131890-S Metody Optymalizacji **Letni 2023/2024**

Zajęcia 6: Simulated Annealing

Daniel Kaszyński

6.1 Funkcja Schaffer

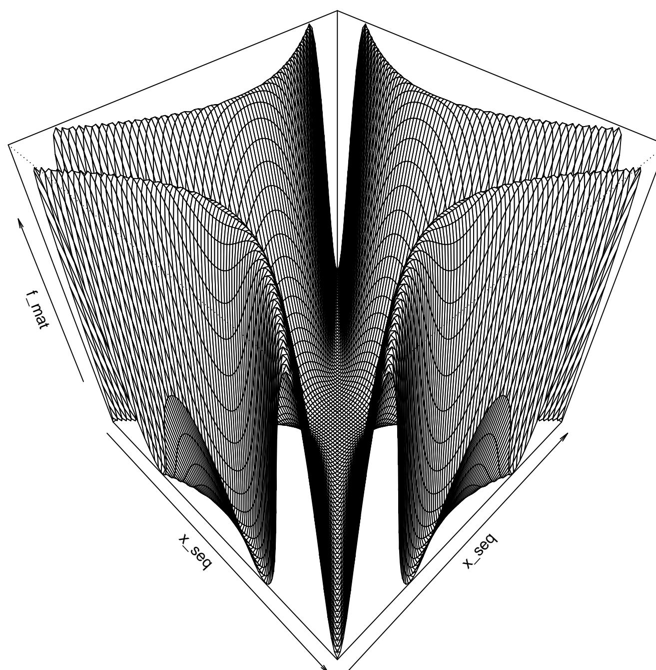
Przy pracy nad metodami optymalizacyjnymi nieraz zdarza się, iż musimy dokładnie przetestować działanie tworzonych przez nas algorytmów. Proste funkcje tylko do pewnego stopnia zapewniają nam możliwość sprawdzenia działania naszych rozwiązań. Chcąc sprawdzić charakterystyki metod optymalizacji na trudniejszych przypadkach warto jest sięgnąć po bardziej złożone funkcje.

Istnieje wiele złożonych funkcji, na których można testować złożone algorytmy optymalizacyjne. Lista przykładowych funkcji testowych umieszczona jest między innymi na stronie Wikipedii.

Jedną z popularnych funkcji do testowania algorytmów optymalizacyjnych jest funkcja Schaffer-a. Wyrażana jest ona przy pomocy wzoru:

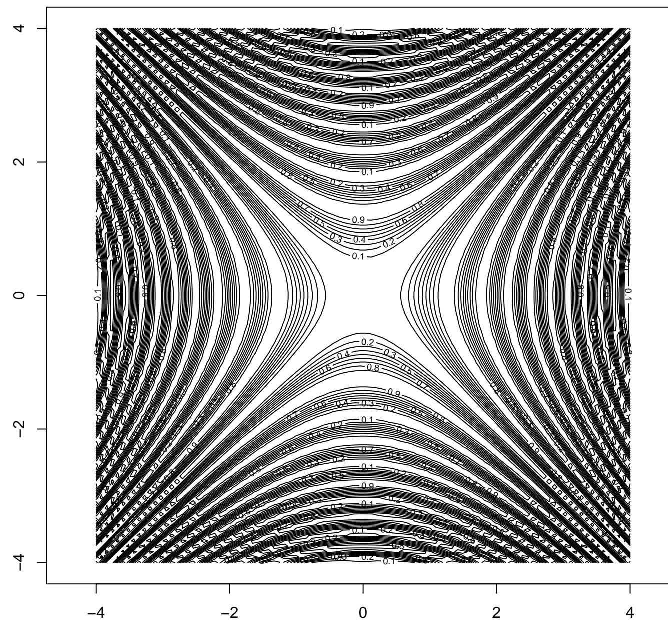
$$f(x_1, x_2) = 0.5 + \frac{sin^2(x^2 - y^2) - 0.5}{[1 + 0.001(x^2 + y^2)]^2} \tag{6.1}$$

Funkcja ta posiada minimum w punkcie $f(0, 0) = 0$ i przyjmuje wartości z zakresu $< 0, 1 >$.



Rysunek 6.1: Wykres funkcji Schaffer-a w przestrzeni 3D.

