

# Projektowanie efektywnych algorytmów

*Autor:*

Tymon Tobolski (181037)

Jacek Wieczorek (181043)

*Prowadzący:*

Prof. dr hab. inż Adam Janiak

Wydział Elektroniki

III rok

Cz TN 13.15 - 15.00

16 stycznia 2012

# 1 Cel projektu

Celem projektu jest zaimplementowanie i przetestowanie metaheurystycznego algorytmu genetycznego dla problemu szeregowania zadań na jednym procesorze przy kryterium minimalizacji ważonej sumy opóźnień zadań.

## 2 Opis problemu

**Jednoprocesorowy problem szeregowania zadań przy kryterium minimalizacji ważonej sumy opóźnień zadań.**

Danych jest  $n$  zadań (o numerach od 1 do  $n$ ), które mają być wykonane bez przerwań przez pojedynczy procesor, mogący wykonywać co najwyżej jedno zadanie jednocześnie. Każde zadanie  $j$  jest dostępne do wykonania w chwili zero, do wykonania wymaga  $p_j > 0$  jednostek czasu oraz ma określoną wagę (priorytet)  $w_j > 0$  i oczekiwany termin zakończenia wykonywania  $d_j > 0$ . Zadanie  $j$  jest spóźnione, jeżeli zakończy się wykonywać po swoim terminie  $d_j$ , a miarą tego opóźnienia jest wielkość  $T_j = \max(0, C_j - d_j)$ , gdzie  $C_j$  jest terminem zakończenia wykonywania zadania  $j$ . Problem polega na znalezieniu takiej kolejności wykonywania zadań (permutacji) aby zminimalizować kryterium  $TWT = \sum_{j=1}^n w_j T_j$ .

## 3 Opis algorytmu

Przebieg algorytmu :

```
1  best = S_0 // stan początkowy
   population = generateRandomPopulation(S_0, 2*k) // losowa populacja
                  początkowa

   while n > 0 // n - ilość iteracji
       nextGen = ... TODO

       foreach i in nextGen
           if rand() < M
               nextGen[i] = mutate(nextGen[i])
           end
       end
11  end

       all = population + nextGen
       sort(all)

       population = []
       for i in (0..2k)
           population[i] = all[i]
       end

21  best = population[0]
```

end

gdzie :

- F - funkcja kosztu/celu
- M - prawdopodobieństwo mutacji

## 4 Implementacja

Jezykiem implementacji algorytmu jest *Scala* w wersji 2.9.1 działająca na *JVM*.

```
// generyczna klasa algorytmu genetycznego
abstract class Genetic[A, R : Ordering] extends Function1[A, A] {
  import scala.Ordering.Implicits._

  def K: Int // population size (will be doubled!)
  def F(x: A): R // cost function
  def N: Int // number of iterations
  8 def M: Double // mutation probability
  def crossover(a: A, b: A): (A, A) // crossover function
  def mutation(a: A): A
  def newRandom(a: A): A

  def bestOf(as: List[A]): A = as.minBy(F)
  def mutate(a: A) = if(math.random < M) mutation(a) else a

  def apply(s0: A) = {
    18 def inner(n: Int, population: List[A], best: A): A = {
      val nextGen = population.grouped(2).flatMap {
        case a :: b :: Nil =>
          val (x,y) = crossover(a,b)
          mutate(x) :: mutate(y) :: Nil
        case _ => Nil
      }

      val newPopulation = (population ++ nextGen).sortBy(F).take(2*K)
      val newBest = bestOf(newPopulation)
    28 if(n > 0) inner(n-1, newPopulation, newBest)
      else newBest
    }

    val initial = (1 to (2*K)).map(i => newRandom(s0)).toList

    inner(N, initial, initial.head)
  }
  38 }

// Klasa reprezentujaca zadanie
case class Task(index: Int, p: Int, d: Int, w: Int){
  override def toString = index.toString
}
```

```

    }

    // Klasa reprezentująca uporządkowanie zadan
    case class TaskList(list: Array[Task]){
      lazy val cost = ((0,0) /: list){
        case ((time, cost), task) =>
48         val newTime = time + task.p
          val newCost = cost + math.max(0, (newTime - task.d)) * task.w
          (newTime, newCost)
      }. _2

      override def toString = "%s : %d" format (list.map(_.toString).mkString(
        "[", ", ", "]", " "), cost)
    }

    trait Common {
      def selections[A](list: List[A]): List[(A, List[A])] = list match {
58         case Nil => Nil
          case x :: xs => (x, xs) :: (for((y, ys) <- selections(xs)) yield (y,
            x :: ys))
      }

      implicit def taskListOrdering = new Ordering[TaskList]{
        def compare(x: TaskList, y: TaskList): Int = x.cost compare y.cost
      }

      implicit def arraySwap[T](arr: Array[T]) = new {
68         def swapped(i: Int, j: Int) = {
          val cpy = arr.clone
          val tmp = cpy(i)
          cpy(i) = cpy(j)
          cpy(j) = tmp
          cpy
        }
      }
    }

    }

78 // Implementacja algorytmu genetycznego
    val GA = (n: Int, k: Int) => new Genetic[TaskList, Int] with Common {
      def N = n
      def M = 0.01
      def K = k
      def F(tasks: TaskList) = tasks.cost

      def crossover(a: TaskList, b: TaskList) = pmx(b,a)
      def mutation(tasks: TaskList) = TaskList(randomPermutation(tasks.list))
      def newRandom(tasks: TaskList) = TaskList(randomPermutation(tasks.list))

88      def pmx(ta: TaskList, tb: TaskList): (TaskList, TaskList) = {
        def zeros(n: Int) = new Array[Task](n)

        val (a, b, n) = (ta.list, tb.list, ta.list.length)

        val rand = new Random
        var ti = rand.nextInt(n)
        var tj = rand.nextInt(n)
        while(ti == tj){ tj = rand.nextInt(n) }
98      val (i,j) = if(ti < tj) (ti, tj) else (tj, ti)

        val (af, ar) = a.splitAt(i)
        val (am, ab) = ar.splitAt(j-i)

