Grafy i Rekurencje

Janusz Marecki

Spis treści

Rozdział I Koncepcja grafu ***7***

I. 1 Czym jest graf **7**

I. 2 Relacje związane z grafem **8**

I. 3 Wierzchołki grafu **9**

I. 4 Pod-grafy **10**

I. 5 Orientacja grafu **11**

I. 6 Szczególne przypadki grafów **13**

I. 7 Izomorfizm grafów **15**

Rozdział II Spójność grafu ***16***

II. 1 Łańcuchy w grafie **16**

II. 2 Grafy spójne **17**

II. 3 Drzewa **21**

II. 4 Drzewa właściwe **25**

II. 5 Grafy silnie spójne **26**

II. 6 Porządek grafu **28**

II. 7 Rozkład grafu **29**

Rozdział III Algorytmy grafowe ***31***

III. 1 Przeszukiwanie grafu **31**

III. 2 Klasy silnie spójne **33**

III. 3 Maksymalne skojarzenia **36**

III. 4 Najkrótsze ścieżki z jednym źródłem **39**

III. 5 Najkrótsze ścieżki z wieloma źródłami **44**

III. 6 Drzewa rozpinające **47**

Rozdział IV Algorytmy numeracji ***50***

IV. 1 Słowa **50**

IV. 2 Permutacje **51**

IV. 3 Kod Gray'a **54**

Rozdział V Funkcje rekursywne ***58***

V. 1 Funkcje rekursywnie prymitywne **58**

V. 2 Konstrukcja funkcji rekursywnie prymitywnych **60**

V. 3 Kodowanie ciągów **63**

V. 4 Rekurencja nie prymitywna **66**

V. 5 Rodzaje funkcje rekursywnych **67**

V. 6 Teza Church'a **68**

Rozdział VI Rekursywność i obliczalność ***69***

VI. 1 Funkcje rekursywne i Turing – obliczalne **69**
VI. 2 Rekursywność funkcji Turing – obliczalnych **70**
VI. 3 Abstrakcja funkcjonalna **73**
VI. 4 Język λ – term **76**
VI. 5 Obliczenia na λ – termach **77**
Rozdział VII Równania rekurencyjne ***81***
VII. 1 Zastosowanie równań rekurencyjnych **81**
VII. 2 Równania rekurencyjne liniowe **82**
VII. 3 Rekurencyjne przepołowienie **84**
VII. 4 Równania rekurencyjne liniowe dowolnego stopnia **86**
Rozdział VIII Przykładowe procesy rekurencyjne ***90***
VIII. 1 Stan rejestru przesuwającego **90**
VIII. 2 Stan procesu montażu **92**
VIII. 3 Fundusz emerytalny **95**
VIII. 4 Fundusz amortyzacji **96**
VIII. 5 Kredyty ratalne **97**

Spis rysunków

Rysunek 1 Przykładowy graf 7
Rysunek 2 Grafy kompletne stopnia 1, 2, 3, 4 10
Rysunek 3 Grafy wygenerowane przez zbiór wierzchołków i krawędzi 11
Rysunek 4 Przykładowy graf zorientowany 12
Rysunek 5 Grafy: planarny i nieplanarny 14
Rysunek 6 Graf dwuczęściowy przedstawiony na dwa różne sposoby 14
Rysunek 7 Graf K 3,3 nie będący grafem planarnym 15
Rysunek 8 Dwa grafy izomorficzne 15
Rysunek 9 Graf nieskierowany posiadający łańcuch 16
Rysunek 10 Punkt artykulacji w grafie 18
Rysunek 11 Graf podatny na lemat o grafach spójnych 18
Rysunek 12 Cykl i łańcuch w grafie 20
Rysunek 13 Graf, las, drzewa 21
Rysunek 14 Drzewo rozpinające graf 23
Rysunek 15 Przodkowie, potomkowie, koła i korzenie 24
Rysunek 16 Graf podstawowy i graf zredukowany 27
Rysunek 17 Graf i jego domknięcie przechodnie 27
Rysunek 18 Rozkład grafu 30
Rysunek 19 Las oparty na grafie G 33
Rysunek 20 Proces wyznaczania komponentów silnie spójnych 36
Rysunek 21 Graf dwudzielny i graf skojarzeń 37
Rysunek 22 Graf przepływów i ścieżka powiększająca 39
Rysunek 23 Rezultat działania algorytmu Forda – Fulkersona 39
Rysunek 24 Znajdowanie wierzchołka minimalnego 40
Rysunek 25 Działanie algorytmu Dijkstry 41
Rysunek 26 Niewykonalność algorytmu Dijkstry 42
Rysunek 27 Graf dla algorytmu Floyda - Warshalla 45
Rysunek 28 Podział ścieżki dla algorytmu Floyda – Warshalla 47
Rysunek 29 Numerowanie permutacji 53
Rysunek 30 Drzewo wywołań rekurencyjnych przy generowaniu permutacji 53
Rysunek 31 Kod Gray'a i cykl Hamiltona 54
Rysunek 32 Uporządkowane drzewo podzbiorów 55
Rysunek 33 Podzbiory i cykl Hamiltona 56
Rysunek 34 Pokrycie płaszczyzny kolejnymi liczbami naturalnymi 64
Rysunek 35 Kodowanie maszyny Turinga 71
Rysunek 36 Proces arytmetyzacji 73
Rysunek 37 Drzewa reprezentujące termy 78
Rysunek 38 Zmienne wolne i połączone 79
Rysunek 39 Zmiana nazw zmiennych w drzewie 79
Rysunek 40 Przerzutnik danych 90
Rysunek 41 Rejestr przesuwający 90
Rysunek 42 Przykładowa modyfikacja rejestru przesuwającego 91
Rysunek 43 Linia montażowa 92

Słowo wstępne

Na początku chciałbym podziękować wszystkim czytelnikom, którzy zdecydowali się poszerzać swoją wiedzę w dziedzinie grafów i rekurencji przy wykorzystaniu tego właśnie skryptu. Na rynku jest wiele doskonałych pozycji, które poruszają przedstawioną tematykę, jednak są one często bardzo obszerne i rzadko ograniczone tylko do grafów i rekurencji.

Zawarta w tym skrypcie teoria grafów jak również wnikliwe wyjaśnienie istoty rekurencji pozwala na zrozumienie całości materiału bez konieczności korzystania z innych pozycji książkowych z dziedziny informatyki.

Czym są „Grafy i rekurencje”?

Świat, w którym się znajdujemy, pomimo swojej złożoności i niezwykłości da się jednak przedstawić na wiele interesujących sposobów. Wprowadzając bardzo ogólny podział można doszukać się w nim:

Obiektów
Własności
Relacji

Ten powyższy podział może być z powodzeniem reprezentowany przez odpowiednie grafy, stąd tak wielkie znaczenie tej struktury danych w informatyce. Każdy świat nieustannie się jednak zmienia, stąd by był on dobrze reprezentowany przez grafy, one same muszą podlegać zmianom. Często zmiany te są wynikiem działania algorytmów.

Za algorytm w dużym uproszczeniu możemy uważać „przepis informatyczny”, który na podstawie danych wejściowych zwraca nam dane wyjściowe będące rezultatami obliczeń. Możemy sobie wyobrazić graf, którego wierzchołki będą zbiorami danych, natomiast łączące je krawędzie – algorytmami. Taki graf będzie wycinkiem rzeczywistości mówiącym nam, że grafy i algorytmy są ze sobą ściśle powiązane.

Gdy grafy zawierają pętle, zaczynamy mówić o rekurencji.

Algorytmy rekurencyjne często najlepiej odzwierciedlają trudne do analizy problemy, stąd ich popularność w środowisku programistów. Mimo iż na pierwszy rzut oka trudno oszacować ich efektywność i poprawność, istnieją odpowiednie metody, które pozwalają nam to zrobić.

Dla kogo jest ten skrypt?

Skrypt „Grafy i rekurencje” jest przeznaczony dla studentów informatyki, którzy zaczynają poznawać algorytmy komputerowe i sposoby ich zastosowania. Stanowi on doskonałe podłoże przygotowawcze do innych przedmiotów takich jak badania operacyjne, bazy danych, sieci komputerowe, analiza algorytmów, zagadnienia sztucznej inteligencji itp.

Czytelnik pracujący z tym skryptem powinien jednak posiadać znajomość przynajmniej jednego języka programowania, w celu zrozumienia działania algorytmów i umiejętności przetwarzania jego rezultatów. Algorytmy przedstawione są najczęściej w pseudo języku informatycznym, który bazuje niekiedy na bardziej skomplikowanych strukturach danych, stąd wskazane jest posiadanie przez czytelnika gotowych bibliotek z zaimplementowanymi podstawowymi strukturami danych.

Jak czytać ten skrypt?

Tak jak wspomniano na początku, by zrozumieć przedstawione w skrypcie zagadnienia, nie ma konieczności korzystania z innych książek informatycznych. Warto jednak zapoznać się z rozdziałem I, który stanowi wprowadzenie do tematyki grafów.

Czytanie rozdziału III, w którym zaprezentowane zostały podstawowe algorytmy operujące na grafach powinno być poprzedzone zaznajomieniem się z rozdziałem II, który omawia spójność grafów.

Rozdział IV może być czytany oddzielnie, pod warunkiem, że czytelnik potrafi budować i analizować podstawowe algorytmy rekurencyjne. W przeciwnym wypadku zaleca się zapoznanie z rozdziałem V.

W rozdziale V poświęconym funkcjom rekursywnym znajdujemy genezę dzisiejszych algorytmów, czyli najbardziej prymitywne funkcje informatyczne, które są wykonywane bezpośrednio przez procesor. Rozdział ten może być czytany oddzielnie.

Rozdział VI skierowany jest głównie do czytelników, którzy dobrze opanowali już sztukę budowania algorytmów rekurencyjnych i zależy im na pogłębieniu teoretycznej wiedzy z tej dziedziny.

Rozdział VII, poświęcony równaniom rekurencyjnym może być czytany oddzielnie, jednak zalecana jest podstawowa znajomość aparatu matematycznego pod kątem wielomianów i szeregów potęgowych.

W rozdziale VIII przedstawione zostały problemy rekurencyjne, z którymi możemy się spotkać na co dzień, dlatego może od być czytany oddzielnie.

Rozdział I Koncepcja grafu

I. 1 Czym jest graf

Jeśli mamy do czynienia ze schematami obrazującymi różnorodne zjawiska i poszukujemy metody prezentacji tych schematów, wykorzystanie grafów okazuje się niezastąpione. Otaczający nas świat często da się podzielić na obiekty i na zachodzące pomiędzy nimi zależności, co dokładnie odpowiada wierzchołkom grafu, oraz połączeniom między wierzchołkami.

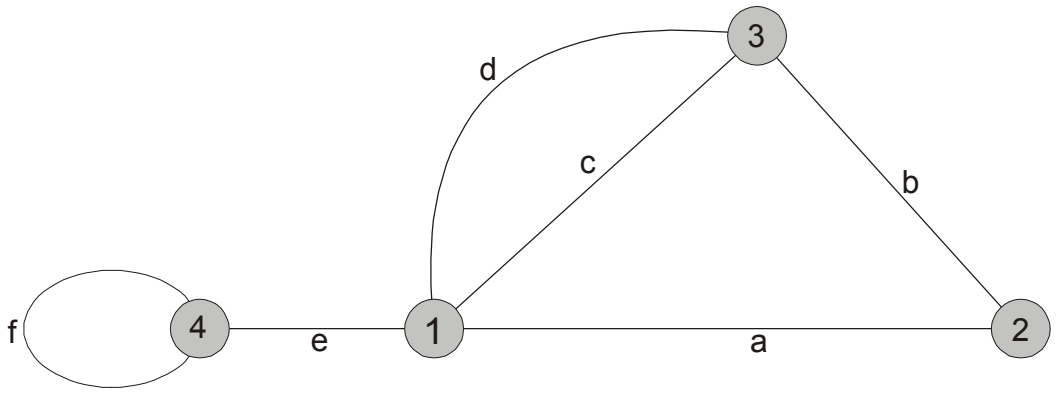
Każdy graf G jest parą (X, E) dwóch rozłącznych zbiorów skończonych: $X=\{x_1, x_2, ..., x_n\}$ gdzie $n > 0$ oraz $E=\{e_1, e_2, ..., e_m\}$ gdzie $m > 0$, przy czym dla każdego i e_i jest parą elementów ze zbioru X. Zbiór X nazywamy *zbiorem wierzchołków*, natomiast E *zbiorem krawędzi*. *Stopień grafu* G jest ilością jego wierzchołków.

Aby uniknąć nieporozumień, będziemy używali oznaczeń X(G)oraz E(X), by oznaczyć odpowiednio zbiór wierzchołków oraz zbiór wierzchołków grafu G. W niektórych publikacjach można też spotkać oznaczenia n(G) i m(G).

Przykład 1

Rozpatrzmy zbiór X={1, 2, 3, 4} oraz E={a, b, c, d} gdzie: a={1,2}, b={2,3}, c={3,1}, d={1,3}, e={1,4}, f={4,4}.

Każda krawędź (tutaj mała litera alfabetu} jest parą dwóch wierzchołków. Tak zdefiniowany graf obrazuje rysunek 1.



Rysunek 1 Przykładowy graf

Gdy mamy daną krawędź e={x, y} lub stosując inną notację: e=xy, mówi się, że x i y są *krańcami* e. Ponadto stosuje się oznaczenie: e łączy x i y, lub e jest krawędzią o krańcach x i y. Wtedy, gdy x=y mówi się, że e jest *pętlą*.

Dwie krawędzie są zwane *podobnymi*, gdy mają takie same krańce. Zbiór krawędzi podobnych nazywany jest *krawędzią powtórzoną*. W powyższym przykładzie f jest cyklem, natomiast d i c są krawędziami podobnymi.

DEFINICJA Graf nazywamy prostym, gdy nie posiada pętli oraz zbioru krawędzi powtórzonych.

W informatyce największe zastosowanie ma analiza grafów prostych.

I. 2 Relacje związane z grafem

Wierzchołek $x \in X$ oraz krawędź $e \in E$ są nazywane *incydentnymi* jeśli x jest krańcem e, możemy zatem zdefiniować relację X nad E, nazywaną *relacją incydencji*. Można również zamienić kolejność i rozważać relację E nad X.

Dwa wierzchołki nazywamy sąsiadującymi, gdy są incydentne z tą samą krawędzią. W ten sposób definiuje się relację na zbiorze X^2 nazywaną *relacją sąsiedztwa*.

Zbiór wierzchołków sąsiadujących z x jest nazywany zbiorem sąsiadów x i oznaczany przez N(x).

Wprowadzenie relacji incydencji oraz relacji sąsiedztwa sugeruje sposób przedstawienia grafu. W rzeczywistości próby narysowania na kartce papieru grafu dużych rozmiarów mogłyby zakończyć się niepowodzeniem. Gdy graf jest zdefiniowany przy użyciu jednej z przedstawionych relacji możemy go reprezentować przez macierz odpowiadającą wybranej relacji.

- Macierz sąsiedztwa

Z każdym grafem G stopnia n utożsamia się macierz n x n, $A(G) = [a_{ij}]$, której wiersze i kolumny są indeksowane przez kolejne wierzchołki.

a_{ij} = ilość krawędzi o krańcach x_i i x_j.

Zauważamy łatwo, że tak zdefiniowana macierz jest symetryczna, ponadto gdy graf jest prosty, wówczas na diagonali macierzy znajdują się 0, a pozostałe elementy należą do zbioru {0,1}.

- Macierz incydencji

Z każdym grafem G posiadającym n wierzchołków i m krawędzi utożsamiamy macierz n x m, B(G) = [b_{ij}] , gdzie wiersze indeksowane są przez kolejne wierzchołki ze zbioru X, natomiast kolumny, krawędziami ze zbioru E.

b_{ij} = ilość przypadków, gdy x_i jest incydentny z b_j.

Widzimy, że $b_{ij} \in \{0, 1, 2\}$ i że suma elementów w jednej kolumnie wynosi 2.

Przykład 2

Z grafem z przykładu 1 utożsamiamy następujące tabele:

	1	2	3	4
1	0	1	2	1
2	1	0	1	0
3	2	1	0	0
4	1	0	0	1

A(G)

	a	B	c	d	e	f
1	1	0	1	1	1	0
2	1	1	0	0	0	0
3	0	1	1	1	0	0
4	0	0	0	0	1	2

B(G)

W formie macierzowej, indeksując w porządku naturalnym wiersze przez 1,2,3,4 oraz kolumny przez a, b, c, d, e, f otrzymujemy:

$$A(G) = \begin{pmatrix} 0 & 1 & 2 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 \end{pmatrix} \qquad B(G) = \begin{pmatrix} 1 & 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 2 \end{pmatrix}$$

I. 3 Wierzchołki grafu

DEFINICJA Stopniem wierzchołka x, d(x) nazywamy ilość krawędzi incydentnych z x. Pętla występująca w wierzchołku liczona podwójnie

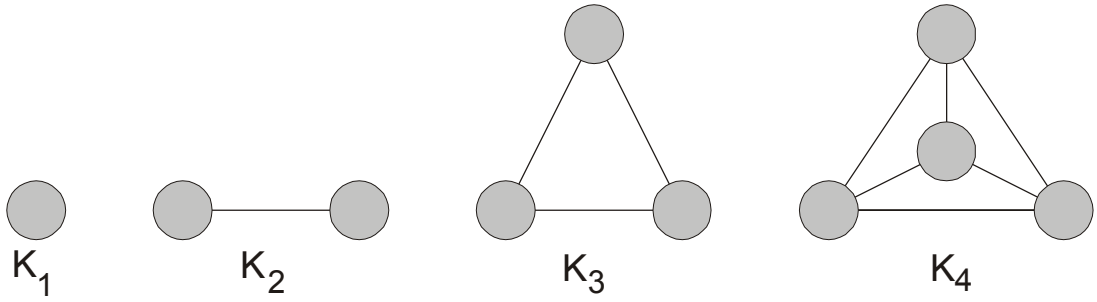
Ponadto oznaczamy:

$\delta(G) = \min \{d(x) \mid x \in X\}$

$\Delta(G) = \text{Max } \{d(x) \mid x \in X\}$

W przypadku, gdy $\delta(G) = \Delta(G) = k$, graf jest nazywany k - *regularnym*. Zauważmy, że gdy G jest grafem prostym, wówczas $\Delta(G) \leq n-1$ oraz dla każdego x, $|N(x)| = d(x)$.

Graf prosty, którego wszystkie wierzchołki są stopnia n-1 jest nazywany *kompletnym*. Nazywamy go K_n. W konsekwencji każde dwa wierzchołki K_n są sąsiadujące. Mamy więc: $K_n = (X, P_2(X))$ gdzie $|X| = n$. K_3 jest nazywany trójkątem. Na rysunku nr 2 przedstawione są grafy kompletne stopnia ≤ 4:



Rysunek 2 Grafy kompletne stopnia 1, 2, 3, 4

Wierzchołek x , taki że $d(x) = 0$ nazywamy *wolnym*. Wierzchołek x jest *doczepiony*, gdy $d(x) = 1$. Krawędzią *doczepioną* nazywamy krawędź incydentną z wierzchołkiem krańcowym.