Na wykresie funkcji możemy zaobserwować, iż w okolicach punktu $(0, 0)$ znajduje się dolina zawierająca ekstremum globalne funkcji. Dolina ta przybiera kształt "X"i otoczona jest drastycznymi skokami w wartościach funkcji. Przestrzeń leżąca za tymi skokami charakteryzuje się występowaniem spadków i wzrostów wartości funkcji we wzorcu przypominającym falowanie. Warto również zauważyć, iż warstwice fal leżą prawie że równolegle do siebie nawzajem.



Rysunek 6.2: Wykres warstwicy funkcji Schaffer-a.

Funkcja ta znakomicie nadaje się do testowania zdolności algorytmów do eksploracji różnych sekcji przestrzeni funkcji w celu odnalezienia globalnego ekstremum. Funkcja Schaffer-a posiada zróżnicowaną topografię. Bez odpowiedniej eksploracji, algorytmy mogą utknąć w ekstremum lokalnym. Może to prowadzić do suboptymalnych rozwiązań, które nie są globalnie najlepsze.

Aby uniknąć utknięcia w ekstremum lokalnym, można zastosować kilka różnych technik eksploracyjnych. Należą do nich między innymi metody stochastyczne, czyli algorytmy oparte na losowości, takie jak algorytmy ewolucyjne czy algorytmy genetyczne. Innymi sposobami mogą być metody wielopunktowe, populacyjne czy adaptacyjne strategie eksploracyjne.

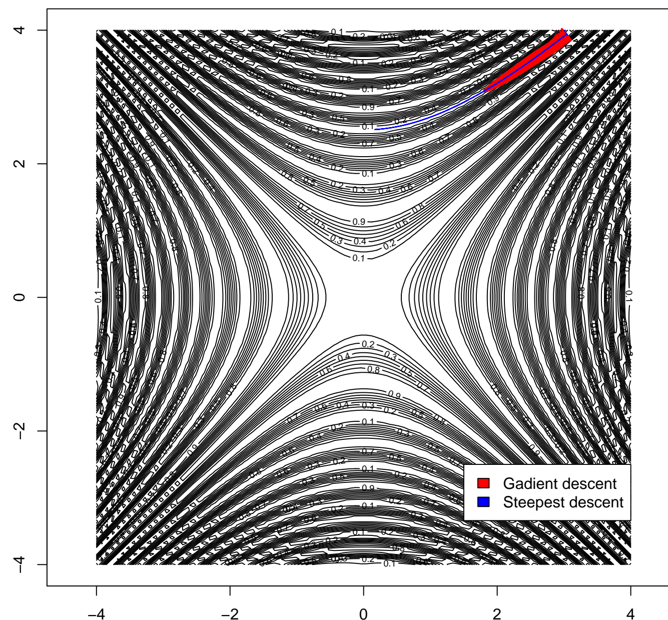
6.2 Testy algorytmów przeszukiwania lokalnego

Na początek warto ustalić, jak na funkcji Schaffer-a zachowują się poznane przez nas na poprzednich wykładach algorytmy optymalizacyjne, takie jak algorytm metody gradientu prostego (gradient descent) czy metoda najszybszego spadku (steepest descent).

Przykład 1. Przyjmijmy jako rozpatrywaną funkcję f funkcję Schaffer-a oraz punkt startowy $x_{start} = (3, 4)$.

Ponadto, przyjmijmy jako paramety metody gradientu prostego wartości learning rate $\alpha = 0.02$ oraz maksymalny limit iteracji $K = 30000$. Podobnie przy metodzie najszybszego spadku załóżmy, że maksymalny learning rate $\alpha = 5$, ilość iteracji przeszukiwania najlepszego learning rate $g = 1000$ oraz maksymalny limit iteracji $K = 30000$.

Przy podjęciu próby optymalizacji funkcji z punktu startowego x_{start} przy pomocy algorytmu metody gradientu prostego oraz metody najszybszego spadku udało się zbliżyć do ekstremum lokalnego funkcji. Ścieżki uzyskane w kolejnych krokach iteracji tych algorytmów możemy zaobserwować na wykresie poniżej:



Rysunek 6.3: Ścieżki stworzone przez algorytmy gradient descent oraz steepest descent na bazie funkcji Schaffer-a.