```

```

108      val (bf, br) = b.splitAt(i)
          val (bm, bb) = br.splitAt(j-i)

          val ax = zeros(i) ++ bm ++ zeros(n - j)
          val bx = zeros(i) ++ am ++ zeros(n - j)

          a.zipWithIndex.foreach { case (e, i) => if(ax(i) == null && !ax.
              contains(e)) ax(i) = e }
          b.zipWithIndex.foreach { case (e, i) => if(bx(i) == null && !bx.
              contains(e)) bx(i) = e }

          ax.zipWithIndex.foreach { case (e, i) => if(e == null) ax(i) = a.
              dropWhile(ax.contains).head }
          bx.zipWithIndex.foreach { case (e, i) => if(e == null) bx(i) = b.
              dropWhile(bx.contains).head }

          (TaskList(ax), TaskList(bx))
    }
118 }

```

## 5 Testy

Test algorytmu genetycznego przeprowadzony został dla trzech zestawów testów o różnej ilości zadań, każdy składający się ze 125 instancji.

Jako wyniki testów przedstawiamy średni czas liczenia wszystkich instancji dla danego rozmiaru problemu -  $\bar{t}$ , a także średni błąd względny rozwiązań dla każdej instancji -  $\bar{x}$ . Według wzoru :

$$\bar{t} = \frac{\sum_{j=1}^m \frac{\sum_{i=1}^z t_i}{z}}{m} \quad (1)$$

$$\bar{x} = \frac{\sum_{j=1}^m \frac{\sum_{i=1}^z x_i}{z}}{m} \quad (2)$$

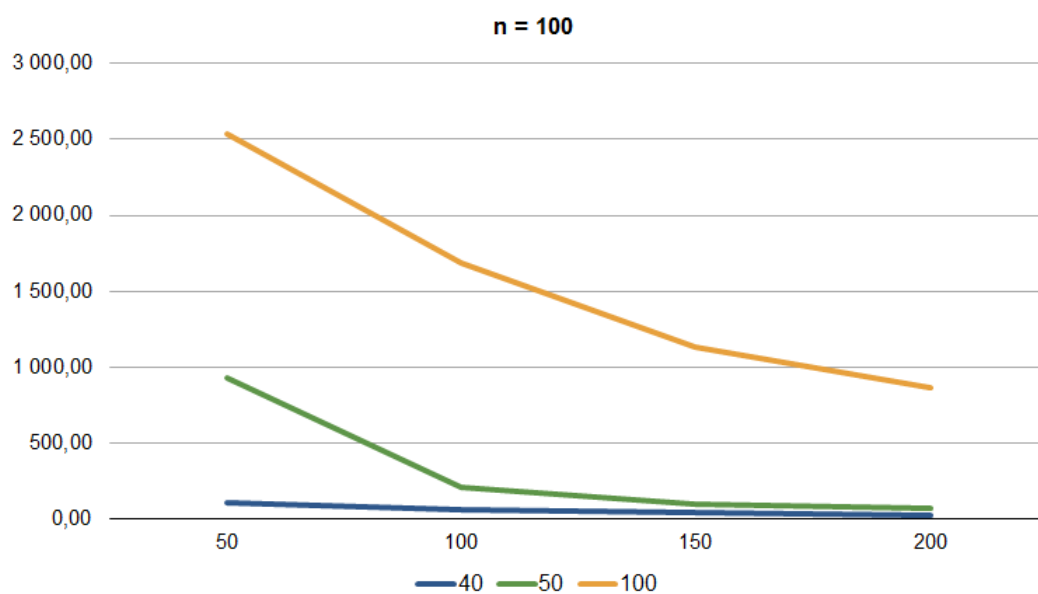
gdzie :

- $z$  - ilość rozwiązań w instancji
- $m$  - ilość instancji danego problemu

## 5.1 Średnia różnica dla zmiennego $k$ i stałego $n = 100$

$k \backslash m$	40	50	100
50	108,04	928,76	2 535,24
100	57,86	212,16	1 682,69
150	38,76	92,89	1 133,71
200	26,19	69,96	859,94

Tabela 1: Diff,  $n = 100$

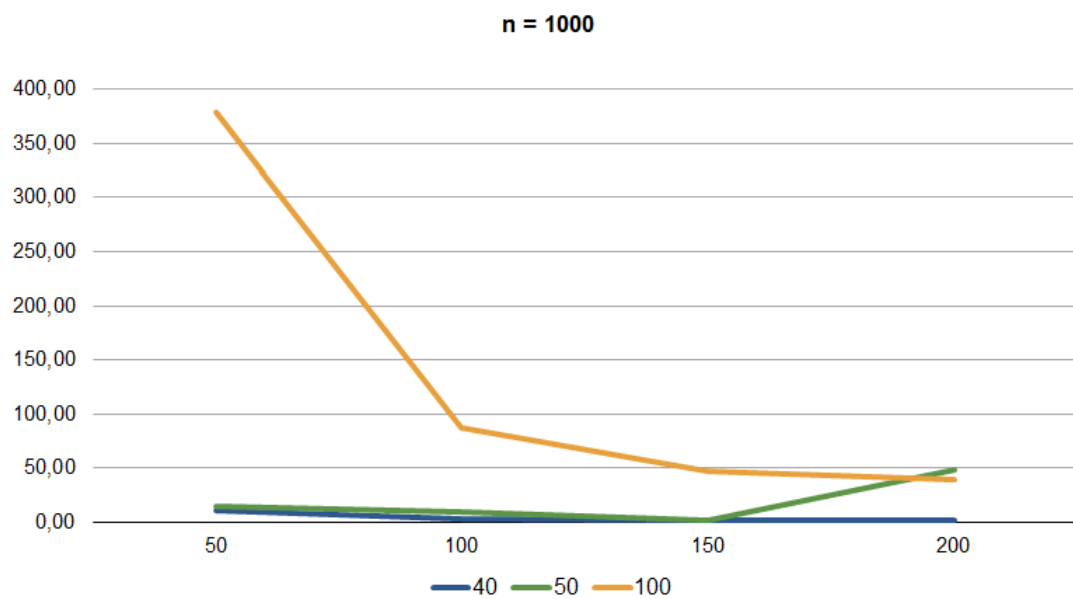


Rysunek 1: Diff,  $n = 100$

## 5.2 Średnia różnica dla zmiennego $k$ i stałego $n = 1000$

$k \backslash m$	40	50	100
50	10,99	14,33	378,84
100	2,59	9,17	87,50
150	2,53	2,14	46,63
200	1,44	48,89	39,68

