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 swap (zamiana miejscami dwóch elementów), 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
```

gdzie :

- S - funkcja generująca listę możliwych ruchów
- F - funkcja kosztu/celu
- NS - funkcja generująca nowy stan
- SR - funkcja generująca nowy losowy stan

4 Implementacja

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

```scala
//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 Kmax: 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 >= Kmax) {
                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
}

trait Common {
    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, k: Int, t: Int) => new TabuSearch[TaskList, (Int, Int),
    Int] with Common {
    def N = n
    def Kmax = k
    def Tsize = t
    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

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 :

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

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

gdzie :

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

Ze względu na złożoność obliczeniową algorytmu testy zostały podzielone na dwa etapy. Pierwsza część testów obejmowała ustalenie najlepszych parametrów k i t_{size} dla niewielkiej liczby instancji problemu.

5.1 Testowanie parametrów k i t_{size}

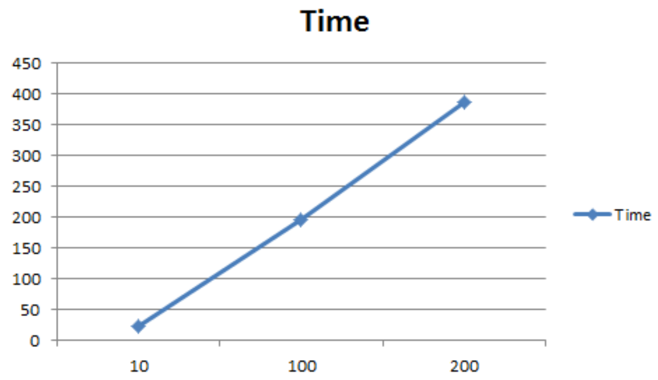
n	k	t	time	dif
10	4	7	32.40	38.84
10	**5**	**7**	**22.68**	**38.84**
10	6	7	23.21	38.84
10	7	7	23.02	38.84
10	4	8	22.29	46.47
10	5	8	22.56	46.47
10	6	8	22.07	46.47
10	7	8	22.21	46.47
10	4	9	22.15	47.23
10	5	9	22.00	47.23
10	6	9	21.83	47.23
10	7	9	21.88	47.23
10	4	10	21.79	47.23
10	5	10	21.53	47.23
10	6	10	21.68	47.23
10	7	10	21.48	47.23
100	4	7	197.91	5.48
100	**5**	**7**	**197.09**	**3.37**
100	6	7	197.72	6.17
100	7	7	197.88	6.52
100	4	8	192.55	4.50
100	5	8	191.70	4.06
100	6	8	191.43	4.43
100	7	8	190.61	6.84
100	4	9	186.19	5.27
100	5	9	186.65	5.67
100	6	9	186.17	3.93
100	7	9	185.71	7.22
100	4	10	181.78	4.20
100	5	10	181.64	4.65
100	6	10	181.43	4.97
100	7	10	181.47	6.14

Z otrzymanych wyników wywnioskowaliśmy, iż najlepszą parą parametrów k i t_{size} są wartości 5 i 7.

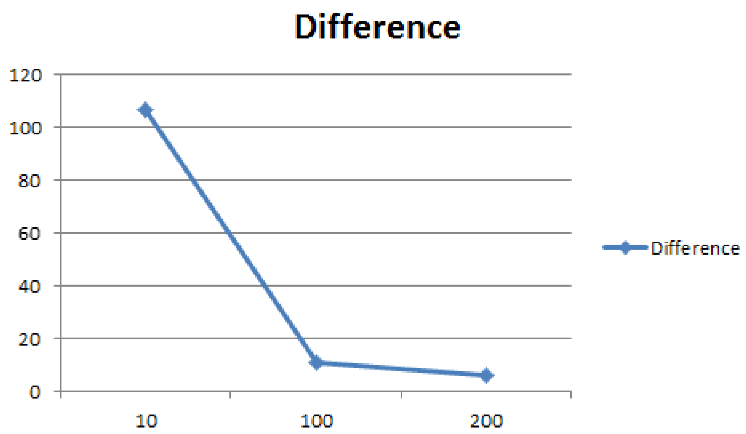
5.2 Pełny przebieg dla parametrów $k = 5$ i $t_{size} = 7$