DEFINICJA Dopełnieniem grafu $G = (X, E)$ nazywamy graf $G' = (X, P_2(X) - E)$.

I. 4 Pod-grafy

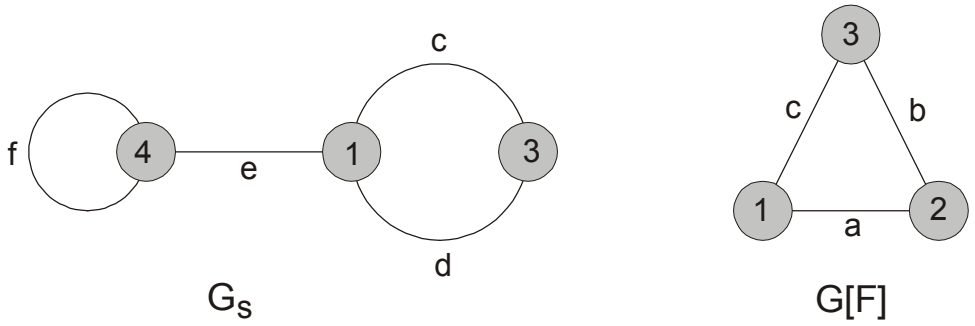
W grafie $G = (X, E)$ wyróżnimy dwa podzbiory: $Y \subseteq X$ oraz $F \subseteq E$. Mówimy, że $H = (Y, F)$ jest pod-grafem G, gdy $\forall\, e \in F$ krańce e są w Y.

Gdy F jest utworzony z wszystkich krawędzi ze zbioru E, których krańce są w Y, otrzymany graf jest nazywany *pod-grafem wygenerowanym przez Y*. Nazywamy go G_y, gdyż jest on jednoznacznie zdefiniowany przez Y.

W podobny sposób pod-grafem wygenerowanym przez F, oznaczanym G[F], nazywamy pod-graf G, którego zbiorem krawędzi jest F, a którego zbiór wierzchołków jest utworzony z krańców krawędzi należących do F.

Przykład 3

Rozpatrzmy graf pokazany na przykładzie 1. Na rysunku 3 przedstawione zostały pod-grafy wygenerowane przez S={1,3,4} oraz F={a, b, c}:



Rysunek 3 Grafy wygenerowane przez zbiór wierzchołków i krawędzi

DEFINICJA Kliką nazywamy graf posiadający taki zbiór wierzchołków, z których każde dwa ze sobą sąsiadują.

W grafie prostym, klika generuje zawsze graf kompletny. Graf przeciwny do kliki definiuje się następująco:

DEFINICJA Klatką nazywamy Graf, którego żadne dwa wierzchołki nie są ze sobą połączone:

Inaczej mówiąc, klatka jest generowana przez pod-graf który nie posiada krawędzi. W szczególności każdy wierzchołek klatki nie posiada pętli.

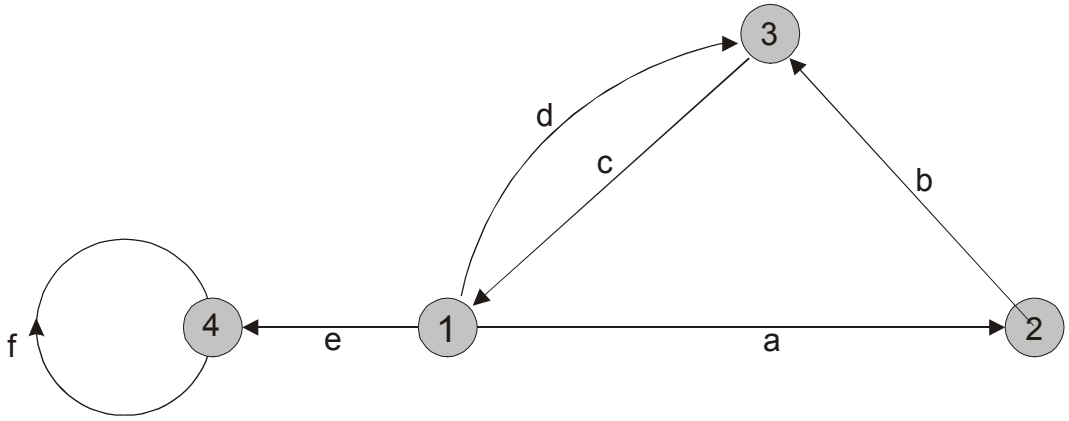
I. 5 Orientacja grafu

Graf G = (X, E) jest *zorientowany*, gdy każda krawędź e∈E jest *uporządkowanym* zbiorem dwóch wierzchołków.

Można również powiedzieć, że orientacja grafu zamienia każdą krawędź w *łuk* , który jest parą wierzchołków. Pierwszym elementem tej pary jest *wierzchołek początkowy*, a drugim *wierzchołek końcowy*. Każdy łuk oznacza się przez (x, y) lub xy.

Przykład 4

Dodajmy orientację do grafu z przykładu 1. Przypomnijmy, że X={1,2,3,4}, E = {a, b, c, d, e, f}, gdzie: a=(1,2), b=(2,3), c=(3,1), d=(1,3), e=(1,4), f= (4,4). Ten zorientowany graf pokazany jest na rysunku nr 4.



Rysunek 4 Przykładowy graf zorientowany

Graf zorientowany bez cyklu nazywamy prostym, gdy pomiędzy dwoma dowolnymi wierzchołkami występuje co najwyżej jeden łuk.

Niech będzie dany wierzchołek x. Mówimy, że y jest następnikiem (poprzednikiem) x, gdy xy (yx) jest łukiem. Zbiory: następników oraz poprzedników x są odpowiednio zapisywane: $N^+(x)$ i $N^-(x)$. Oczywistym jest, że $N(x) = N^+(x) \cup N^-(x)$.

Relacja incydencji oraz sąsiedztwa jest tutaj zdefiniowana tak samo jak w przypadku grafów niezorientowanych. Jedyna różnica występuje w macierzy incydencji, które elementy przyjmują wartości:

$$b_{ij} = \begin{cases} 1 & gdy\, x_i \ \ jest\ wierzcholkiem\ poczatkowym\ a_j \\ -1 & gdy\, x_i \ \ jest\ wierzcholkiem\ koncowym\ a_j \\ 0 & w\ innych\ przypadkach \end{cases}$$

Tabela sąsiedztwa i incydencji grafu zorientowanego z rysunku nr 4 przedstawia się następująco:

	1	2	3	4
1	0	1	1	1
2	0	0	1	0
3	1	0	0	0
4	0	0	0	1

A(G)

	a	b	c	d	E	f
1	1	0	-1	1	1	0
2	-1	1	0	0	0	0
3	0	-1	1	-1	0	0
4	0	0	0	0	-1	0

B(G)

I. 6 Szczególne przypadki grafów

W wielu algorytmach opartych na grafach wykorzystuje się ich szczególne przypadki, które są zawsze zawężeniem klasy wszystkich grafów do pewnego niekoniecznie skończonego podzbioru. Są to np.:

I. 6. 1 Grafy bez pętli

Są to grafy G=(E,V), dla których prawdziwa jest reguła: $\forall x \in E\ (x,x) \notin V$. W reprezentacji macierzowej, grafy te będą miały na diagonali zera.

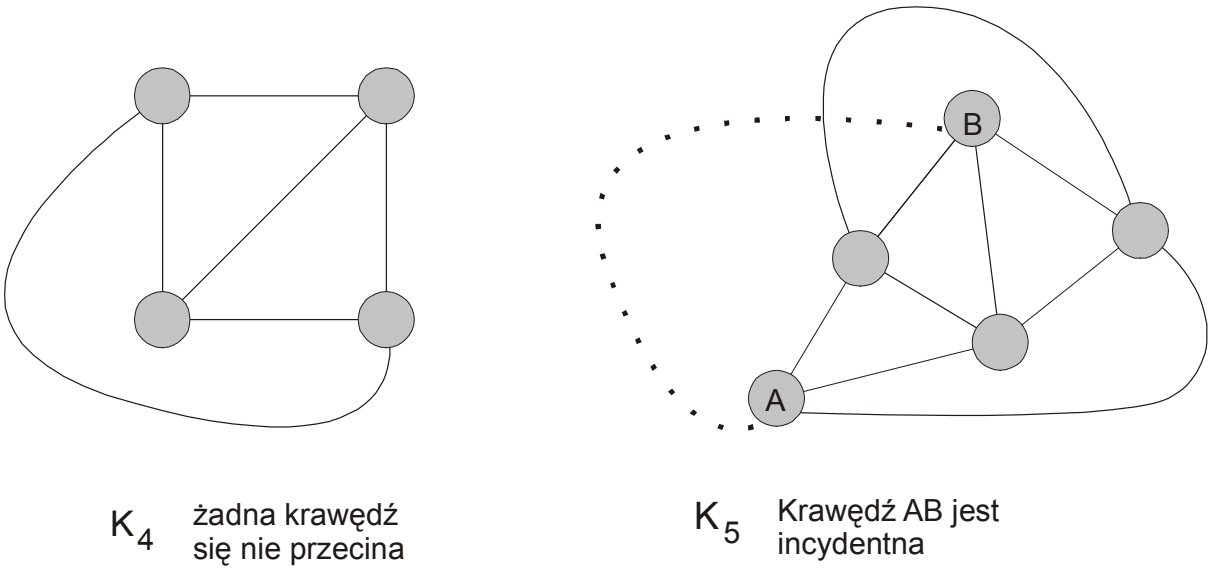
Przykład grafu z pętlą przedstawia rysunek nr 1, natomiast na rysunku nr 2 przedstawione zostały grafy bez pętli.

I. 6. 2 Grafy kompletne

Są to takie grafy, w których dla każdego wierzchołka istnieje krawędź (bądź łuk w grafach skierowanych) łącząca go z wszystkimi pozostałymi wierzchołkami. Przykłady tych grafów przedstawia rysunek nr 2.

I. 6. 3 Grafy planarne

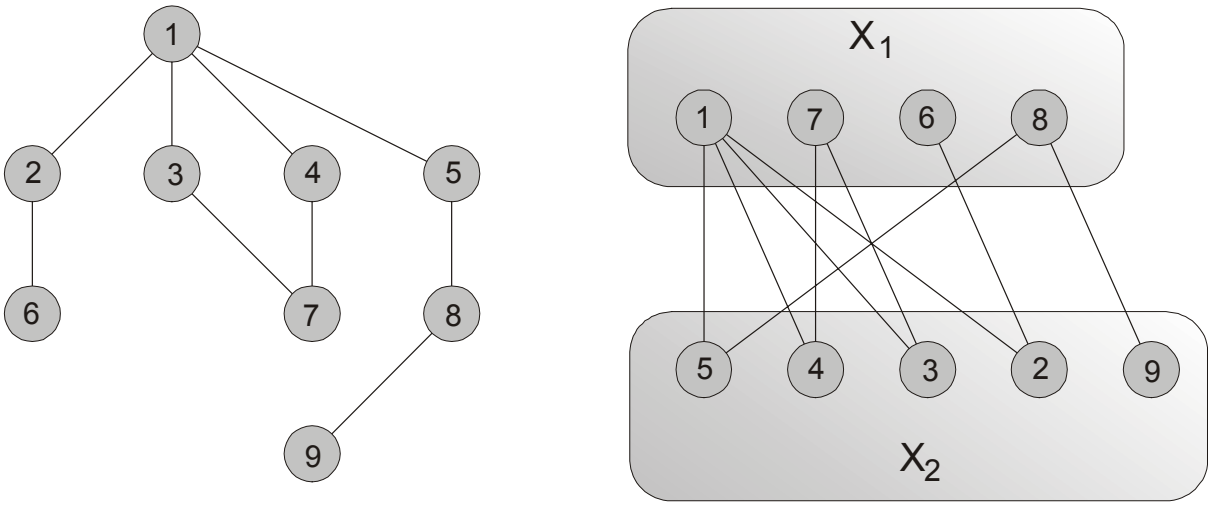
Grafem planarnym nazywa się graf, który może być narysowany na płaszczyźnie dwuwymiarowej w taki sposób, by dwie dowolne krawędzie nie przecinały się ze sobą. Łatwo zobaczyć, że graf K_4 jest grafem planarnym, natomiast graf dla grafu K_5 nie da się tak rozmieścić krawędzi, by uniknąć choćby jednego ich przecięcia. Sytuację tą przedstawia rysunek nr 5.



Rysunek 5 Grafy: planarny i nieplanarny

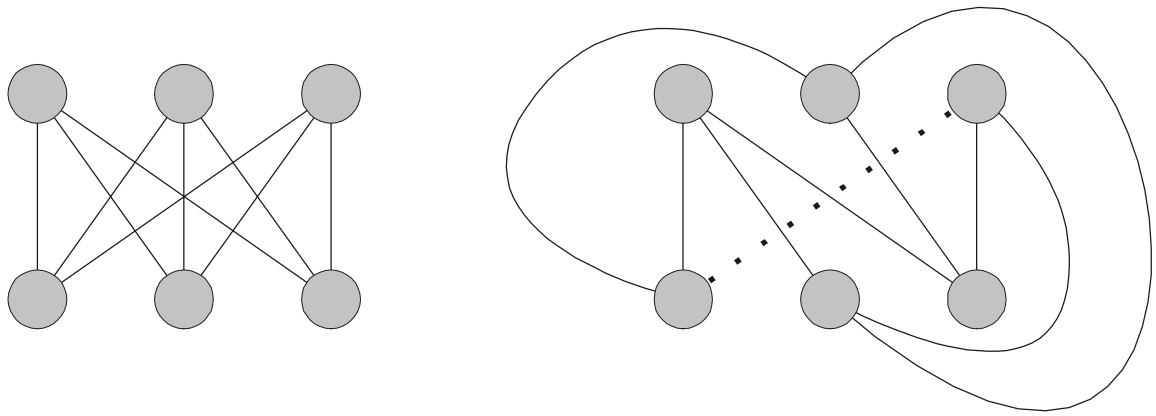
I. 6. 4 Grafy dwuczęściowe

Grafem dwuczęściowym (bipart) G=(E,V) nazywany taki graf, którego zbiór wierzchołków E da się podzielić na dwa rozłączne podzbiory X_1, X_2, tak by $X_1 \cup X_2 = E$ oraz by istniały tylko krawędzie łączące 1 wierzchołek z X1 z jednym wierzchołkiem z X2. Ponadto musi zostać zachowana poprzednia struktura grafu, tzn. jeśli istniała krawędź (x,y) to graf dwuczęściowy także musi ją posiadać. Rysunek nr 6 przedstawia ten sam graf dwuczęściowy, przedstawiony na dwa różne sposoby:



Rysunek 6 Graf dwuczęściowy przedstawiony na dwa różne sposoby

Łącząc ze sobą grafy kompletne i dwuczęściowe, otrzymujemy graf $K_{p,q}$, w którym $|X_1| = p$, $|X_2| = q$. Na rysunku nr 7 przedstawiony jest graf $K_{3,3}$, który zarazem nie może być grafem planarnym.

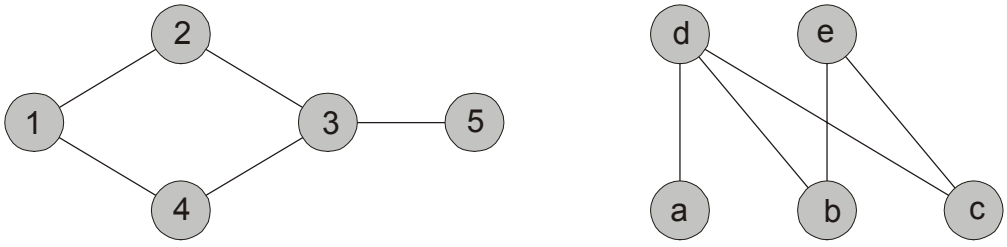


Rysunek 7 Graf K 3,3 nie będący grafem planarnym

I. 7 Izomorfizm grafów

Dwa grafy G=(X,V) oraz H=(Y,W) są izomorficzne jeśli istnieje funkcja wzajemnie jednoznaczna „f", której dziedziną jest zbiór X, a przeciwdziedziną zbiór Y, taka że: $(x,y) \in V \Leftrightarrow (x,y) \in W$. Dwa grafy izomorficzne przedstawione są na rysunku nr 9. Funkcja „f" ma postać:

X	1	2	3	4	5
F(X)	e	b	d	c	a



Rysunek 8 Dwa grafy izomorficzne

Rozdział II Spójność grafu

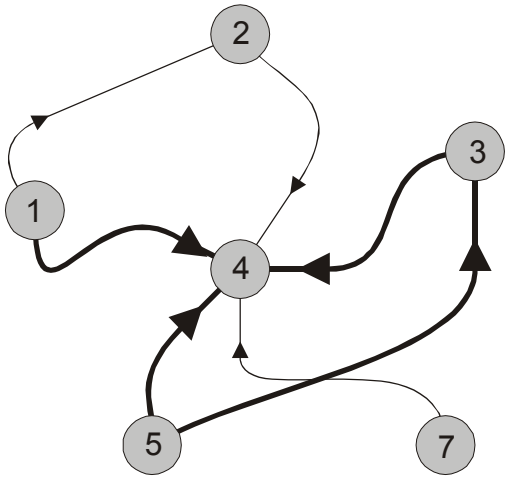
II. 1 Łańcuchy w grafie

DEFINICJA Łańcuch w grafie G=(X,V) jest ciągiem $c=(x_0, x_1, ..., x_n)$ wierzchołków grafu G, dla których zachodzi zależność:

$$\forall i \ (1 \le i \le n) \Rightarrow [\ (x_{i-1}x_i) \in V \text{ lub } (x_i x_{i-1}) \in V)\ ]$$

gdzie x_0, x_n są krańcami łańcucha, a „n" jego długością.

Nie należy mylić ze sobą pojęć: droga ≠ łańcuch, gdyż w łańcuchu mogą wystąpić łuki w dwóch różnych kierunkach, natomiast w drodze poszczególne łuki mają jeden kierunek. Na rysunku nr 9 przedstawiony jest przykładowy graf i występujący w nim łańcuch c=(4,3,5,4,1).



Rysunek 9 Graf nieskierowany posiadający łańcuch

DEFINICJA Łańcuch nazywamy elementarnym, jeśli każdy wierzchołek wchodzący w jego skład jest inny.

Łańcuch złożony z jednego wierzchołka ma długość 0.

Patrząc na rysunek nr 9 możemy wskazać łańcuch elementarnym np. c=(1,4,5).

Własność 1

Z każdego łańcucha xy można wyodrębnić łańcuch, który jest elementarny.

