Politechnika
Wrocławska

Sprawozdanie 1
Problem szeregowania zadań na 1 maszynie

Optymalizacja procesów dyskretnych

Aleksander Żołnowski - 272536

3 kwietnia 2025

1 Wstęp

Zadanie do wykonania dotyczyło rozwiązania problemu szeregowania zadań na jednej maszynie. Należało znaleźć minimalny czas zakończenia wszystkich zadań na maszynie C_{max} przy uwzględnieniu parametrów każdego z zadań w modelu RPQ, gdzie

- R - czas po którym zadanie jest gotowe
- P - czas produkcji
- Q - czas stygnięcia

W ramach zajęć zaimplementowano i zbadano różne metody rozwiązania tego problemu. Badania były prowadzone dla 4 zbiorów danych `data1.txt`, `data2.txt`, `data3.txt` i `data4.txt`

2 Przygotowanie danych

Pierwszym etapem było wczytanie do pamięci programu danych z plików tekstowych, gdzie zawarte są informacje o etapach. Plik zawiera liczbe zadań oraz wartości parametrów każdego z zadań. Dane są wczytywane do property klasy `Scheduler` jaką jest wektor struktur `Task`, przechowujący informacje o id, R, P i Q każdego z zadań.

```
class Scheduler
{
private:
    std::vector<Task> tasks;

public:
    void readFileContent(std::string dataFile);
    void quickSort(int low, int high);
    int calculateC_max();
    const std::vector<Task>& getTasks() const;
    void sortRQ(int maxIterations, int tabuSize);
    void schrage();
    void display_order();

private:
    int partition(int low, int high);
};
```

(a) Definicja klasy Scheduler

```
struct Task
{
    int id;
    int R;
    int P;
    int Q;
};
```

(b) Definicja struktury Task

Dane z plików były wczytywane przy użyciu metody `readFileContent`, gdzie należało przekazać nazwe pliku, który ma być wczytany.

```
1  void Scheduler::readFileContent(string dataFile)
2  {
3      ifstream file(dataFile);
4
5      if (!file.is_open())
6      {
7          cerr << "Error: Can not open the '" + dataFile + "' file" << endl;
8          return;
9      }
10
11     int numberOfTasks = 0;
12     file >> numberOfTasks;
13
14     tasks.reserve(numberOfTasks);
15
16     for (int i = 0; i < numberOfTasks; i++)
17     {
18         Task task;
19         task.id = i + 1;
20         file >> task.R >> task.P >> task.Q;
21         tasks.push_back(task);
22     }
23
24     file.close();
25 }
```

3 Obliczenie C_{max}

Pierwszym sposobem było obliczenie C_{max} dla każdego zbioru danych w kolejności w jakiej zostały wczytane z plików. W tym celu wykorzystano podstawowa funkcje do obliczania C_{max}.

```
int Scheduler::calculateC_max()
{
    int time = 0;
    int C_max = 0;

    for (const Task& task : tasks)
    {
        if (time < task.R)
        {
            time = task.R;
        }
        time += task.P;
        C_max = max(C_max, time + task.Q);
    }

    return C_max;
}
```

w ten sposób otrzymano następujące wyniki

	Data:1	Data:2	Data:3	Data:4	Suma
123	25994	33465	57403	51444	168306

Rysunek 2: Wyniki dla nieposortowanych danych

4 Sortowanie po R

Kolejnym krokie było zaimplemenotwanie algorytmu sortującego zadania po czasie gotowości R. W tym celu zastosowano algorytm `QuickSort`

```
int Scheduler::partition(int low, int high)
{
    int pivot = tasks[high].R;

    int i = low - 1;

    for (int j = low; j <= high - 1; j++)
    {
        if(tasks[j].R <= pivot)
        {
            i++;
            swap(tasks[i], tasks[j]);
        }
    }

    swap(tasks[i+1], tasks[high]);

    return i + 1;
}

void Scheduler::quickSort(int low, int high)
{
    if(low < high)
    {
        int pi = partition(low, high);

        quickSort(low, pi - 1);
        quickSort(pi + 1, high);
    }
}
```

Dzięki zastosowaniu sortowania otrzymaliśmy poniższe wyniki, które się znacznie polepszyły