5.2.1 n = 40

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



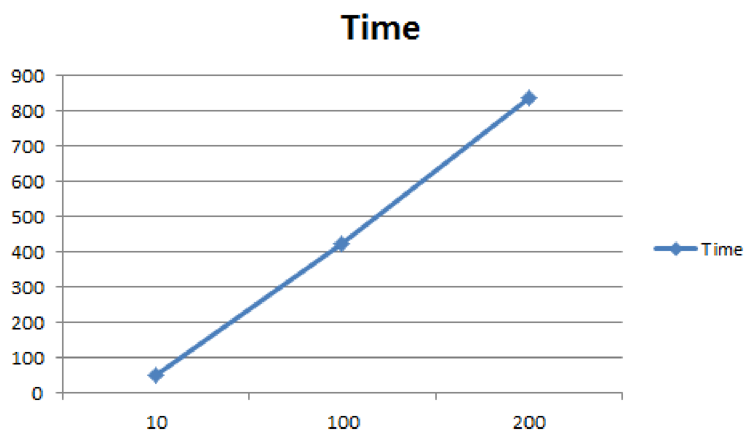
Rysunek 1: Czas rozwiązywania w zależności od parametru T_d



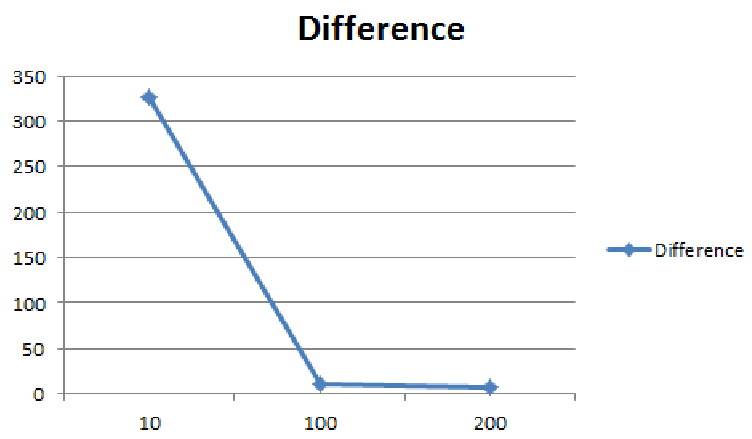
Rysunek 2: Błąd względny rozwiązywania w zależności od parametru T_d

5.2.2 n = 50

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



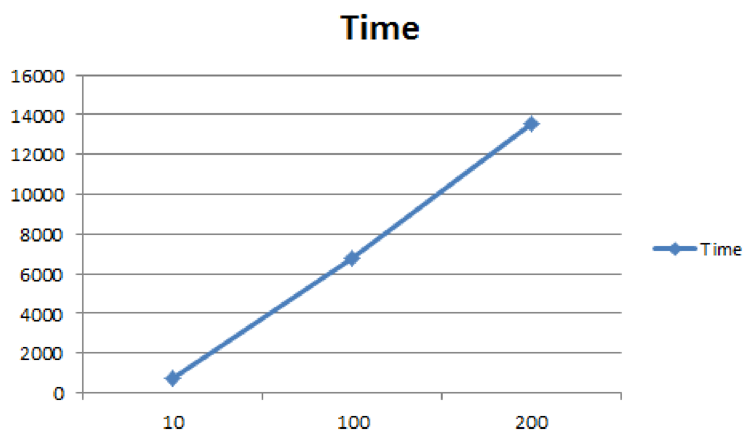
Rysunek 3: Czas rozwiązywania w zależności od parametru T_d