Dowód: Indukcyjny ze względu na ilość powtórzeń w łańcuchu:
0 powtórzeń, wówczas badany łańcuch jest elementarny.
Założenie: Własność prawdziwa dla „n" powtórzeń

Teza : Własność prawdziwa dla „n+1” powtórzeń. Weźmy dowolny łańcuch c. Podążając po kolejnych wierzchołkach tego łańcucha napotykamy powtarzające się wierzchołki. Gdy napotkamy już „n” powtórzeń (w całym łańcuchu jest ich n+1), wówczas z założenia indukcyjnego da się wyodrębnić podłańcuch. Będzie to zarazem podłańcuch łańcucha wyjściowego o n+1 powtórzeniach.

Własność 2

Najkrótszy łańcuch grafu G jest elementarny

Dowód: Załóżmy, że xy jest najkrótszym łańcuchem grafu G i posiada powtarzający się element v. Wtedy łańcuch ten będzie posiadał postać: (x,.....,v,.....,v,....t). Usuwając zaciemnioną część otrzymamy krótszy łańcuch – sprzeczność.

II. 2 Grafy spójne

Mamy dany graf G=(E,V). Zdefiniujmy relację $R_c \subseteq (E \times E)$,która będzie relacją spójności w grafie.

$x R_c y \Leftrightarrow$ w grafie G istnieje łańcuch xy.

Relacja R_c jest relacją równoważności, gdyż:

Zwrotność - $\forall x \in E$ (x) jest łańcuchem o długości 0

Symetria - $\forall x,y \in E$ xy jest łańcuchem na mocy faktu, że orientacja łuków w łańcuchu nie gra roli, wnioskujemy że yx też jest łańcuchem

Przechodniość - $\forall x,y,z \in E$ jeśli xz i zy są łańcuchami, wówczas c=(x,...,z,z,.....y) również jest łańcuchem.

Klasy równoważności relacji R_c nazywamy komponentami spójnymi grafu G.

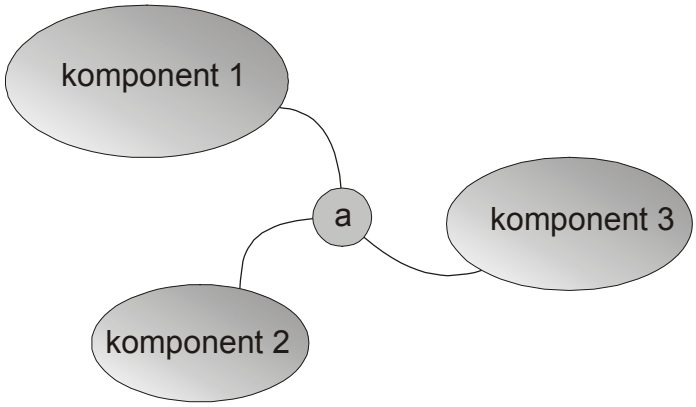
DEFINICJA Graf nazywamy spójnym, jeśli posiada *tylko jeden* komponent spójny.

Graf przedstawiony na rysunku nr 9 jest spójny, gdyż posiada jeden komponent spójny, a $\{1,2,3,4,5,6,7\} \in [1]_{Rc}$, czyli istnieje jedna klasa abstrakcji, w skład której wchodzą wszystkie wierzchołki grafu G.

DEFINICJA Punkt artykulacji „a” w grafie G=(E,V) jest takim wierzchołkiem ze zbioru E, że graf $G=(E-\{a\},V_1)$ nie jest spójny. (V_1 powstaje z V przez usunięcie wszystkich łuków, których jednym z końców jest wierzchołek „a”)

Uwaga: Nie w każdym grafie da się znaleźć punkt artykulacji.

Powyższą definicję ilustruje rysunek nr. 10, w którym punkt „a” łączy ze sobą spójne komponenty grafu G.



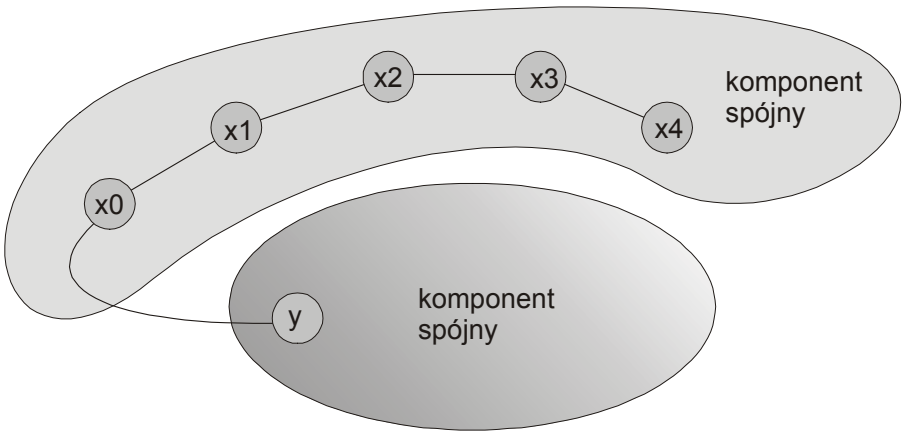
Rysunek 10 Punkt artykulacji w grafie

Łatwo udowodnić, że „a” jest połączony z *każdym* z komponentów grafu G. Załóżmy, że istnieje komponent, który nie jest połączony z „a”, wówczas wyjściowy graf G (zawierający „a”) posiadałby więcej niż jeden komponent spójny, czyli G nie byłby spójny – sprzeczność.

Lemat dotyczący grafów spójnych:

Każdy graf spójny stopnia ≥ 2 posiada co najmniej 2 wierzchołki nie będące punktami artykulacji.

Dowód: W grafie G (rysunek nr 11) weźmy najdłuższy łańcuch $c=(x_0, x_1, ..., x_n)$. Załóżmy, że x_0 jest punktem artykulacji. Wówczas odrzucając wierzchołek x_0 otrzymamy graf niespójny, zatem w grafie G musi istnieć drugi komponent spójny, w którym istnieje taki punkt y, że $x \, R_c \, y$. Wtedy jednak łańcuch $c_1=(y, x_0, x_1, ..., x_n)$ jest dłuższy od łańcucha c – sprzeczność. Stosując podobne rozumowanie dla wierzchołka x_n otrzymujemy dowód lematu.



Rysunek 11 Graf podatny na lemat o grafach spójnych

Klasa GS grafów spójnych jest generowana przez poniższy schemat indukcji:

Baza: $\{K_1\}$

Zasada indukcji: Jeśli $G \in GS$, wówczas Graf $G + \{x\} \in GS$, dla $d(x) \geq 1$.

Własność 1

Każdy graf spójny jest generowany przez powyższy schemat.

Dowód: W jedną stronę dowód jest natychmiastowy. Każdy graf wygenerowany przez powyższy schemat jest spójny, bowiem K_1 jest spójny, a gdy do grafu spójnego G dodamy wierzchołek „x", który łączy się przynajmniej jednym łukiem z grafem G, wówczas otrzymamy graf spójny.

W drugą stronę stosujemy dowód indukcyjny ze względu na stopień grafu.

stopień = 1, schemat generuje $\{K_1\}$ – graf o stopniu 1

Założenie – schemat potrafi wygenerować grafy spójne stopnia „n"

Teza – schemat potrafi wygenerować grafy spójne stopnia „n+1".

Weźmy G=(E,V), graf spójny stopnia „n+1". Na mocy $\exists x \in E$, że G-{x} jest spójny, ponadto G-{x} jest stopnia „n" czyli spełnia założenie indukcyjne, zatem G-{x} jest spójny. Teraz dodajemy do G-{x} wierzchołek {x}, który na mocy faktu, że G jest spójny, jest połączony z G-{x}, czyli $d(x) \geq 1$.W konsekwencji schemat pozwolił nam wygenerować graf G, co kończy dowód.

DEFINICJA Cyklem $c=(x_0, x_1, ..., x_n, x_0)$ nazywamy łańcuch, którego krańce są takie same.

W grafie z rysunku nr 9 możemy łatwo znaleźć cykl: c=(1,4,3,5,4,1) i cykl elementarny c_e(1,4,1), spełniający zależność: $\#C_e = 1 + \#Wierzch(C_2)$, gdzie $Wierzch(C_2)$ jest zbiorem różnych wierzchołków cyklu C_e.

Cykl elementarny w grafie G jest pod-grafem spójnym G, którego wszystkie wierzchołki są stopnia = 2.

Własność 2

Z każdego cyklu c, przechodzącego przez łuk „e" można wyodrębnić cykl elementarny, który przechodzi przez łuk „e".

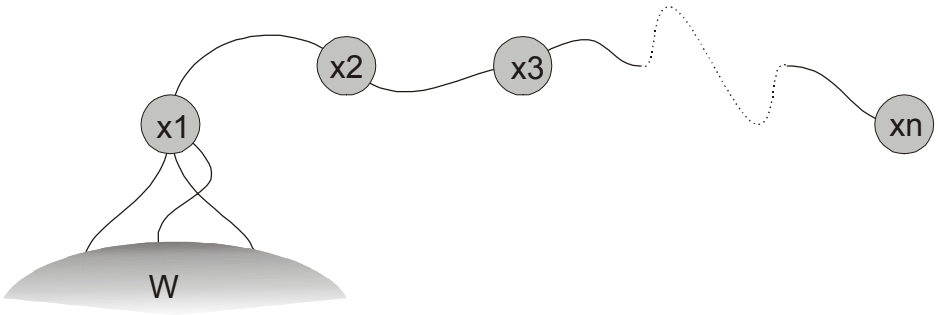
Dowód: Analogiczny jak przy łańcuchach.

Własność 3

Każdy graf G=(X,E) ,taki że $\delta(G) \geq k \geq 2$ dopuszcza cykl elementarny długości $\geq k+1$ oraz łańcuch elementarny o długości $\geq k$.

Dowód:

Na rysunku nr 12 przedstawiona została sytuacja, która pozwala na łatwe zrozumienie dowodu.



Rysunek 12 Cykl i łańcuch w grafie

Na początku wybieramy maksymalny łańcuch w grafie G=(X,E), oznaczając go c=(x1, ..., xn). Korzystając z założenia, że $\forall x_i \quad 1 \le i \le n \quad k \le \delta(x_i)$) wnioskujemy, że $k \le n$, bowiem każdy wierzchołek łańcucha c jest połączony z co najmniej k innymi wierzchołkami, czyli istnieje poszukiwany łańcuch o długości $\ge k$.

W celu znalezienia cyklu o długości $\ge k+1$ spójrzmy na wierzchołek x_1 na rysunku nr 12. Zauważamy, że w zbiorze wierzchołków W, które są bezpośrednio połączone z x_1 występują tylko wierzchołki x_3, ..., x_n, bowiem gdyby były w W inne wierzchołki np. $s \notin \{x_2, .., x_n\}$, wówczas łańcuch $(s, x_1, x_2, ..., x_n)$ byłby dłuższy od łańcucha c. Ponadto w W nie występuje x_2, bo c jest łańcuchem elementarnym. Z drugiej strony x_1 posiada co najmniej k sąsiadów, z czego wniosek, że $x_n \in W$, czyli istnieje cykl elementarny $(x_n, x_1, x_2, ..., x_n)$ o długości n+1. Korzystając z wcześniej udowodnionego faktu, że istnieje łańcuch elementarny o długości $\ge k$ wnioskujemy istnienie cyklu elementarnego o długości $\ge k+1$.

Własność 4

Mamy G=(X,E) #X = n oraz #E = m. Jeśli $m \ge n$, wówczas G nie posiada cyklu.

Dowód:

Weźmy graf G i wyodrębnijmy z niego pod-graf H=(Y,V) taki, że $Y \subseteq X$, przy czym w zbiorze Y znajdują się takie wierzchołki z X, że $\forall x \in X \quad d(x) \ge 2$. Zbiór krawędzi $V \subseteq E$, przy czym w zbiorze V znajdują się tylko te krawędzie, których krańce należą do X. Niemożliwe jest, by #Y = 0, bo wówczas w grafie wyjściowym G, $m < n$, zatem każdy wierzchołek grafu H posiada co najmniej dwóch różnych sąsiadów, stąd na podstawie *własności 2* w H istnieje cykl. H jest pod-grafem, więc cykl ten występuje też w G, co kończy dowód.

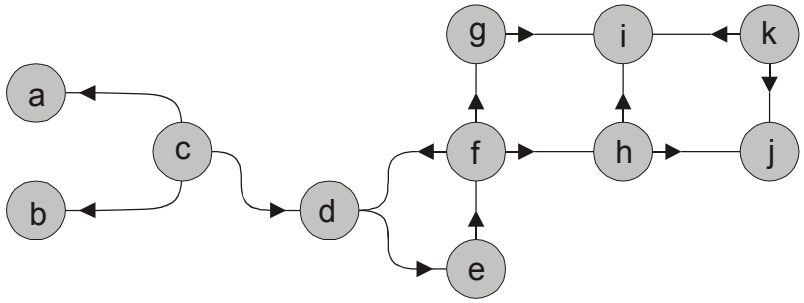
II. 3 Drzewa

DEFINICJA

Drzewo jest grafem spójnym bez cyklu.
Las jest grafem bez cyklu.

Na rysunku nr 13 możemy wyróżnić np. drzewa: (pod-grafy ograniczone do wierzchołków}
T1 = {a,b,c,d,e}
T2 = {k,i,j}
T3 = {d,f,g,h,j}

Lasem jest np. pod-graf z wierzchołkami $T1 \cup T2$.



Rysunek 13 Graf, las, drzewa

Własność 1

Dla danego grafu G=(X,E) #X = n oraz #E = m stopnia $n \geq 2$ następujące warunki są równoważne:

(1) G jest bez cyklu i bez wierzchołków wolnych
(2) G jest spójny i m = n-1
(3) G jest bez cyklu i m = n-1

Na początku dowodzimy (2) $\Rightarrow$ (3). Wniosek jest oczywisty (wykorzystujemy *własność 4* z podrozdziału o grafach spójnych).

Dla dowodu (3) $\Rightarrow$ (2) zakładamy, że G jest bez cyklu i nie jest spójny. Istnieją zatem takie wierzchołki $x \in X$, że d(x) = 0. Gdy utworzymy pod-graf H=(Y,V) wyrzucając te wierzchołki, wówczas dla tego pod-grafu będzie zachodziła zależność $\#V \geq \#Y$ i na mocy *własności nr 4* z poprzedniego rozdziału otrzymamy cykl w pod-grafie H, który będzie cyklem w grafie G, co da nam sprzeczność z założeniem.

Dowód (1) $\Rightarrow$ (2). G nie ma wierzchołków wolnych, więc G jest spójny, pozostaje więc dowieść, że m = n-1. Gdy G jest bez cyklu wtedy m < n, ponadto ze spójności wynika, że $m \geq n-1$ (wychodząc od K_2 dodanie do grafu nowego wierzchołka implikuje dodanie nowej krawędzi). Ostatecznie otrzymujemy m=n-1.

Dowód (2) $\Rightarrow$ (1). Graf ma m < n, czyli na mocy *wniosku 4* nie posiada cyklu, ponadto jest spójny, czyli nie posiada także wierzchołków wolnych.

Każdy graf spełniający jedną z powyższych własności jest drzewem.

Własność 2

Klasa drzew $\wp$ jest generowana przez następujący schemat indukcji:

Baza: $\{K_1\}$

Reguła: $T \in \wp \Rightarrow T+x \in \wp$ (x jest wierzchołkiem dołączonym w T+x)

Dowód, że schemat generuje drzewa jest oczywisty. Dowód w drugą stronę będzie oparty na indukcji ze względu na ilość wierzchołków w drzewie:

P(1) jest prawdziwe, bowiem $K_1 \in$ Bazy.

Zakładamy p(n), czyli że schemat wygeneruje nam wszystkie drzewa o n wierzchołkach. Weźmy dowolne drzewo T o n+1 wierzchołkach. Z definicji drzewa widzimy, że istnieje w nim jakiś wierzchołek dołączony x , taki że d(x)=1. Usuwając ten wierzchołek dostajemy drzewo T'= T - x, które spełnia założenia schematu, czyli $T = T' + x \in \wp$,co kończy dowód.

Własność 3

Mamy dany graf T=(X,E). Poniższe warunki są równoważne:

(1) T jest drzewem.

(2) T jest maksymalnym grafem bez cyklu ($\forall (x , y) \notin E$, T + (x , y) zawiera cykl)

(3) pomiędzy dowolnymi 2 wierzchołkami w T istnieje dokładnie jeden łańcuch elementarny.

(4) T jest najmniejszym grafem spójnym ($\forall e \in E$ T - e nie jest spójny)

Dowód:

(1) $\Rightarrow$ (3). T jest drzewem, więc jest grafem spójnym bez cyklu, czyli pomiędzy 2 dowolnymi wierzchołkami istnieje łańcuch elementarny (istnienie łańcucha nie elementarnego implikowałoby występowanie cyklu). Gdyby istniał inny łańcuch pomiędzy tymi wierzchołkami, wówczas łącząc odpowiednio ze sobą te dwa łańcuchy otrzymalibyśmy cykl, co jest sprzeczne z założeniem.

(3) $\Rightarrow$ (2) Weźmy dwa dowolne wierzchołki $x,y \in X$, aby $(x , y) \notin E$. Poprzednik implikacji mówi, że istnieje pomiędzy nimi dokładnie jeden łańcuch elementarny c. Dodając do T krawędź (x,y) otrzymamy nowy łańcuch łączący x i y, który po połączeniu z łańcuchem c da nam cykl (x,y, „c",x).

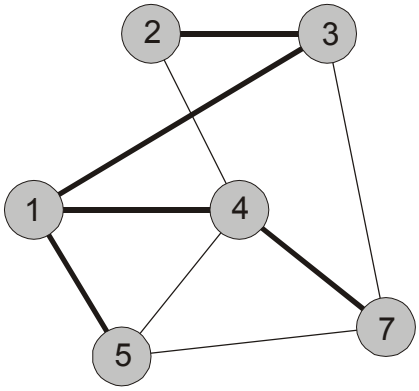
(2) $\Rightarrow$ (4) Weźmy dowolne $x,y \in X$. Jeśli $(x,y) \in E$, wówczas istnieje łańcuch łączący x i y. Jeśli $(x , y) \notin E$, to z założenia T + (x , y) zawiera

cykl, czyli w T istniał łańcuch łączący x i y, zatem T jest spójny. Gdyby $\exists$ $e \in E$ T-e jest spójny, wówczas T posiadałby cykl – sprzeczność.

(4) $\Rightarrow$ (1) T jest grafem spójnym. Gdyby T nie był drzewem, wówczas posiadałby on cykl c, zatem istniałyby wierzchołki x i y $\in X$ takie, że po usunięciu krawędzi (x, y) graf nadal byłby spójny – sprzeczność.

