie operatora transpozycji, który pozwala nam na zamienienie dwóch dowolnych elementów ze zbioru. W praktyce dla problemu komiwojażera stosuje często operator λ_2 *interchange*, który jest bardzo podobny do transpozycji, ale zamieniane są również punkty pośrednie. Rysunek 3.5 prezentuje na czym polegają i jak wyglądają takie trasy w przypadku obu operatorów. To co jest ważne, to fakt, że zastosowanie operatora jest zależne od natury problemu, niektóre problemy nie mogą bowiem być reprezentowane, jak w naszym przypadku, przez wektor kolejnych miast i będą wymagać zastosowania innego rodzaju operatora. Inne problemy mogą dotyczyć tego, że operator może nie być możliwy technicznie lub merytorycznie do zaimplementowania, ale w naszym przypadku zamiana kolejności odwiedzania miast jak najbardziej jest sensowna.



Rysunek 3.5: Operator transpozycji i λ_2 interchange

3.1.3 Reguła decyzyjna

Reguła decyzyjna jest taka sama jak w przypadku *weighted GS-RS*, ale prawdopodobieństwo dzielące przestrzeń decyzyjną dla siły globalnej algorytmu będzie wyliczane automatycznie w każdej iteracji. Do wyznaczenia reguły decyzyjnej w punkcie x^c_{k+1} wykorzystamy funkcję nazywaną **funkcją aktywacji** lub **funkcją akceptacji** oznaczoną symbolem A. Funkcja ta przyjmuje trzy parametry: wartość funkcji celu w potencjalnym nowym punkcie, wartość funkcji celu w obecnym punkcie oraz parametr temperatury. Parametr temperatury, tak jak w fizycznym modelu wyżarzania, będzie początkowo wysoką wartością, a z czasem działania algorytmu zacznie ona spadać. Zapis tej reguły decyzyjnej przedstawia równanie (3.3):

$$x_{k+1} = \begin{cases} x^c_{k+1} & \text{z p-stwem } A(f(x^c_{k+1}), f(x_k), t_k) \\ x_k & \text{z p-stwem } 1 - A(f(x^c_{k+1}), f(x_k), t_k) \end{cases} \tag{3.3}$$

Sposób aktualizacji wartości temperatury nazywamy *Annealing schedule* i występuje on najczęściej w sposób wykładniczy z parametrem α przyjmującym zazwyczaj wartości bliskie 1, np. 0.995. Wartość ta zależy najczęściej od tego ile chcemy wykonać iteracji. Przy mniejszych wartościach temperatura zaczyna spadać szybciej. Równanie (3.4) opisuje zmianę tego parametru w kolejnych krokach algorytmu:

$$t_{k+1} = \alpha\, t_k \quad \alpha \in (0, 1) \tag{3.4}$$

3.1.4 Funkcja aktywacji

Za funkcja aktywacji stosowaną w algorytmie S.A. najczęściej przyjmuję się funkcję Metropolisa. Zanim przedstawimy wzór tej funkcji warto zauważyć, że niezależnie od typu wybranej funkcji musi ona spełniać pewne własności:

1. Jeżeli nowy punkt jest lepszy to przechodzimy do niego zawsze:

$$f(x^c_{x+1}) < f(x_k), \quad x^c_{k+1} \succ x_k \Rightarrow A = 1 \tag{3.5}$$

2. Jeżeli nowy punkt jest gorszy to zweryfikujmy o ile i dostosujmy prawdopodobieństwo:

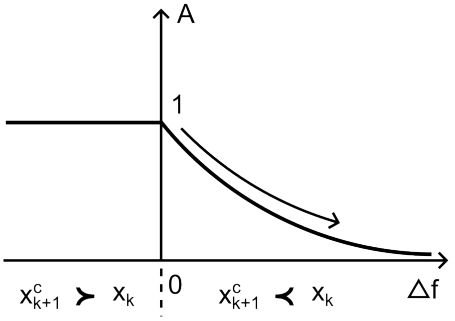
$$\begin{aligned} f(x^c_{x+1}) - f(x_k) &= \Delta f \\ f(x^c_{x+1}) > f(x_k) &= \Delta f > 0 \end{aligned} \tag{3.6}$$

   Zatem zależności między różnicą funkcji celu, a wartością funkcji aktywacji można przedstawić za pomocą funkcji przedstawionej na rysunku 3.6.

