Politechnika Wrocławska

Wydział Elektroniki, Fotoniki i Mikrosystemów

Sterowanie Procesami Dyskretnymi

Oliwier Woźniak, Dzmitry Mandrukevich

kierunek studiów: Automatyka i robotyka

specjalność: Robotyka

Algorytm NEH dla $F^*||C_{max}$

Prowadzący: dr inż. Radosław Grymin

Wrocław 30 marca 2024

Spis treści

1 Opis problemu

Celem zadania jest znalezienie algorytmu, znajdującego optymalną permutację zbioru zadań, dla których suma czasów wykonywania jest minimalna. Zadania wykonywane są na wielu maszynach, każda z maszyn pracuje niezależnie, jednak każde zadanie musi być wykonane na wszystkich maszynach. Jedna maszyna może wykonywać jedynie jedno zadanie jednocześnie. Maszyna po wykonaniu pierwszego zadania może od razu przejść do wykonywania kolejnego. Kolejność wykonywania zadań na maszynach jest zawsze taka sama. Zadanie wykonane na maszynie pierwszej, przechodzi na maszynę drugą, o ile jest ona wolna, w przeciwnym przypadku dołącza do kolejki FIFO zadań przypisanych do maszyny.

1.1 Algorytm NEH

NEH jest naiwnym rozwiązaniem wcześniej opisanego problemu. Spośród wszystkich dostępnych zadań wybierane jest zadanie o największej wadze $w_i = \sum_{k \in M} p_{i,k}$. Następnie wybrane zadanie zostaje umieszczone we wszystkich dostępnych miejscach (na początku, na końcu i pomiędzy każdym z zadań), najlepsze uszeregowanie zostaje zapisane i przechodzimy do następnej iteracji. Uszeregowane zadania powinny dać optymalny wynik. Algorytm ten jest skuteczny, jednak powolny, co zostanie opisane w następnym rozdziale.

```
// Obliczanie czasu wykonywania wszystkich zadań w podanej kolejności
int timeCmax (int n, int m, Dane dane[])
{
    int Cmax[m];
    Cmax[0] = dane[0].stream[0];
    // Obliczanie dla pierwszego zadania
    for (int i=1; i<m; i++){
        Cmax[i] = Cmax[i-1] + dane[0].stream[i];
    }
    // Obliczanie dla wszystkich kolejnych
    for (int i=1; i<n; i++){
        Cmax[0] += dane[i].stream[0];
        for (int j=1; j<m; j++){
            Cmax[j] = max(Cmax[j-1], Cmax[j]) + dane[i].stream[j];
        }
    }
    return Cmax[m-1];
}
```

Rysunek 1: Obliczanie czasu dla danej permutacji

```
void sortQueue (int n, int m, Dane *dane)
{
    Dane obliczenia[n];
    Dane wynik[n];
    wynik[0] = dane[0];
    int mintime, timeC;
    for (int i=1; i<n; i++){
        obliczenia[0] = dane[i];
        przerzuc(1, i, wynik, obliczenia);
        przerzuc(0, i+1, obliczenia, wynik);
        mintime = timeCmax(i+1, m, obliczenia);
        for (int j=0; j<i; j++){
            swap (obliczenia[j], obliczenia[j+1]);
            timeC = timeCmax(i+1, m, obliczenia);
            if (timeC < mintime){
                mintime = timeC;
                przerzuc(0, i+1, obliczenia, wynik);
            }
        }
    }
    przerzuc(0, n, wynik, dane);
}
```