DEFINICJA Drzewem rozpinającym grafu G nazywamy taki pod-graf, który posiada wszystkie wierzchołki z G, podzbiór krawędzi G oraz jest drzewem.

Na rysunku nr 14 pokazano drzewo rozpinające graf (krawędzie drzewa są pogrubione).



Rysunek 14 Drzewo rozpinające graf

Własność 3

G jest grafem spójnym $\Leftrightarrow$ istnieje drzewo rozpinające graf G.

Dowód ($\Leftarrow$) jest oczywisty, ponieważ drzewo z założenia jest grafem spójnym.

Dowód ($\Rightarrow$) : indukcyjny ze względu na ilość wierzchołków grafu. P(1) jest prawdziwe, gdyż istnieje drzewo złożone z jednego wierzchołka.

Zakładamy p(n) ,czyli że dowolny każdy graf spójny o n wierzchołkach posiada drzewo rozpinające. Weźmy dowolny graf spójny G o n+1 wierzchołkach. Z lematu o grafach spójnych wiemy, że istnieje wierzchołek x nie będący punktem artykulacji grafu G. Zatem G-x jest grafem spójnym o n wierzchołkach, czyli z założenia indukcyjnego istnieje dla niego drzewo rozpinające T'. Łącząc jakikolwiek wierzchołek drzewa T' z punktem x otrzymamy drzewo rozpinające grafu G co dowodzi tezę indukcyjną.

DEFINICJA Drogą w grafie zorientowanym G=(X,V) nazywamy ciąg wierzchołków $\mu = (x_0, x_1, ..., x_n)$, taki że $\forall i \ \ 1 \leq i \leq n \ \ x_{i-1}x_i \in V$. Krańcami μ są x_0 i x_n , a długością n.

Droga elementarna to ciąg, w którym każdy wierzchołek jest inny.

Z każdej drogi łączącej wierzchołki x i y możemy wyodrębnić drogę elementarną między x i y (dowód jest analogiczny jak dla łańcuchów).

Droga elementarna odwiedzająca wszystkie wierzchołki grafu jest nazywana *drogą Hamiltona*.

Przyjmujemy oznaczenia: *przodek = Ascendant, potomek = Descendant* i dla każdego wierzchołka definiujemy: zbiór jego przodków i potomków:

$\forall x \in X \quad Des(x) = \{y \in X \mid \exists \text{ droga } xy\}$

$\forall x \in X \quad Asc(x) = \{y \in X \mid \exists \text{ droga } yx\}$

Jeśli $x,y \in X$ oraz $(x,y) \in E$ wtedy x jest *poprzednikiem* y, a y jest *następnikiem* x.

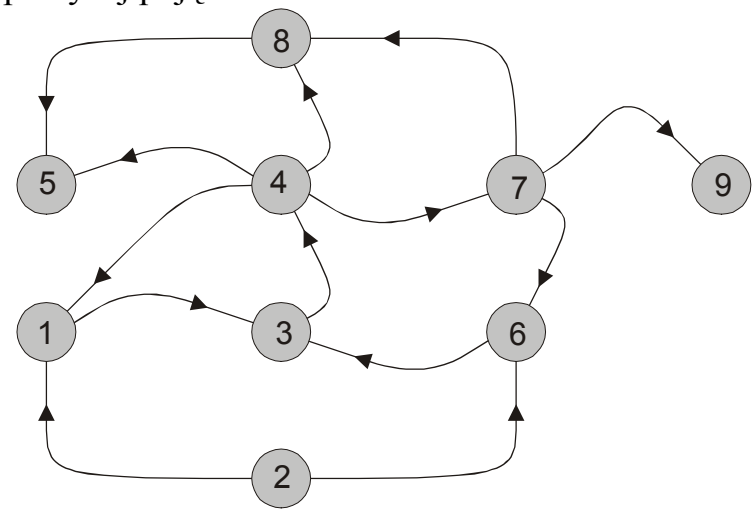
Korzeniem w grafie G=(X,E) jest taki wierzchołek $r \in X$, że $Des(r) \cup \{r\} = X$.

Odległość: Gdy $y \in Des(x)$,to d(x,y) jest długością drogi xy.

Gdy $y \notin Des(x)$, to $d(x,y) = +\infty$

Koło: Kołem nazywamy zamknięty ciąg wierzchołków $(x_1, x_2, ..., x_n)$, który jest permutacją cykliczną i $\forall i \quad 2 \le i \le n \quad (x_{i-1},x_i) \in E$ oraz $(x_n,x_1) \in E$.

Koło elementarne jest ciągiem wierzchołków, które nie mogą się powtórzyć. Na rysunku nr 15 przedstawiony jest przykładowy graf obrazujący przedstawione powyżej pojęcia.



Rysunek 15 Przodkowie, potomkowie, koła i korzenie

Des(3) = {4,5,6,7,8,9,1}
Następnik(3) = {4}
Asc(3) = {1,2,4,6,7}
Poprzednik(3) = {1,6}
Koło = {7,6,3,4,1,3,4}
Koła elementarne = {1,3,4} , {1,6,3,4}
Korzeń = {2}, gdyż Des(2) = {1,3,4,5,6,7,8,9}
d(3,5) = 2

II. 4 Drzewa właściwe

DEFINICJA: Drzewem właściwym nazywamy drzewo posiadające korzeń.

Własność 1

Dla grafu G=(E,V) poniższe definicje są równoważne:
G jest drzewem właściwym
G jest bez cykli i posiada korzeń
G posiada korzeń i (n-1) łuków
G posiada korzeń r oraz $\forall x \in X \; \exists!$ droga rx
G jest spójny i $\exists\, x_0 \in X \mid d^-() = 0$ oraz $\forall x \in X \;\; x \neq x_0 \;\Rightarrow d^-(x) = 1$

Dowód:

(1)⇒(2) Jeśli G jest drzewem, to z definicji nie posiada cykli, a jeśli jest ponadto drzewem właściwym, to posiada korzeń.

(2)⇒(4) Jeśli G posiada korzeń r, to każdy inny wierzchołek $x \in G$ jest osiągalny, czyli istnieje droga rx. Gdyby istniały dwie różne drogi rx, wówczas G posiadałby cykl i mielibyśmy sprzeczność z założeniem.

(4)⇒(3) Jeśli dla n-1 wierzchołków $x \in G$ istnieje od korzenia dokładnie jedna droga rx, to znaczy, że graf G posiada n-1 łuków.

(3)⇒(5) G posiada korzeń r, czyli $Des(r) \cup \{r\} = X$, zatem G jest spójny. $Asc(r) = \varnothing$, bowiem w przeciwnym wypadku: istniałoby n-1 dróg od korzenia do pozostałych wierzchołków + łuk do korzenia = n łuków; sprzeczność, czyli $d^-(r) = 0$. Podobnie, $\forall x \in X \;\; x \neq r \;\Rightarrow d^-(x) = 1$ gdyż gdyby $d^-(x) = 0$ nie istniałaby droga rx, ponadto gdyby $\exists x \in X \;\; x \neq r$ i $d^-(x) = 1+h$, to G posiadałby co najmniej n-1 + h łuków, stąd h = 0.

(5)⇒(1) $\forall x \in X \;\; x \neq x_0 \;\Rightarrow x \notin Des(x)$, gdyż w przeciwnym przypadku $Asc(x) \cap Des(x) \neq \varnothing$, a wtedy uwzględniając $d^-(x_0) = 0$ otrzymalibyśmy graf niespójny. Zatem G jest bez cykli i w dodatku spójny, czyli G jest drzewem. Gdyby $\exists x \in X \;\; x \neq r$ i $r \notin Asc(x)$ wówczas uwzględniając fakt, że r nie posiada poprzednika mielibyśmy $r \notin Des(x)$ i w konsekwencji niespójność grafu G. Wynika z tego, że $\forall x \in X \;\; x \neq x_0 \; r \in Asc(x)$, czyli r jest korzeniem i ostatecznie G jest drzewem właściwym.

Własność 2

Klasa drzew właściwych $\wp$ jest generowana przez następujący schemat indukcji:

Baza: $\{K_1\}$

Reguła: $T \in \wp \;\Rightarrow\; T+x \in \wp$ (gdzie $d^-(x) = 1$ i $d^+(x) = 0$; x jest liściem)

Schemat indukcyjny bez wątpienia generuje drzewa właściwe. Udowodnimy, że każde drzewo właściwe może być przez ten schemat wygenerowane.

Dowód indukcyjny ze względu na ilość wierzchołków drzewa właściwego.

P(1) jest prawdziwe, gdyż $K_1 \in$ Bazy.

Załóżmy, że p(n) jest prawdziwe, czyli schemat potrafi wygenerować każde drzewo właściwe o n wierzchołkach. Weźmy dowolne drzewo T o n+1 wierzchołkach. Drzewo nie posiada cykli, więc $\exists x \in T$, że $d^-(x) = 1$ i $d^+(x) = 0$, czyli x jest liściem. Tworzymy drzewo T' usuwając wierzchołek x i otrzymujemy drzewo o n wierzchołkach, które spełnia założenie indukcyjne, zatem drzewo T = T' + x może być wygenerowane przez schemat indukcyjny.

DEFINICJA Drzewem właściwym rozpinającym graf G nazywamy pod-graf , który jest drzewem, posiada korzeń, te same wierzchołki co G i podzbiór łuków G.

Własność 3

Graf G posiada 1 korzeń $\Leftrightarrow$ Posiada drzewo właściwe rozpinające go.

II. 5 Grafy silnie spójne

DEFINICJA: Silna spójność

Dla danego grafu G=(X,E) definiujemy relację $R_{ss} \subseteq X^2$. $\forall x \in X$ mamy

$$(x,y) \in R_{ss} \Leftrightarrow \exists \text{ droga xy lub yx}$$

Można sprawdzić, że R_{ss} jest relacją równoważności.

Komponentami s-spójnymi grafu G nazywamy klasy równoważności relacji R_{ss}.

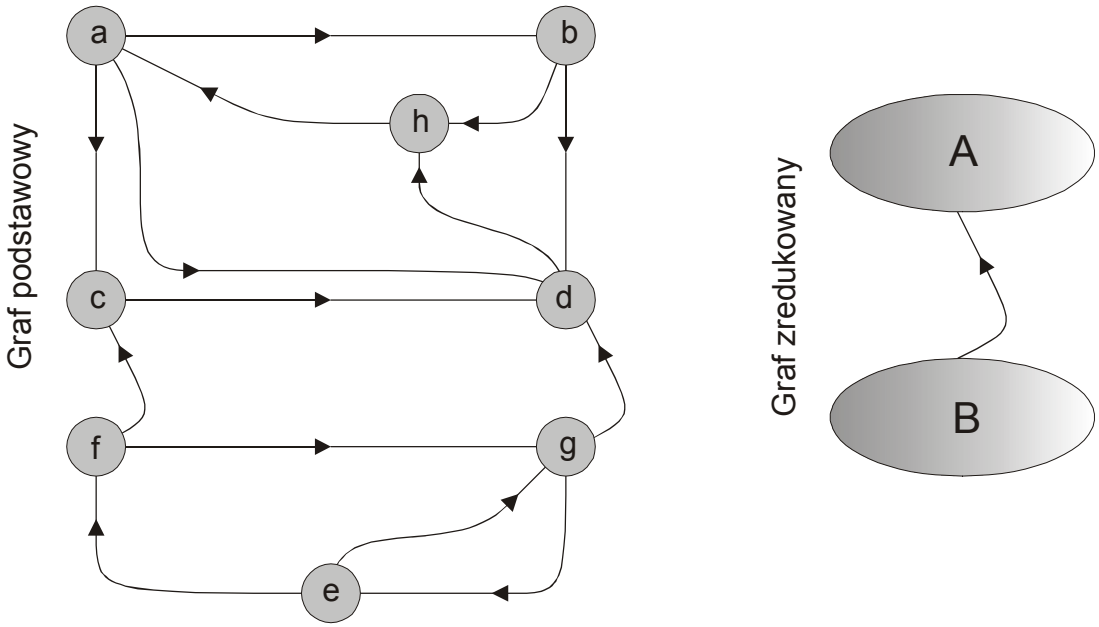
Grafem *silnie spójnym* nazywamy graf posiadający 1 spójny komponent.

Na rysunku nr 15 mamy dwa komponenty silnie spójne:

A={a,b,c,d,h}

B={e,f,g}

Grafem zredukowanym nazywamy graf, którego wierzchołkami są klasy abstrakcji relacji R_{ss}. W klasie zredukowanym nie występują cykle, gdyż gdyby pomiędzy dwoma klasami abstrakcji istniał cykl, wówczas każdy element z pierwszej klasy byłby w relacji R_{ss} z każdym elementem drugiej klasy i w konsekwencji te klasy byłyby takie same, czyli graf zredukowany posiadałby jeden wierzchołek.



Rysunek 16 Graf podstawowy i graf zredukowany

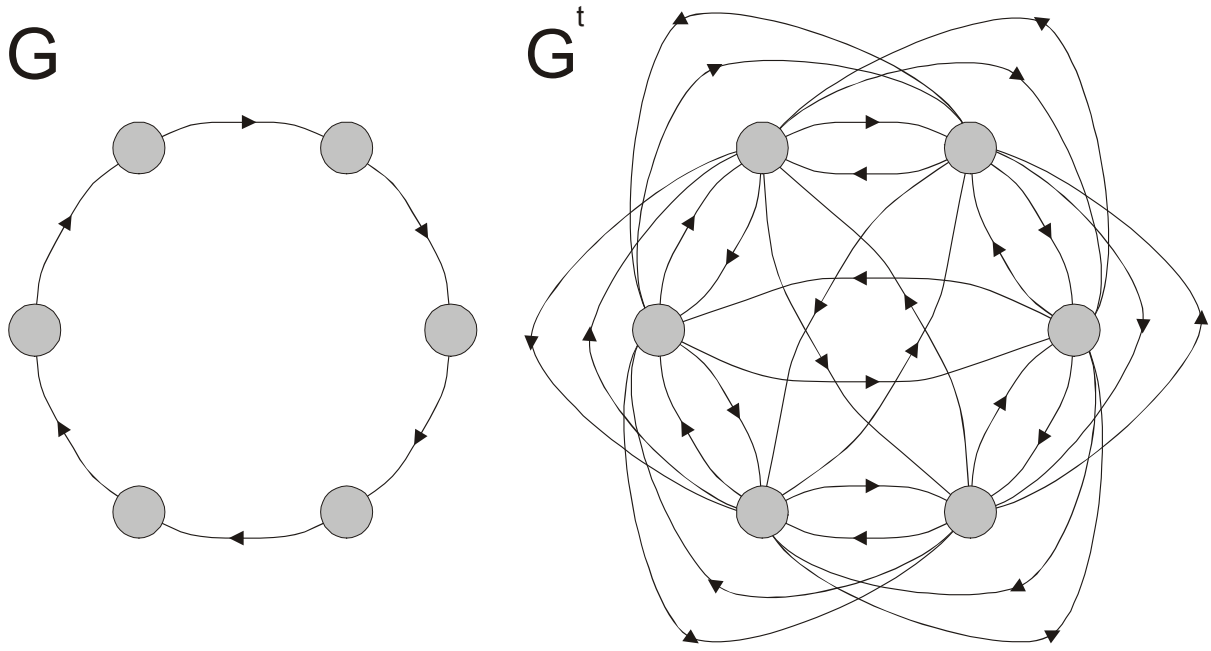
DEFINICJA Domknięcie przechodnie grafu zorientowanego.

Mówimy, że xy jest *łukiem przechodnim* w grafie G=(X,E) jeśli istnieje w tym grafie inna ścieżka łącząca x i y, długości >1, czyli y jest potomkiem x w grafie G'=(X,E-{x,y}).

$G^t=(X,E^t)$ jest domknięciem przechodnim grafu G=(X,E), gdy:

$$E^t=\{(x,y) \mid y \in des_G(x), y \neq x\}$$

Domknięcie przechodnie dodaje do grafu wszelkie możliwe drogi łączące wierzchołek x i y. Na rysunku nr 17 przedstawiono graf złożony z jednego cyklu i jego domknięcie przechodnie.



Rysunek 17 Graf i jego domknięcie przechodnie

Rysunek nr 17 jest przykładem na to, że domknięcie przechodnie grafu silnie spójnego powoduje powstanie grafu kompletnego. Dowód jest tutaj oczywisty. Weźmy dowolne x i y, wierzchołki grafu G. Ze spójności grafu G otrzymujemy istnienie drogi xy. Domknięcie przechodnie będzie zatem zawierało łuk (x,y) dla dowolnych x i y, czyli G^t jest kompletny.

Gdy do domknięcia przechodniego dodamy pętle dla każdego wierzchołka, wówczas otrzymamy *domknięcie przechodnio-zwrotne* oznaczane przez G*.

Są dwa podejścia do tworzenia z G grafu G^t:

(1) dodajemy do G wszystkie możliwe łuki przechodnie.

(2) tworzymy najmniejszy graf przechodni zawierający łuki G.

Ponadto gdy mamy 2 grafy: $H \subseteq G$ wtedy $H^t \subseteq G^t$.

II. 6 Porządek grafu

Dla danego zbioru A możemy zdefiniować relację porządku R. Gdy R jest słabym porządkiem, co oznaczamy jako ≤, zachodzą zależności:

(1) zwrotność $\forall x \in A\ xRx$

(2) anty-symetria $\forall x,y \in A\ (xRy \wedge yRx) \Rightarrow x=y$

(3) przechodniość $\forall x,y,z \in A\ (xRy \wedge yRz) \Rightarrow xRz$

Gdy R jest silnym porządkiem, co oznaczamy jako <, zachodzą zależności:

(1) niezwrotność $\forall x \in A\ (x,x) \notin R$

(2) przechodniość

Z tymi relacjami utożsamiamy:

a) graf relacji słabego porządku ≤

$$G_{\leq} = (E,U) \qquad (x,y) \in U \Leftrightarrow x \leq y$$

Graf ten jest zwrotny (przy każdym wierzchołku występuje pętla), anty-symetryczny i przechodni.

b) graf relacji słabego porządku <

$$G_{<} = (X,V) \qquad (x,y) \in V \Leftrightarrow x < y$$

Graf ten jest niezwrotny (brak pętli) i przechodni.

Własność 1

Jeśli < jest relacją silnego porządku, wówczas $G_{<}$ nie posiada koła.

Jeśli ≤ jest relacją słabego porządku, wówczas $G_{\leq}$ nie posiada koła o długości >1

Dowód pierwszej zależności wynika bezpośrednio z przechodniości $G_<$. Gdyby istniało koło $(x_1,x_2, .., x_n)$ wówczas $(x_1<x_2) \wedge ... \wedge (x_{n-1}<x_n) \Rightarrow x_1<x_n$. Ponadto $x_n<x_1$;sprzeczność. Dowód drugiej zależności jest analogiczny, a koła o długości 1 są cyklami, których istnienie wynika ze zwrotności $G_\leq$.

