Optymalizacja procesu produkcyjnego metodą zachłanną

Współczesne Metody Heurystyczne - Zadanie projektoweJW2

**Zespół:**  
Maciej Szerling  
Adrian Wiśniewski

**Prowadzący:**  
prof. dr hab. Jacek Wojciechowski

Spis treści

[Zadanie projektowe. 3](#_Toc282727792)

[Wybór i opis algorytmu. 3](#_Toc282727793)

[Algorytmy zachłanne 3](#_Toc282727794)

[Uruchamianie 11](#_Toc282727795)

[Wejście. 11](#_Toc282727796)

[Wyjście. 11](#_Toc282727797)

[Rozwiązanie przykładowego zadania optymalizacyjnego 12](#_Toc282727798)

[Rozwiązanie przedstawione w postaci 3d 13](#_Toc282727799)

[Testy 14](#_Toc282727800)

[Porównanie jakości rozwiązań dla badanych algorytmów 15](#_Toc282727801)

[Badanie czasu rozwiązania w zależności od liczby kolorów karoserii 16](#_Toc282727802)

[Wnioski 17](#_Toc282727803)

[Wnioski końcowe 18](#_Toc282727804)

[Wykaz literatury. 18](#_Toc282727805)

## Zadanie projektowe.

Fabryka produkująca samochody oferuje n kolorów karoserii. Koszt wyczyszczenia i przestawienia linii do malowanie z koloru i-tego na j-ty wynosi cij (np. koszt przestawienia linii z koloru czarnego na biały może być wyższy od kosztu przestawienia z białego na czarny). Zakładamy, że możemy rozpocząć malowanie od dowolnego koloru i przechodząc przez wszystkie oferowane kolory wracamy do koloru startowego, a następnie powtarzamy cykl malowania. Kosztem pojedynczego cyklu jest suma kosztów cij przejścia przez kolejne kolory i powrotu do koloru startowego. Znaleźć najtańszą sekwencję malowania samochodów.

Zastosować metodę zachłanną i jako referencję porównać z metodą pełnego przeglądu. Napisać odpowiedni program komputerowy (włączając w to generator danych). Porównać jakość rozwiązań otrzymanych obydwoma metodami. Zbadać zależność czasu rozwiązania od liczby kolorów n dla metody zachłannej oraz pełnego przeglądu.

## Wybór i opis algorytmu.

Nasze zadanie jest analogiczne do problemu komiwojażera. Problem można zamodelować w postaci grafu, w którym konfiguracja koloru linii produkcyjnej odpowiada wierzchołkowi, a koszty przestawienia z jednego koloru na drugi­­­­ – wagom krawędzi. Znalezienie najtańszej sekwencji zmian kolorów linii produkcyjnej jest równoważne ze znalezieniem w tym grafie cyklu Hamiltona o minimalnym koszcie.

### Algorytmy zachłanne

Działanie algorytmów zachłannych opiera się o zasadę: “Wybierz najlepsze rozwiązanie dostępne w tej chwili”. Algorytmy te w żaden sposób nie gwarantują znalezienia rozwiązania optymalnego, natomiast ich zaletą jest niska złożoność obliczeniowa, pozwalająca znaleźć satysfakcjonujące rozwiązania w niedługim czasie, nawet dla bardzo dużych zbiorów danych.

#### Algorytm zachłanny wierzchołkowy „Greedy Vertices”

Algorytm ten działa w następujący sposób:

1. Dla każdego wierzchołka rozwiąż problem cząstkowy korzystając z metody:

Rozpoczynając od ustalonego wierzchołka wybierz krawędź o najmniejszym koszcie łączącą go z nieodwiedzonymdo tej pory wierzchołkiem i dołącz tę krawędź do budowanego cyklu. Zakończ gdy odwiedzisz wszystkie wierzchołki, dołączając do cyklu krawędź zamykającą.

1. Z rozwiązań cząstkowych wybierz najlepsze.

#### Pseudokod

SOLVE(E, V)

bestCost = MAX

bestPath = NIL

FOR EACH vertex v IN V #1

cost, path = SOLVE\_FROM\_VERTEX(E, V,v)

IF cost<bestCost THEN

bestCost = cost

bestPath = path

RETURN (bestCost, bestPath)

SOLVE\_FROM\_VERTEX(E, V, v)

path=NIL

current=v

cost=0

MARK\_VISITED(v)

WHILE EXIST\_NOT\_VISITED\_VERTEX(V)#2

u = FIND\_NEAREST\_UNVISITED\_VERTEX(V, current)#3

path += u

cost += EDGE\_COST(E, current, u)

MARK\_VISITED(u)

current = u

cost += EDGE\_COST(E, current, v)

RETURN (cost, path)

#### Złożoność obliczeniowa

W tym algorytmie wykonujemy 3 zagnieżdżone pętle, które iterują po wszystkich wierzchołkach. Wynika z tego, że algorytm ma złożoność gdzie n to liczba kolorów.

