Politechnika Wrocławska

Wydział Elektroniki, Fotoniki i Mikrosystemów

Sterowanie Procesami Dyskretnymi

Oliwier Woźniak, Dzmitry Mandrukevich

kierunek studiów: Automatyka i robotyka

specjalność: Robotyka

Algorytm programowania dynamicznego dla problemu $\Sigma(w_i T_i)$

Prowadzący: dr inż. Radosław Grymin

Wrocław 20 marca 2024

Spis treści

1 Opis problemu

Celem zadania jest znalezienie algorytmu, znajdującego optymalną permutację zbioru zadań, dla których kara za opóźnienie jest minimalna. Każde zadanie charakteryzują trzy liczby: czas wykonywania, waga, termin wykonania. Za każdy cykl opóźnienia w wykonaniu zadania kara rośnie liniowo z mnożnikiem wagi.

2 Algorytmy PD

Programowanie dynamiczne polega na dzieleniu zadań na mniejsze zadania, korzystając z reguły „dziel i rządź". Dzięki temu skomplikowane zadania można podzielić na zbiór mniejszych, prostszych zadań. W wyniku takiej operacji może zmaleć złożoność obliczeniowa całego problemu, albo zadanie w ogóle może zostać rozwiązane (jeśli wcześniej nie było).

2.1 Użyty algorytm

Pierwszym problemem jaki musieliśmy rozwiązać przy poszukiwaniu rozwiązania optymalnego było napisanie funkcji oceniającej karę dla danej permutacji. Iterujemy zadania zgodnie z ich ułożeniem w vectorze, następnie sprawdzamy czy zostały wykonane w terminie. W przypadku jeśli zadanie przekroczyło określiny termin, to obliczamy karę i dodajemy ją do licznika, w przeciwnym przypadku nic nie robimy. Funkcja kończy swoje zadanie, gdy cała tablica zostanie sprawdzona.

```
int sumaWiti (int n, vector<Dane> dane){
    int czas=0, suma_kar=0;
    for (int i = 0; i < n; ++i){
        czas+=dane[i].czas_zadania;
        if (dane[i].termin<czas){
            suma_kar+=(czas - dane[i].termin)*dane[i].kara;
        }
    }
    return suma_kar;
}
```

Rysunek 1: Obliczanie kary dla danej permutacji

W użytym przez nas algorytmie zaczynamy od posortowania zadań według założonego terminu wykonania. W wielu przypadkach pozwala to na skrócenie analizy w późniejszych częściach algorytmu. Użyliśmy sortowania bąbelkowego, ponieważ jest ono najprostsze, a następny algorytm nie pozwala na użycie dużych zbiorów danych, ze względu na dużą złożoność obliczeniową.

```
void sortTermin (int n, vector <Dane> &dane){
    for (int i=0; i<n; i++){
        for (int j=1; j<n-i; j++){
            if (dane[j-1].termin > dane[j].termin){
                swap(dane[j-1], dane[j]);
            }
        }
    }
}
```

Rysunek 2: Sortowanie po terminie wykonania

Następnie posortowany vector zostaje przekazany do kolejnego algorytmu sortującego, tym razem poszukujemy optymalnego rozwiązania. Algorytm najpierw wybiera element który ma być zamieniany (wybierane są od 0 do n-1), a następnie wymieniany jest on z poprzednimi elementami. W momencie w którym zostanie znalezione lepsze rozwiązanie, algorytm zaczyna iterować tablice od początku. Jeśli rozwiązanie nie jest lepsze, to elementy zamieniają się z powrotem i badane są kolejne elementy. Zdecydowanie lepiej widać działanie patrząc na napisany kod.

```
void sorting (int n, vector <Dane> &dane)
{
    int timemin = sumaWiti(n,dane), timeW;
    for (int i=0; i<n; i++){
        for (int j=i+1; j<n; j++){
            swap(dane[i], dane[j]);
            timeW = sumaWiti(n,dane);
            if (timeW < timemin){
                timemin = timeW;
                i=0; j=0;
            } else {
                swap(dane[i], dane[j]);
            }
        }
    }
}
```

Rysunek 3: Szukanie optymalnego rozwiązania

2.2 Algorytm PD dla problemu w_iT_i

W przypadku problemu w_iT_i istnieją różne algorytmy poprawnego rozwiązania, rozważany będzie dla porównania algorytm opisany w [1] przez dr inż. Andrzeja Gnatowskiego.

