Politechnika
Wrocławska

Sprawozdanie 2
Problem WiTi

Optymalizacja procesów dyskretnych

Aleksander Żołnowski - 272536

16 kwietnia 2025

1 Wstęp

W ramach zadania należało rozwiązać problem WiTi, polegający na optymalnym szeregowaniu zadań na jednej maszynie w celu minimalizacji całkowitej kary za opóźnienia. Dla każdego zadania podano trzy parametry: `P`, `W` oraz `D`, gdzie:

- P - czas produkcji
- W - waga zadania, odzwierciedlająca koszt jego opóźnienia
- D - termin dostarczenia, po którym naliczana jest kara

Celem było znalezienie takiego uporządkowania zadań, które minimalizuje łączną karę wynikającą z przekroczenia terminów. W ramach zajęć problem został rozwiązany poprzez implementację algorytmu PD.

2 Przygotowanie danych

Pierwszym etapem było zaimplementowanie funkcji odpowiedzialnej za wczytywanie danych z plików tekstowych. Pliki te zawierają liczbę zadań oraz wartości parametrów każdego z nich: czas wykonania (P), wagę (W) i termin dostarczenia (D).
Dane te są wczytywane do właściwości klasy **Scheduler**, którą stanowi wektor struktur **Task**. Każda struktura `Task` przechowuje informacje o pojedynczym zadaniu, czyli id, czasie wykonania (P), wadze (W) oraz terminie (D).

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

public:
    void readFileContent(std::string filePath);
    int delayCost();
    void displayOrder();
    const std::vector<Task>& getTasks() const;
    void witiDP();
};
```

(a) Definicja klasy Scheduler

```
struct Task
{
    int id;
    int P;
    int W;
    int D;
};
```

(b) Definicja struktury Task

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

```
void Scheduler::readFileContent(string filePath)
{
    ifstream file(filePath);

    if(!file.is_open())
    {
        cerr << "Can not open file " + filePath << endl;
        return;
    }

    int size = 0;
    file >> size;

    tasks.reserve(size);

    for (int i = 0; i < size; i++)
    {
        Task task;
        task.id = i + 1;
        file >>  task.P >> task.W >> task.D;

        tasks.push_back(task);
    }

    file.close();
}
```

3 Obliczenie kary za opóźnienia

Po wczytaniu danych do pamięci programu, przystąpiono do obliczenia kar za opóźnienia dla każdego zestawu danych, w kolejności, w jakiej zadania zostały wczytane z plików. Do tego celu zaimplementowano funkcję `delayCost()`, której zadaniem jest obliczenie całkowitego kosztu opóźnień na podstawie aktualnej kolejności zadań.

```
int Scheduler::delayCost()
{
    int currentTime = 0;
    int coast = 0;

    for(const Task& task : tasks)
    {
        currentTime += task.P;

        if(currentTime > task.D)
        {
            coast += (currentTime - task.D) * task.W;
        }
    }

    return coast;
}
```

w ten sposób otrzymano następujące wyniki

Data	10	11	12	13	14	15	16	17	18	19	20	SUM
time	3994	6323	7778	7322	6851	6388	5925	5468	6189	5656	13252	75146

Rysunek 2: Wyniki dla nieposortowanych danych

4 Algorytm DP

Kolejnym etapem było zaimplemetowanie algorytmu DP, którego celem jest znalezienia optymalnego ułożenia zadań minimalizującego całkowity koszt opóźnienia. Algorytm DP ,czyli programowania dynamicznego wykorzystuje reprezentację podzbiorów zadań za pomocą masek bitowych, aby znaleźć permutację zadań minimalizującą całkowitą karę za opóźnienia. Dla każdej możliwej kombinacji zadań obliczany jest minimalny koszt wykonania, a następnie na podstawie wartości pomocniczych odtwarzana jest optymalna kolejność. Podejście to zapewnia dokładne rozwiązanie dla małych instancji problemu.

```
void Scheduler::witiDP()
{
    int n = tasks.size();
    int size = 1 << n;
    vector<int> dp(size, INT_MAX);
    vector<int> prev(size, -1);
    vector<int> lastTask(size, -1);

    dp[0] = 0;

    for (int mask = 0; mask < size; ++mask)
    {
        int time = 0;
        for (int i = 0; i < n; ++i)
        {
            if (mask & (1 << i))
            {
                time += tasks[i].P;
            }
        }

        for (int j = 0; j < n; ++j)
        {
            if (!(mask & (1 << j)))
            {
                int nextMask = mask | (1 << j);
                int completionTime = time + tasks[j].P;
                int tardiness = max(0, completionTime - tasks[j].D);
                int cost = dp[mask] + tardiness * tasks[j].W;

```

```
                if (cost < dp[nextMask])
                {
                    dp[nextMask] = cost;
                    prev[nextMask] = mask;
                    lastTask[nextMask] = j;
                }
            }
        }
    }

    vector<Task> optimalOrder;
    int mask = size - 1;
    while (mask)
    {
        int taskIndex = lastTask[mask];
        optimalOrder.push_back(tasks[taskIndex]);
        mask = prev[mask];
    }

    reverse(optimalOrder.begin(), optimalOrder.end());

    tasks = optimalOrder;
}
```

Dzięki zastosowaniu algorytmu DP otrzymaliśmy następujące wyniki dla naszych zbiorów danych:

Data	10	11	12	13	14	15	16	17	18	19	20	SUM
time	766	799	742	688	497	440	423	417	405	393	897	6467

Rysunek 3: Wyniki po sortowaniu wedlug R

5 Podsumowanie

W ramach zadania zaimplementowano algorytm programowania dynamicznego (DP) w celu rozwiązania problemu WiTi. Otrzymane wyniki jednoznacznie pokazują, że zastosowanie algorytmu DP znacząco poprawia jakość rozwiązania w porównaniu do bazowego ułożenia zadań. W każdym przypadku całkowity koszt opóźnień został zredukowany, często nawet kilkukrotnie. Metoda DP okazała się niezwykle skuteczna dla mniejszych instancji problemu, oferując dokładne i zoptymalizowane wyniki.

Data	10	11	12	13	14	15	16	17	18	19	20	SUM
123	3994	6323	7778	7322	6851	6388	5925	5468	6189	5656	13252	75146
PD	766	799	742	688	497	440	423	417	405	393	897	6467

Rysunek 4: Podsumowanie wyników