Rysunek 2: Kod naiwnego algorytmu NEH

1.2 Algorytm QNEH

Algorytm QNEH, w odróżnieniu od zwykłego algorytmu NEH, jest oparty o operacje grafowe. Należy zauważyć, że wszystkie możliwe permutacje tworzone przy dokładaniu nowej wartości są w pewnym stopniu podobne, taka obserwacja umożliwia nam podzielenie wcześniejszego zadania na mniejsze (mniej złożone). W związku z tym analizowane są dwie ścieżki podróży zadania: „najdłuższa wchodząca ścieżka" i „najdłuższa dochodząca scieżka". Po analizie grafu pod oboma kątami wybierane jest optymalne miejsce umieszczenia nowego zadania, czyli te miejsce, gdzie czas dochodzenia i wchodzenia jest największy. Algorytm dodatkowo przyśpiesza liczenie czasu wykonania zadań, co dodatkowo zwiększa jego efektywność.

```
    void timeCmaxLeftToRight (int last, int m, Dane dane[], int **Cmax)
{
    Cmax[0][0] = dane[0].stream[0];
    for (int i=1; i<m; i++){
        Cmax[0][i] = Cmax[0][i-1] + dane[0].stream[i];
    }
    for (int i=1; i<last; i++){
        Cmax[i][0] = Cmax[i-1][0] + dane[i].stream[0];
        for (int j=1; j<m; j++){
            Cmax[i][j] = max(Cmax[i][j-1], Cmax[i-1][j]) + dane[i].stream[j];
        }
    }
}
```

Rysunek 3: Funkcja licząca czas dochodzenia

```
    void timeCmaxRightToLeft (int last, int m, Dane dane[], int **Cmax)
{
    --last;
    Cmax[last][m-1] = dane[last].stream[m-1];
    for (int i=m-2; i>-1; i--){
        Cmax[last][i] = Cmax[last][i+1] + dane[last].stream[i];
    }
    for (int i=last-1; i>-1; i--){
        Cmax[i][m-1] = Cmax[i+1][m-1] + dane[i].stream[m-1];
        for (int j=m-2; j>-1; j--){
            Cmax[i][j] = max(Cmax[i][j+1], Cmax[i+1][j]) + dane[i].stream[j];
        }
    }
}
```

Rysunek 4: Funkcja licząca czas wchodzenia

```
int qTimeCmax (int ind, int m, int ile, int dane[], int *Cmltr[], int *Cmrtl[])
{
    int Cmax[m], Cm;
    if (ind == 0){
        Cmax[m-1] = Cmrtl[0][m-1] + dane[m-1];
        for (int i=m-2; i>-1; i--){
            Cmax[i] = max(Cmax[i+1], Cmrtl[0][i]) + dane[i];
        }
        return Cmax[0];
    }
    if (ind == ile){
        Cmax[0] = Cmltr[ind-1][0] + dane[0];
        for (int i=1; i<m; i++){
            Cmax[i] = max(Cmax[i-1], Cmltr[ind-1][i]) + dane[i];
        }
        return Cmax[m-1];
    }

    Cmax[0] = Cmltr[ind-1][0] + dane[0];
    Cm = Cmax[0] + Cmrtl[ind][0];
    for (int i=1; i<m; i++){
        Cmax[i] = max(Cmax[i-1], Cmltr[ind-1][i]) + dane[i];
        if (Cmax[i]+Cmrtl[ind][i] > Cm){
            Cm = Cmax[i] + Cmrtl[ind][i];
        }
    }
    return Cm;
}
```

Rysunek 5: Funkcja licząca czas wykonywania zadań

```
void qsortQueue (int n, int m, Dane *dane, int **CmLeftToRight, int **CmRightToLeft)
{
    Dane obliczenia[n];
    Dane wynik[n];
    wynik[0] = dane[0];
    int mintime, timeC, ind=0;
    for (int i=1; i<n; i++){
        timeCmaxLeftToRight(i, m, wynik, CmLeftToRight);
        timeCmaxRightToLeft(i, m, wynik, CmRightToLeft);
        ind=0;
        obliczenia[0] = dane[i];
        przerzuc(1, i, wynik, obliczenia);
        przerzuc(0, i+1, obliczenia, wynik);
        mintime = qTimeCmax (ind, m, i, dane[i].stream, CmLeftToRight, CmRightToLeft)
        for (int j=0; j<i; j++){
            swap (obliczenia[j], obliczenia[j+1]);
            timeC = qTimeCmax (j+1, m, i, dane[i].stream, CmLeftToRight, CmRightToLeft
            if (timeC < mintime){
                mintime = timeC;
                ind = j + 1;
                przerzuc(0, i+1, obliczenia, wynik);
            }
        }
    }
    przerzuc(0, n, wynik, dane);
}
```