#### Złożoność pamięciowa

Przed wejściem do funkcji SOLVE musimy zaalokować pamięć na:

* Krawędzie E –
* Wierzchołki V –

Następnie w pętli #1 alokujemy

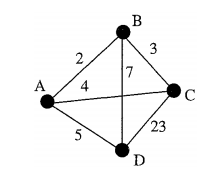
* Dla listy bestPath -
* W metodzie SOLVE\_FROM\_VERTEX -

Pozostałe alokacje mają złożoność O(1)

Na tej podstawie dochodzimy do wniosku, że złożoność pamięciowa algorytmu wynosi .

#### Przypadek złośliwy

Rozpatrzmy następujący przypadek:



Zastosujmy algorytm zachłanny wierzchołkowy rozpoczynając od wierzchołka A. Wykonujemy następujące kroki:

1. Z wierzchołka A wybieramy najbliższy wierzchołek, czyli B. Ścieżka: AB, koszt 2.
2. Z wierzchołka B idziemy do najbliższego wierzchołka, czyli C. Ścieżka: ABC koszt 5.
3. Z wierzchołka C możemy iść tylko do D (A był już wcześniej odwiedzony). Ścieżka ABCD, koszt 28.
4. Zamykamy cykl. Otrzymany cykl: ABCDA, koszt 33.

Analogiczne kroki wykonujemy rozpoczynając kolejno z innych wierzchołków. Najlepsze rozwiązanie to cykl CBADC o koszcie 31.

Optymalne rozwiązanie to ACBDA, którego koszt wynosi jedynie 19. Otrzymaliśmy więc rozwiązanie bardzo odległe od optymalnego. Nie zabezpieczyło nas przed nim sprawdzenie wszystkich rozwiązań wybierając za każdym razem inny wierzchołek jako startowy.

Algorytm zachłanny krawędziowy „Greedy Edges”

W algorytmie krawędzie są sortowane od najmniejszych do największych wag. Następnie posortowane krawędzie są przeglądane kolejno i w miarę możliwości dodawane do tworzonego cyklu. Rozpatrywana krawędź może być dodana tylko wtedy, gdy budowany cykl nie zawiera już innej krawędzi, zaczynającej lub kończącej się w tym samym wierzchołku, co krawędź rozpatrywana (wynika to z natury samego cyklu). Ponadto dodawana krawędź nie może tworzyć cyklu z żadną ścieżką, utworzoną z wybranych do tej pory krawędzi, chyba że jest to krawędź kończąca cały cykl. Algorytm kończy działanie gdy utworzy pełny cykl.

#### Pseudokod

SOLVE(E, V)

path = NIL

cost = 0

pathEdges = NIL

disabledEdges = NIL

usedVertices = NIL

SORT\_BY\_EDGE\_COST(E) #1

FOR EACH edge (u,v) IN E #2

IF (u,v) IN disabledEdges #3

CONTINUE

IF CREATES\_CYCLE((u,v), pathEdges) #4

CONTINUE

cost += EDGE\_COST(E, u, v)

pathEdges += (u,v)

usedVertices += u

usedVertices += v

FOR EACH vertex w IN V #5

disabledEdges += (u,w)

disabledEdges += (w,v)

IF usedVertices == V THEN

BREAK

path = CONSTRUCT\_PATH(pathEdges)

(u,v) = GET\_ENDS(bestPath)

cost += EDGE\_COST(E, v, u)

path += (v,u)

RETURN (cost, path)

#### Złożoność obliczeniowa

Oszacowanie złożoności w tym przypadku jest nieco bardziej kłopotliwe, ponieważ algorytm operuje na krawędziach, których liczba wynosi . Przyjmijmy oznaczenia:

* **n** – ilość wierzchołków, równa ilości kolorów
* **E** – ilość krawędzi, w przypadku grafu pełnego
* **Osort** – złożoność metody sortującej **#1**
* **Oforeach** – złożoność pętli **#2**
* **Odisabled** – złożoność sprawdzenia, czy krawędź może być dodana do cyklu **#3**
* **Ocycle** – złożoność sprawdzenia, czy krawędź przedwcześnie tworzy cykl **#4**
* **Overtex** – złożoność pętli **#5**
* **Oalgorithm** – złożoność całego algorytmu

Złożoność algorytmu wynosi:

Algorytm w fazie przygotowywania danych wykonuje sortowanie wszystkich krawędzi, a następnie uruchamia pętlę iterującą po wszystkich krawędziach. Pętla dla każdej krawędzi sprawdza, czy dana krawędź może zostać dodana do cyklu poprzez sprawdzenie dwóch warunków. Dla dokładnie *n* krawędzi warunki te będą spełnione i zostanie wykonana reszta ciała pętli.

