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

19 grudnia 2011

1 Cel projektu

Celem projektu jest zaimplementowanie i przetestowanie metaheurystycznego algorytmu tabu search 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

Algorytm *Tabu Search* wykorzystywany jest do otrzymywania wyników optymalnych, lub niewiele różniących się od optymalnych dla problemów optymalizacyjnych. Idę algorytmu jest przeszukiwanie przestrzeni możliwych rozwiązań, stworzonej za pomoca sekwencji ruchów, zawierających ruchy niedozwolone (*tabu*). W celu uniknięcia zakleszczenia w lokalnym minimum, algorytm dokonuje dywersyfikacji poprzez sprawdzenie, czy w ciągu ostatnich k operacji wystąpiło lepsze rozwiązanie niż dotychczasowe minimum. W przeciwnym wypadku losujemy nowe rozwiązanie.

Przebieg algorytmu :

```
old = S_0          // stan poczatkowy
best = old         // najlepsze znalezione rozwiazanie
tabu = []          // pusta lista
k = 0              // zmienna dywersyfikacji

while n > 0 // n - ilosc iteracji
        if k > k_max
                k = 0
                old = SR(old)
        else
                possibleMoves = S(old)
                candidates = []
                foreach move in possibleMoves
                        if not P(move)
                                newPossibleState = NS(old, move)
                                candidates <- newPossibleState
                        end
                end

                new, move = LocateBestCandidate(candidates)
                tabu <- move

                if tabu.length > t_size
                        removeFirst(tabu)
                end

                if F(new) < F(best)
                        best = new
                        k = 0
                else
                        k = k + 1
                end

                old = new
                n = n - 1
        end
end
```

4 Implementacja

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

```
//generyczna klasa algorytmu tabu search
abstract class TabuSearch[A, T, R : Ordering] extends Function1[A, A] {
    import scala.Ordering.Implicits._

    def N: Int
    def Tsize: Int

    def F(x: A): R  // cost function
    def S(x: A): TraversableOnce[T]  // new moves generator
    def NS(x: A, t: T): A // new state generator
    def SR(x: A): A // new random state generator

```

```
    val tabu = new scala.collection.mutable.Queue[T]

    def P(t: T): Boolean

    def apply(s0: A) = {
        tabu.clear()
        def inner(bestState: A, oldState: A, n: Int, k: Int): A = {
            if(n <= 0) {
                bestState
            } else if(k >= 5) {
                inner(bestState, SR(oldState), n, 0)
            } else {
                val newStates = S(oldState).toList.filterNot { m => P(m) }
                    map { m => (NS(oldState, m), m) }
                val (newState, newTabu) = newStates.minBy { e => F(e._1) }

                tabu enqueue newTabu
                if(tabu.length > Tsize) tabu.dequeue

                if(F(newState) < F(bestState)) inner(newState, newState, n
                    -1, 0)
                else inner(bestState, newState, n-1, k+1)
            }
        }

        inner(s0, s0, N, 0)
    }

    override def toString = "TS(%d)" format N
}

// 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 {
        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) = {
            val cpy = arr.clone
            val tmp = cpy(i)
            cpy(i) = cpy(j)
            cpy(j) = tmp
            cpy
        }
    }

}

// Implementacja algorytmu Tabu Search
val TS = (n: Int) => new TabuSearch[TaskList, (Int, Int), Int] with Common {
    def N = n
    def Tsize = 7
    def F(tasks: TaskList) = tasks.cost

    def S(tasks: TaskList) = (0 until tasks.list.length).combinations(2).map
         { idx => (idx(0), idx(1)) }

    def NS(tasks: TaskList, move: (Int, Int)) = TaskList(tasks.list.swapped(
        move._1, move._2))

    def SR(tasks: TaskList) = TaskList(randomPermutation(tasks.list))

    def P(move: (Int, Int)) = {
        tabu.exists { case (a,b) => a == move._1 || b == move._1 || a ==
            move._2 || b == move._2 }
    }
}
```

5 Testy

Testy algorytmu symulowanego wyżarzania przeprowadzone zostały dla trzech zestawów testów o różnym rozmiarze porblemu n, każdy składający się ze 125 instancji. Parametry podstawowe jak T_{min} i T_{max} w przypadku każdego testu były takie same. Zmieniany natomiast był parametr T_d.