R jest relacją pokrycia , co oznaczamy $<\cdot$, gdy zachodzi warunek:

$$\forall x,y \quad x<\cdot y \Leftrightarrow (x < y \ \wedge \ \neg \exists z \ x < z < y)$$

Z relacją pokrycia $<\cdot$ utożsamiamy graf Hasse'a G_h. Graf G_h powstaje przez usunięcie z grafu $G_<$ wszystkich łuków przechodnich.

Własność 2

$(G_h)^t = G_<$

Dowód:

Udowodnimy inkluzję „$\subset$“.$\subset G_<$ zatem $(G_h)^t = (G_<)^t = G_<$.

Teraz udowodnimy inkluzję „$\supset$“. Weźmy dowolny łuk (x,y) z grafu $G_<$. Jeśli xy, wówczas (x,y) należy do G_h i tym bardziej do $(G_h)^t$. Gdy natomiast $(x,y) \notin <\cdot$, wówczas korzystając z tego, że $G_<$ nie posiada koła otrzymamy taki ciąg wierzchołków grafu G $(x,x_1,x_2, ..., x_n, y)$, że $\{(x,x_1),(x_1,x_2), ..., (x_n,y)\} \subset <\cdot$. Stąd (x,y) należy do domknięcia przechodniego grafu G_h, co kończy dowód.

II. 7 Rozkład grafu

Jeśli graf G=(X,E) posiada wierzchołki $x \in X$ takie, że $d^-(x)=0$, to każdy taki wierzchołek jest *źródłowy* w grafie G.

Rozkład grafu przeprowadzamy usuwając w kolejnych krokach wierzchołki źródłowe. Niech dla grafu G stopnia k, zbiór S(G) będzie zawierał wszystkie jego wierzchołki źródłowe. Tworzymy kolejno zbiory:

$S_0 = S(G)$

$S_1 = S(G - \{S_0\})$

$S_2 = S(G - \{S_0 \cup S_1\})$

.....

$S_i = S(G - \{S_0 \cup S_1 \cup \ ... \cup S_{i-1}\})$

.....

$S_k = 0$

Własność 1

Graf G nie posiada koła $\Leftrightarrow$ dla powyższego schematu rozkładu zachodzi:

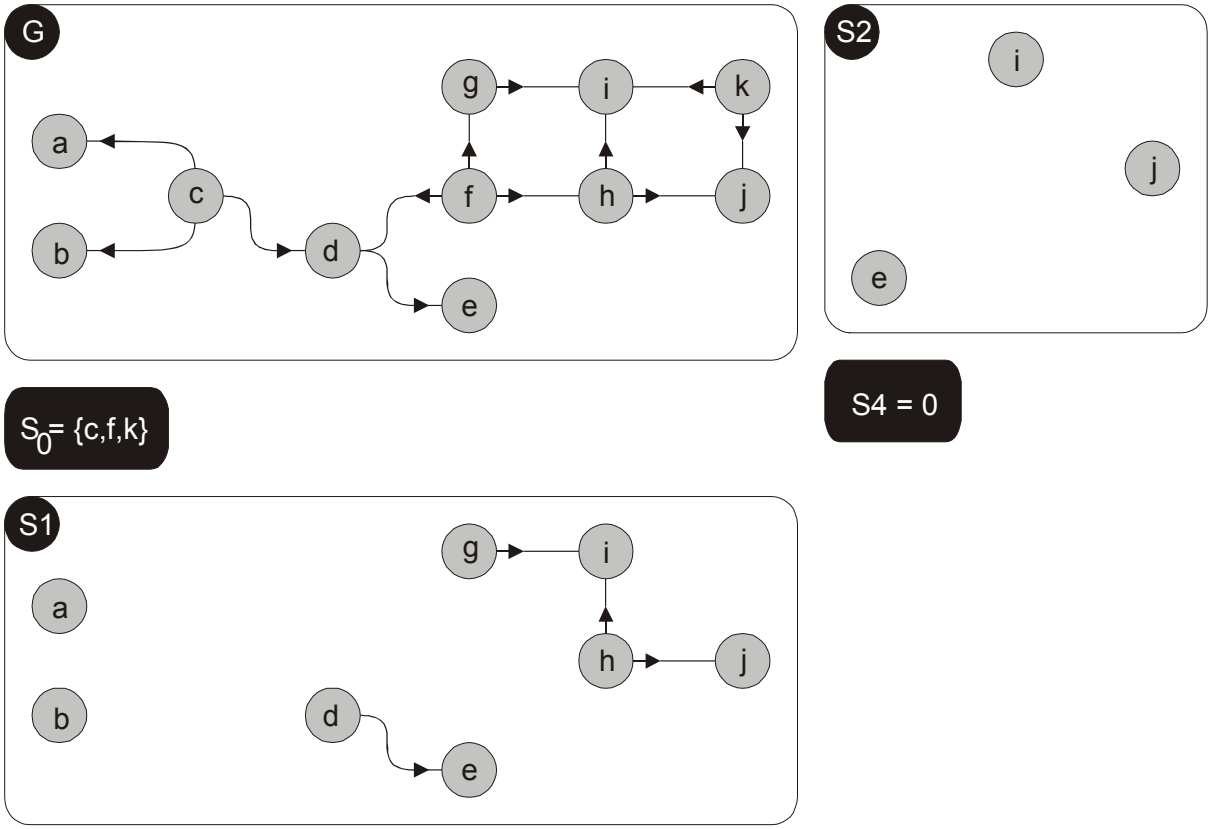
$$X = S_0 \cup S_1 \cup \ ... \cup S_{k-1} , \quad Si \cap Sj = \varnothing$$

Dowód:

„⇒”. Jeśli graf nie posiada koła, wówczas w kolejnych krokach będziemy usuwali z grafu G wierzchołki źródłowe do momentu, gdy G nie będzie już pusty. Ponieważ G jest stopnia k, więc powyższy rozkład będzie skończony i w rezultacie suma mnogościowa zbiorów S_i da nam cały zbiór wierzchołków X.

„⇐” Przeprowadzimy dowód nie-wprost. Załóżmy, że G posiada koło, wówczas $\exists j \geq 0$, że od pewnego momentu $S_j = S_{j+1} = S_{j+2} = ... = S_{k-1} \neq \emptyset$, stąd $\exists x$ wierzchołek grafu G, że $\forall S_i \; x \notin S_i$; sprzeczność.

Przykładowy rozkład grafu przedstawiony został na rysunku nr 18.



Rysunek 18 Rozkład grafu

Rozdział III Algorytmy grafowe

III. 1 Przeszukiwanie grafu

Wyróżniamy dwa główne sposoby przeszukiwania grafu:

1) przeszukiwanie wszerz
2) przeszukiwanie w głąb

We wszystkich typach przeszukiwania musimy uwzględnić niespójność grafów, dlatego właściwy algorytm przeszukiwania, który na wejściu otrzymuje wierzchołek startowy, musimy zastosować dla wszystkich spójnych komponentów badanego grafu. Najczęściej więc metodę przeszukiwania stosuje się kolejno dla wszystkich wierzchołków grafu (gdyż nie wiemy, które komponenty są spójne) sprawdzając czy wybierany wierzchołek nie został już wcześniej odwiedzony.

Kolejność wyboru sąsiadów może zależeć np. od ich numerów, tzn. mając dwóch sąsiadów o numerach 4 i 6 wybieramy na początku 4, a potem 6.

III. 1. 1 Przeszukiwanie wszerz

Przy tym przeszukiwaniu odwiedzamy kolejno sąsiadów danego wierzchołka, a dopiero później potomków tych sąsiadów. Ogólną ideę przeszukiwania wszerz przedstawia poniższy schemat:

Dane:

tablica visited[1..n] = 1,0 (visited[i]=1 – gdy wierzchołek I został już odwiedzony, 0 gdy nie został jeszcze odwiedzony)

kolejka queue – w której znajdują się wierzchołki grafu, które będą kolejno odwiedzane.

Algorytm:

```
Dla i=1..n podstaw visited[i]←0
Dla i=1..n wykonuj
{
        Wyzeruj(queue)
        Jeśli (visited[i] = 0) Dodaj(queue) ← i
        Dopóki queue nie jest pusta wykonuj
        {
                x←Głowa(queue)
                visited[x] = 1                  (odwiedzenie wierzchołka)
                ZwolnijGłowę(queue)
                Dla każdego sąsiada s wierzchołka x wykonuj
                {
                        Jeśli visited[s] = 0 Dodaj(queue) ← s
                }
        }
}
```

Przeglądając graf z rysunku nr 16 otrzymamy rezultat:
(a, b, c, d, h, e, f, g)

III. 1. 2 Przeszukiwanie w głąb

Przy tym przeszukiwaniu odwiedzamy na początku dzieci danego sąsiada wierzchołka x, a później innych sąsiadów wierzchołka x. Ideę przeszukiwania w głąb przedstawia poniższy schemat.

Dane:
tablica visited[1..n] = 1,0 (visited[i]=1 – gdy wierzchołek I został już odwiedzony, 0 gdy nie został jeszcze odwiedzony)

Algorytm rekurencyjny:

```
DFS(x)
{
        visited[x] = 1                  (odwiedzenie wierzchołka)
        Dla każdego sąsiada s wierzchołka x wykonuj
        {
                Jeśli visited[s] = 0  DFS(s)
        }
}
```

wywołanie algorytmu przeszukiwania w głąb

```
Dla i=1..n podstaw visited[i]←0
Dla i=1..n wykonuj
{
        Jeśli (visited[i] = 0)
        DFS(i)
}
```

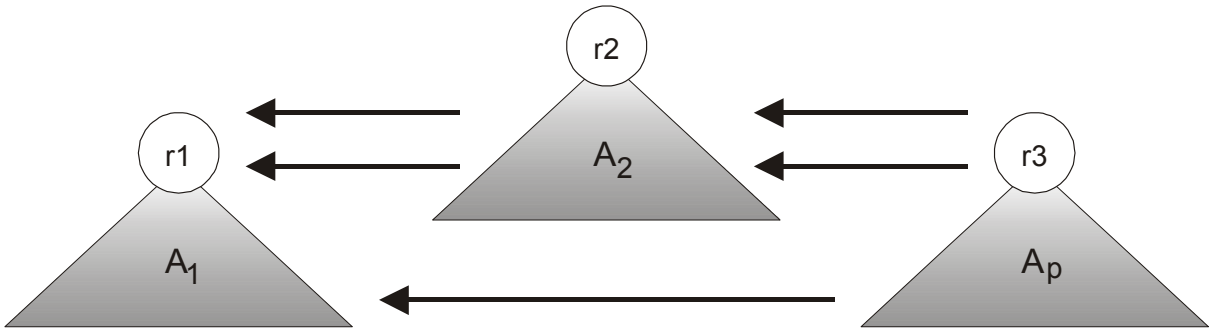
Przeglądając graf z rysunku nr 16 otrzymamy rezultat:
(a, b, d, h, c, e, f, g)

III. 2 Klasy silnie spójne

W rozdziale tym rozpatrzymy algorytm dzielący dowolny graf zorientowany na komponenty silnie spójne.

Rozważmy graf zorientowany G=(X,E) stopnia n. Wybierzmy wierzchołek r1∈X. Oznaczmy przez A_1 maksymalne drzewo o korzeniu r_1, zbudowane z łuków grafu G. Jeśli A_1 nie zawiera wszystkich wierzchołków grafu G, wybierzmy dowolny wierzchołek r_2, który nie należy do drzewa A_1. Tworzymy maksymalne drzewo A_2 o wierzchołku r_2, zbudowane z wierzchołków grafu G-A_1. Kontynuując ten proces otrzymamy las drzew rozłącznych przykrywających w całości wierzchołki grafu G.

Rozpatrzmy G_i pod-graf oparty na wierzchołkach drzewa A_i. Rozpatrzmy łuki grafu G, które łączą różne pod-grafy G_i. Jeśli istnieje łuk pomiędzy G_i i G_j ,gdzie i<j, wówczas musi on prowadzić od G_j do G_i , bowiem w przeciwnym przypadku G_i nie byłby maksymalny. W konsekwencji, jeśli drzewa $(A_i)_{1\le i\le p}$ zostały utworzone w wyniku przeglądania grafu (np. w głąb) zachodzą następujące własności: (Rysunek nr 19)



Rysunek 19 Las oparty na grafie G

- Pomiędzy kolejnymi G_i, łuki grafu G - $\cup_{1\le i\le p} G_i$ są skierowane tak jak na rysunku nr 19.
- Wewnątrz każdego G_i, łuki G_i – A_i łączą tylko wierzchołki znajdujące się w A_i.

Dla potrzeb algorytmu będziemy używali grafu dualnego do G (graf otrzymany przez zmianę orientacji krawędzi) oznaczanego prze ${}^dG$. Ponadto przez $C_G(X)$ będziemy oznaczali komponent silnie spójny zawierający wierzchołek X.

Lemat 1

Dla 2 maksymalnych drzew (A,r) i (B,r) o wspólnym korzeniu r, otrzymanych przy przeglądaniu w głąb kolejno grafów G i ${}^dG$ zachodzi poniższa własność:

$x \in C_G(r) \Leftrightarrow x \in (A,r) \cap (B,r)$.

Dowód: jeśli $x \in C_G(r)$, wówczas w grafie G istnieje ścieżka (r..x) , stąd $x \in (A,r)$, oraz ścieżka (x..r) stąd $x \in (B,r)$. Dowód w drugą stronę jest analogiczny.

Lemat 2

Dla każdego wierzchołka x grafu G_i zachodzi: $C_G(x) = C_{Gi}(x)$.

Dowód: Weźmy wierzchołek $y \in C_G(x)$ i $y \neq x$. Z definicji istnieje droga od y do x, czyli y nie może należeć do G_j gdy $j<i$, bowiem nie ma żadnego łuku od G_j do G_i. Z tego samego powodu istnieje droga od x do y i y nie należy do żadnego G_k, gdy $k>i$. Stąd y należy do G_i.

Rozpatrzmy teraz sekwencję drzew (A_1,r_1), (A_2,r_2) ... (A_p,r_p) otrzymanych w wyniku przeglądania grafu G w głąb. Ponumerujmy wszystkie wierzchołki każdego drzewa według porządku postfix poczynając od drzewa (A_1,r_1),a kończąc na drzewie (A_p,r_p); W szczególności r_p będzie posiadał numer n.

Lemat 3

Komponent silnie spójny oparty na ostatnim korzeniu $C_G(r_p)$ jest zbiorem wierzchołków odwiedzonych stosując przeglądanie w głąb grafu ${}^dG$ począwszy od wierzchołka r_p.

Dowód: Oznaczmy (B,r_p) drzewo otrzymane w wyniku przeszukiwania w głąb grafu ${}^dG$ począwszy od R_p. Pokażemy najpierw, że wierzchołki drzewa (B,r_p) są zarazem wierzchołkami drzewa (A_p,r_p). W rzeczywistości z konstrukcji sekwencji $(A_i,r_i)_{1\le i\le p}$ wynika, że nie ma żadnych łuków dochodzących do

drzewa (A_p,r_p), innymi słowy, w grafie ${}^dG$ nie ma żadnych łuków wychodzących z (A_p,r_p). W rezultacie każde drzewo grafu ${}^dG$ o korzeniu r_p może zawierać tylko wierzchołki znajdujące się w (A_p,r_p), stąd wniosek, że każdy wierzchołek drzewa (B,r_p) należy do (A_p,r_p).

O wierzchołku powiemy, że jest *uzgodniony*, jeśli znaleziono już dla niego komponent silnie spójny.

Lemat 4

Wybierzmy wierzchołek nie uzgodniony, który posiada największy numer postfix. $C_G(r)$ jest zbiorem wierzchołków dotychczas nie uzgodnionych, znalezionych w wyniku przeszukiwania w głąb grafu ${}^dG$ począwszy od r.

Dowód: Wybierzmy drzewo (A_i,r_i) do którego należy r. Oznaczmy przez S zbiór wierzchołków (A_i,r_i) które są uzgodnione. Zauważamy, że dla numeracji postfix, jeśli S nie jest pusty, zawiera wierzchołek r_i. W drzewie (A_i,r_i) rozpatrzmy pod-drzewo (C,r) o korzeniu r. Prawdziwe są własności:

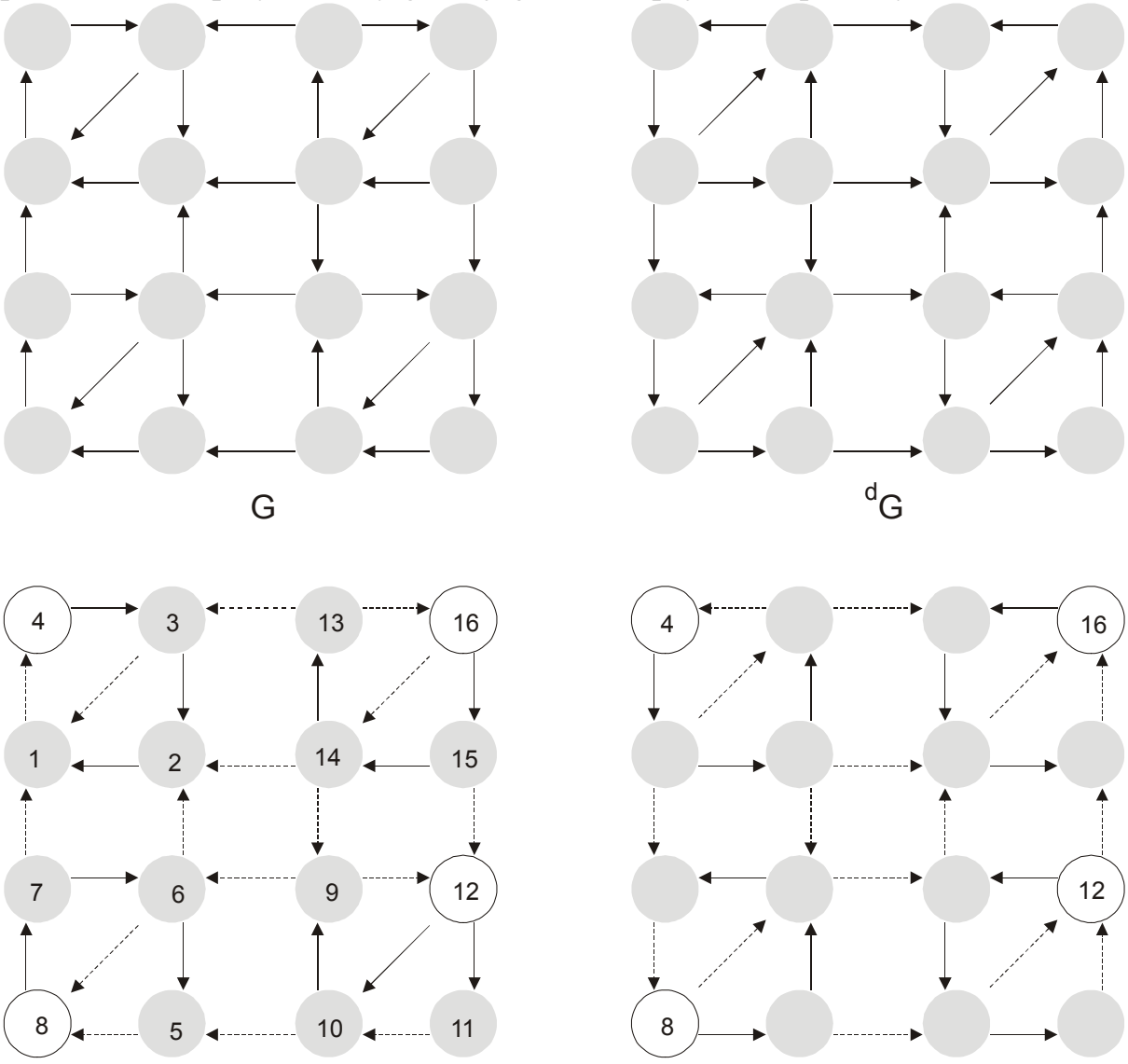
- nie ma żadnych łuków od (C,r) do S, ponieważ gdyby istniał taki łuk, wówczas istniałaby droga od r do wierzchołka s już uzgodnionego, czyli droga od s do r w ${}^dG_i$, czyli r byłby wierzchołkiem odwiedzonym przy przeszukiwaniu ${}^dG$.
- nie ma łuków wracających do (C,r) wychodzących z wierzchołka nie uzgodnionego h spoza (C,r), gdyż w przeciwnym przypadku istniałby łuk $h \rightarrow c \in (C,r)$, zatem c należałby do innego pod-drzewa opartego na korzeniu, którego numer jest mniejszy od numeru korzenia r.