Rysunek 6: Implementacja algorytmu sortowania QNEH

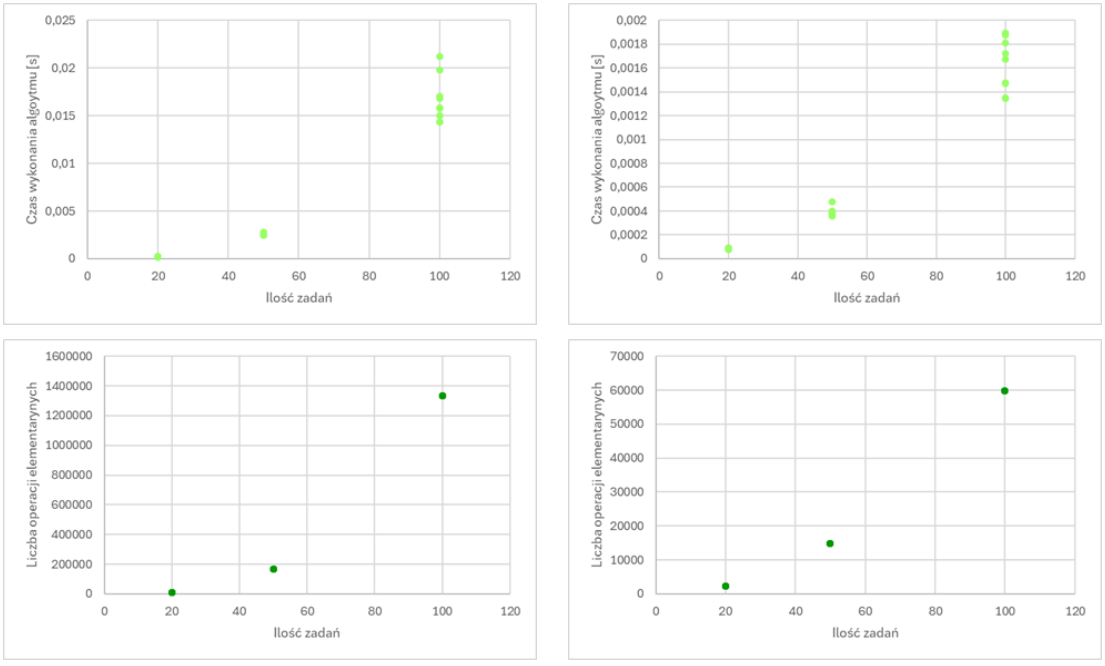
2 Porównanie algorytmu NEH i QNEH

Już po samym przedstawieniu sposobu działania obu algorytmów można stwierdzić że QNEH będzie miał zdecydowanie szybsze działanie, jednak należy jeszcze sprawdzić jak bardzo. Oba algorytmy zostały poddane testom ilościowym, aby porównać ich złożoności obliczeniowe. Wszystkie wykonane próbki dawały poprawny wynik (zgodny z tym podanym przez dr. Mariusza Makuchowskiego).