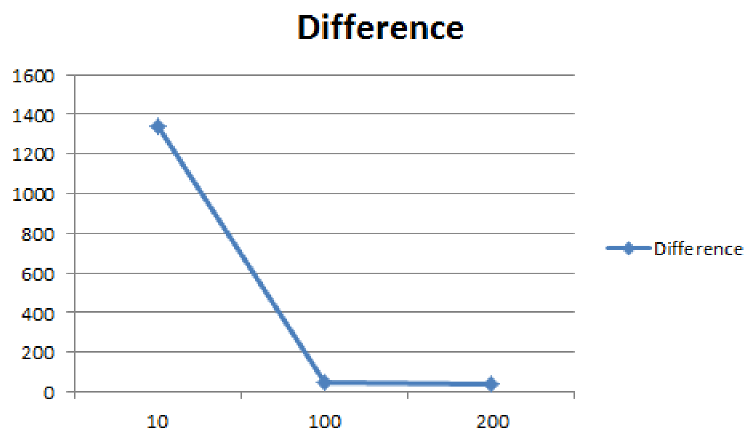
Rysunek 4: Błąd względny rozwiązywania w zależności od parametru T_d

5.2.3 n = 100

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

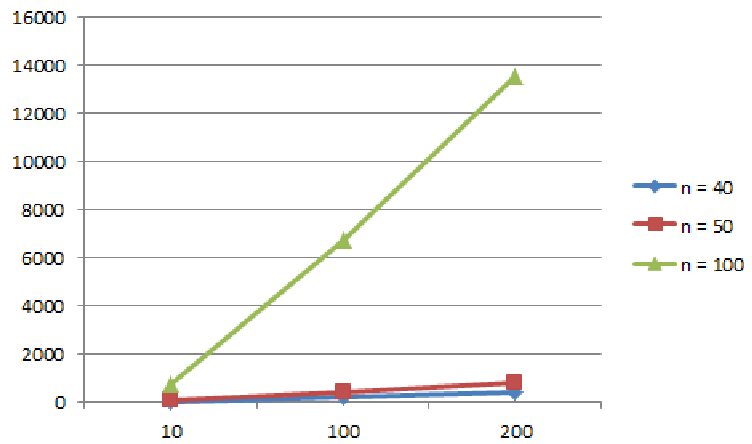


Rysunek 5: Czas rozwiązywania w zależności od parametru T_d

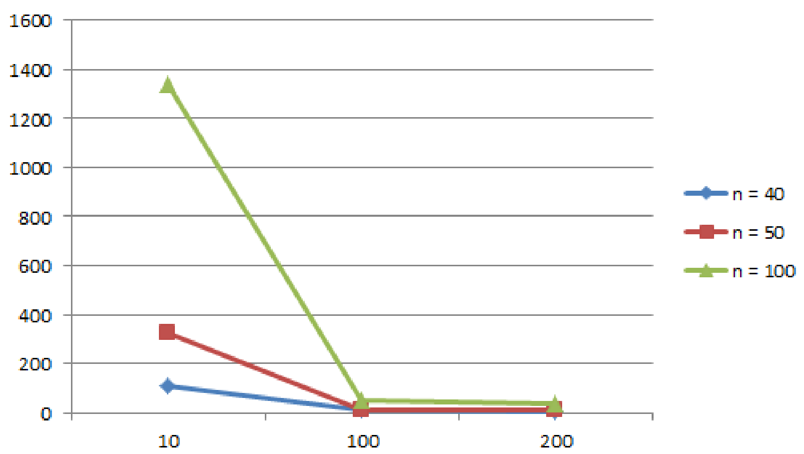


Rysunek 6: Błąd względny rozwiązywania w zależności od parametru T_d

5.2.4 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

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. Ważnymi parametrami są również k - zmienna dywersyfikacji oraz t_{size} - rozmiar listy tabu. Odpowiednio dobrane parametry wraz z czynnikiem losowym minimalizują podatność algorytmu na zapętlenie w lokalnym minimum.

Algorytm tabu search pozwala na znalezienie przybliżonego rozwiązania problemu $sNPh$. Wiąże się to jednak z koniecznością dobrania odpowiednich parametrów, co nie jest zadaniem łatwym. 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.