W konsekwencji przeglądając graf ${}^dG$ w głąb począwszy od r, odwiedzamy tylko wierzchołki w (C,r) lub wierzchołki już uzgodnione.

Możemy teraz zapisać już ostateczny algorytm wyznaczania komponentów silnie spójnych w grafie G:

1. przeglądanie w głąb grafu G, numerując kolejne wierzchołki w porządku postfix
2. konstrukcja grafu dualnego ${}^dG$.
3. Dopóki istnieje wierzchołek nie uzgodniony wykonywanie operacji: Przeglądanie w głąb grafu ${}^dG$ począwszy od wierzchołka o największym numerze. Komponent silnie spójny dla tego wierzchołka jest zbiorem nie uzgodnionych dotychczas wierzchołków spotkanych przy przeglądaniu grafu ${}^dG$.

Złożoność obliczeniowa każdego etapu, to O(n+m). Na rysunku nr 20 przedstawiono przykładowy graf i jego silnie spójne komponenty.



Rysunek 20 Proces wyznaczania komponentów silnie spójnych

III. 3 Maksymalne skojarzenia

Problem znajdywania maksymalnego skojarzenia w grafie dwudzielnym jest szczególnym przypadkiem problemu znajdowania maksymalnego przepływu w grafie wykorzystując algorytm Forda – Fulkerson'a.

Weźmy graf dwudzielny G=(X,E) to znaczy taki, którego zbiór wierzchołków możemy podzielić na dwa rozłączne zbiory U i V, by $X = U \cup V$, a ponadto dla każdej krawędzi $(a,b) \in E$, takiej że $a,b \in X$ zachodzi $a \in U \Leftrightarrow b \in V$, czyli krawędzie istnieją tylko pomiędzy wierzchołkami z różnych zbiorów.

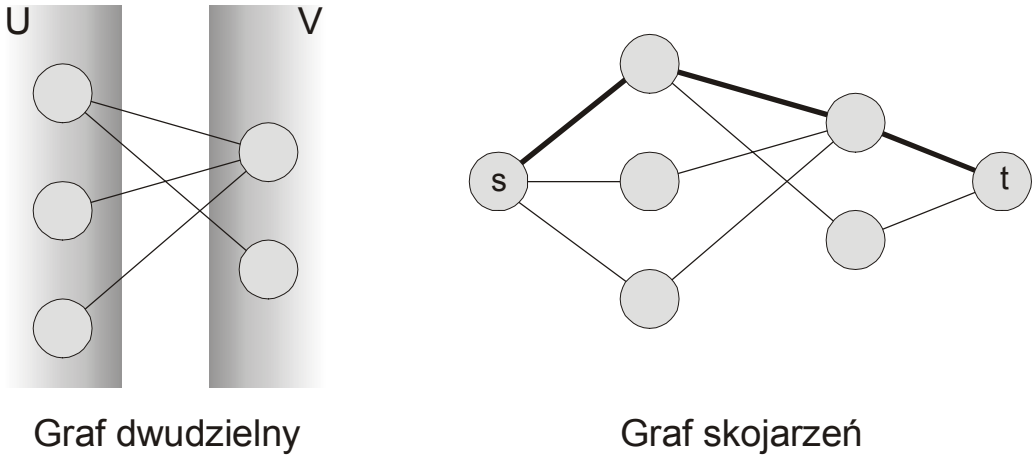
Problem znajdowania maksymalnego skojarzenia w grafie G sprowadza się do znalezienia największego pod względem inkluzji podzbioru F zbioru krawędzi w grafie G, czyli $F \subseteq G$, spełniającego następującą zależność:

$\forall (a,b) \in F$ i $(c,d) \in F$ zachodzi $a \neq b \neq c \neq d$, czyli każda krawędź zbioru F posiada inny wierzchołek krańcowy.

Z naszego grafu tworzymy graf skojarzeń $G_s = (Y,H)$, którego zbiór wierzchołków jest powiększony o wierzchołek startowy s , oraz wierzchołek końcowy t: $Y = X \cup \{s,t\}$. Zbiór krawędzi H jest powiększeniem zbioru E o krawędzie pomiędzy s i U, oraz krawędzie pomiędzy V i t. Ogólnie możemy zapisać, że:

$H = E \cup \{(s,x) \text{ dla } \forall x \in U\} \cup \{(x,t) \text{ dla } \forall x \in U\}$

Poszczególnemu skojarzeniu będzie zatem odpowiadał zbiór dróg długości 3 od s do t, przy czym każda droga przechodzi przez inny wierzchołek oraz żadne dwie drogi nie mogą mieć wspólnych wierzchołków. Powyższy schemat ilustruje rysunek nr 21.



Rysunek 21 Graf dwudzielny i graf skojarzeń

Metoda Forda-Fulkersona wykorzystana w problemie znajdowania maksymalnego skojarzenia będzie więc znajdowała coraz to większy w sensie inkluzji zbiór dróg od s do t. *Ścieżką powiększającą* będziemy nazywali taką ścieżkę od s do t, która pozwoli nam na powiększenie zbioru dróg (zależność między tą ścieżką a zbiorem dróg podamy później).

Ogólny szkic algorytmu Forda-Fulkersona wygląda następująco:

Zainicjowanie zbioru dróg na 0
while istnieje ścieżka powiększająca p
    do powiększ zbiór dróg wzdłuż p
return(zbiór dróg)

Przyjrzyjmy się teraz w jaki sposób znajduje się ścieżkę powiększającą oraz jak na jej podstawie zmienia się aktualny zbiór dróg. W tym celu z naszego grafu skojarzeń zbudujemy graf przepływu.

Graf przepływu $G_p=(Y,P)$ jest grafem skierowanym, którego łuki spełniają zależności:

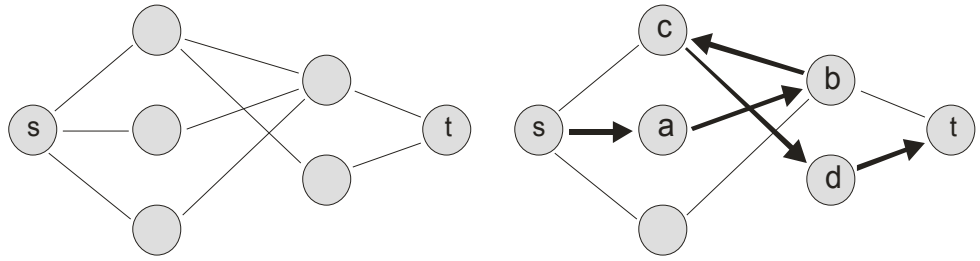
- dla każdych 2 wierzchołków a,b , które w grafie skojarzeń znajdują się na tej samej drodze od $s\rightarrow a\rightarrow b\rightarrow t$, do zbioru łuków P dokładamy łuki (s,a) , (a,b) , (b,t).
- wszystkie pozostałe krawędzie grafu G_s dodają do zbioru P grafu G_p łuki skierowane od strony prawej do lewej, czyli albo łuki od t do wierzchołków z V albo łuki od V do U lub od V do s.

Twierdzenie

Jeśli w grafie dualnym ${}^dG_p$ istnieje ścieżka od $s\rightarrow t$ wówczas w grafie skojarzeń istnieje większy przepływ.

Dowód: Przypatrzmy się rysunkowi nr 22. Jeśli omawiana ścieżka byłaby długości 3, np. $s\rightarrow n\rightarrow m\rightarrow t$, wówczas moglibyśmy w aktualnym grafie skojarzeń wybrać połączenie wierzchołków n i m zwiększając jednocześnie przepływ w grafie, jeżeli natomiast ścieżka ta byłaby dłuższa, oznaczałoby to, że musiały na niej wystąpić przejścia od V do U. Każde takie przejście usuwa jedno skojarzenie wierzchołka z V z wierzchołkiem z U, ale jeśli ścieżka ma dojść do t, wówczas będzie wtedy dodane jeszcze jedno skojarzenie U z V. W rzeczywistości przechodząc z wierzchołka $a\in U$ do $b\in V$ dodajemy skojarzenie (ab), natomiast przechodząc z $b\in U$ do $a\in V$ usuwamy skojarzenie (ab).

Pozostaje nam jeszcze rozpatrzyć przejścia do wierzchołków s oraz t, jednak możemy je zaniedbać, bowiem dla nowo otrzymanego układu skojarzeń połączenie s z U oraz V z t możemy wygenerować od nowa. Powstaje nam nowy graf skojarzeń G_s dla którego powtarzamy powyższy algorytm do momentu, gdy nie będzie istniała ścieżka powiększająca od s do t. Szkic działania algorytmu przedstawia rysunek nr 22.

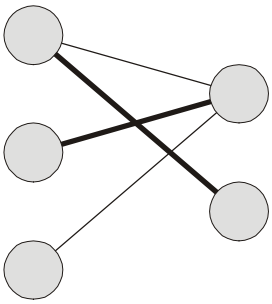


Graf przepływów

Graf dualny do grafu przepływów
i ścieżka powiększająca (sabcdt)

Rysunek 22 Graf przepływów i ścieżka powiększająca

Brak ścieżki powiększającej dla aktualnego grafu skojarzeń oznacza, że znaleźliśmy już maksymalne pod względem inkluzji skojarzenie w grafie dwudzielnym G, co przedstawia rysunek nr 23.



Maksymalne skojarzenie

Rysunek 23 Rezultat działania algorytmu Forda – Fulkersona

III. 4 Najkrótsze ścieżki z jednym źródłem

Problem znajdowania najkrótszej ścieżki z jednym źródłem odnosi się bez wątpienia do grafów ważonych G=(X,E) stopnia n, czyli takich, że istnieje funkcja wagi *f* która dla każdego łuku ze zbioru E jednoznacznie określa jego wagę. Zakładając, że dla każdego łuku (a,b) zachodzi $f((a,b)) \geq 0$, do znalezienia najkrótszych ścieżek od jednego źródła najczęściej wykorzystuje się algorytm Dijkstry.

Przedstawiony poniżej algorytm jest algorytmem zachłannym, mimo to zawsze daje nam optymalne rozwiązanie.

W algorytmie Dijkstry wykorzystujemy następujące zmienne:

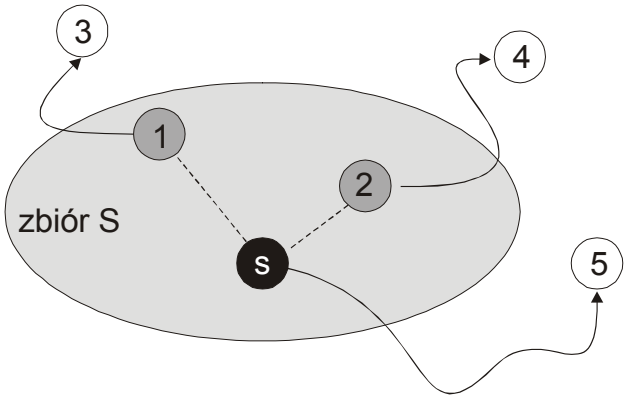
- $s \in X$ jest wierzchołkiem startowym (jedynym źródłem)
- d[1..n] Tablica o wartościach dodatnich (ew. zerowych). Reprezentuje ona aktualnie minimalne odległości wszystkich wierzchołków od źródła. Po zakończeniu algorytmu jeśli d[i] = w, to $\delta(s,i) = w$.
- zbiór S wierzchołków, dla których została już znaleziona minimalna odległość od źródła.
- tablica p[1..n]- nie jest konieczna do obliczenia minimalnych odległości od źródła, jednak wykorzystujemy ją do odtworzenia minimalnej ścieżki ; jeśli p[i] = j, oznacza to, że na tej ścieżce poprzednikiem wierzchołka i jest wierzchołek j.

Sam algorytm przedstawia się następująco:

for i ← 1 **to** n **do**
    d[i] ← ∞
    p[i] ← s *Inicjacja źródła*
d[s] ← 0
S ← {s}

while X ≠ S **do**
    v ← Minimal(X-S)
    S ← S ∪ {v}
    for każdy wierzchołek u sąsiadujący z v **do**
        Relaksacja(u,v)

Na początku następuje inicjalizacja źródła s. W tablicy d tylko d[s] = 0, ponieważ znajdujemy się w źródle. Występującą w algorytmie pętla *while* będzie wykonywana n-1 razy, dodając przy każdym jej przejściu jeden wierzchołek do zbioru S. Pierwszą instrukcją tej pętli jest wybranie ze zbioru X–S wierzchołka v, który jest najbliżej źródła, co obrazuje rysunek nr 24.



Rysunek 24 Znajdowanie wierzchołka minimalnego

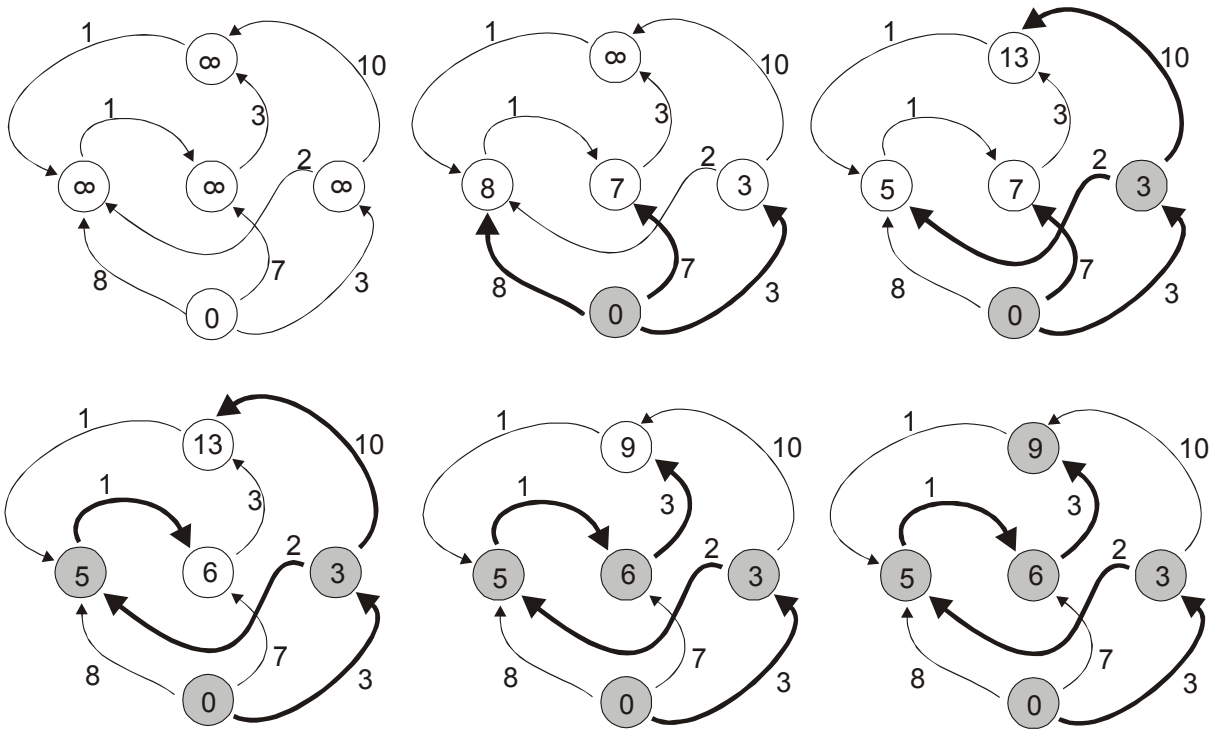
Przy znajdowaniu wierzchołka minimalnego musimy brać pod uwagę odległości „kandydatów” od źródła s. Dysponując tablicą d, wybieramy z niej najmniejszy element, który nie należy jeszcze do zbioru S.

Następna instrukcja pętli *while* dodaje do zbioru S wierzchołek minimalny v. Teraz pozostaje nam tylko dla każdego sąsiada u wierzchołka v wykonać funkcję relaksacji, która przedstawia się następująco:

Relaksacja(u,v)
if d[u] > d[v] + f(v,u) **then**
d[u] ← d[v] + f(v,u)

p[u] ← v

Działanie algorytmu na przykładowym grafie przedstawia rysunek nr 25.



Rysunek 25 Działanie algorytmu Dijkstry

Na powyższym rysunku wierzchołki zaciemnione należą do zbioru S, a pogrubione krawędzie reprezentują aktualne minimalne ścieżki do wszystkich wierzchołków.

Do wypisania wierzchołków leżących na minimalnej ścieżce od źródła s do wierzchołka v służy funkcja rekurencyjna:

```
Wypisz(s,v)
u←p[v]
if u≠s then Wypisz(s,u)
```

By wypisać także wierzchołki krańcowe tej ścieżki należy wywołać:

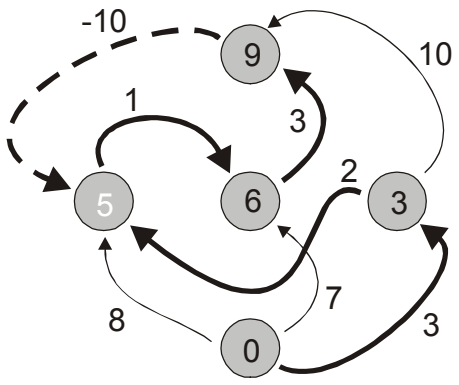
```
Print(s)
Wypisz(s,v)
Print(v)
```

Algorytm Dijkstry rzeczywiście daje optymalne rozwiązanie, dla każdego wierzchołka v zachodzi: $d[u] = \delta(s,u)$.