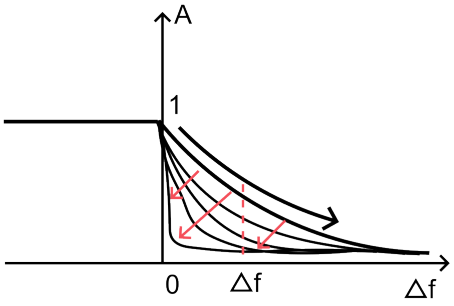
3. Wraz ze spadkiem temperatury między kolejnymi iteracjami wartość funkcji aktywacji powinna maleć:

$$t \downarrow => A \downarrow \tag{3.7}$$

   Wpływ tej zależności na funkcję aktywacji ukazuje rysunek 3.7.



Rysunek 3.6: Kształt funkcji aktywacji



Rysunek 3.7: Zmiana funkcji aktywacji w kolejnych iteracjach

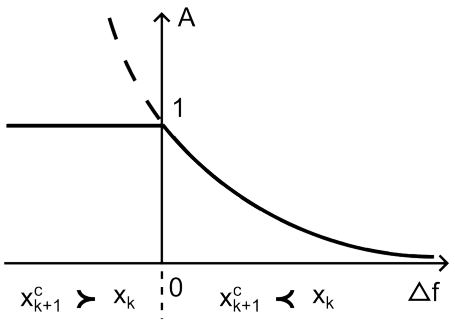
Znając wymagania stawiane funkcji aktywacji w algorytmie S.A. wprowadźmy zatem równanie opisujące funkcję Metropolisa:

$$A(f(x_k), f(x^c_{k+1}), t_k) = \min(1, e^{-\frac{\Delta f}{t_k}}) \tag{3.8}$$

Rysunek 3.8 ukazuje funkcję opisaną równaniem (3.8). Można następnie zweryfikować, czy spełnia ona stawiane jej założenia.

1. Wartości mniejsze od zera wynikające z członu e^{-x} są "ściągane" do wartości 1 przez człon wyboru wartości minimalnej.

2. Im większa różnica między rozwiązaniami tym mniejsze prawdopodobieństwo przejścia do nowego rozwiązania.



Rysunek 3.8: Funkcja Metropolisa

Sprawdźmy, czy spełnia ona trzecią zależność utrzymując stałą wartość Δt w iteracji $k+1$. Ponieważ wartość t_k jest w mianowniku potęgi liczby e i spada z iteracji na iterację, to powoduje to wzrost wartości potęgi e^x. Ponieważ wartość e^x jest w mianowniku to wzrastająca wartość w mianowniku ostatecznie skutkuje spadkiem całego członu, a zatem zostaje spełniona trzecia zależność.

$$e^{-\frac{\Delta t}{t_k}} = \frac{1}{\uparrow e^{\frac{\Delta t}{t_k}\downarrow}\uparrow} \downarrow \tag{3.9}$$

Graniczną formą algorytmu S.A. jest algorytm *Greedy Search*. Stosowany jest wtedy, kiedy parametr t_k jest bliski zeru. Z drugiej strony, na samym początku trwania algorytmu nie jest on dosłownie algorytmem *Random Search*. Należy również zauważyć, że przy bardzo dużych różnicach wartości funkcji celu, prawdopodobieństwo może być na tyle niskie, że nie przejdziemy do tego punktu. Zatem algorytm tworzy swego rodzaju blokadę niepozwalającą "zniszczyć" wypracowanej wartości funkcji celu.