Tabela 2: Diff,  $n = 1000$

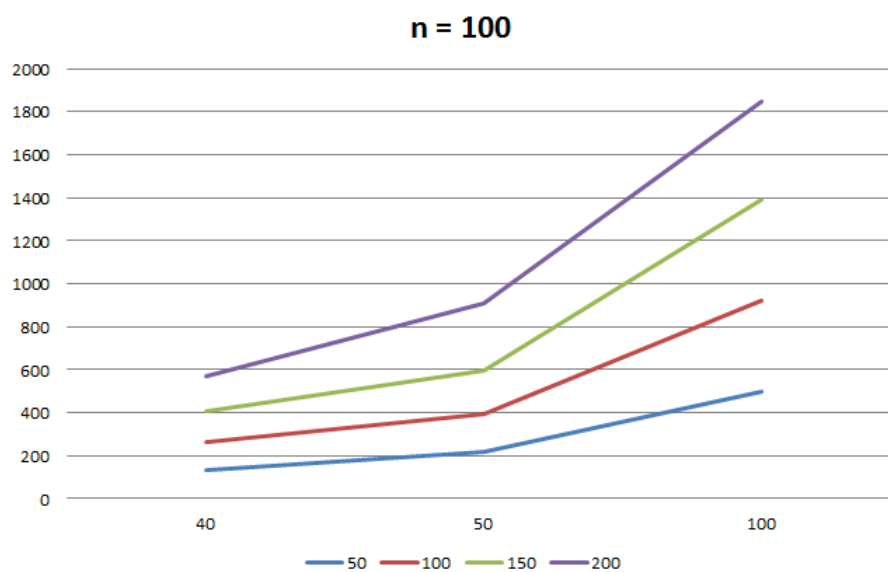


Rysunek 2: Diff,  $n = 1000$

### 5.3 Średnia czas rozwiązywania dla zmiennego $k$ i stałego $n = 100$

$k \backslash m$	40	50	100
50	136,45	219,75	497,38
100	261,98	391,01	918,89
150	409,40	598,76	1 394,21
200	572,62	908,32	1 847,48

Tabela 3: Time,  $n = 100$



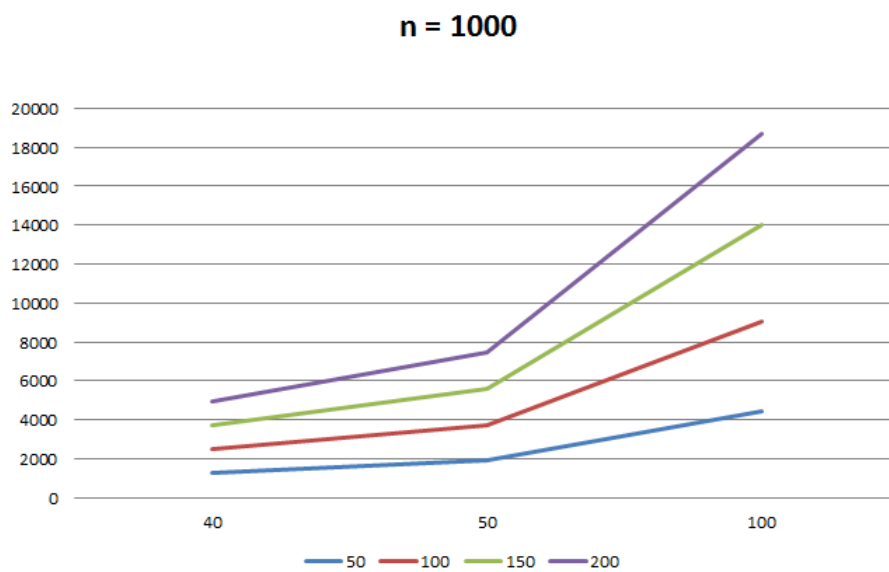
Rysunek 3: Time,  $n = 100$



## 5.4 Średnia czas rozwiązywania dla zmiennego $k$ i stałego $n = 1000$

$k \backslash m$	40	50	100
50	1 285,72	1 904,94	4 428,43
100	2 492,99	3 706,92	9 026,62
150	3 713,95	5 595,20	13 981,76
200	4 954,24	7 452,48	18 717,00

