<table>
<tr><td colspan="3">Wydział Nauk Inżynieryjnych ANS w Nowym Sączu
Optymalizacja dyskretna - projekt</td></tr>
<tr><td colspan="2">Temat: Programowanie dyskretne - zagadnienie przydziału. Rozwiązanie za pomocą algorytmu węgierskiego. Operacje na tablicach, funkcje.</td><td>Symbol:
OD_P1</td></tr>
<tr><td>Nazwisko i imię:

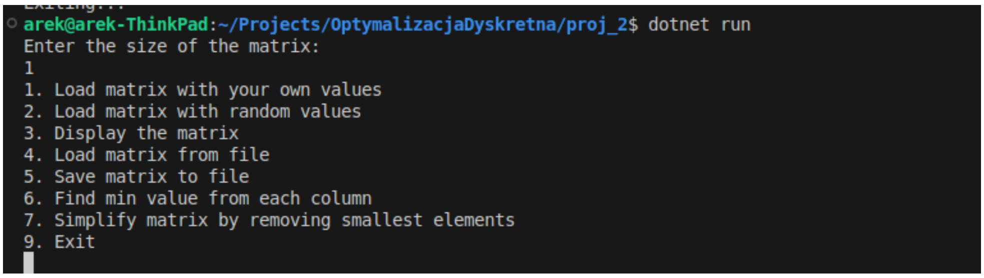
Ryczek Arkadiusz</td><td>Ocena sprawozdania:</td><td>Zaliczenie:</td></tr>
<tr><td>Data wykonania ćwiczenia:</td><td colspan="2">Oceniane efekty uczenia się:
EUU1=................. ,EUU2=................. ,EUU3=...................., EUK1=.....................</td></tr>
</table>

Wstęp

Celem laboratoriów była implementacja algorytmu węgierskiego do rozwiązywania problemu przydziału zadań pracownikom. Problem przydziału polega na znalezieniu optymalnego przypisania zadań do pracowników tak, aby zminimalizować całkowity koszt lub zmaksymalizować całkowitą efektywność. Algorytm węgierski jest efektywną metodą rozwiązania tego problemu.

Zadanie 1

Jako że zadanie 1 jest takie samo jak na wcześniejszych laboratoriach, nie będę go tu powtarzał, za to zamieszczę zrzuty ekranu(Rys. 1 (s. 1)) z jego działania.



Rysunek 1: Menu programu.

Zadanie 2 i 3

Zadania 2 i 3 zostały połączone w jeden program, który implementuje kompletny algorytm węgierski do rozwiązywania zagadnienia przydziału.

Zadanie 2

Zadanie polega na napisaniu funkcji, która dokona redukcji wyczytanej tablicy - pierwszy i drugi krok algorytmu węgierskiego. Po redukcji w każdym wierszu i kolumnie powinno być przynajmniej jedno zero, a pozostałe elementy są nieujemne.

Implementacja obejmuje:

- Redukcję wierszową - odejmowanie minimum z każdego wiersza
- Redukcję kolumnową - odejmowanie minimum z każdej kolumny

Zadanie 3

Zadanie polega na napisaniu kompletnego programu rozwiązującego zagadnienie przydziału przy użyciu algorytmu węgierskiego. Program implementuje wszystkie kroki algorytmu:

- Redukcję tablicy (zadanie 2)

