# Algorytmy i struktury danych II - projekt zaliczeniowy

## Skład grupy:

Maryna Makas

Volha Rai

Yaucheni Luferchyk

## Dokumentacja

Wszystkie programy napisane w Python. Wszystkie funkcje i zmienne napisane po angielsku z polskimi komentarzami.

Najważniejsze biblioteki wykorzystane w rozwiązaniach: math, random, collections, matplotlib.

### Rozwiązanie Problemu 1:

**Problem.** Ustalić, w jaki sposób są transportowane odcinki z fabryki do miejsca budowy płotu. Możliwie szybko i możliwie małym kosztem zbudować płot.

Nad rozwiązaniem pierwszego problemu pracowały Panie Makas i Rai.

**Postwiliśmy sobie za cel**:

> Ustalić, w jaki sposób transportować odcinki z fabryki do miejsca budowy płotu w sposób szybki i tani.

Aby rozwiązać ten problem, nasz zespół potrzebował następujących algorytmów:

1. Algorytm Edmonsa-Karpa;
2. BFS (wyszukiwanie grafu wszerz);
3. Znajdowanie maksymalnego skojarzenia w grafie dwudzielnym;
4. Tworzenie sieci rezydualnej;
5. Tworzenie otoczki wypukłej za pomocą algorytmu Grahama.

Po kolei rozparzmy wszystkie wyżej wymienione algorytmy.

#### Algorytm Edmonsa-Karpa

Plik algorytm_edmonsa_karpa.py

Dla rozwiązywania Problemu №1 wykorzystałyśmy algorytm Edmondsa Karpa, który pomaga znaleźć maksymalny przepływ w sieci przepływowej.

Do wczytywania pliku z danymi używa się funkcji `ReadGraphFromFile()` .

Funkcja `Bfs()` implementuje algorytm BFS, który znajduje ścieżkę w grafie rezydualnym.

Główna funkcja `EdmondsKarp()` implementuje algorytm Edmonda-Karpa, który wielokrotnie używa BFS do znajdowania ścieżek powiększających i aktualizowania przepływu w sieci.

**Złożoność pamięciowa i czasowa:**

Najbardziej kosztowną operacją w algorytmie Edmonda-Karpa jest wielokrotne wykonywanie BFS, co prowadzi do całkowitej złożoności O(VE^2).

Pamięć na przechowywanie grafu przepustowościowego ( `residual_graph` ) to O(V + E). Pamięć na przechowywanie odwiedzonych wierzchołków ( `visited` ), kolejki ( `queue` ) i rodziców ( `parent` ) to O(V). Całkowita złożoność pamięciowa to O(V + E).

***Złożoność czasowa*:** O(VE^2)

***Złożoność pamięciowa:*** O(V + E)

## BFS (wyszukiwanie grafu wszerz)

Plik bfs.py

W kontekście tego problemu, BFS jest używany do znalezienia ścieżki od źródła (source) do ujścia (sink) w grafie rezydualnym.

**Inicjalizacja**:

`visited` to zbiór przechowujący odwiedzone wierzchołki (aby uniknąć ponownego odwiedzenia tych samych wierzchołków).

`queue` to kolejka FIFO używana do przechowywania wierzchołków, które należy przetworzyć.

`parent` to słownik, który przechowuje informacje o rodzicach wierzchołków.

**Pętla główna**:

`u = queue.popleft()` usuwa i zwraca pierwszy wierzchołek z kolejki do przetworzenia.

**Przetwarzanie sąsiadów**:

Pętla `for` iteruje po wszystkich sąsiadach wierzchołka oraz po ich przepustowościach ( `capacity` ).

`if` sprawdza, czy wierzchołek `v` nie został wcześniej odwiedzony ( `v not in visited` ) i czy przepustowość krawędzi między `u` a `v` jest większa od zera ( `capacity > 0` ).

**Sprawdzenie, czy osiągnięto ujście**:

Jeśli aktualnie przetwarzany wierzchołek `v` jest wierzchołkiem ujściowym (`if v == sink`), funkcja zwraca `True`, co oznacza, że znaleziono ścieżkę od źródła do ujścia.