Tabela 4: Time,  $n = 1000$



Rysunek 4: Time,  $n = 1000$

## 6 Wnioski

Po przeprowadzeniu analizy wyników testów, jednoznacznie widać, że rozmiar instancji ma znaczący wpływ na czas działania algorytmu. Znaczną część czasu wykonywania algorytmu zajmuje obliczanie funkcji kosztu, której czas jest zależny od rozmiaru instancji. Głównym parametrem, od którego zależna jest dokładność wyniku, a co za tym idzie czas dochodzenia do rozwiązania jest parametr  $n$ , określający ilość iteracji. Ważnym parametrem jest również  $k$  - wielkość populacji. Odpowiednio dobrane parametry wraz z czynnikiem losowym (mutacja) pozwalają uzyskać zadowalające wyniki.

Algorytm genetyczny pozwala na znalezienie przybliżonego rozwiązania problemu  $sNPh$ . Wiąże się to jednak z koniecznością dobrania odpowiednich parametrów, co nie jest zadaniem łatwym, a także z wybraniem odpowiedniego sposobu krzyżowania populacji. W miarę poprawy wyników poprzez dobierane parametry, wzrasta czas wykonania algorytmu. W celu obliczenia problemu, musimy odpowiedzieć sobie na pytanie, jak dokładne rozwiązanie nas interesuje i ile czasu możemy na nie poświęcić.

## 7 Porównanie

Ostatnim etapem projektu jest porównanie algorytmów symulowanego wyżarzania, tabu search oraz algorytmu genetycznego pod względem dokładności otrzymanych wyników oraz czasu wynonywania.

### 7.1 Porównanie średniego czasu wykonywania algorytmów

Alg	40	50	100
G(k=50, n=100)	136,45	219,75	497,38
G(k=100, n=100)	261,98	391,01	918,89
G(k=150, n=100)	409,40	598,76	1 394,21
G(k=200, n=100)	572,62	908,32	1 847,48
G(k=50, n=1000)	1 285,72	1 904,94	4 428,43
G(k=100, n=1000)	2 492,99	3 706,92	9 026,62
G(k=150, n=1000)	3 713,95	5 595,2	13 981,76
G(k=200, n=1000)	4 954,24	7 452,48	18 717,00
TS(n=10, k=4, t=7)	23,11	46,81	333,83
TS(n=10, k=5, t=7)	25,96	50,96	368,84
TS(n=10, k=6, t=7)	26,23	39,14	327,85
TS(n=10, k=7, t=7)	19,42	38,48	314,94
TS(n=100, k=4, t=7)	188,06	378,68	3 028,87
TS(n=100, k=5, t=7)	208,98	375,12	3 009,74
TS(n=100, k=6, t=7)	206,57	382,79	3 004,64
TS(n=100, k=7, t=7)	185,13	387,76	3 003,10
SA(0,990000)	1,89	2,01	3,09
SA(0,999000)	15,88	18,74	30,36
SA(0,999900)	167,36	189,63	305,09