Łatwo można zaobserwować, iż żadnemu z algorytmów nie udało się wydostać z początkowej doliny, w której okolicy znajdował się punkt startowy x_{start}. Poruszały się one bardzo wolno i wymagały przy tym dużej liczby iteracji. Co więcej, utknęły one dążąc do lokalnego minimum funkcji. Takie zjawisko nazywa się pułapką lokalnego ekstremum. Na podstawie tego można wnioskować, że gradient descent oraz steepest descent nie posiadają własności przeszukiwania globalnego (ang. *global search*). W przypadku takich funkcji jak funkcja Schaffer-a warto jest wprowadzić mechanizmy ucieczki pozwalające na przeszukiwanie innych regionów.

6.3 Simulated Annealing

Symulowane wyżarzanie (ang. *simulated annealing*) to probabilistyczny algorytm optymalizacji inspirowany procesem wyżarzania w metalurgii. W metalurgii wyżarzanie jest techniką stosowaną w celu zmniejszenia defektów i poprawy struktury krystalicznej materiałów poprzez ich ogrzewanie, a następnie stopniowe

chłodzenie. Podobnie w kontekście optymalizacji symulowane wyżarzanie służy do znalezienia globalnego minimum funkcji poprzez naśladowanie procesu stopniowego chłodzenia.

Pierwszym krokiem algorytmu *simulated annealing* jest jego inicjalizacja. W tej fazie ustalane są parametry startowe algorytmu, do których należą:

- f - rozpatrywana funkcja celu,
- x_0 - punkt startowy algorytmu,
- d - otoczenie,
- t_0 - temperatura początkowa,
- α - spadek temperatury,
- K - liczba iteracji algorytmu,

oraz parametry wewnętrzne takie jak:

- A_k - wartość funkcji aktywacji,
- t_k - temperatura w każdej iteracji.

Symulowane wyżarzanie wykorzystuje parametr temperatury, który kontroluje prawdopodobieństwo przyjęcia gorszych rozwiązań w miarę postępu algorytmu. Początkowo temperaturę ustawia się na wysoką wartość i stopniowo zmniejsza się ją w kolejnych iteracjach, zgodnie z przyjętą strategią wyżarzania.

W każdej iteracji algorytmu generowane jest rozwiązanie sąsiednie. Tego sąsiada uzyskuje się dokonując niewielkiej losowej zmiany w bieżącym rozwiązaniu w obrębie otoczenia d. Potem następuje ocena funkcji celu dla bieżącego rozwiązania oraz sąsiedniego rozwiązania. Jeżeli sąsiednie rozwiązanie jest lepsze (tj. ma niższą wartość funkcji celu), akceptuje się je jako nowe, aktualne rozwiązanie. Jeżeli rozwiązanie sąsiada jest gorsze, należy je zaakceptować z prawdopodobieństwem określonym przez funkcję aktywacji A_k i aktualną temperaturę t_k. Ta funkcja aktywacji (lub inaczej prawdopodobieństwa) pozwala algorytmowi czasami akceptować gorsze rozwiązania na początku procesu optymalizacji, co pomaga zapobiegać utknięciu algorytmu w lokalnych minimach. Następnie obniżana jest temperatura zgodnie z przyjętą strategią. Wraz ze spadkiem temperatury maleje prawdopodobieństwo przyjęcia gorszych rozwiązań i zwiększa się prawdopodobieństwo zbieżności algorytmu w stronę minimum globalnego. Kroki te powtarzane są w kolejnych iteracjach, aż do spełnienia kryterium zatrzymania, jekim może być osiągnięcie maksymalnej liczby iteracji lub osiągnięcie zadowalającego rozwiązania.

Symulowane wyżarzanie jest skuteczne w znajdowaniu niemal optymalnych rozwiązań złożonych problemów optymalizacyjnych, w których tradycyjne metody oparte na gradiencie mogą sprawiać problemy, szczególnie gdy funkcja celu jest niewypukła lub zaszumiona. Pozwalając algorytmowi okazjonalnie akceptować gorsze rozwiązania, symulowane wyżarzanie może zbadać szerszy zakres przestrzeni rozwiązań i uniknąć uwięzienia w lokalnych minimach.