Zapamiętując informacje o krawędziach, których można użyć, w postaci macierzywartości logicznych, można sprowadzić ten test do pobrania odpowiedniej wartości z macierzy. Pozwala to na zredukowanie złożoności**Odisabled**do .

Przechowując informacje o początkach i końcach ścieżek utworzonych przez wybrane krawędzie w postaci dwóch tablic, można także zmniejszyć złożoność **Ocycle**do . Tablice te zawierają wskazania na utworzone ścieżki pod indeksami odpowiednio początkowych i końcowych wierzchołków. Wystarczy pobrać ścieżkę kończącą się w wierzchołku początkowym wybranej krawędzi i ścieżkę zaczynającą się w wierzchołku docelowym. Jeżeli oba wskazania istnieją i wskazują na tą samą ścieżkę, oznacza to, że krawędź faktycznie tworzy przedwczesny cykl.

Złożoność **Osort**ściśle zależy od wybranego algorytmu sortowania. Podczas implementacji algorytmu został wybrany przez nas algorytm quicksort o złożoności . Pamiętając, że algorytm ten działa na zestawie wszystkich krawędzi, złożoność **Osort**można wyrazić jako:

Pętla sprawdzająca wszystkie krawędzie ma złożoność **Oforeach**równą , natomiast iterująca po wszystkich wierzchołkach **Overtex**równą.

Agregując powyższe wyprowadzenia, wzór na złożoność obliczeniową całego algorytmu można zapisać w postaci:

#### Złożoność pamięciowa

Przed wejściem do funkcji SOLVE musimy zaalokować pamięć na:

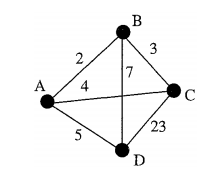
* Krawędzie E –
* Wierzchołki V –

Następnie:

* Zakładając, że metoda sortująca SORT\_BY\_EDGE\_COST korzysta z algorytmu quicksort, alokujemy w niej pesymistycznie O(E) danych na stos wywołań rekurencyjnych, czyli jej złożoność pamięciowa wynosi
* W pętli #2 złożoność pamięciowa wynosi O(n)

Na podstawie powyższych stwierdzamy, iż złożoność pamięciowa całego algorytmu wynosi .

#### Przypadek złośliwy



Zastosujmy algorytm zachłanny krawędziowy. Na początku sortujemy krawędzie wg. kosztów:

AB = 2, BC = 3, CA = 4, AD = 5, BD = 7, CD = 23

1. Zgodnie z algorytmem rozpoczynamy od krawędzi AB. Ścieżka = AB, Koszt 2.
2. Przechodzimy do ścieżki BC. Dołączamy ją do istniejącej ścieżki. Ścieżka = ABC, koszt 5.
3. Przechodzimy do ścieżki CA, odrzucamy ją, gdyż zamknęłaby cykl.
4. Przechodzimy do ścieżki AD, dołączamy ją do istniejącej ścieżki. Ścieżka = DABC koszt 10.
5. Domykamy cykl krawędzią CD. Cykl DABC, koszt 33.

Stosując algorytm zachłanny krawędziowy otrzymujemy rozwiązanie: DABC o koszcie 33. W porównaniu do przypadku optymalnego o koszcie 19, otrzymane rozwiązanie jest wysoce niedoskonałe.

#### Algorytm pełnego przeglądu “Full Search”

Rozwiązanie to polega na przejrzeniu wszystkich dostępnych możliwości i wybraniu z nich najlepszego. Algorytm pozwala nam na znalezienie minimum globalnego, ale jego złożoność ogranicza jego zastosowanie jedynie do problemów bardzo małych rozmiarów.