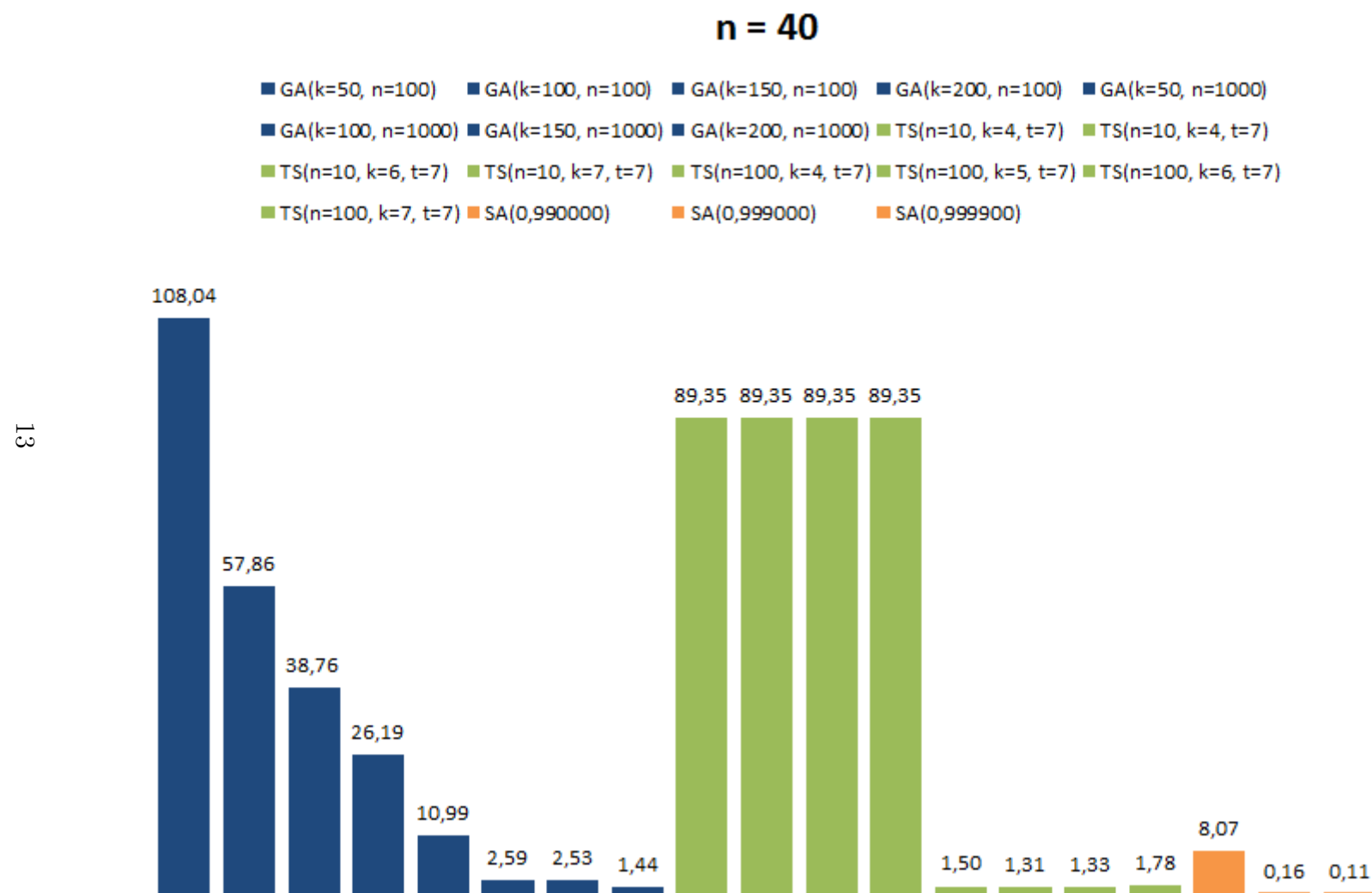
Tabela 5: Średni czas wykonywania algorytmów