Z wykorzystaniem języka programowania **R** możemy napisać własną implementację algorytmu *simulated annealing*. Przykładowa implementacja tej metody może wyglądać następująco:

```
1 simulated_annealing <- function(f, x0, d, t0, a, K = 100){
2   #' Symulowane Wyzarzanie
3   #'
4   #' @description Funkcja odpowiedzialna za aproksymacje ekstremum globalnego
5   #' dla funkcji f w K krokachprzy pomocy algorytmu symulowanego wyzarzania
6   #' (ang. simulated annealing).
7   #'
8   #' @param f funkcja. Funkcja celu algorytmu.
```

```
#' @param x0 wektor numeryczny. Punkt startowy algorytmu.
#' @param d skalar. Badane otoczenie.
#' @param t0 skalar. Temperatura poczatkowa.
#' @param a skalar. Parametr okreslajacy tempo spadku temperatury.
#' @param K skalar. Opcjonalny parametr okreslajacy maksymalny limit iteracji (domyslnie
  100).
#'
#' @usage simulated_annealing(f, x0, d, t0, a, K)
#'
#' @returns
#' Lista wynikow zawierajaca nastepujace elementy:
#' * x_opt: znalezione rozwiazanie,
#' * f_opt: wartosc funkcji docelowej w znalezionym rozwiazaniu,
#' * x_hist: historia badanych rozwiazan,
#' * f_hist: historia wartosci funkcji docelowej,
#' * t_eval: czas, jaki uplynal podczas obliczen algorytmu.

start_time <- Sys.time()
n <- length(x0)
results <- list(x_opt = x0,
                f_opt = f(x0),
                x_hist = matrix(NA, nrow = K, ncol = n),
                f_hist = rep(NA, K),
                A_k = rep(NA, K),
                t_k = rep(NA, K),
                t_eval = NA)

results$x_hist[1,] <- x0
results$f_hist[1] <- f(x0)

x <- x0
t_k <- t0
for(k in 2:  K){
  # opis wyboru sasiedniego rozwiazania z otoczenia
  x_c <- x + runif(n, min = -d, max = d)
  A_k <- min(1, exp(-(f(x_c) - f(x)) / (t_k)))

  # sprawdzenie, czy nowe rozwiazanie
  # powinno zostac przyjete jako nowe
  # oraz czy jest najlepsze do tej pory
  if(runif(1) < A_k){
    x <- x_c
    if(f(x) < results$f_opt){
      results$x_opt <- x
      results$f_opt <- f(x)
    }
  }

  results$x_hist[k,] <- x
  results$f_hist[k] <- f(x)

  results$A_k[k] <- A_k
  results$t_k[k] <- t_k

  t_k <- t_k*a
}

# roznica czasu pomiedzy koncem i poczatkiem algorytmu
results$t_eval <- Sys.time() - start_time
return(results)
}
```

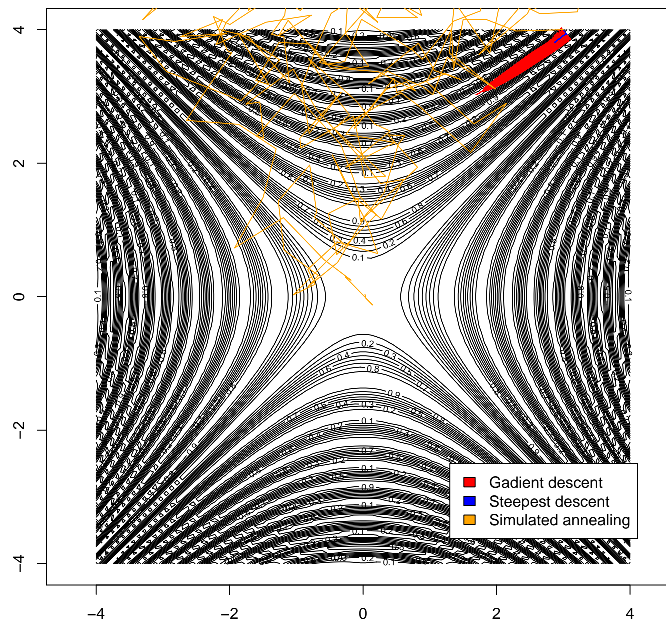
Listing 1: Implementacja algorytmu *simulated annealing*

Warto zwrócić uwagę na to, że do wyboru sąsiedniego rozwiązania z otoczenia wykorzystano funkcję *runif()*. Funkcja ta dostarcza losowe odchylenia zgodnie z rozkładem normalnym w podanym przedziale od *min* do *max*.