#### Pseudokod

GLOBAL

bestPath = NIL

bestCost = MAX

SOLVE(E, V)

FOR EACH vertex v IN V

SOLVE\_RECURSIVE(V,E,0,v)

RETURN (bestCost, bestPath)

SOLVE\_RECURSIVE(V, E, accumulatedCost, accumulatedPath)

IF LENGTH(accumulatedPath) = LENGTH(V)

accumulatedCost += EDGE\_COST(E,GET\_ENDS(accumulatedPath))

IF accumulatedCost<bestCost

bestCost = accumulatedCost

bestPath = accumulatedPath

ELSE

last\_visited = LAST\_VISITED(accumulatedPath)

FOR EACH vertex destination IN (V-accumulatedPath)

cost = EDGE\_COST(E, last\_visited, destination)

SOLVE\_RECURSIVE(accumulatedPath+destination,

accumulatedCost+cost)

RETURN

#### Złożoność obliczeniowa

Określmy złożoność algorytmu w postaci:

Gdzie:

* n – liczba kolorów
* a,b – stałe
* – czas rozwiązywania problemu o rozmiarze n

Powyższe równanie możemy oszacować jako:

#### Złożoność pamięciowa

Ponieważ algorytm pełnego przeglądu jest algorytmem rekurencyjnym, za każdym wywołaniem funkcji SOLVE\_RECURSIVE musimy zaalokować pamięć na argumenty wywołania. Zwiększa się także głębokość stosu. Całkowita pamięć potrzebna na alokację stosu wynosi O(n), natomiast pamięć niezbędna do zarezerwowania dla argumentów jest równa , zatem złożoność pamięciowa algorytmu wynosi .

#### Przypadek złośliwy

Ponieważ, za każdym razem rozpatrujemy wszystkie rozwiązania przypadki złośliwe nie istnieją.

## Uruchamianie

### Wejście.

Na wejściu naszego programu podajemy:

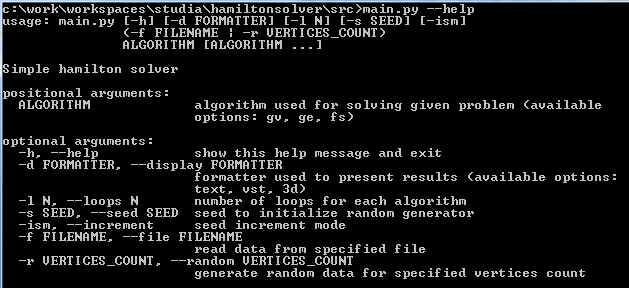
* plik w formacie testowym zawierający informacje o ilości kolorów oraz koszcie przestawienia lakierni z jednego koloru na drugi. Plik ma taką strukturę, gdzie kolejne wiersze i kolumny oznaczają kolor a wartość w tabeli koszt przestawienia lakierni.   
  0 1 3 4

1 0 3 4

3 2 0 1

2 2 1 0

* wykorzystywany algorytm. Możliwy wybór: ge, gv, fs
* formater uzyskanych wyników. Możliwy wybór: text, 3d, simple\_text
* ilość powtórzeń jaką ma się wykonać każdy algorytm
* ziarno który posłuży do zainicjowania generatora pseudolosowego
* opcjonalnie flagę oznaczającą uruchomienie trybu w którym ziarno jest zmieniane z każdym uruchomieniem pętli



### Wyjście.

Na wyjściu możemy otrzymać wyniki w trzech formatach:

* tekstowym, w formacie:  
  *[Algorithm] ([loops] - [avg time]: [min time] : [max time]  
  [Solution cost]  
  [Solution path]*
* w postaci wykresu 3d
* w prostej tekstowej postaci, format ten jest używany do późniejszego przetwarzania wyników

## Rozwiązanie przykładowego zadania optymalizacyjnego

Korzystając z naszego programu rozwiążmy problem rozmiaru 10 dla kosztów danych tabelą:

['x', 8.4, 7.6, 2.6, 5.0, 4.5, 6.5, 7.9, 1.0, 0.3]

[8.3, 'x', 7.6, 0.1, 4.5, 7.2, 2.3, 9.4, 9.0, 0.4]

[0.3, 5.4, 'x', 3.8, 2.2, 4.2, 0.3, 2.2, 4.4, 5.0]

[2.4, 2.3, 2.2, 'x', 2.9, 0.3, 8.3, 5.6, 6.4, 1.9]

[9.9, 8.6, 1.2, 3.3, 'x', 7.1, 9.3, 4.2, 8.3, 6.7]

[3.1, 5.9, 8.8, 8.4, 5.1, 'x', 0.4, 2.5, 7.9, 4.2]

[1.8, 5.5, 7.0, 6.7, 3.8, 4.4, 'x', 7.8, 5.2, 3.9]