## 7.2 Porównanie dokładności otrzymanych wyników

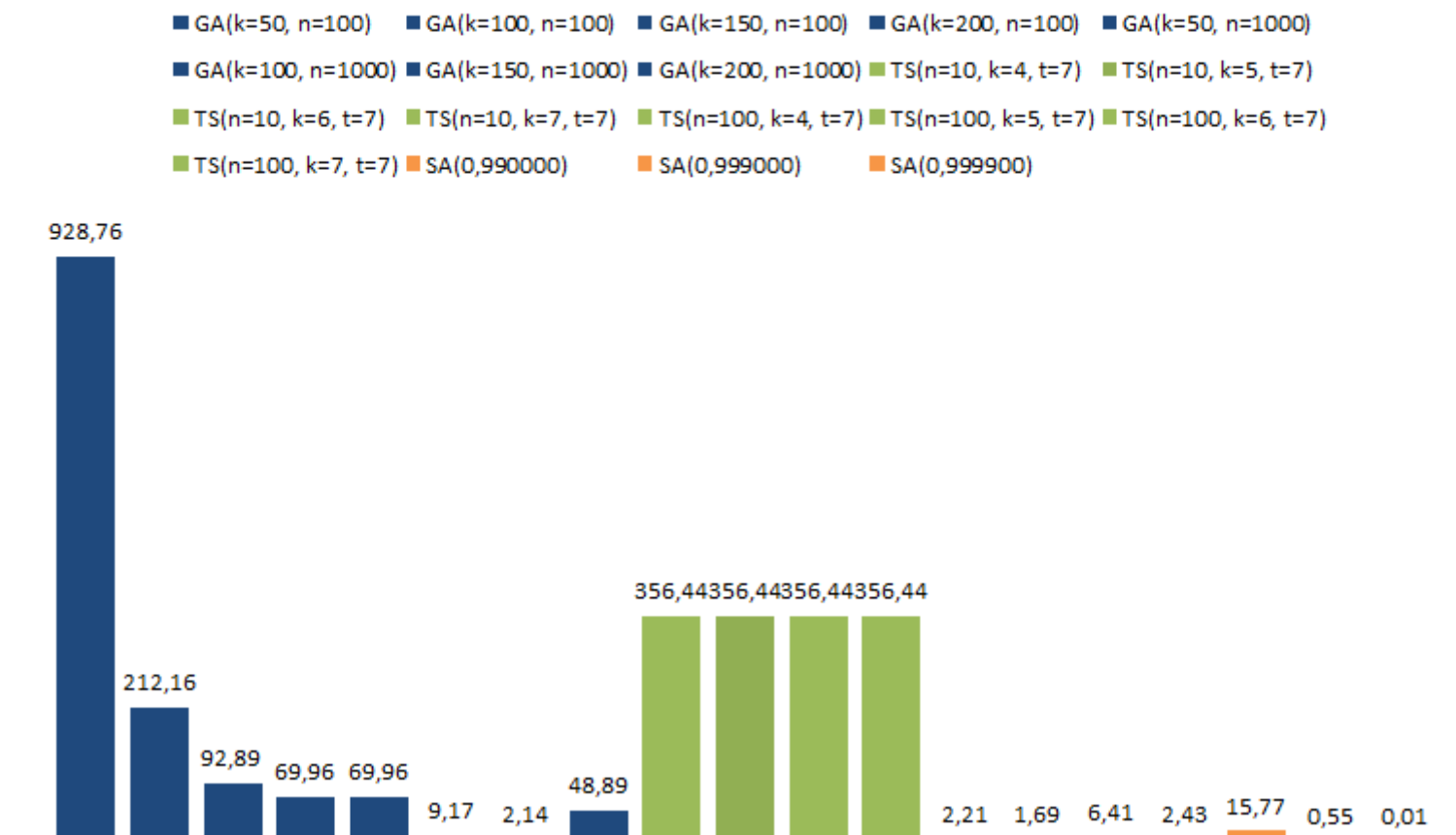
Alg	40	50	100
GA(k=50, n=100)	108,04	928,76	2 535,24
GA(k=100, n=100)	57,86	212,16	1 682,69
GA(k=150, n=100)	38,76	92,89	1 133,71
GA(k=200, n=100)	26,19	69,96	859,94
GA(k=50, n=1000)	10,99	14,33	378,84
GA(k=100, n=1000)	2,59	9,17	87,50
GA(k=150, n=1000)	2,53	2,14	46,63
GA(k=200, n=1000)	1,44	48,89	39,68
TS(n=10, k=4, t=7)	89,35	356,44	1 170,06
TS(n=10, k=5, t=7)	89,35	356,44	1 170,06
TS(n=10, k=6, t=7)	89,35	356,44	1 170,06
TS(n=10, k=7, t=7)	89,35	356,44	1 170,06
TS(n=100, k=4, t=7)	1,50	2,21	28,78
TS(n=100, k=5, t=7)	1,31	1,69	30,62
TS(n=100, k=6, t=7)	1,33	6,41	33,63
TS(n=100, k=7, t=7)	1,78	2,43	33,78
SA(0,990000)	8,07	15,77	337,97
SA(0,999000)	0,16	0,55	9,93
SA(0,999900)	0,11	0,01	0,57