Dowód: Załóżmy, że powyższa zależność nie zachodzi, to znaczy istnieje taki wierzchołek q, że $d[q] > \delta(s,q)$. Oznacza to, że algorytm **najpierw** wygenerował najkrótszą ścieżkę od s do q, a **później** znalazł krótszą ścieżkę, która nie została uwzględniona. Oznaczmy **pierwszą** ścieżkę jako $sx_1x_2..x_nq$, a **drugą** jako $sy_1y_2..y_mq$. Zatrzymajmy się w momencie *t*, gdy w pętli while algorytm wybrał wierzchołek minimalny q, którego poprzednikiem był x_n.

Jeśli rzeczywiście znaleziona **później** druga ścieżka jest krótsza, to każda jej pod-ścieżka $sy_1y_2..y_j$ gdzie $j \leq m$ jest krótsza od pierwszej ścieżki. Oznacza to, że w etapach algorytmu poprzedzających *t* do zbioru S będą dodawane wierzchołki $y_1..y_m$ i ostatecznie druga ścieżka zostanie znaleziona w pierwszej kolejności, a poprzednikiem q będzie y_m co daje sprzeczność.

Algorytm Dijkstry nie będzie działał prawidłowo, gdy w grafie skierowanym będą istniały łuki ujemne, bowiem wówczas po n krokach będzie zachodziła równość S = X a tablica d będzie zawierała wartości, które niekoniecznie będą wartościami najkrótszych ścieżek. Posłużmy się grafem z rysunku nr 25 zmieniając wartość krawędzi (9,5) na –10.



Rysunek 26 Niewykonalność algorytmu Dijkstry

Widzimy, że gdy będziemy odwiedzać kolejno wierzchołki 5,6,9,5 .. wówczas wartości w tablicy d dla tych wierzchołków będą stawały się coraz mniejsze, a algorytm tego nie uwzględni.

Możemy wprowadzić modyfikację do algorytmu Dijkstry, tzn. stosując zamiast zbioru S, kolejkę Q wierzchołków, które należy jeszcze odwiedzić. Zmodyfikowany algorytm będzie wyglądał następująco:

for i ← 1 **to** n **do**
    d[i] ← ∞
    p[i] ← s
d[s] ← 0
Q ← {s}

repeat
    x ← *DeQueue*(Q)
    for każdy wierzchołek u sąsiadujący z x **do**
        if d[u] > d[x] + f(x,u) **then**
        d[u] ← d[x] + f(x,u)

            p[u] ← x
            if u∉Q **then** *EnQueue*(Q,u)
until Q jest pusta

W powyższym algorytmie kursywą zostały zaznaczone operacje DeQueue(Q) oraz EnQueue(Q,u) odpowiednio usuwające z kolejki pierwszy element oraz dodające na koniec kolejki element u.

Podzielmy kolejne etapy wykonywania algorytmu na *„kroki”*. Podczas pierwszego kroku ma miejsce inicjalizacja kolejki, czyli dodanie do niej wierzchołka s.

Dla k>0, krok k jest jednym przejściem pętli *repeat*, czyli wszystkimi operacjami wykonywanymi na wierzchołkach znajdujących się w kolejce pod koniec kroku k-1. Prawdziwe jest twierdzenie:

jeśli istnieje ścieżka minimalna s → x zawierająca k łuków, wówczas na początku kroku k d(x) jest długością tej ścieżki.

Dowód indukcyjny ze względu na wartość k:

k=0, wtedy za x możemy przyjąć tylko źródło, które w kroku 0 będzie miało poprawnie obliczone d(s) = 0

Załóżmy, że twierdzenie jest prawdziwe dla k. Udowodnimy, że jest prawdziwe dla k+1. Jeśli wiemy, że istnieje wierzchołek y oraz ścieżka s → x zawierająca k+1 łuków, wówczas będzie istniał wierzchołek x, łuk (x,y), oraz

dla wierzchołka x będzie istniała ścieżka s → y zawierająca k łuków. Skorzystawszy z założenia indukcyjnego widzimy, że na początku kroku k zostaje obliczone d(x), co gwarantuje nam, że w kroku k+1 w kolejce znajdzie się wierzchołek x. W kolejnym kroku algorytmu zostanie obliczona wartość d[x], co kończy dowód.

III. 5 Najkrótsze ścieżki z wieloma źródłami

Często mając skierowany graf G chcielibyśmy znaleźć najkrótsze odległości między dowolnymi dwoma wierzchołkami. Problem taki możemy spotkać, gdy np. mając mapę drogową chcemy podać odległości łączące dowolne miasta na tej mapie.

Teoretycznie problem ten moglibyśmy rozwiązać stosując algorytm Dijkstry kolejno dla każdego wierzchołka grafu G, to znaczy każdorazowo podstawiając za źródło kolejny wierzchołek grafu. Rozwiązanie to jednak nie jest stosowane ze względu na długi czas działania algorytmu. Do znalezienia minimalnych ścieżek między dwoma dowolnymi wierzchołkami grafu skierowanego, bez cykli ujemnych G stosuje się często algorytm **Floyda-Warshalla**.

Sam algorytm jest prosty i czytelny w zapisie, pozwala na znalezienie minimalnych i ich długości między dowolną parą 2 wierzchołków, jednak przekonanie się o jego poprawności nie jest natychmiastowe.

Standardowo na wejściu algorytmu mamy macierz $H=(h_{ij})$ rozmiarów $n \times n$ grafu G, tzn. $h_{ij}=k$, jeśli istnieje łuk (i,j) długości k

$h_{ij}=\infty$, jeśli nie ma łuku (i,j)

Algorytm znajduje kolejne macierze $D^{(i)}$ dla i=0..n , gdzie $D^{(0)}$ jest macierzą H, natomiast $D^{(n)}$ to macierz minimalnych odległości, która jednocześnie jest macierzą grafu G^t będącego domknięciem tranzytywnym grafu G.

Algorytm Floyda-Warshalla wygląda następująco:

```
n←Rozmiar(H)
D(0)←H
for k←1 to n do
        for i←1 to n do
                for j←1 to n do
                        d(k)ij ← min{d(k-1)ij , d(k-1)ik + d(k-1)kj}
return D(n)
```

Jak widać algorytm dla kolejnych wartości k, przegląda każdą możliwą ścieżkę i→j i sprawdza czy przechodząc przez k, czyli i→k→j nie otrzymamy ścieżki krótszej.

Jeśli chcielibyśmy odtworzyć minimalną ścieżkę, wówczas istnieje potrzeba pamiętania macierzy poprzedników S, która na początku będzie przyjmowała wartości: $s_{ij} = i$ jeśli istnieje łuk (i,j)

$s_{ij} = Nil$ jeśli nie istnieje łuk (i,j)

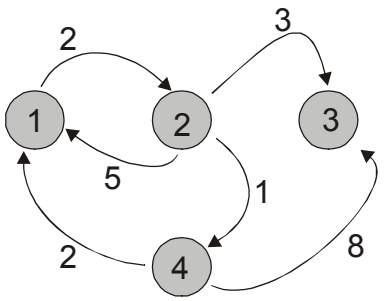
Modyfikacje wartości macierzy S będą następowały zaraz po wykonaniu w algorytmie instrukcji $d^{(k)}_{ij} \leftarrow \min\{d^{(k-1)}_{ij}, d^{(k-1)}_{ik} + d^{(k-1)}_{kj}\}$. Jeśli funkcja min wybierze drugi składnik, wówczas dodamy instrukcję $s_{ij} \leftarrow k$. Po zakończeniu działania algorytmu ścieżkę a→b będziemy mogli odtworzyć wywołaniem:

```
if s_ab ≠ nil then
        Pokaż(a,b)
        print(b)
else nie istnieje ścieżka a→b
```

gdzie Pokaż(x,y) jest funkcją rekurencyjną zdefiniowaną następująco:

```
Pokaż(i, j)
        k←s_ij
        if i=k then print(k)
        else
Pokaż(i, k)
Pokaż(k+1, j)
```

Na rysunku nr 27 Pokazany jest przykładowy graf, dla którego zbudujemy ciąg macierzy D, oraz macierz S.



Rysunek 27 Graf dla algorytmu Floyda - Warshalla

$D^{(0)}$

0	2	∞	∞
5	0	3	1
∞	∞	0	∞
2	∞	8	0

$D^{(1)}$

0	2	∞	∞
5	0	3	1
∞	∞	0	∞
2	4	8	0

$D^{(2)}$

0	2	5	3
5	0	3	1
∞	∞	0	∞
2	4	7	0

S

Nil	1	Nil	Nil
2	Nil	2	2
Nil	Nil	Nil	Nil
4	Nil	4	Nil

S

Nil	1	Nil	Nil
2	Nil	2	2
Nil	Nil	Nil	Nil
4	1	4	Nil

S

Nil	1	2	2
2	Nil	2	2
Nil	Nil	Nil	Nil
4	1	2	Nil

$D^{(3)}$

0	2	5	3
5	0	3	1
∞	∞	0	∞
2	4	7	0

$D^{(4)}$

0	2	5	3
3	0	3	1
∞	∞	0	∞
2	4	7	0

S

Nil	1	2	2
2	Nil	2	2
Nil	Nil	Nil	Nil
4	1	2	Nil

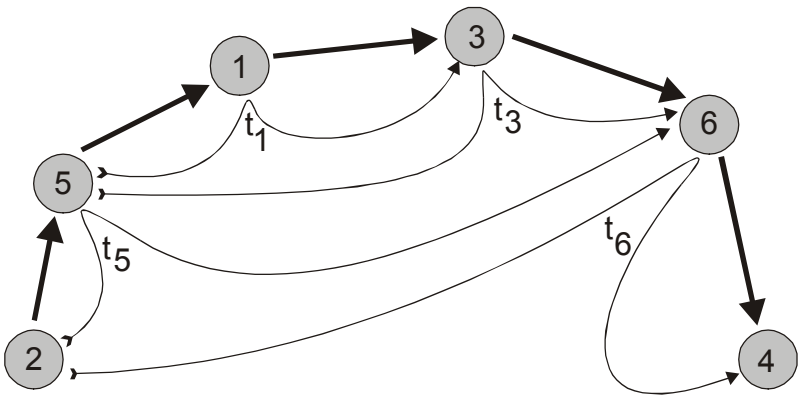
Nil	1	2	2
4	Nil	2	2
Nil	Nil	Nil	Nil
4	1	2	Nil

Jak łatwo zauważyć, potrójne pętle for pociągają za sobą złożoność obliczeniową algorytmu Floyda – Warshalla rzędu $O(n^3)$.

Dowód poprawności

Załóżmy, że w grafie skierowanym bez cykli ujemnych G=(X,E) będzie istniała minimalna ścieżka (ax_1x_2 .. x_kb) łącząca 2 wierzchołki a i b. Ponumerujmy kolejno wszystkie wierzchołki $x \in X$. Wybierzmy ze zbioru { x_1,x_2, .., x_k } wierzchołek x_j o największym numerze. Pod-ścieżki znajdujące się na ścieżce minimalnej też są minimalne, dlatego będziemy mieli $a \rightarrow b = a \rightarrow x_j \rightarrow b$. Oznaczmy czas operacji rozbicia ścieżki minimalnej przez t_j. Wśród wierzchołków leżących wewnątrz nowo powstałych ścieżek nie ma wierzchołków o numerze większym od x_j. Operację rozbicia ścieżek kontynuujemy dopóki nie otrzymamy ścieżek złożonych z pojedynczych łuków. Ponadto każde rozbicie oparte o wierzchołek x_i będzie następowało w czasie t_i, a numery wierzchołków będą malały, dlatego *odwracając sytuację*: będziemy w kolejnych odcinkach czasu (kolejnych etapach działania algorytmu) sprawdzali, czy dla dowolnej drogi $x \rightarrow y$ korzystniejsze jest przejście przez wierzchołek x_i czyli $x \rightarrow x_i \rightarrow y$ przy czym x_i będą posiadały coraz większe numery. Taki wybór

wierzchołków x_i gwarantuje nam pierwsza pętla for algorytmu. Podział ścieżki na pod-ścieżki obrazuje rysunek nr 28.



Rysunek 28 Podział ścieżki dla algorytmu Floyda – Warshalla

III. 6 Drzewa rozpinające

Dla każdego grafu spójnego G=(X,E) możemy skonstruować drzewo rozpinające T=(X,F), często jednak spotykamy się z grafami, w których krawędzie posiadają wagę, tzn. z grafem G utożsamiamy funkcję p: $E \to \Re$, która każdej krawędzi przypisuje wartość rzeczywistą. Każde drzewo rozpinające posiada zatem ściśle określoną wagę:

$P(T) = \sum_{e \in T} p(e)$.

Rozpatrzymy dwa algorytmy pozwalające na znalezienie minimalnego drzewa rozpinającego (do znalezienia drzewa maksymalnego wystarczy przyjąć funkcję wagową –p).

Wniosek 1

Każde drzewo rozpinające grafu G stopnia n, posiada n-1 krawędzi.

Dowód indukcyjny ze względu na n jest stosunkowo prosty. Lemat ten pozwala nam na wyciągnięcie następującego wniosku:

Wniosek 2

$\forall e \in E(G) - E(T)$ T+e posiada unikalny cykl

Dowód: Niech e=(x,y) oraz w drzewie T istnieje ścieżka $x \to y$. Dodając krawędź e, która nie leży na tej ścieżce tworzymy cykl. Jest to cykl elementarny, bowiem w drzewie T istnieje tylko jedna ścieżka łącząca x i y.

Cykl drzewa T+e oznaczmy jako $C_T(e)$

Wniosek 3

Drzewo rozpinające T grafu G jest minimalne $\Leftrightarrow \forall_{e\in E(G)-E(T)}\forall_{f\in CT(e)}\ p(f)\leq p(e)$

Dowód: $\Rightarrow$ Weźmy krawędź e=(x,y) grafu G, która nie należy do drzewa T. Dodając tę krawędź do tego drzewa otrzymamy unikalny cykl $C_T(e)$ oparty na ścieżce s i krawędzi e. Oznaczmy ten cykl przez $c=(ya_1a_2...a_nx)$. Jeśli krawędź e **nie jest największa**, wówczas w znalezionym cyklu wyodrębnijmy krawędź dłuższą f. Usuńmy teraz z cyklu c krawędź f. Otrzymamy ścieżkę s' łączącą wierzchołki $x,y,a_1,a_2, .., a_n$. Jeśli teraz zastąpimy w drzewie T ścieżkę s ścieżką s' otrzymamy nowe drzewo minimalne – sprzeczność.

Dowód w drugą stronę jest oczywisty.

Wniosek 4

Jeśli p jest iniekcją, wówczas drzewo rozpinające jest unikatowe.

Przedstawimy teraz 2 algorytmy znajdowania drzewa minimalnego:

III. 6. 1 Algorytm Kruskal'a

Tworzymy zbiór krawędzi grafu G uporządkowany według rosnących wag: $E_p = (e_1,e_2, ..., e_n)$. Na znalezienie drzewa minimalnego pozwala nam sekwencja:

$T_1 \leftarrow \varnothing$

$T_{i+1} \leftarrow T_i + e_k$

gdzie k jest najmniejszym indeksem j takim, że ej nie tworzy cyklu z krawędziami w T_i. T_n jest drzewem minimalnym.

Dowód:

Załóżmy, że T_n nie jest minimalne. Niech e będzie krawędzią nie należącą do drzewa T_n. Stąd istnieje cykl $C_T(e)$ oparty na ścieżce s taki, że $\exists_{f\in s}\ p(e)<p(f)$

Z drugiej strony algorytm działający na uporządkowanym zbiorze E^p zbudował drzewo z krawędzi krótszych od e, co daje sprzeczność.

Algorytm Kruskal'a jest przejrzysty i efektywny, bowiem każdą krawędź grafu rozpatrujemy tylko raz, wystarczy jedynie na początku posortować je rosnąco w czasie O(n logn). Zachodzi jednak potrzeba sprawdzania czy dodawana krawędź nie spowoduje powstania cyklu, co w niektórych implementacjach grafu jest czasochłonne, dlatego często stosuje się:

III. 6. 2 Algorytm Prim'a

Dla zbioru $X=\{x_1,x_2, ..., x_n\}$ wierzchołków grafu G mamy sekwencję:
$T_1 \leftarrow \varnothing$
$S_1 \leftarrow \{1\}$
$T_{i+1} \leftarrow T_i + e_i$
$S_{i+1} \leftarrow S_i + x_{i+1}$

gdzie $e_i = xx_{i+1}$ jest najmniejszą krawędzią łączącą jeden wierzchołek ze zbioru S_i z jednym wierzchołkiem ze zbioru $X-S_i$.

W każdym etapie j działania algorytmu Prima T_i jest częścią końcowego drzewa minimalnego T_n. Taka konstrukcja powoduje, że końcowe drzewo T_n spełnia *Wniosek 3*.

Rozdział IV Algorytmy numeracji

Algorytmy numeracji służą do przyporządkowywania wszystkim elementom badanej przestrzeni unikatowych numerów.

IV. 1 Słowa

Mamy dany zbiór A nazywany *alfabetem*, składający się z elementów: $A=\{a_1, a_2, a_3, ..., a_n\}$ gdzie: $a_1<a_2<a_3< ... <a_n$. Zakładamy, że dla każdego elementu zbioru A jesteśmy w stanie jednoznacznie określić element *następny* oraz *poprzedni*, co pozwala nam jednoznacznie określić element *pierwszy* i *ostatni* w alfabecie A. Dla alfabetu A zdefiniujemy więc 4 funkcje:

- $\text{Następny}_A(a_i) = a_j$, gdzie $a_i<a_j$ i $!\exists_k$, że $a_i < a_k < a_j$
- $\text{Poprzedni}_A(a_i) = a_j$, gdzie $a_i>a_j$ i $!\exists_k$, że $a_i > a_k > a_j$
- $\text{Pierwszy}_A = a_1$
- $\text{Ostatni}_A = a_n$.

Utwórzmy zbiór W_p *słów* długości p opartych na języku A. Łatwo zauważamy, że wszystkich możliwych słów jest n^p.

Przykład:

Dla alfabetu $A=\{a,b,c\}$ Zbiór $W_2=\{aa, ab, ac, ba, bb, bc, ca, cb, cc\}$

Dla uporządkowania elementów zbioru W_p i w konsekwencji ponumerowania jego elementów zastosujemy porządek leksykograficzny, który określa poniższa reguła:

Mamy 2 słowa: $m=b_1b_2b_3...b_p$, oraz $m'=c_1c_2c_3...c_p$

$m <_L m' \Leftrightarrow (\exists_{i\le p} \forall_{j<i}\ b_j=c_j$ oraz $b_i<c_j)$

Dla alfabetu A i powyższego porządku leksykograficznego, określamy stałe: $First=a_1^p$ oraz $Last = a_n^p$.