	Data:1	Data:2	Data:3	Data:4	Suma
SortR	14239	33465	40042	39616	127362

Rysunek 3: Wyniki po sortowaniu wedlug R

5 Sortowanie RQ

Po wykonaniu sortowania po czasie gotowości zadań, algorytm został rozbudowany o dodatkowa optymalizacje polegająca na przeszukiwaniu i zamianie zadań w celu znalezienia lepszego ułożenia, które da lepszy czas. W tym celu wykorzystano metode przeszukiwania lokalnego implementując Tabu Search, czyli huerystyczny algorytm polegający na wielokrotnym testowaniu różnych permutacji zadań w celu wybrania najlepszej kolejności.

```
void Scheduler::sortRQ(int maxIterations, int tabuSize)
{
    quickSort(0, tasks.size() - 1);

    vector<Task> bestSolution = tasks;
    int bestC_max = calculateC_max();
    unordered_set<string> tabuList;

    for (int iter = 0; iter < maxIterations; iter++)
    {
        vector<Task> bestNeighbor = tasks;
        int bestNeighborC_max = bestC_max;
        int swapIdx1 = -1, swapIdx2 = -1;

        for (size_t i = 0; i < tasks.size() - 1; i++)
        {
            for (size_t j = i + 1; j < tasks.size(); j++)
            {
                swap(tasks[i], tasks[j]);
                int newC_max = calculateC_max();
                string move = to_string(i) + "," + to_string(j);
                if (newC_max < bestNeighborC_max && tabuList.find(move) == tabuList.end())
                {
                    bestNeighborC_max = newC_max;
                    bestNeighbor = tasks;
                    swapIdx1 = i;
                    swapIdx2 = j;
                }
                swap(tasks[i], tasks[j]);
            }
        }

        if (swapIdx1 != -1)
        {
            tabuList.insert(to_string(swapIdx1) + "," + to_string(swapIdx2));
            if (tabuList.size() > tabuSize) tabuList.erase(tabuList.begin());
            tasks = bestNeighbor;
            bestC_max = bestNeighborC_max;
        }
    }
}
```

W ten sposób otrzymaliśmy jeszcze lepsze wyniki niż dla samego sortowania po R:

	Data:1	Data:2	Data:3	Data:4	Suma
SortRQ	13966	20918	33683	33878	102445

Rysunek 4: Wyniki dla sortowania RQ

6 Algorytm Schrage

Ostatnim sposobem, który zaimplementowaliśmy i zbadaliśmy był algorytm Schrage'a. Służy on do szeregowania zadań polegając na dynamicznym dodawaniu dostępnych zadań do zbioru gotowych zadań i wybieraniu z nich tego, które ma największą wartość . Algorytm ten pozwala skutecznie minimalizować czas zakończenia wszystkich zadań, szczególnie w przypadku ograniczeń wynikających z różnych wartości .

```
void Scheduler::schrage()
{
    quickSort(0, tasks.size() - 1);

    vector<Task> N = tasks;
    vector<Task> G;
    vector<Task> order;

    int t = 0;
    int C_max = 0;

    while(!N.empty() || !G.empty())
    {
        while(!N.empty() && N.front().R <= t)
        {
            G.push_back(N.front());
            N.erase(N.begin());
        }

        if(!G.empty())
        {
            vector<Task>::iterator maxQ_it = max_element(G.begin(), G.end(), compareQ);

            Task e = *maxQ_it;
            G.erase(maxQ_it);

            order.push_back(e);
            t += e.P;
            C_max = max(C_max, t + e.Q);
        }
        else
        {
            t = N.front().R;
        }
    }

    tasks = order;
}
```

Sposób ten pozwolił osiągnać nam najlepszy czas ze wszystkich zaimplementowanych metod

	Data:1	Data:2	Data:3	Data:4	Suma
Schrage	13981	21529	31683	34444	101637

Rysunek 5: Wyniki algorytmu Schrage

7 Podsumowanie

Przebadano i zaimplementowano różne rozwiązania problemu szerogowania zadań na jednej maszynie, mającego na celu minimalizację czasu zakończenia wszystkich zadań C_{max}. Badania zostały przeprowadzone na 4 zbiorach danych, na których testowane były różne algorytmy i podejścia. Najlepsze wyniki uzyskano wykorzystując algorytm Schrage.

	123	SortR	SortRQ	Schrage
Data:1	25994	14239	13966	13981
Data:2	33465	33465	20918	21529
Data:3	57403	40042	33683	31683
Data:4	51444	39616	33878	34444
Suma:	168306	127362	102445	101637

Rysunek 6: Podsumowanie wyników