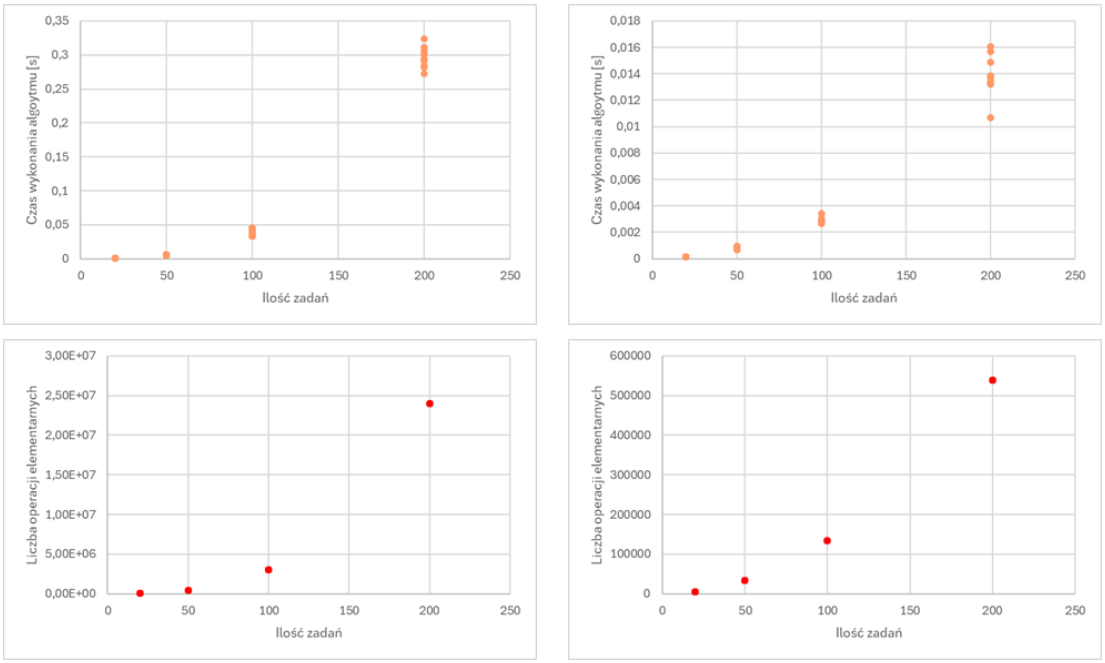
2.1 Analiza ze względu na ilość zadań

Pierwszym badanym parametrem będzie wpływ ilości zadań na długość wykonywania/ilość operacji elementarnych. W tym celu zostały wybrane próbki, gdzie ilość maszyn jest stała.

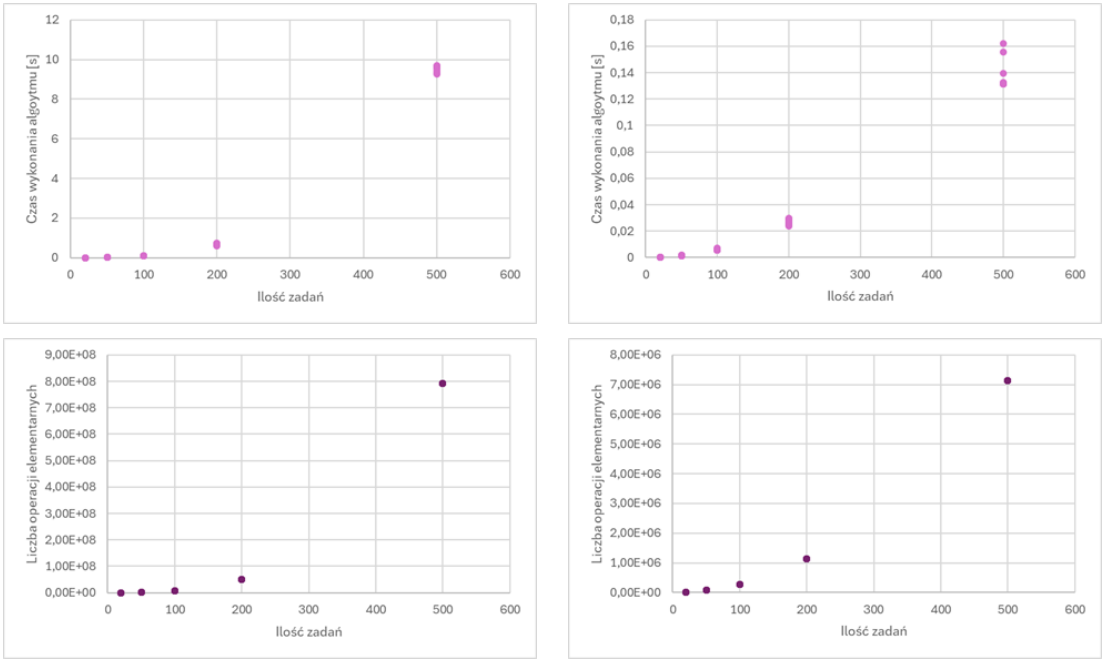
Wszystkie prezentowane zestawienia mają wykresy dla NEH po lewej, natomiast dla QNEH po prawej.



