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 $TWT = \Sigma_{j=1}^{n} w_j T_j$.

3 Opis algorytmu

Przebieg algorytmu :

```
best = S_0                                          // stan poczatkowy
population = generateRandomPopulation(S_0, 2*k) // losowa populacja
    poczatkowa

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

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

    all = population + nextGen
    sort(all)

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

    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
    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) = {

        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)

            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)
    }
}

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

// Klasa reprezentujaca uporzadkowanie zadan
case class TaskList(list: Array[Task]){
    lazy val cost = ((0,0) /: list){
        case ((time, cost), task) =>
            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 {
        def swapped(i: Int, j: Int) = {
68          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)

        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)
108
        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 }
```

I \ k	50	100	150	200
40	108.04	57.86	38.76	26.19
50	928.76	212.16	92.89	69.96
100	2,535.24	1,682.69	1,133.71	859.94

Tabela 1: Diff, n=100

I \ k	50	100	150	200
40	10.99	2.59	2.53	1.44
50	14.33	9.17	2.14	48.89
100	378.84	87.50	46.63	39.68

Tabela 2: Diff, n=1000

5 Testy

Test algorytmu tabu search przeprowadzony został dla trzech zestawów testów o różnej ilośći 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 wzgledny rozwiązań dla każdej instancji - $\bar{x}$. Według wzoru :

$$\bar{t} = \frac{\Sigma_{j=1}^{m} \frac{\Sigma_{i=1}^{z} t_i}{z}}{m} \tag{1}$$

$$\bar{x} = \frac{\Sigma_{j=1}^{m} \frac{\Sigma_{i=1}^{z} x_i}{z}}{m} \tag{2}$$

gdzie :

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

k \ I	40	50	100
50	136.45	219.75	497.38
100	261.98	391.01	918.89
150	409.4	598.76	1394.21
200	572.62	908.32	1847.48

Tabela 3: Time, n=100

k \ I	40	50	100
50	1285.72	1904.94	4428.43
100	2492.99	3706.92	9026.62
150	3713.95	5595.2	13981.76
200	4954.24	7452.48	18717

Tabela 4: Time, n=1000

6 Wnioski

TODO