Przykład 2. Przyjmijmy jako rozpatrywaną funkcję f funkcję Schaffer-a oraz punkt startowy $x_{start} = (3, 4)$.

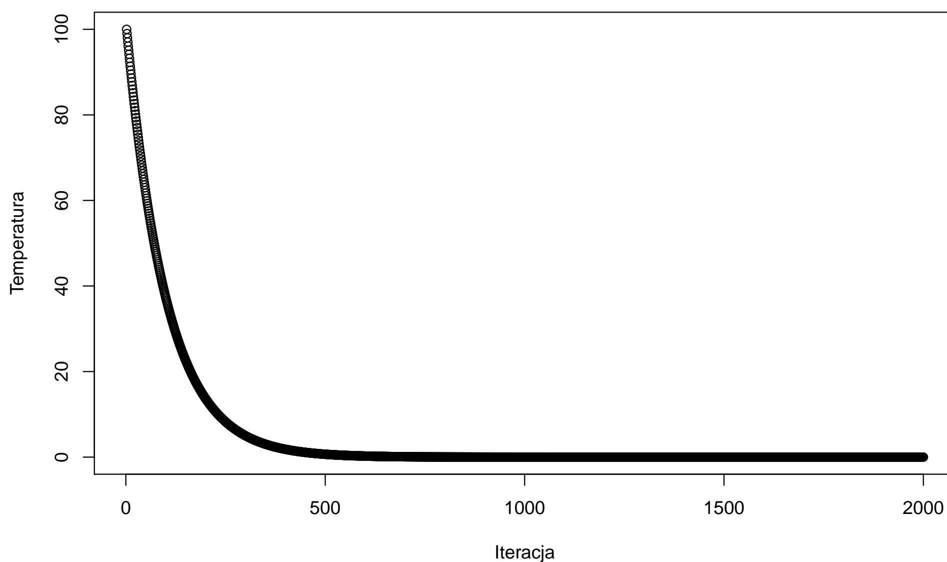
Ponadto, przyjmijmy jako paramety metody gradientu prostego oraz metody najszybszego spadku takie same wartości jak w przypadku poprzedniego przykładu z wyjątkiem zmniejszenia maksymalnej iteracji do 2000. Załóżmy również, iż algorytm *simulated annealing* przyjmuje parametry $d = 0.8$, $t_0 = 100$, $\alpha = 0.99$ oraz maksymalny limit iteracji $K = 2000$.

Przy podjęciu próby optymalizacji funkcji z punktu startowego x_{start} przy pomocy algorytmu *simulated annealing* możemy zauważyć, iż eksploruje on rózne regiony badanej funkcji. Nie zatrzymuje sie on w minimach lokalnych a nawet opuszcza ekstermum lokalne w celu badania dalszych wartości. Ścieżki uzyskane w kolejnych krokach iteracji tych algorytmów możemy zaobserwować na wykresie poniżej:



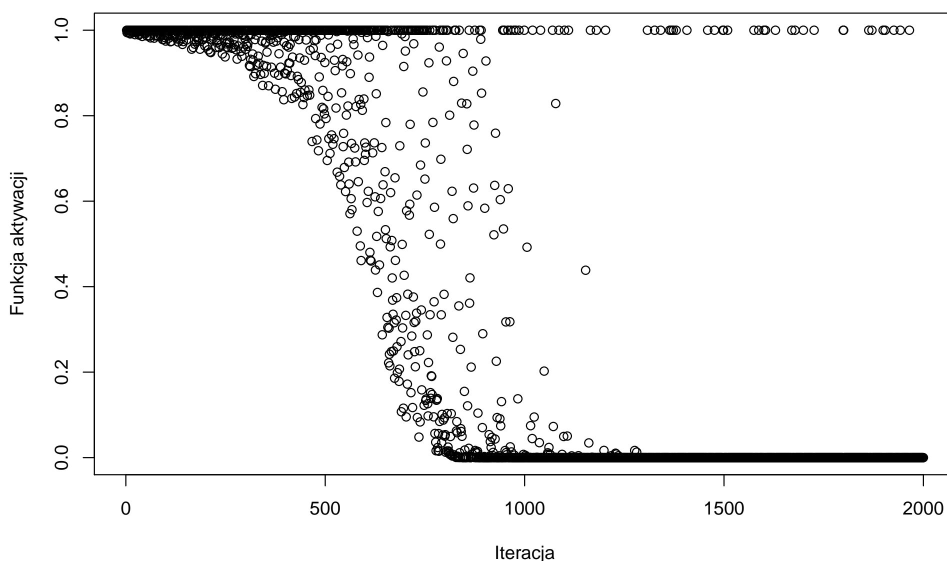
Rysunek 6.4: Ścieżka stworzona przez przeszukiwanie algorytmem *simulated annealing* w porównaniu do gradient descent oraz steepest descent na bazie funkcji Schaffer-a.