Opisany przez niego algorytm wygląda następująco:

Algorytm 1 Pseudokod dla algorytmu programowania dynamicznego.

procedure PD(n, P, W, T)
  $\mathcal{I} \leftarrow \mathcal{N}$
  for $i = 1$ **to** n **do**
    $v(\mathcal{I}^i) \leftarrow \arg\min_{j \in \mathcal{I}^i} \left\{ F\left(\mathcal{I}^{L(\mathcal{I}^i \setminus \{j\})}\right) + f_j(p(\mathcal{I}^i)) \right\}$
    $F(\mathcal{I}^i) \leftarrow F\left(\mathcal{I}^{L(\mathcal{I}^i \setminus \{v(\mathcal{I}^i)\})}\right) + f_{v(\mathcal{I}^i)}(p(\mathcal{I}^i))$
  end for
  for $i = n$ **to** 1 **do**
    $\pi(i) \leftarrow v(\mathcal{I})$
    $\mathcal{I} \leftarrow \mathcal{I} \setminus \{v(\mathcal{I})\}$
  end for
end procedure

Rysunek 4: Pseudokod dla algorytmu PD

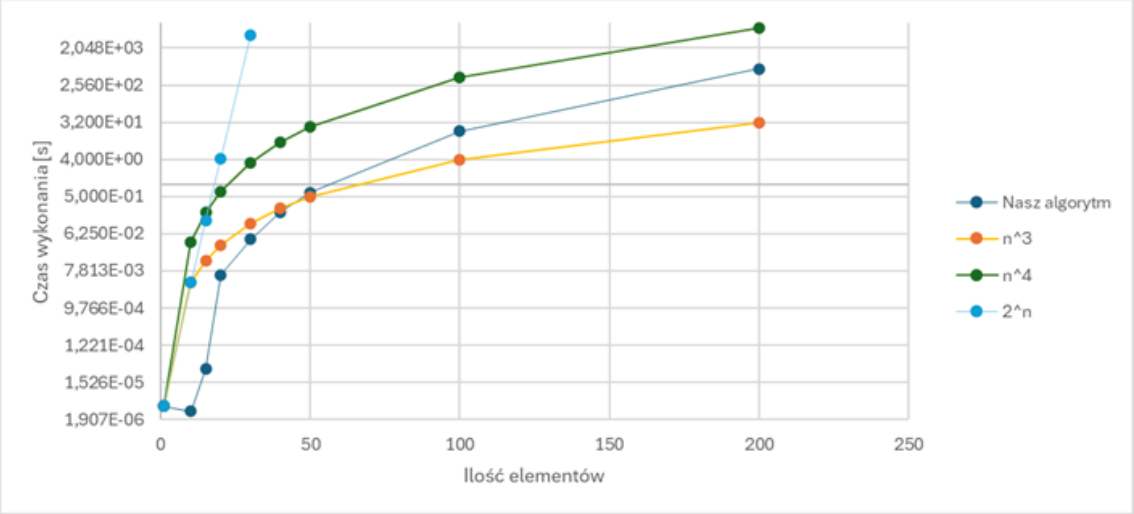
Z racji że wszystkie parametry, oraz działanie tego kodu są szeroko opisane w internecie, nie będziemy się w to bardziej zagłębiać i przejdziemy od razu do porównania działania obu rozwiązań.

3 Analiza użytego rozwiązania

Nasze rozwiązanie jest amatorskie i możliwe że nie znajduje optymalnego rozwiązania dla każdego problemu. Jednak dla podanych przez Dr inż. Mariusza Makuchowskiego przypadków, udaje się każdorazowo znaleźć podane optymalne rozwiązanie (przynajmniej o podanej karze).