[4.9, 0.3, 0.5, 7.0, 9.8, 5.9, 3.9, 'x', 5.0, 9.8]

[7.7, 5.4, 8.6, 2.3, 5.1, 9.5, 5.8, 4.6, 'x', 5.5]

[9.5, 0.1, 7.8, 8.2, 8.8, 7.4, 8.1, 5.2, 5.6, 'x']

Otrzymane rozwiązania (uśrednione dla 10 pętli):

|  |  |  |  |
| --- | --- | --- | --- |
|  | Greedy Edges | Greedy Vertices | Full Search |
| Rozwiązanie problemu | [8,7,2,0,9,1,3,5,6,4] | [7,1,3,5,6,0,9,8,4,2] | [0,8,9,1,3,5,6,4,7,2] |
| Czas rozwiązania problemu [ms] | 0.115262768189 | 0.573561398662 | 25997.1340619 |
| Koszt rozwiązania | 18.7 | 17.3 | 16.2 |
| O ile gorsze od optymalnego | 15,4% | 10% | 0% |

Analizując powyższą tabelę możemy dojść do następujących wniosków:

* Najlepsze rozwiązanie otrzymaliśmy przy wykorzystaniu metody Full Search. Jest to rozwiązanie optymalne. Dojście do tego rozwiązanie trwało jednak dziesiątki tysięcy razy dłużej niż w przypadku pozostałych metod.
* Drugie w kolejności pod względem optymalności było rozwiązanie otrzymane za pomocą algorytmu Greedy Vertices. Było o 10% gorsze od rozwiązania optymalnego.
* Najgorsze pod względem optymalności, ale za to najszybciej obliczone, jest rozwiązanie uzyskane przy użyciu algorytmu Greedy Edges.

### Rozwiązanie przedstawione w postaci 3d

#### 

Rysunek : Greedy Edges

#### 

Rysunek : Greedy Vertices

#### 

Rysunek : Full Search

## Testy

Platforma testowa

Wszystkie badania zostały wykonane na komputerze klasy PC o następującej konfiguracji:  
Procesor: AMD Athlon II X4 640 3.00Ghz  
Pamięć: 4.0 GB  
System: Windows 7 Professional 64bit  
Do badania została użyta aplikacja napisana w języku Python.

Testy zostały przeprowadzone badając 10 różnych przypadków (10 różnych ziaren generatora liczb losowych) dla każdej liczby kolorów karoserii. Następnie wyniki scalono i uśredniono.

### Porównanie jakości rozwiązań dla badanych algorytmów



Analizując powyższą tabelę możemy dojść do następujących wniosków:

* Najbardziej optymalne rozwiązania daje zawsze algorytm Full Search, lecz ze względu na jego złożoność nie jesteśmy w stanie osiągnąć wyników w zadowalającym czasie dla problemu o rozmiarze większym niż 11 kolorów.
* Z algorytmów heurystycznych najbardziej optymalne rozwiązanie dla wszystkich przypadków generuje algorytm Grendy Vertices.

### Badanie czasu rozwiązania w zależności od liczby kolorów karoserii



Rysunek : Wyniki w postaci wykresu w skali logarytmicznej

Rysunek : Wynik w postaci wykresu w skali liniowej

### Wnioski

Analizując dane zebrane w tabeli oraz wykresy możemy dojść do następujących wniosków:

* Dla wyszukiwania Full Search nie jesteśmy uzyskać wyników w zadowalającym czasie już dla problemów większych niż 11 kolorów
* Potwierdziły się teoretyczne oszacowane złożoności algorytmów (n liczba kolorów):
  + Algorytm Full Search jest bardzo wolny (**złożoność n!)**
  + Czas rozwiązania przy użyciu algorytmu Greedy Edges rośnie proporcjonalnie do
  + Czas rozwiązania zadania przy użyciu algorytmu „Greedy Vertices” rośnie proporcjonalnie do

## Wnioski końcowe

Z przeprowadzonego przez nas eksperymentu wynika, że z pośród badanych algorytmów nie możemy wybrać jednego „najlepszego”. Wszystko zależy od kilku czynników: jak duży problem chcemy rozwiązać, jak bliskich optymalnemu rozwiązań oczekujemy, jakimi mocami obliczeniowymi i czasem na rozwiązanie zadania dysponujemy. Dla małych problemów wybierzemy FullSearch dla większych GreedyVertices, a dla ogromnych Greedy Edges.

## Wykaz literatury.

Z.Michalewicz, D. Fogel, Jak to rozwiązać, czyli nowoczesna heurystyka