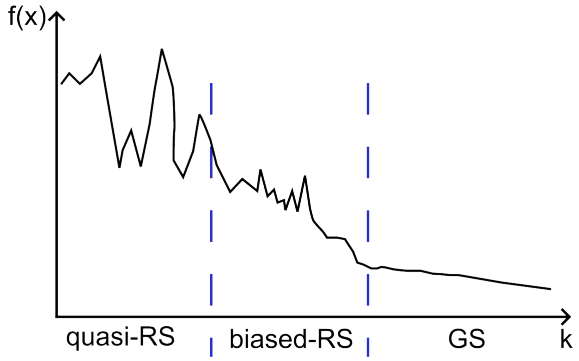
Wracając jeszcze do algorytmu GS-RS, widać wyraźne podobieństwo między tymi dwoma metodami, a to co je wyróżnia to właśnie endogenizacja parametru prawdopodobieństwa, tzn. w każdej iteracji wyznaczamy na nowo wartość prawdopodobieństwa przejścia do gorszego rozwiązania.

Uruchamiając algorytm S.A. w praktyce można zaobserwować trzy główne fazy działania algorytmu, wykres zmiany funkcji celu w kolejnych iteracjach przedstawiono na rysunku 3.9. Pierwsza faza to jest w zasadzie spacer losowy, wartość funkcji celu swobodnie fluktuuje i można ją nazwać fazą *quasi-RS*. Druga faza, gdy już parametr temperatury trochę spadnie, mamy tzw. *biased-RS* i częstotliwość wybierania rozwiązań gorszych znacząco maleje. Ostatnia faza, gdy temperatura już jest bliska zeru mamy już algorytm niemal w pełni przypominający *Greedy Search*.

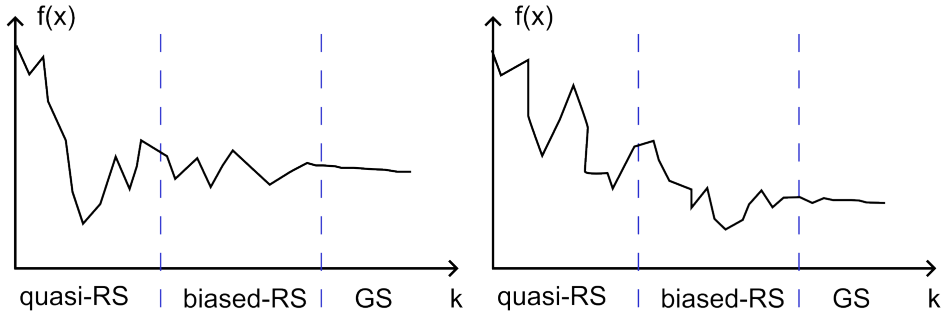
To na co warto zwrócić uwagę to fakt, że nie zawsze na sam koniec działania algorytmu dostaniemy najlepsze wyniki. Możemy znaleźć je w pierwszej lub drugiej fazie działania i otrzymać dużo lepsze wyniki, jednak siła mocy globalnej może być na tyle duża, że będziemy dalej przeszukiwać przestrzeń rozwiązań i opuścimy te lepsze ekstremum. Rysunek 3.10 ukazuje znalezienie takich wyników w pierwszej i drugiej fazie działania algorytmu wyżarzania symulowanego.

Implementację algorytmu S.A. przedstawiono na Listingu 1

```
simulated_annealing <- function(f, x, d = 0.05, t = 100, a = 0.995, K = 100){
  #' SIMULATED ANNEALING
  #'
```