**Brak ścieżki**:

Jeśli kolejka się opróżni, a ujście nie zostało osiągnięte, funkcja zwraca `False`, co oznacza, że nie ma dostępnej ścieżki od źródła do ujścia w grafie przepustowościowym.

**Złożoność pamięciowa i czasowa:**

Złożoność czasowa oraz pamięciowa zależy od liczby wierzchołków (V) i liczby krawędzi (E) w grafie.

***Złożoność czasowa:*** *O(V + E), bo:*

1. Odwiedzenie wszystkich wierzchołków zajmuje O(V) czasu.
2. Sprawdzenie wszystkich krawędzi zajmuje O(E) czasu.

***Złożoność pamięciowa:*** *O(V), bo:*

1. Zbiór `visited` może zawierać do V wierzchołków.
2. Kolejka `queue` może przechowywać do V wierzchołków.
3. Słownik `parent` może zawierać do V wpisów, ponieważ każdy wierzchołek może mieć jednego rodzica.

## Znajdowanie maksymalnego skojarzenia w grafie dwudzielnym

Plik **maksymalne_skojarzenie_w_grafie_dwudzielnym.py**

Do konwersji grafu dwudzielnego na sieć przepływową istnieje funkcja `BipartiteToFlowGraph()` .

Po pierwsze, inicjaluzujemy słownik, który będzie służył do przechowywania struktury sieci przepływowej (ona jest wynikiem konwersji grafu dwudzielnego na sieć przepływową).

Dodajemy źródło ‘s’ do słownika `flow_graph`:

```
flow_graph[0] = {x: 1 for x in X}
```

Potem dodajemy ujście ‘t’:

```
t = max(max(X), max(Y)) + 1
for y in Y:
    if y not in flow_graph:
        flow_graph[y] = {}
    flow_graph[y][t] = 1
```

Obliczamy indeks wierzchołka ujściowego `t`, który jest o 1 większy od największego wierzchołka w zbiorze `X` lub `Y` . Następnie dodajemy wierzchołek `t` do słownika `flow_graph` dla każdego wierzchołka `y` ze zbioru `Y`. Przepustowość krawędzi między `y` a `t` wynosi 1.

Ponadto, musimy dodać krawędzi grafu dwudzielnego:

```
for x in X:
        if x not in flow_graph:
            flow_graph[x] = {}
        for y in bipartite_graph[x]:
            flow_graph[x][y] = 1
```

Czyli iterujemy przez każdy wierzchołek `x` ze zbioru `X` i dodajemy krawędzie do każdego wierzchołka `y`, z którym `x` jest połączony w grafie dwudzielnym `bipartite_graph`. Każda krawędź ma przepustowość równą 1.

## Tworzenie sieci rezydualnej

Plik tworzenie_sieci_rezydualnej.py

Ten plik zawiera definicję klasy `ResidualGraph` , która służy do tworzenia grafu rezydualnego na podstawie grafu przepływowego.

W tej klasie znajduje się metoda `BuildResidualGraph()` , która buduje graf rezydualny na podstawie grafu przepływowego.

Ten plik jest potrzebny do programu algorytm_edmonsa_karpa.py , aby znaleźć maksymalny przepływ w sieci.

**Złożoność pamięciowa i czasowa:**

***Złożoność czasowa:*** *O(V+E), bo:*

Budowanie grafu rezydualnego wymaga przejścia przez wszystkie wierzchołki i krawędzie oryginalnego grafu

***Złożoność pamięciowa:*** *O(V+E), bo:*

Każda krawędź w oryginalnym grafie jest przechowywana jako dwie krawędzie w grafie rezydualnym (jedna w kierunku oryginalnym i jedna w przeciwnym kierunku z przepustowością 0).

## Tworzenie otoczki wypukłej za pomocą algorytmu Grahama

Plik **otoczka_wypukla_grahama.py**

Ten program imlementuje algorytm Grahama do znajdowania otoczki wypukłej zbiou punktów na płaszczyźnie.