Tabela 6: Dokładność wyników wyrażona w procentach

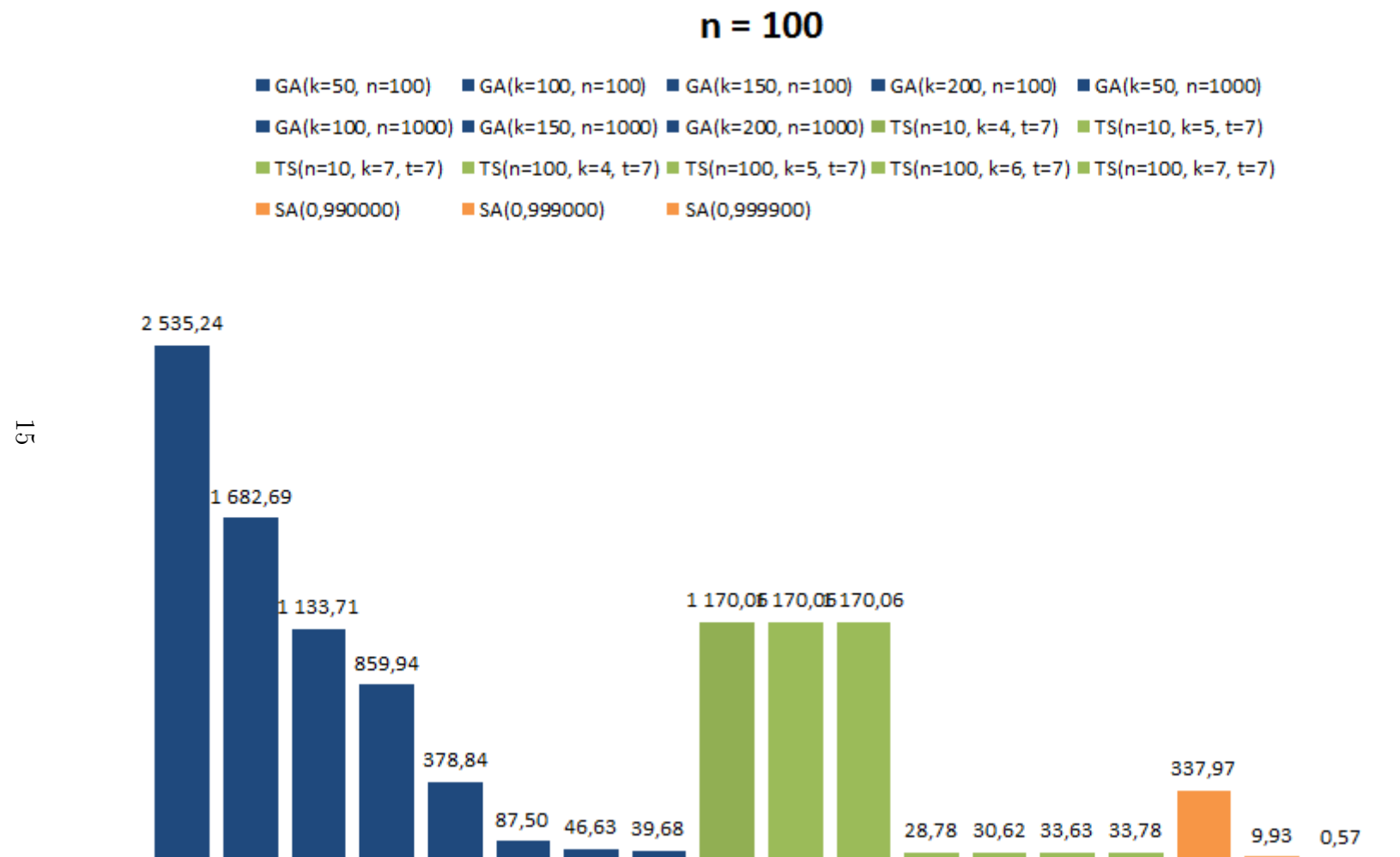


Rysunek 6: 40 zadań

**n = 50**



Rysunek 7: 50 zadań



Rysunek 8: 100 zadań



Po przeprowadzeniu analizy testów zarówno czasu wykonywania algorytmów jak i różnicy od wartości optymalnej najlepszym algorytmem okazują się algorytm symulowanego wyżarzania. Nie tylko uzyskał on wyniki najbardziej zbliżone do optymalnych w najkrótszym czasie, ale jest także najprostszym algorytmem w implementacji spośród omawianych.

Algorytmy tabu search i genetyczne, mimo iż cieszą się większym uznaniem, wiążą się z dobraniem szeregu parametrów, od których zależy jakość rozwiązania. Nie jest to zadanie trywialne i wymaga od programisty przeprowadzenia szeregu testów na optymalność wyniku. Są również podatne na "utknięcie" w lokalnym minimum, dlatego należy zadbać o odpowiednią metodę dywersyfikacji rozwiązań.