Podczas kolejnych iteracji algorytmu wartość parametru temperatury t_k sukcesywanie spada. Zmniejsza to losowość występującą w zachowaniu algorytmu podczas przeszukiwania i koncentruje go na dotychczas najlepszym znalezionym rozwiązaniu. Algorytm z czasem więc przestaje skupiać się na eksploracji nowych wartości, a stara się zoptymalizować najlepsze dotychczasowe rozwiązanie. Spadek temperatury w kolejnych iteracjach algorytmu możemy zaobserwować na wykresie poniżej:



Rysunek 6.5: Wykres wartości temperatury na przestrzeni kolejnych iteracji algorytmu.

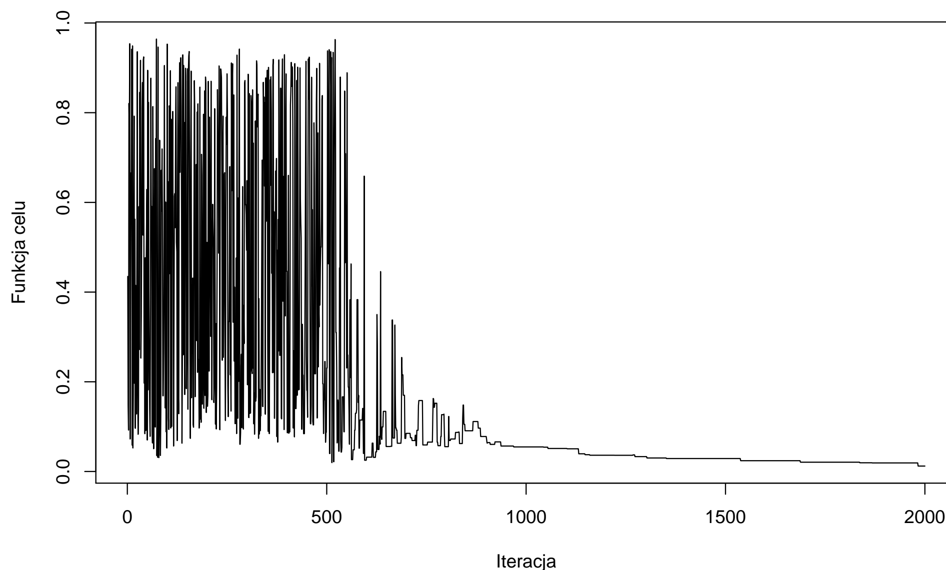
Warto również przyjrzeć się wartościom funkcji aktywacji w kolejnych iteracjach algorytmu. Spadają one na przestrzeni działania algorytmu od wartości bliskich 1 do wartości bliskich 0. Wartości funkcji aktywacji możemy zobaczyć na wykresie poniżej:



Rysunek 6.6: Wykres wartości funkcji aktywacji na przestrzeni kolejnych iteracji algorytmu.