Dzięki funkcji math.atan2() obliczamy kąt pomiędzy dwoma punktami, aby później w algorytmie Grahama moglibyśmy ich posortować.

```
def Angle(point0, p):
    return math.atan2(p.y - point0.y, p.x - point0.x)
```

**Złożoność pamięciowa i czasowa:**

***Złożoność czasowa:*** *O(nlogn).*

***Złożoność pamięciowa:*** *O(n).*

## Cały program:

```
import math
import random
from collections import deque, defaultdict

class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return f"({self.x}, {self.y})"

    def DistanceBetween2Points(self, other):
        return math.sqrt(math.pow(self.x - other.x, 2) + math.pow(self.y

def Def(p1, p2, p3):
    return p1.x * p2.y + p2.x * p3.y + p3.x * p1.y - p2.y * p3.x - p1.y

def Angle(point0, p):
    return math.atan2(p.y - point0.y, p.x - point0.x)

def Graham(points):
    point0 = min(points, key=lambda p: (p.y, p.x))
    points.remove(point0)
    points.sort(key=lambda p: (Angle(point0, p), point0.DistanceBetween2
    stack = [point0, points[0]]
    for p in points[1:]:
        while len(stack) > 1 and Def(stack[-2], stack[-1], p) <= 0:
```

```
            stack.pop()
        stack.append(p)
    return stack

def GeneratePoints(number, x_min, x_max, y_min, y_max):
    points = []
    for _ in range(number):
        x = random.randint(x_min, x_max)
        y = random.randint(y_min, y_max)
        points.append(Point(x, y))
    return points

def Bfs(residual_graph, source, sink, parent):
    visited = set()
    queue = deque([source])
    visited.add(source)
    while queue:
        u = queue.popleft()
        for v, capacity in residual_graph[u].items():
            if v not in visited and capacity > 0:
                queue.append(v)
                visited.add(v)
                parent[v] = u
                if v == sink:
                    return True
    return False

def EdmondsKarp(graph, source, sink):
    residual_graph = defaultdict(dict)
    for u in graph:
        for v in graph[u]:
            residual_graph[u][v] = graph[u][v]
            residual_graph[v][u] = 0
    parent = {}
    max_flow = 0
    while Bfs(residual_graph, source, sink, parent):
        path_flow = float('Inf')
        s = sink
        while s != source:
            path_flow = min(path_flow, residual_graph[parent[s]][s])
            s = parent[s]
        max_flow += path_flow
        v = sink
        while v != source:
            u = parent[v]
```

```
            residual_graph[u][v] -= path_flow
            residual_graph[v][u] += path_flow
            v = parent[v]
    return max_flow

if __name__ == "__main__":

    #generowanie losowych punktow
    points = GeneratePoints(10, 0, 10, 0, 10)
    print("Wygenerowane punkty: ")
    for point in points:
        print(point)

    #oblicznie otoczki wypuklej za pomoca alg. Grahama
    convex_hull = Graham(points)
    print("\nPunkty otoczki wypuklej: ")
    for point in convex_hull:
        print(point)

    #tworzymy graf z punktow otoczki wypuklej
    graph = defaultdict(dict)
    n = len(convex_hull)
    source = 0  #zakladamy ze 0 - zrodlo
    sink = n + 1  #nowe ujscie

    #dodawanie krawedzi miedzy fabryka a punktami otoczki
    for i in range(n):
        graph[source][i + 1] = 1 #przepustowosc 1 (relacja przyjzni)
        graph[i + 1][sink] = 1 #laczenie punktow otoczki z ujsciem

    #dodawamie krawedzi miedzy punktami otoczki z losowymi przepustowosc
    for i in range(n):
        for j in range(i + 1, n):
            capacity = random.choice([0, 1]) #losowe okreslenie czy sa p
            graph[i + 1][j + 1] = capacity
            graph[j + 1][i + 1] = capacity

    print("\nWygenerowany graf z losowymi przepustowosciami: ")
    for u in graph:
        for v in graph[u]:
            print(f"{u} -> {v} : {graph[u][v]}")

    #obliczenie maks. przeplywu przy pomocy alg. Edmondsa-Karpa
    max_flow_value = EdmondsKarp(graph, source, sink)
    print("\nMaksymalny przeplyw (algorytm Edmondsa-Karpa):", max_flow_va
```

*Zakładamy, że zaprzyjaźnieni płaszczacy mają 1 na krawędzi pomiędzy sobą, odpowiedzio 0 w przyciwnym wypadku.