Rysunek 7: Porównanie zmiany czasu wykonywania i ilości operacji elementarnych dla 5 maszyn



Rysunek 8: Porównanie zmiany czasu wykonywania i ilości operacji elementarnych dla 10 maszyn

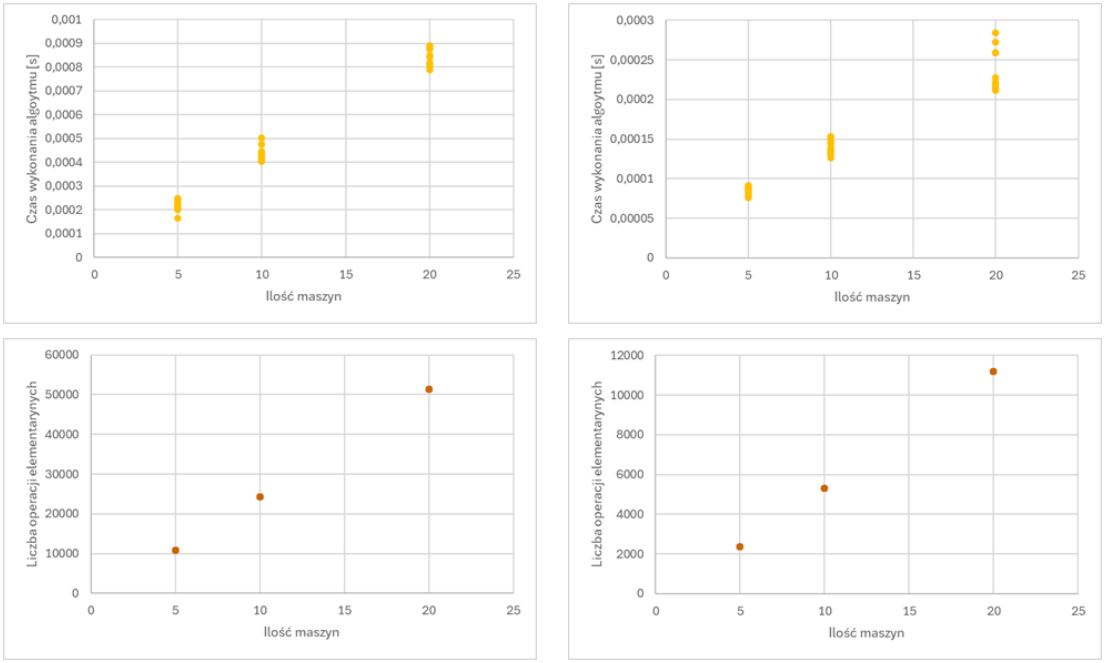


Rysunek 9: Porównanie zmiany czasu wykonywania i ilości operacji elementarnych dla 20 maszyn