Rysunek 3.9: Wartość funkcji celu w kolejnych iteracjach działania algorytmu S.A



Rysunek 3.10: Inne przypadki zmiany wartości funkcji celu w algorytmie S.A.

```
#' INPUT:
#'       - f: funkcja celu
#'       - x: punkt poczatkowy
#'       - d: dlugosc sasiedztwa
#'       - t: temperatura poczatkowa
#'       - a: wspolczynnik spadku temperatury
#'       - K: maksimum iteracji
#'
#' OUTPUT:
#'       - x_opt: znalezione rozwiazanie
#'       - f_opt: wartosc funkcji celu w rozwiazaniu
#'       - x_hist: historia przeszukiwanych rozwiazan
#'       - f_hist: historia wartosci funkcji celu
#'       - a_k: historia wartosci funckji aktywacji
#'       - t_k: historia wartosci temperatury
#'       - t_eval: czas trwania dzialania algorytmu

start_time <- Sys.time()
results <- list(x_opt = x,
                f_opt = f(x),
                x_hist = matrix(NA, nrow = K, ncol = length(x)),
                f_hist = rep(NA, K),
                a_k = rep(NA, K),
                t_k = rep(NA, K),
```

```
                    t_eval = NA )

  results$x_hist[1,] <- x
  results$f_hist[1] <- f(x)
  results$a_k[1] <- 1
  results$t_k[1] <- 1

  for(k in 2:  K){
    # wybor punktu kandydata
    x_c <- x + runif(length(x), min = -d, max = d)
    # wartosci funkcji celu w punkcie obecnym i kandydacie
    f_c <- f(x_c)
    f_k <- f(x)
    # wyznaczenie wartosci funkcji aktywacji
    A_k <- min(1, exp(-(f_c - f_k)/(t)))

    # warunek przejscia do gorszego rozwiazania
    if (runif(1) < A_k){
      x <- x_c

      if(f_c < results$f_opt){
        results$x_opt <- x
        results$f_opt <- f_c
      }
    }

    results$x_hist[k,] <- x
    results$f_hist[k] <- f_c
    # zmiana temperatury w kolejnych iteracjach
    t <- t * a
    results$a_k[k] <- A_k
    results$t_k[k] <- t
  }
  results$t_eval <- Sys.time() - start_time
  return(results)
}
```

Listing 1: Implementacja algorytmu Simulated Annealing

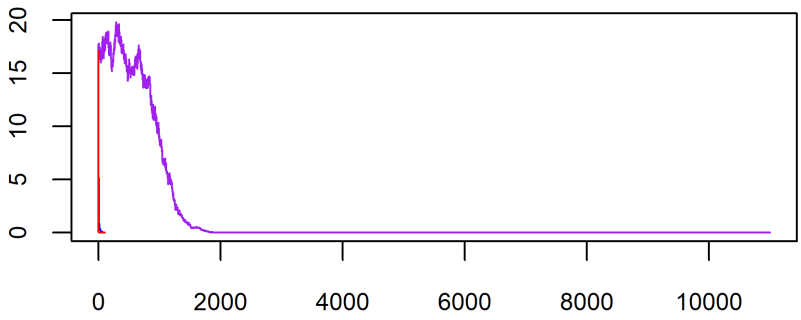
Listing 2 przedstawia kod wykorzystany do porównania algorytmu SA z algorytmami GD i SD.

```
library(numDeriv)
source("grad_descent.R")
source("steepest_descent.R")
source("simulated_annealing.R")

# cost function
f <- function(x) 1/8*x[1]^2 + x[2]^2
# inital vector
x <- c(3, 4)

gd <- gradient_descent(f, x, a = 0.1, K = 100)
sd <- steepest_descent(f, x, a = 2, grid_size = 100, K = 100)
sa <- simulated_annealing(f, x, K = 1000)

yy <- seq(-6, 6, length = 400)
xx <- seq(-9, 9, length = 400)
ff <- function(x1, x2) 2*x1^2 + x2^2
zz <- outer(xx,yy, ff)

# eksploracji przestrzeni
contour(xx, yy, zz, asp = T)
lines(gd$x_hist, type='l', col ='red')
lines(sd$x_hist, type='l', col ='blue')
```