## Rozwiązanie Problemu 2:

**Problem.** Zapisać opowieść-melodię w maszynie Informatyka, zamieniając wcześniej „poli" na „boli" oraz próbując oszczędzić wykorzystane miejsce. Znaleźć rozwiązanie problemu ewentualnej zamiany innych fragmentów opowieści-melodii, który niepokoi Heretyka oraz Informatyka.

Nad rozwiązaniem drugiego problemu pracował Pan Luferchyk samodzielnie.

Całe rozwiązanie problemu znajduje się w pliku **haffman_codes.py**

Kod implementuje kodowanie Huffmana, które jest techniką kompresji danych bez strat. Rozwiązanie składa się z kilku funkcji i klas, które umożliwiają kodowanie i dekodowanie tekstu przy użyciu drzewa Huffmana oraz modyfikację tekstu poprzez zamianę słów.

Klasa `HuffmanNode` reprezentuje węzeł w drzewie Huffmana.

Funkcja `BuildHuffmanTree()` tworzy drzewo Huffmana z danych częstotliwości znaków. Oprócz tego, funkcja tworzy węzły dla każdego znaku i ich częstotliwości, a następnie umieszcza je w kopcu. Ważną rzeczą jest to, że `BuildHuffmanTree()` łączy dwa węzły o najmniejszych częstotliwościach w nowy węzeł (dopóki nie pozostanie tylko jeden węzeł - korzeń drzewa).

Funkcja `BuildCodes()` tworzy słownik kodów Huffmana dla każdego znaku. Czyli przechodzi przez drzewo Huffmana, przepisując "0" do lewej gałęzi i "1" do prawej, budując kod binarny dla każdego znaku.

Funkcja `HuffmanEncoding()` koduje dane wejściowe przy użyciu kodowania Huffmana.

Oczywiste jest to, że musi istnieć funkcja dekodująca, więc funkcja `HuffmanDecoding()` dekoduje dane przy użyciu słownika kodów Huffmana.

Aby zobaczyć działanie tego algorytmu istnieje funkcja `VisualizeHuffmanTree()` , która, jak wskazuje nazwa, wizualizuje drzewo Huffmana.

**Złożoność pamięciowa i czasowa:**

***Złożoność czasowa:*** *O(nlogn).*

***Złożoność pamięciowa:*** *O(n).*

## Rozwiązanie Problemu 3:

**Problem.** Ustalić jak najszybciej grafik pracy strażników i jak najmniejszą liczbę odsłuchań melodii dla każdego strażnika.

Podczas pracy nad rozwiązywaniem tego problemu zaimplementowaliśmy swój własny algorytm.

Podzieliliśmy ten problem na dwa podproblemy:

1. Wyznaczanie kolejności strażników;
2. Znalezienie optymalnego algorytmu, który byłby w stanie wyznaczyć grafik strażników.

Pani Rai szukała rozwiązania pierwszego podproblemu, a Pani Makas i Pan Luferchyk zajęli się drugim podproblemem.

## Podproblem pierwszy:

> Jak wyznaczyć kolejność strażników?

Zdecydowaliśmy, że dla rozkładu pracy strażników potrzebujemy tylko ośmiu strażników, aby każdy miał tydzień odpoczynku.

Zakładamy, że ilość wszystkich chętnych do pracy zawsze jest wielokrotnością cyfry 8, żebyśmy mogli podzielić całość na 8 równych części i wybrać 8 strażników z największą ilością energii.

Aby nie tworzyć otoczki wypukłej jeszcze jeden raz, połączyliśmy 1 i 3 problem.

Zakładamy, że maksymalna ilość energii płaszczka to ilość punktów otoczki wypukłej.

Za pomocą biblioteki numpy dzielimy ilość płaszczaków na 8, oraz za pomocą funkcji max wybieramy strażników z największą ilością energii.