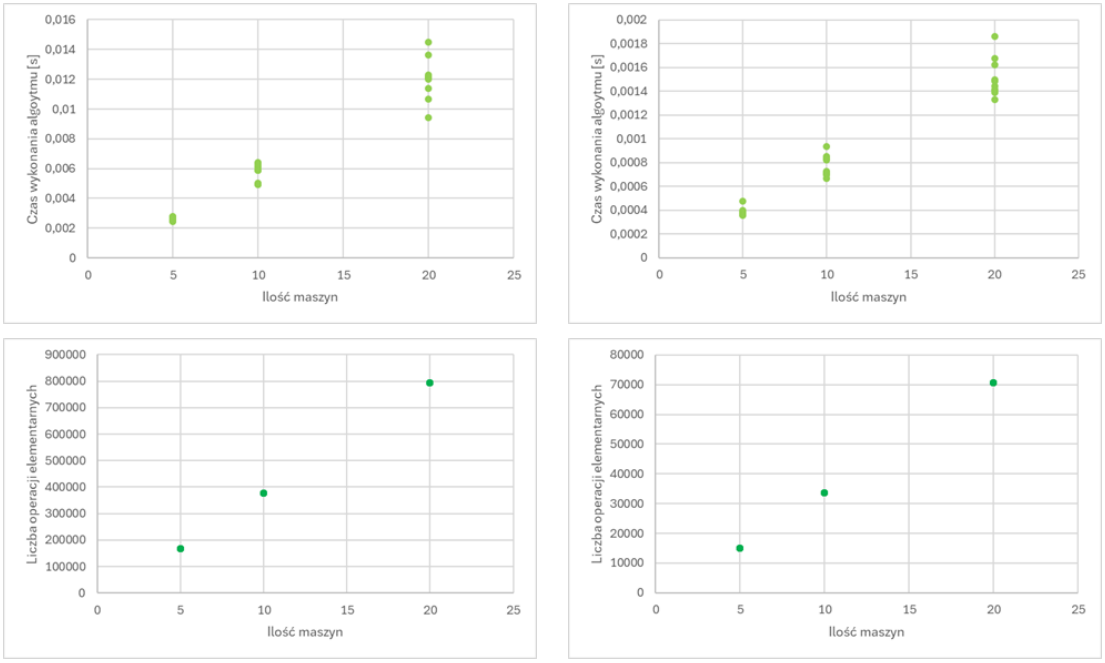
2.2 Analiza ze względu na ilość maszyn

Analogiczną analizę przeprowadzono dla wpływu ilości maszyn na działanie programu. Podobnie jak poprzednio użyto danych o identycznej ilośći zadań.

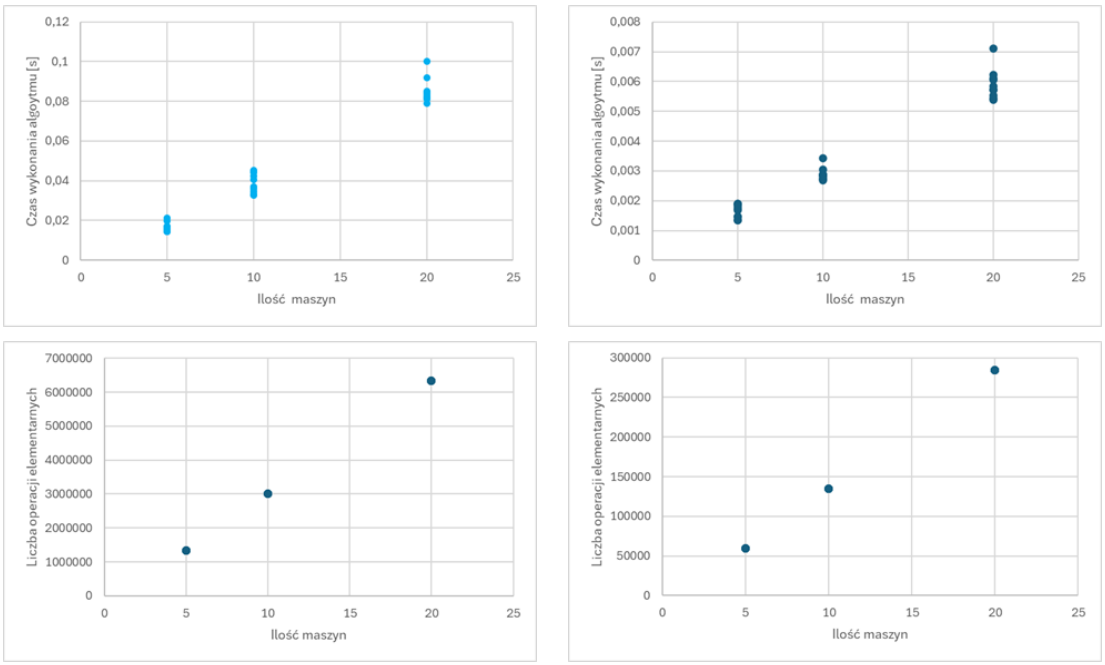
Tak samo jak w poprzednim podrozdziale, wykresy dla NEH znajdują się po lewej stronie, zaś dla QNEH po prawej.