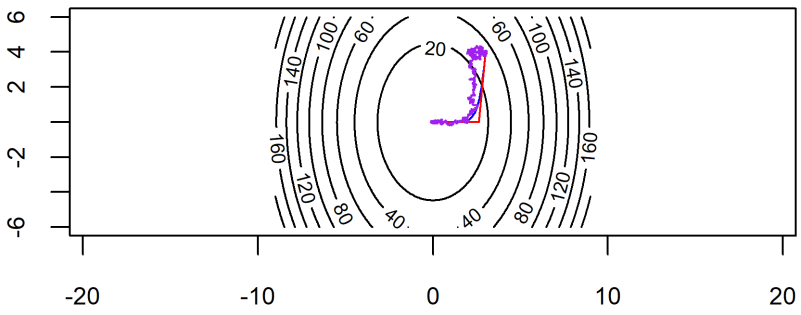
```
lines(sa$x_hist, type="l", col='purple')

# wartosci funkcji celu
plot(sa$f_hist, type="l", col='purple')
lines(gd$f_hist, type='l', col ='red')
lines(sd$f_hist, type='l', col ='blue')

# wartosc funkcji aktywacji
plot(sa$a_k)
```

Listing 2: Kod wykorzystany do porównania algorytmów SA z algorytmami GD i SD

Na Rysunku 3.11 przedstawiono wartość funkcji celu w kolejnych iteracjach algorytmu S.A. Algorytmowi temu zajmuje dużo dłużej zejście do minimum globalnego, co wynika z globalnej natury algorytmu w początkowych etapach jego działania; trajektorię tę oznaczono fioletowym kolorem na Rysunku 3.12. Rysunek 3.13 przedstawia wartości funkcji aktywacji w kolejnych iteracjach.



Rysunek 3.11: Wartość funkcji celu w kolejnych iteracjach algorytmu S.A.

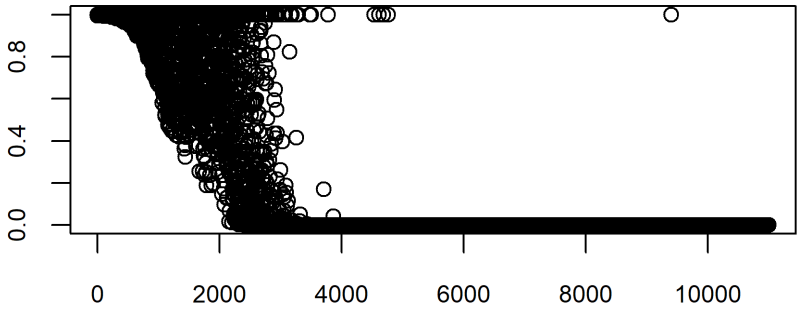


Rysunek 3.12: Trajektoria wektora x w kolejnych iteracjach algorytmów SA, GD i SD

Następnie Listing 3 przedstawia implementację działania algorytmów na bardziej skomplikowanym przypadku wykorzystując funkcję celu w postaci funkcji Schaffera:

$$f(x,y) = 0.5 + \frac{sin^2(x^2 - y^2) - 0.5}{[1 + 0.001(x^2 + y^2)]^2} \tag{3.10}$$

```
library(numDeriv)
source("grad_descent.R")
```