Podczas wyszukiwania rozwiązania można wyróżnic trzy fazy, łatwe do zaobserwowania na wykresie funkcji aktywacji. Na początku funkcja aktywacji posiada duże wartości (mniej więcej do 200-300 iteracji). Oznacza to, że algorytm często akceptuje rozpatrywanych kandydatów na optymalny x, nawet jeśli wartości funkcji celu nie są w ich przypadku pożądane. Potem następuje druga faza, w której dobre rozwiązania mają wysokie prawdopodobieństwo, a słabe rozwiązania mają niskie prawdopodobieństwo akceptacji (mniej więcej do 1000-1200 iteracji). Ostatnią fazą jest faza, w której przyjmowanie jest zachowanie modelu przełącznikowego - lepsze rozwiązania przyjmują wartość 1, a gorsze 0. Jedynki występują znacznie rzadziej, gdyż cięższej jest znaleźć lepsze rozwiązanie im dłużej działa algorytm, ponieważ jest wtedy mniejsza szansa na polepszenie wyniku.

Wymienione trzy fazy działania algorytmu odzwierciedlają również wartości funkcji celu, uzyskiwane podczas działania algorytmu. Wykres przedstawiający wspomniane wartości jest umieszczony poniżej:



Rysunek 6.7: Wykres wartości funkcji celu na przestrzeni kolejnych iteracji algorytmu.

Podobnie jak w przypadku funkcji aktywacji, widoczną są trzy fazy działania algorytmu. Na poczatku wartości funkcji celu przyjmują wartości rozrzucone od minimum do maksimum funkcji, gdyż akceptowane były prawie wszystkie napotkane rozwiązania. Następnie częstotliwość i natężenie skoków wartości funkcji zmalała i nie były one już tak drastyczne. W końcowej fazie wartości funkcji celu ustabilizowały się i powoli malały.

Algorytm *simulated annealing* jest bardzo czuły na definicję otoczenia. Poprawność jego działania zależy w znacznym stopniu od poprawnego dobrania parametrów startowych. Ponadto, cierpi on na przekleństwo wymiarowości występujące w problemach optymalizacji. Wraz ze wzrostem wymiarów funkcji, na których pracujemy, znacznie maleje prawdopodobieństwo wybrania prawidłowego kierunku przy analizie losowego sąsiedniego rozwiązania. Zmniejsza to skuteczność algorytmu w przypadku wielowymiarowych problemów.

Simulated Annealing to algorytm stochastyczny, gdyż przy każdym uruchomieniu algorytmu tworzona jest inna ścieżka. Jest to podstawowa **metoda metaheurystyczna**.

6.4 Wstęp do metod populacyjnych i algorytmów genetycznych

Algorytm *simulated annealing* jest dobrym wstępem do algorytmów populacyjnych. Mają one za zadanie zwiększyć skuteczność algorytmów, poprzez wykonanie kilku uruchomień algorytmów z różnymi parametrami startowymi. Większa ilość uruchomień w przypadku algorytmów zawierających losowość zwiększa szansę na uzyskanie poprawnego wyniku. Dodatkowo poszczególne uruchomienia algorytmów mogą współdzielić część informacji, dzięki czemu jeszcze bardziej zwiększą swoje szanse na znalezienie ekstremum funkcji.

Metody populacyjne w zagadnieniach dotyczących optymalizacji to podejście, w którym rozwiązania są reprezentowane jako osobniki w populacji. Osobniki te podlegają procesowi ulepszania, często poprzez procesy genetyczne inspirowane ewolucją biologiczną. Podstawowym celem tych algorytmów jest znalezienie optymalnego rozwiązania w przestrzeni poszukiwań poprzez eksplorację i eksploatację potencjalnych obszarów.

Do najpopularniejszych metod populacyjnych należą:

- Algorytmy genetyczne, które opierają się na mechanizmach dziedziczenia genetycznego, mutacji i krzyżowania,
- Strategie ewolucyjne, które koncentrują się na ewolucji rozwiązań poprzez modyfikację ich strategii, a nie konkretnych genotypów,
- Algorytmy roju (Particle Swarm Optimization), które opierają się na modelu roju, gdzie rozwiązania są reprezentowane przez cząstki, a proces optymalizacji polega na dostosowywaniu ruchu cząstek w przestrzeni poszukiwań w oparciu o ich doświadczenia.

Wszystkie te metody mają wspólny rdzeń w inspiracji procesami ewolucyjnymi, a ich skuteczność zależy od odpowiedniego dostosowania parametrów i reprezentacji problemu. Algorytmy populacyjne są często stosowane do rozwiązywania problemów optymalizacyjnych w różnych dziedzinach, takich jak inżynieria, nauki przyrodnicze, finanse czy sztuczna inteligencja.