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 :

$$\bar{t} = \frac{\Sigma_{j=1}^{n} \frac{\Sigma_{i=1}^{k} t_i}{k}}{n} \tag{1}$$

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

gdzie :

- k - ilość rozwiązań w instancji
- n - ilość instancji danego problemu

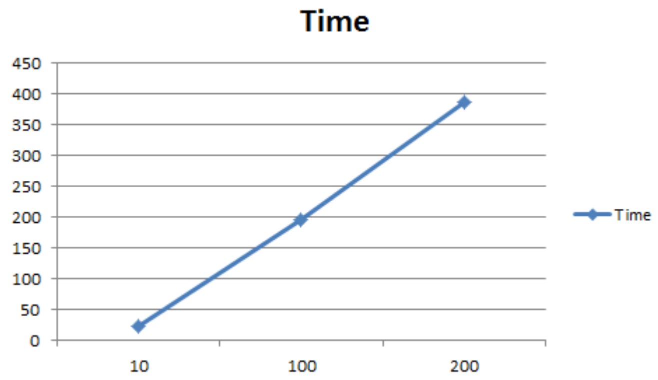
Parametry niezmienne :

- $T_{min} = 0.01$
- $T_{max} = 100$

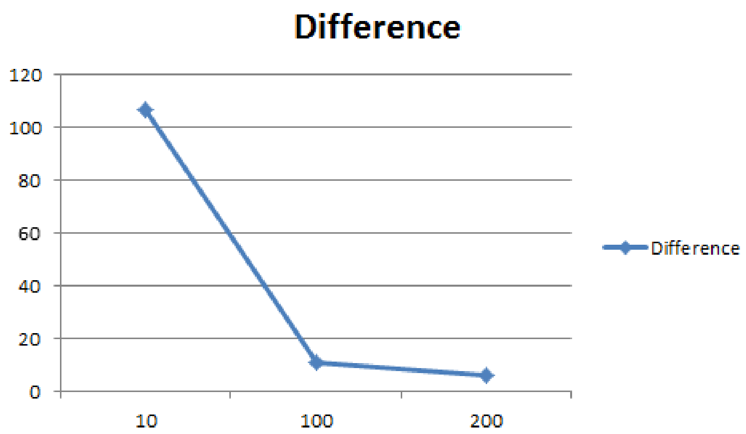
5.1 Pełny przebieg dla parametrów $k = 5$ i $t_{size} = 7$

5.1.1 n = 40

T_d	Time	Difference
10	24,01	106,56
100	195,63	10,97
200	386,48	5,88



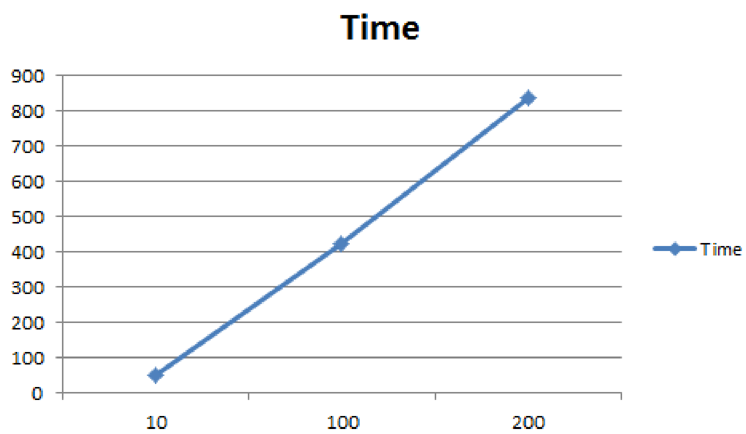
Rysunek 1: Czas rozwiązywania w zależności od parametru T_d



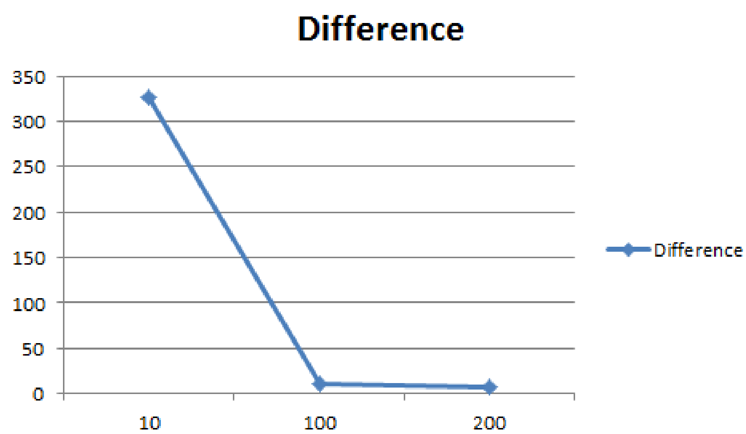
Rysunek 2: Błąd względny rozwiązywania w zależności od parametru T_d