- Znajdowanie maksymalnego skojarzenia
- Tworzenie linii pokrywających wszystkie zera
- Modyfikację tablicy w przypadku niepełnego pokrycia
- Znajdowanie optymalnego przydziału

```
    Console.WriteLine("Hungarian Algorithm - solving assignment problem:");

    // Create a copy of the matrix for the algorithm
    int[,] workMatrix = new int[size, size];
    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size; j++)
        {
            workMatrix[i, j] = matrix[i, j];
        }
    }

    Console.WriteLine("Original matrix:");
    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size; j++)
        {
            Console.Write(workMatrix[i, j] + "\t");
        }
        Console.WriteLine();
    }

    // Step 1: Subtract row minimums
    Console.WriteLine("\nStep 1: Subtract row minimums");
    for (int i = 0; i < size; i++)
    {
        int min = workMatrix[i, 0];
        for (int j = 1; j < size; j++)
        {
            if (workMatrix[i, j] < min)
            {
                min = workMatrix[i, j];
            }
        }

        Console.WriteLine($"Row {i} minimum: {min}");
        for (int j = 0; j < size; j++)
        {
            workMatrix[i, j] -= min;
        }
    }

    Console.WriteLine("After step 1:");
    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size; j++)
        {
```

```
                Console.Write(workMatrix[i, j] + "\t");
            }
            Console.WriteLine();
            }

            // Step 2: Subtract column minimums
            Console.WriteLine("\nStep 2: Subtract column minimums");
            for (int j = 0; j < size; j++)
            {
            int min = workMatrix[0, j];
            for (int i = 1; i < size; i++)
            {
                if (workMatrix[i, j] < min)
                {
                    min = workMatrix[i, j];
                }
            }

            if (min > 0)
            {
                Console.WriteLine($"Column {j} minimum: {min}");
                for (int i = 0; i < size; i++)
                {
                    workMatrix[i, j] -= min;
                }
            }
        }

        Console.WriteLine("After step 2:");
        for (int i = 0; i < size; i++)
        {
            for (int j = 0; j < size; j++)
            {
                Console.Write(workMatrix[i, j] + "\t");
            }
            Console.WriteLine();
        }

        // Main algorithm loop
        int iteration = 0;
        while (true)
        {
            iteration++;
            Console.WriteLine($"\nIteration {iteration}:");

            // Find maximum matching
            int[] assignment = new int[size];
            for (int i = 0; i < size; i++) assignment[i] = -1;

            bool[] rowMatched = new bool[size];
            bool[] colMatched = new bool[size];
            int matchCount = 0;

            // Try to find maximum matching using zeros
            for (int i = 0; i < size; i++)
```

```
        {
            for (int j = 0; j < size; j++)
            {
                if (workMatrix[i, j] == 0 && !rowMatched[i] && !colMatched[j])
                {
                    assignment[i] = j;
                    rowMatched[i] = true;
                    colMatched[j] = true;
                    matchCount++;
                    break;
                }
            }
        }

        Console.WriteLine($"Found {matchCount} matches");

        if (matchCount == size)
        {
            // Found optimal assignment
            int totalCost = 0;
            Console.WriteLine("\nOptimal assignment found:");
            for (int i = 0; i < size; i++)
            {
                Console.WriteLine($"Worker {i+1} -> Job {assignment[i]+1}, Cost
                    : {matrix[i, assignment[i]]}");
                totalCost += matrix[i, assignment[i]];
            }
            Console.WriteLine($"Total assignment cost: {totalCost}");
            break;
            }

        // Step 3: Find minimum vertex cover using Konig's theorem
        bool[] rowCovered = new bool[size];
        bool[] colCovered = new bool[size];

        // Start with unmatched rows
        bool[] visited = new bool[size];
        for (int i = 0; i < size; i++)
        {
            if (!rowMatched[i])
            {
                DfsAlternatingPath(workMatrix, assignment, i, visited,
                    rowCovered, colCovered, size);
            }
        }

        // Apply Konig's theorem: cover unvisited rows and visited columns
        for (int i = 0; i < size; i++)
        {
            if (!visited[i]) rowCovered[i] = true;
        }

        for (int j = 0; j < size; j++)
        {
            for (int i = 0; i < size; i++)
```

```
            {
                if (visited[i] && workMatrix[i, j] == 0)
                {
                    colCovered[j] = true;
                    break;
                }
            }
        }

        // Count covering lines
        int lineCount = 0;
        for (int i = 0; i < size; i++) if (rowCovered[i]) lineCount++;
        for (int j = 0; j < size; j++) if (colCovered[j]) lineCount++;

        Console.WriteLine($"Number of covering lines: {lineCount}");

        // Step 4: Create additional zeros
        Console.WriteLine("Step 4: Create additional zeros");

        // Find minimum uncovered element
        int minUncovered = int.MaxValue;
        for (int i = 0; i < size; i++)
        {
            for (int j = 0; j < size; j++)
            {
                if (!rowCovered[i] && !colCovered[j] && workMatrix[i, j] <
                    minUncovered)
                {
                    minUncovered = workMatrix[i, j];
                }
            }
        }

        Console.WriteLine($"Smallest uncovered element: {minUncovered}");

        // Adjust matrix
        for (int i = 0; i < size; i++)
        {
            for (int j = 0; j < size; j++)
            {
                if (!rowCovered[i] && !colCovered[j])
                {
                    workMatrix[i, j] -= minUncovered;
                }
                else if (rowCovered[i] && colCovered[j])
                {
                    workMatrix[i, j] += minUncovered;
                }
            }
        }

        Console.WriteLine("Matrix after adjustment:");
        for (int i = 0; i < size; i++)
        {
            for (int j = 0; j < size; j++)
```

```
                {
                    Console.Write(workMatrix[i, j] + "\t");
                }
                Console.WriteLine();
            }
        }
        break;
```

Listing 1: Implementacja algorytmu węgierskiego do rozwiązania zagadnienia przydziału.