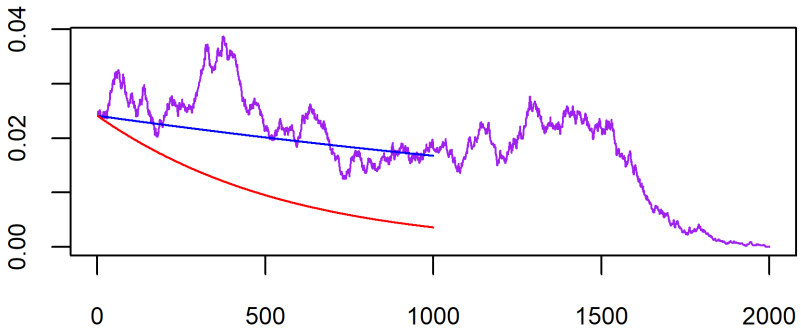


Rysunek 3.13: Wartość funkcji aktywacji w kolejnych iteracjach algorytmu SA

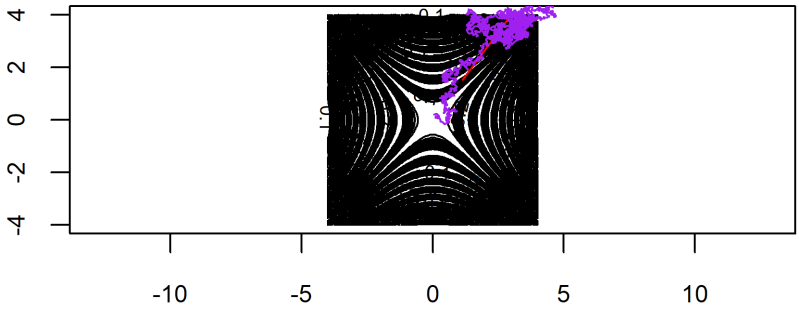
```
source("steepest_descent.R")
source("simulated_annealing.R")

# funkcja celu = funckja Shaffera
f <- function(x){
  outp1 = (sin(x[1]^2 - x[1]^2))^2 - 0.5
  outp2 = (1 + 0.001*(x[1]^2 + x[2]^2))^2
  out = 0.50 + outp1/outp2
  return(out)
}

xx <- seq(-4, 4, length = 100)
ff <- function(x1, x2) {
  outp1 = (sin(x1^2 - x2^2))^2 - 0.5
  outp2 = (1 + 0.001*(x1^2 + x2^2))^2
  out = 0.50 + outp1/outp2
  return(out)
}
zz <- outer(xx,xx, ff)


# inital vector
x <- c(3, 4)
gd <- gradient_descent(f, x, a = 0.1, K = 1000)
sd <- steepest_descent(f, x, a = 0.5, grid_size = 100, K = 1000)
sa <- simulated_annealing(f, x, t = 5, d = 0.1, K = 2000)

# eksploracja
contour(xx, xx, zz, asp = T)
lines(gd$x_hist, type='l', col = 'red')
lines(sd$x_hist, type='l', col = 'blue')
lines(sa$x_hist, type="l", col= 'purple')


## Rysunek wartosci funkcji celu
plot(sa$f_hist, type="l", col = 'purple')
lines(gd$f_hist, type='l', col = 'red')
lines(sd$f_hist, type='l', col = 'blue')
```

Listing 3: Kod wykorzystany do porównania algorytmów SA z algorytmami GD i SD z funkcją aktywacji Schaffer'a

Na Rysunku 3.14 przedstawiono funkcję celu w kolejnych iteracjach działania algorytmów. Funkcja ta sprawia znaczącą trudność algorytmom GD i SD przy zbyt niskich wartościach parametru α. Budowaną trajektorię przez te algorytmy można zaobserwować na Rysunku 3.15. Jak szybko i czy w ogóle algorytm S.A. zbiegnie do minimum globalnego, jest zależne od wprowadzonych parametrów 3.15.



Rysunek 3.14: Wartość funkcji celu w kolejnych iteracjach algorytmu S.A.



Rysunek 3.15: Trajektoria wektora x w kolejnych iteracjach algorytmów S.A., GD i SD

Podsumowując, wyżarzanie symulowane jest algorytmem globalnym, tzn. posiada możliwość wyjścia z ekstremum lokalnego ponieważ zezwalamy na pogorszenie wartości funkcji celu w czasie działania algorytmu. Istnieje jeszcze pewna właściwość algorytmu S.A. mówiąca, że przy pewnych założeniach funkcji celu można wykazać, że algorytm niezależnie od punktu w jakim go zainicjujemy zawsze zbiegnie do ekstremum globalnego. Założenie dotyczy regularności funkcji celu oraz nieskończonej liczby iteracji. Znaczy to tyle, że przy nieskończenie wielkiej liczbie iteracji przeszukamy całą przestrzeń rozwiązań.

Bibliografia

[KGV83] S. Kirkpatrick and C.D. Gelatt and M.P. Vecchi, "Optimization by simulated annealing, "*science*, 1983.