Rysunek 10: Porównanie zmiany czasu wykonywania i ilości operacji elementarnych dla 20 zadań



Rysunek 11: Porównanie zmiany czasu wykonywania i ilości operacji elementarnych dla 50 zadań

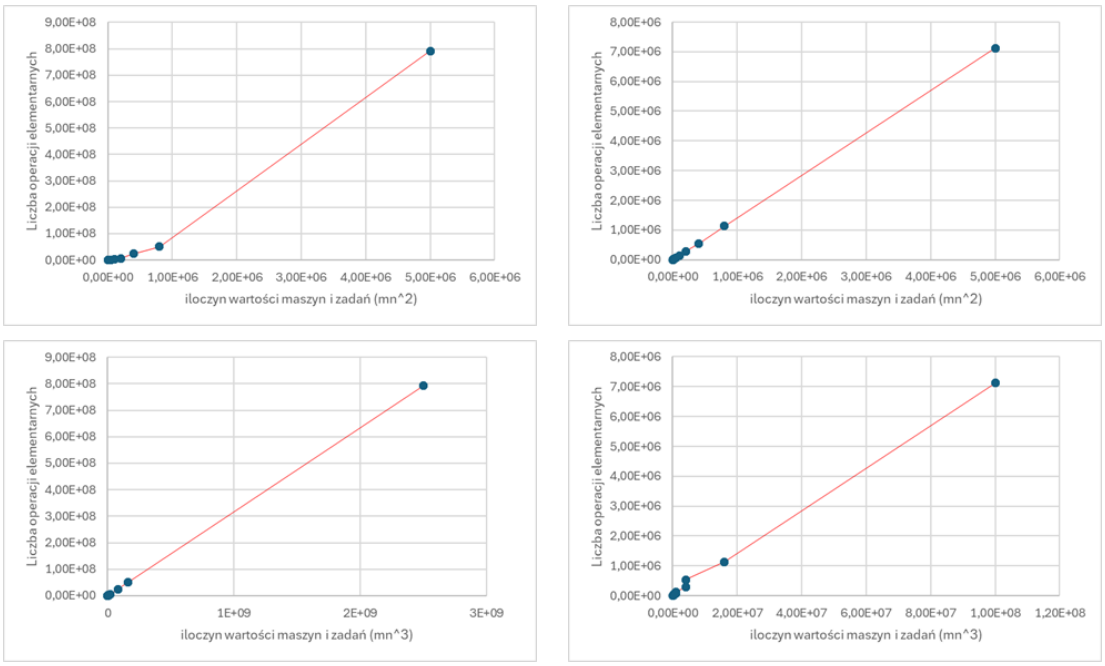


Rysunek 12: Porównanie zmiany czasu wykonywania i ilości operacji elementarnych dla 100 zadań

2.3 Jednoznaczne określenie złożoności obliczeniowej

Obserwując powyższe wykresy, można stwierdzić, że oba algorytmy zachowują się w sposób zbliżony do liniowego dla zmiennej liczby maszyn. Natomiast w przypadku liczby zadań, złożoność rośnie nieliniowo. Na podstawie tych obserwacji i założeniu, że złożoność obliczeniowa dla obu przypadków jest wielomianowa dokonałem dodatkowej analizy związanej z powiązaniem obu zmiennych w funkcję, która w li-

niowy sposób opisuje złożoność obliczeniową. Analizy dokonałem jedynie dla liczby operacji elementarnych, ponieważ dają bardziej jednostajne wyniki.



Rysunek 13: Określenie złożoności obliczeniowej na podstawie iloczynu zmiennych

Jak można zauważyć na powyższych wykresach, zgodnie z podejrzeniami oba algorytmy zachowują się liniowo względem ilości maszyn. Jeśli chodzi zaś o ilość zadań, to NEH ma wyższą złożoność obliczeniową o jeden stopień wyższą niż QNEH. Na podstawie tej analizy można jednoznacznie stwierdzić, że NEH ma złożoność obliczeniową $O(mn^3)$, zaś QNEH ma złożoność rzędu $O(mn^2)$. Gdzie „m” oznacza ilość maszyn, natomiast „n” – ilość zadań