Listing 1 (s. 2) przedstawia kompletną implementację algorytmu węgierskiego w języku C#. Algorytm wykonuje następujące kroki:

1. **Kopiowanie macierzy wejściowej** - tworzenie kopii roboczej macierzy kosztów do manipulacji
2. **Redukcja wierszowa** - odejmowanie minimum z każdego wiersza od wszystkich elementów tego wiersza
3. **Redukcja kolumnowa** - odejmowanie minimum z każdej kolumny od wszystkich elementów tej kolumny (jeśli minimum > 0)
4. **Pętla główna algorytmu** zawierająca iteracje:
   (a) Wyszukiwanie maksymalnego skojarzenia - przypisywanie zer do niesparowanych wierszy i kolumn
   (b) Sprawdzenie warunki stopu - jeśli wszystkie wiersze są sparowane, algorytm kończy działanie
   (c) Zastosowanie twierdzenia Koniga - znajdowanie minimalnego pokrycia wierzchołków przy użyciu przeszukiwania DFS
   (d) Pokrywanie zer liniami - określenie nieodwiedzonych wierszy i odwiedzonych kolumn jako linie pokrywające
   (e) Modyfikacja macierzy - dodawanie/odejmowanie najmniejszego niepokrytego elementu:
      - Odejmowanie od elementów niepokrytych (przecięcie niepokrytych wierszy i kolumn)
      - Dodawanie do elementów pokrytych podwójnie (przecięcie pokrytych wierszy i kolumn)
5. **Wyświetlenie wyniku** - prezentacja optymalnego przydziału z obliczeniem całkowitego kosztu

Algorytm gwarantuje znalezienie optymalnego rozwiązania w skończonej liczbie iteracji dzięki właściwościom redukcji macierzy i twierdzeniu Koniga o pokryciu wierzchołków w grafach dwudzielnych.

Poniższe zdjęcia (Rys. 2 (s. 7) - Rys. 5 (s. 7)) pokazuje działanie połączonego programu z funkcją redukcji tablicy oraz pełnym rozwiązaniem zagadnienia przydziału algorytmem węgierskim.

```
Hungarian Algorithm - solving assignment problem:
Original matrix:
3       6       8       15      10
7       4       3       12      18
9       1       8       13      15
10      4       7       11      20
15      9       5       4       6

Step 1: Subtract row minimums
Row 0 minimum: 3
Row 1 minimum: 3
Row 2 minimum: 1
Row 3 minimum: 4
Row 4 minimum: 4
After step 1:
0       3       5       12      7
4       1       0       9       15
8       0       7       12      14
6       0       3       7       16
11      5       1       0       2
```

Rysunek 2: Macierz oryginalna i odjęcie minimum z wierszy

```
Step 2: Subtract column minimums
Column 4 minimum: 2
After step 2:
0       3       5       12      5
4       1       0       9       13
8       0       7       12      12
6       0       3       7       14
11      5       1       0       0

Iteration 1:
Found 4 matches
Number of covering lines: 4
Step 4: Create additional zeros
Smallest uncovered element: 3
Matrix after adjustment:
0       6       5       12      5
4       4       0       9       13
5       0       4       9       9
3       0       0       4       11
11      8       1       0       0
```

Rysunek 3: Redukcja kolumnowa i wytyczenie linii przecinajacych zera (4 linie), wytyczenie najmniejszego elementu (3) i transformacja macierzy

```
Iteration 2:
Found 4 matches
Number of covering lines: 4
Step 4: Create additional zeros
Smallest uncovered element: 3
Matrix after adjustment:
0       9       8       12      5
1       4       0       6       10
2       0       4       6       6
0       0       0       1       8
11      11      4       0       0

Iteration 3:
Found 4 matches
Number of covering lines: 4
Step 4: Create additional zeros
Smallest uncovered element: 1
Matrix after adjustment:
0       9       8       11      4
1       4       0       5       9
2       0       4       5       5
0       0       0       0       7
12      12      5       0       0
```

Rysunek 4: 2 iteracja

```
Iteration 4:
Found 5 matches

Optimal assignment found:
Worker 1 -> Job 1, Cost: 3
Worker 2 -> Job 3, Cost: 3
Worker 3 -> Job 2, Cost: 1
Worker 4 -> Job 4, Cost: 11
Worker 5 -> Job 5, Cost: 6
Total assignment cost: 24
```

Rysunek 5: Optymalne rozwiązanie

Rysunek 6: Kroki algorytmu węgierskiego - zadania 2 i 3

Podsumowanie

Implementacja algorytmu węgierskiego była dosyć złożonym zadaniem, wymagającym dobrego zrozumienia jego natury, warunków brzegowych oraz efektywnego przeszukiwania macierzy. Pozwoliło to na poćwiczenie umiejętności programistycznych w C# oraz znajdowanie wiadomości na temat algorytmu węgierskiego w dostępnych źródłach internetowych. Mimo początkowych trudności, udało się stworzyć działający program, który skutecznie rozwiązuje problem przydziału zadań pracownikom przy minimalnym koszcie.