Przez to, że wybieramy największą liczbę energii, w większości przypadków będziemy mieli listę płaszczaków z taką samą liczbą energii (ona będzie największa). Więc w poniższym kodzie mamy jeden pattern do przystanków na całej otoczce dla wszystkich strażników.

```
def GenerateAndFindMaxPlaszczaki(punkty_otoczki: list[int]) -> list[int]
    #Tworzymy listu plaszczakow, w której będzie maksymalna ilosc energi
    plaszczaki=[random.randint(0, len(punkty_otoczki)) for _ in range(80

    #Dzielimy na 8 części i wybieramy z maksymalną energią
    parts = np.array_split(plaszczaki, 8)
    max_numbers = [np.max(part) for part in parts]

    return max_numbers
```

## Podproblem drugi:

> Jak znaleźć optymalny algorytm przejścia po punktach otoczki, aby ustalić grafik strażników?

**Mamy dwie wersje tego algorytmu:**

Wersja pierwsza:

Jeśli strażnik zatrzyma się w punkcie, gdzie poprzedni punkt zatrzymania był jaśniejszy, to strażnik odzyska tyle energii, ile miał pierwotnie (to znaczy, że on może posunąć się tak daleko, jak to było wcześniej).

Wersja druga:

Jeśli strażnik zatrzyma się w punkcie, gdzie poprzedni punkt zatrzymania był jaśniejszy, to energia strażnika pozostaje bez zmian, ale energia zostanie wznowiona tylko wtedy, gdy poprzedni punkt zatrzymania był ciemniejszy.

Implementacje tych algorytmów są bardzo podobne, ale mają kilka różnic.

## Uwagi (Założenia)

1. Nie wiemy jak szybki jest strażnik i jak dużo okrążeń on zrobi w określonym czasie (załóżmy, że mamy dobę);
2. Z poprzedniego punktu wynika, że nasz algorytm pozwala wyznaczyć wzorzec (pattern), według którego będzie się powtarzał, co oznacza, że strażnik będzie chodzić w kółko w nieskończoność;
3. Przy przechodzeniu jednego punktu zużywa się jedna jednostka energii;
4. Maksymalna liczba energii strażnika = liczba punktów otoczki wypukłej.

## Opis Algorytmu

***Wejście***

1. Słownik z punktami i ich jasnością w postaci {numer_punktu: jasność}, gdzie ilość punktów = ilości punktów na otoczce.
2. Energia strażnika. Maksymum energii to liczba punktów
3. Flaga wizualizacji algorytmu (True/False)

***Kroki algorytmu:***

1. Tworzymy listę dostępnych punktów dla strażnika. Długość tej listy, a więc liczba punktów w tej liście, równa jest obecnej energii strażnika.

2. Algorytm szuka w tej liście punktu z największą możliwą jasnością, ale mniejszą niż w punkcie, w którym on obecnie jest.
3. Jeżeli takiego punktu nie ma, to wybiera spośród dostępnych punktów ten, który ma największą jasność.
4. Punkt, który otrzymaliśmy, staje się naszym bieżącym punktem, a poprzedni punkt zostaje usunięty ze słownika. Dodajemy ten punkt do listy naszych zatrzymań.
5. Wszystkie te kroki algorytmu powtarzamy, dopóki słownik nie będzie pusty.
6. Algorytm zwraca listę zatrzymań, co stanowi nasze wyjście.
7. Algorytm zwraca listę zatrzymań, co jest naszym wyjściem.

## Jaka wersja algorytmu jest lepsza?

**Wersja pierwsza:**

Będzie wykonywać więcej zatrzymań, ponieważ jeśli kolejne dostępne punkty są ciemniejsze niż obecny, zatrzyma się w tym punkcie, aby odzyskać energię. To podejście pozwala zachować energię, ale liczba odtworzeń melodii nie jest optymalna.

**Wersja druga:**

Jest bardziej optymalna pod względem liczby odtworzeń melodii, ponieważ strażnik zatrzymuje się w punktach tylko wtedy, gdy on musi odzyskać energię.