3.1 Analiza czasowa

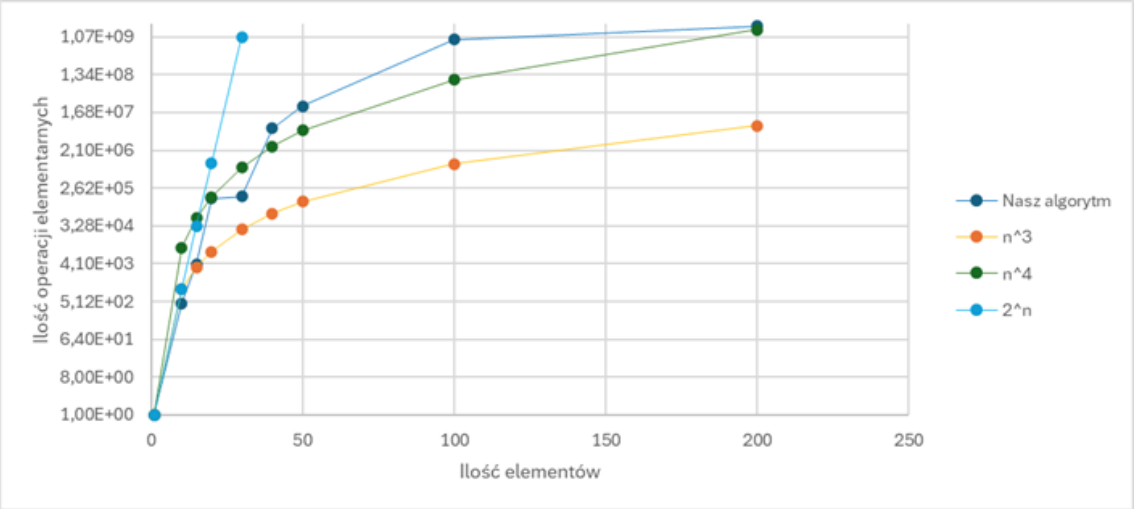
Po wykonaniu wielu testów jednostkowych, można jednoznacznie stwierdzić, że nasze rozwiązanie ma złożoność obliczeniową przynajmniej rzędu $O(n^3)$, jednak ciężko jest jednoznacznie stwierdzić na podstawie wykonanych testów. Można jednak spostrzec, że wzrost jest znacznie wolniejszy niż dla złożoności rzędu $O(2^n)$



Rysunek 5: Porównanie czasów wykonania dla różnych złożoności obliczeniowych

3.2 Analiza iteracyjna

Podobnej analizy dokonaliśmy na podstawie wykonywania elementarnych operacji (w naszym przypadku jest to liczenie sumy kar). Ta analiza dała podobny wynik do analizy czasu wykonania.



Rysunek 6: Porównanie ilości operacji elementarnych dla różnych złożoności obliczeniowych

3.3 Analiza złożoności obliczeniowej

Analiza złożoności obliczeniowej na podstawie kodu jest ciężka do dokonania ze względu na nieliniowy charakter algorytmu. Na podstawie intuicji określiliśmy, że złożoność obliczeniowa jest rzędu $O(n^k)$, gdzie $k \geq 3$ (określono to na podstawie ilości pętli „for"). Wynik ten pokrywa się jednak z wynikami poprzednich analiz.

3.4 Analiza złożoności pamięciowej

W użytym algorytmie nie ma żadnych struktur pamiętających ułożenie elementów, poza vectorem przechowującym wpisane zadania. Zatem złożoność pamięciowa wynosi dokładnie O(n).

3.5 Porównanie algorytmów

Na podstawie przeprowadzonej analizy można jednoznacznie stwierdzić, że użyty przez nas algorytm jest jednoznacznie lepszy (do rozwiązania przedstawionego problemu). Ma niższą złożoność obliczeniową, rzędu $O(n^k)$ w porównaniu z algorytmem przedstawionym w [1], gdzie złożoność obliczeniowa była rzędu $O(n2^n)$. Jeśli zaś chodzi o złożoność pamięciową, to użyty przez nas algorytm nie przechowuje żadnych nadmiarowych informacji związanych z analizowanymi zadaniami. Algorytm

przedstawiony w cytowanym tekście ma złożoność pamięciową $O(2^n)$, czyli wymaga dużego nakładu pamięciowego do działania.

Literatura

[1] dr inż. Andrzej Gnatowski. Lab. 04: Problem witi.