5.1.2 n = 50

T_d	Time	Difference
10	51,38	326,49
100	422,22	11,11
200	836,04	7,26



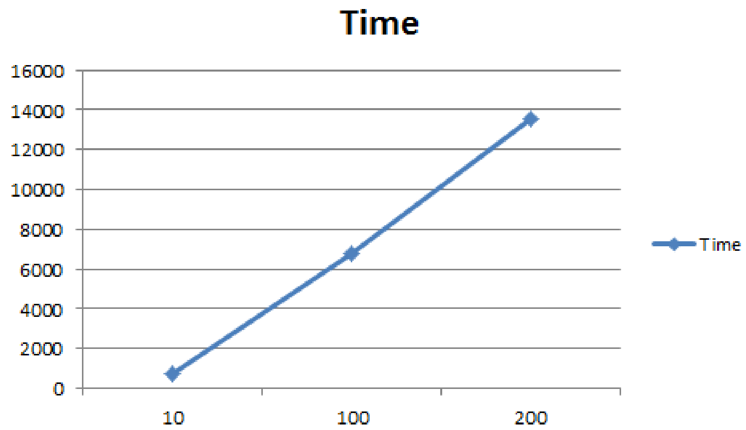
Rysunek 3: Czas rozwiązywania w zależności od parametru T_d



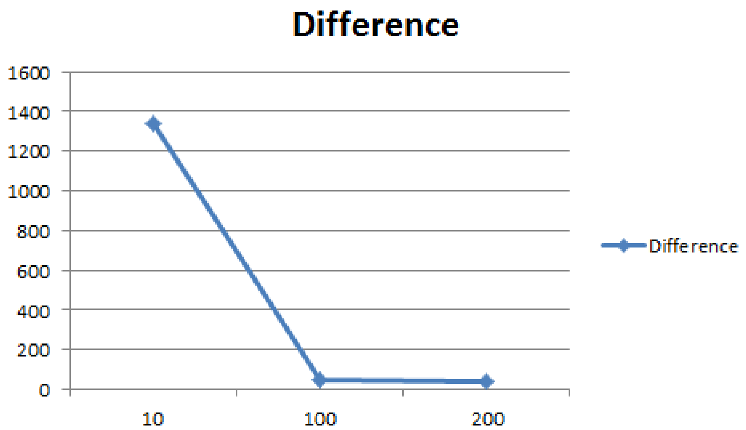
Rysunek 4: Błąd względny rozwiązywania w zależności od parametru T_d

5.1.3 n = 100

T_d	Time	Difference
10	709,64	1338,36
100	6777,35	51,81
200	13555,07	37,12

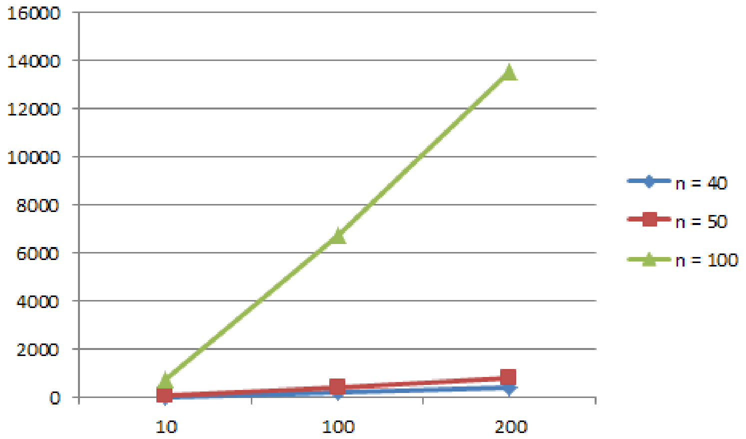


Rysunek 5: Czas rozwiązywania w zależności od parametru T_d

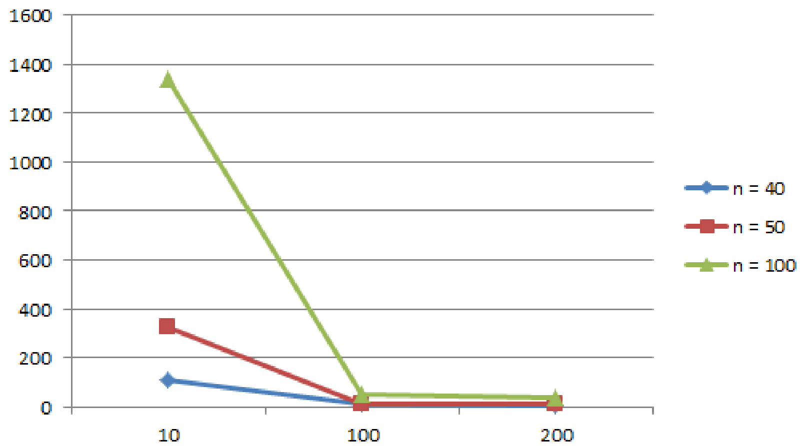


Rysunek 6: Błąd względny rozwiązywania w zależności od parametru T_d

5.1.4 Porównanie wszytskich zadań



Rysunek 7: Czas rozwiązywania w zależności od parametru T_d



Rysunek 8: Błąd względny rozwiązywania w zależności od parametru T_d

6 Wnioski

TODO