Algorytm korzysta z funkcji Next(m) , która jako parametr otrzymuje aktualne słowo, w naszym przypadku tablicę [1..p], a zwraca kolejne słowo w porządku leksykograficznym.

NUMERACJA_LEKSYKOGRAFICZNA(A,p)

```
M ← First
while M ≠ Last do
        Wypisz(M)
        M ← Next(M)
```

Pozostaje zdefiniować funkcję Next(M). M jest tablicą [1..p]

```
Next(słowo)
        i ← p
        while słowo[i] = Ostatni_A  do
                i ← i –1
        if i >0 then
                słowo[i] ← Następny_A(słowo[i])
                for j ← i+1 to p do
                        słowo[j] ← PierwszyA
                return(słowo)
        else return(Null)
```

Gdy słowo jest ostatnie, tzn. nie da się wygenerować kolejnego korzystając z alfabetu A, wówczas funkcja Next zwraca Null. Złożoność obliczeniowa tego algorytmu wynosi $O(n^p)$.

Rzadko spotyka się algorytmy numerujące elementy zbioru według porządku leksykograficznego dla dowolnej długości słowa, gdyż wówczas moglibyśmy tworzyć nieskończenie długie słowa, których komputer nie potrafiłby porównać.

IV. 2 Permutacje

Permutacja na zbiorze $A=\{a_1, a_2, a_3, ..., a_n\}$, gdzie $a_1<a_2<a_3< ... <a_n$ jest słowem długości n, którego poszczególne *litery* występują tylko raz.

$P_n = \{m \in W_n$, że $\forall_{i,j = 1..n}\ m[i] \neq m[j] \}$

Zauważamy, że $P_n \subseteq W_n$, oraz że wszystkich możliwych permutacji jest n!

Przykład:

A={a,b,c} P_3 = {abc, acb, bac, cab, cba}

Minimalne słowo w zbiorze P_n, to: $a_1a_2a_3...a_n$ = *Minimal*.

Maksymalne słowo w zbiorze P_n, to: $a_na_{n-1}a_{n-2}...a_1$.

Podobnie jak w rozdziale IV.1, możemy do numeracji elementów P_n użyć algorytmu leksykograficznego, lecz możemy także zastosować szybki algorytm rekurencyjny.

Zastosujemy funkcję Permutacja(a, b, Tab), gdzie

- Tab - jest tablicą [1..n] różnych elementów alfabetu A. Będzie to zmienna globalna, czyli należy do funkcji Permutacja przekazywać Tab przez referencję.
- a – jest lewą granicą tablicy, na której będziemy przestawiali elementy
- b – jest prawą granicą tablicy, na której będziemy przestawiali elementy.

Wywołanie Permutacja(a, b, Tab) wygeneruje wszystkie permutacje przedziału [a,b] tablicy Tab.

Potrzebujemy także funkcji Zamień(a,b) , która zamieni ze sobą pozycję dwóch elementów tablicy: Tab[a] i Tab[b].

Algorytm, który ponumeruje permutacje P_n bez uwzględnienia porządku leksykograficznego będzie wywoływany:

```
Tab ← Minimal
Permutacja(1,n,Tab)
```

Pozostaje nam jeszcze zdefiniować:

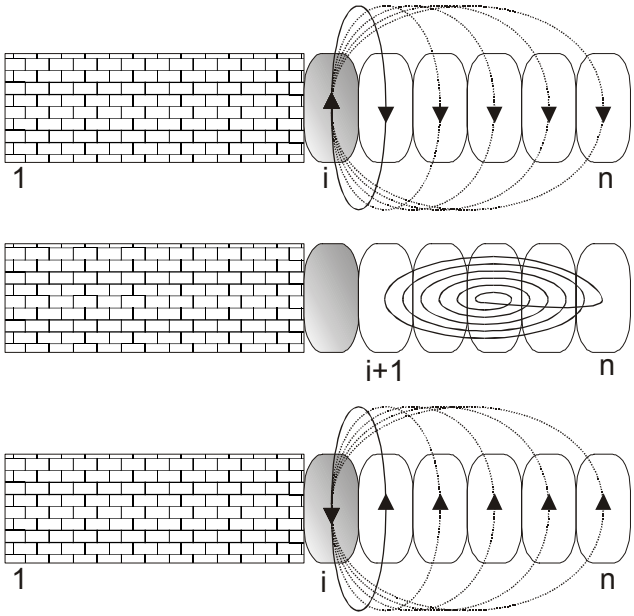
```
Permutacja(i,n,Tab)
if i=n then
        Wypisz(Tab)
else
        for j ← i to n do
                Zamień(Tab[i], Tab[j])
                Permutacja(i+1, n, Tab)
                Zamien(Tab[i], Tab[j])
```

Zauważmy, że wywołanie Permutacja(i, n, Tab) będzie zmieniało zawartość tablicy Tab na przedziale [i..n], natomiast zawartość tablicy Tab na przedziale [1..i-1] nie ulegnie zmianie. Udowodnimy, że powyższe wywołanie wygeneruje wszystkie możliwe permutacje przedziału [i..n].

Dowód indukcyjny ze względu na długość przedziału:

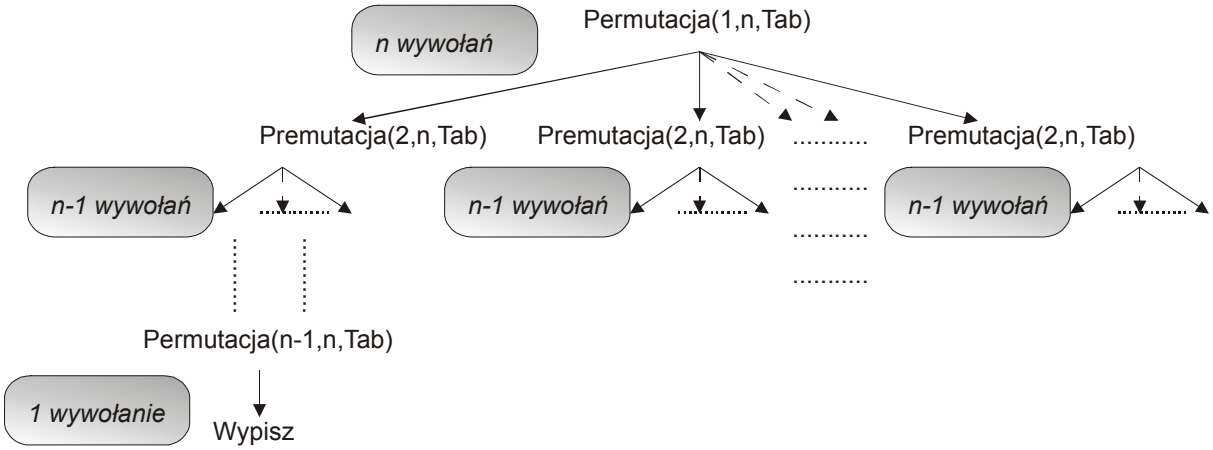
1. Jeśli przedział ma długość 1, wówczas istnieje tylko jedna permutacja tego przedziału, czyli nastąpi wypisanie Tablicy.

2. Zakładamy, że dla przedziału [i+1 .. n] długości n-i-1 algorytm wygeneruje wszystkie permutacje. Chcemy udowodnić, że algorytm wygeneruje wszystkie permutacje dla przedziału [i .. n]. W pętli **for** funkcji „Permutacja” wywołamy n-i razy funkcję Permutacja dla przedziału [i+1 .. n] podstawiając kolejno element Tab[i] na każdą z pozycji i+1 .. n tablicy Tab. Z założenia indukcyjnego, każde pojedyncze wywołanie Permutacja(i+1, n, Tab) wygeneruje nam wszystkie permutacje przedziału [i+1 .. n] tablicy Tab, stąd algorytm rzeczywiście wygeneruje wszystkie permutacje na całym przedziale [i .. n] długości n-i. Szkic rozumowania przedstawiony jest na rysunku nr 29.



Rysunek 29 Numerowanie permutacji

Do policzenia złożoności obliczeniowej powyższego algorytmu zbudujmy drzewo wywołań rekurencyjnych – rysunek nr 30.



Rysunek 30 Drzewo wywołań rekurencyjnych przy generowaniu permutacji

Na rysunku nr 30 widzimy, że ilość liści w drzewie, to n*(n-1)* ... 1 = n! . Jeśli algorytm wypisujący pojedynczą permutacje ma złożoność O(n), wówczas złożoność obliczeniowa rekurencyjnego algorytmu numeracji permutacji, to O(n*n!).

IV. 3 Kod Gray'a

Mówimy, że ciąg permutacji opartych na zbiorze A jest wygenerowany przez Kod Gray'a, jeśli każda następna permutacja różni się od poprzedniej zamianą pozycji dwóch sąsiednich elementów.

Przykład:

A={a,b,c} Zbiór permutacji = {abc, acb, cab, cba, bca, bac}

Twierdzenie:

Dla każdego n istnieje Kod Gray'a generujący P_n, czyli potrafiący wygenerować wszystkie permutacje oparte na zbiorze n elementowym.

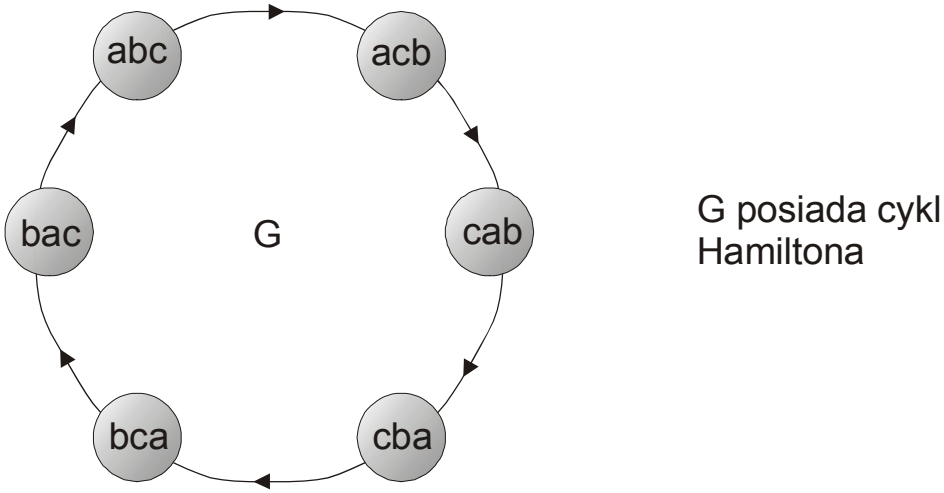
Rozpatrzmy graf $G(P_n)=(P_n, E)$ zdefiniowany w następujący sposób: $\forall$ P, P'$\in P_n$

$(P,P')\in E \quad \Leftrightarrow$ P jest permutacją powstałą z P' przez przestawienie 2 sąsiednich elementów

Twierdzenie:

Istnieje Kod Gray'a dla permutacji P_n wtedy i tylko wtedy, gdy $G(P_n)$ posiada drogę Hamiltona.

Twierdzenie to jest oczywiste, wystarczy spojrzeć na przykład z rysunku nr 31.

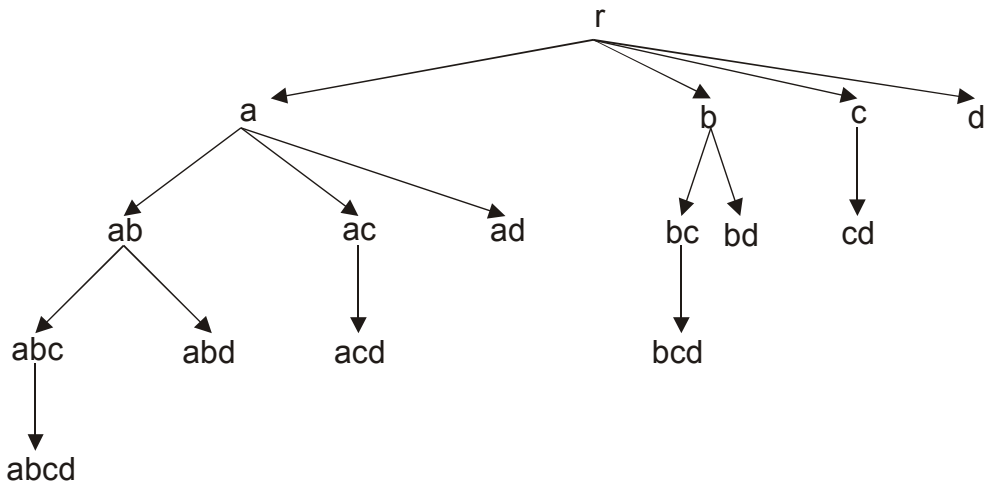


Rysunek 31 Kod Gray'a i cykl Hamiltona

Z kodem Gray'a spotykamy się również, gdy mówimy o generowaniu zbioru podzbiorów danego zbioru A. Przedstawimy dwa algorytmy oparte na kodzie Gray'a , pozwalające na wygenerowanie ciągu podzbiorów zadanego zbioru A w taki sposób, by dwa kolejne elementy tego ciągu były zbiorami różniącymi się *1 elementem*.

Jeśli zbiór A posiada k elementów, wówczas ma on 2^k podzbiorów. Niech $A=\{a_1, a_2, ..., a_n\}$. Z każdym podzbiorem P będziemy wiązali tablicę binarną tab[1..n] tak, by zachodziła zależność $a_i \in P \Leftrightarrow tab[i] = 1$.

Pierwszy algorytm buduje drzewo, którego wierzchołkami są podzbiory A, ponadto kolejno dodawane wierzchołki są ułożone w porządku leksykograficznym. Każde 2 wierzchołki są więc od siebie różne, a na każdej drodze od korzenia drzewa do liścia znajdują się podzbiory różniące się tylko o 1 element . Rysunek nr 32 przedstawia drzewo dla A={a,b,c,d}.



Rysunek 32 Uporządkowane drzewo podzbiorów

W algorytmie aktualna permutacja będzie reprezentowana przez stos S. Elementy znajdujące się na stosie będą elementami aktualnego podzbioru. Na początku S={a}. Każdy nowy stan stosu będzie nowym podzbiorem. Operacja Top_S zwróci wartość na wierzchołku stosu. Operacja Pop_S zdejmie element ze stosu. Operacja $Push_S(X)$ umieści na stosie element X. Algorytm wygląda następująco:

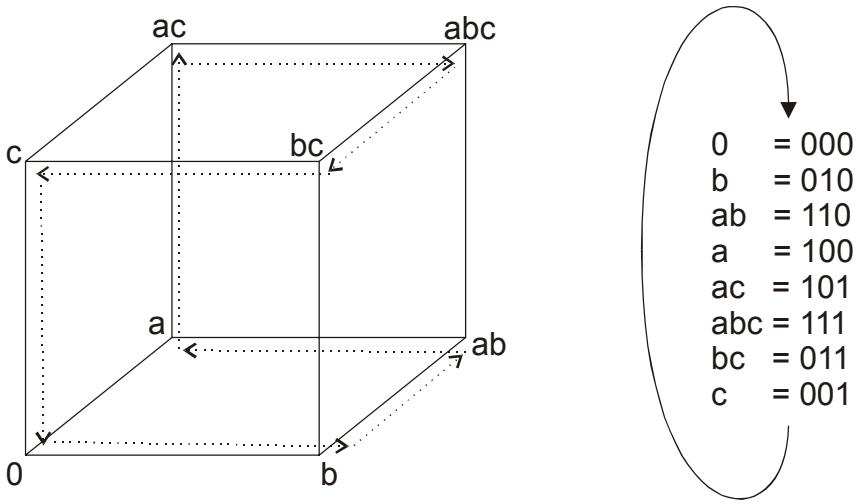
while S ≠∅ **do**
    Wypisz(S)
    if Top_S < $Ostatni_A$ **then**
        $Push_S(Następny_A(Top_S))$
    else
        Pop_S
        if S ≠∅ **then**
            $Push_S(Następny_A(PopS))$

Pozostaje jeszcze wypisać korzeń, czyli podzbiór pusty.

Drugi z algorytmów wygeneruje ciąg złożony z *wszystkich* podzbiorów A reprezentowanych przez tablice *tab* w taki sposób, by kolejne tablice różniły się na jednym bicie. W rezultacie dla grafu G(P(A)) = G(P(A), E) takiego, że $\forall X,Y \in P(A)$

$(X,Y) \in E \Leftrightarrow$ (X i Y różnią się jednym elementem)

będzie istniała droga Hamiltona, co przedstawia rysunek nr 33.



Rysunek 33 Podzbiory i cykl Hamiltona

Powyższy schemat numeracji można uzyskać posługując się następującym algorytmem rekurencyjnym:

$$G(1) = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \qquad G(n+1) = \begin{bmatrix} 0 & G(n) \\ 1 & G(n)^R \end{bmatrix}$$

Macierz G(1) jest wyjściowa. Macierz G(n+1) tworzymy z macierzy G(n) w następujący sposób: odkładamy kolejno, macierz G(n) a po niej macierz $G(n)^R$ – odbicie horyzontalne macierzy G(n). Do nowo powstałej macierzy dodajemy nową pierwszą kolumnę – wektor składający się z 2^n zer i 2^n jedynek. Zauważamy, że kolejne wiersze nowo powstałej macierzy G(n+1) różnią się między sobą na jednym bicie. W części G(n) i $G(n)^R$ z założenia wiersze różniły się na jednym bicie. Wiersze 2^n, oraz 2^{n+1} różnią się tylko na pierwszym elemencie.

Ponumerujmy teraz kolumny macierzy G(n) od prawej do lewej. Dla każdego n, zbudujmy ciąg T(n) przekształceń kolejnych wierszy G(n) (jeśli np. wiersz 011 zmienia się na 010, wówczas zmieniliśmy 2-gi bit, czyli do T(n) dodajemy 2). Piszemy zatem:

T(1) = (1)
T(2) = (1,2,1)
T(3) = (1,2,1,3,1,2,1)
..... *(wnioskujemy, że dla dowolnego n zachodzi)*
T(n) = (T(n-1), n, T(n-1))

Dzięki temu możemy podać złożoność obliczeniową powyższego algorytmu generowania kodu Gray'a, która wynosi $O(2^n)$

Kolejnym problemem jest znalezienie *następnej* permutacji w porządku leksykograficznym, to znaczy, że mając zbiór $A=\{a_1, a_2, a_3, ..., a_n\}$, gdzie $a_1<a_2<a_3<...<a_n$ i pewną permutację elementów $p = (a_{i1}a_{i2} ... a_{in})$ szukamy następnej permutacji $p' = (a_{j1}a_{j2} ... a_{jn})$.

Przykład:
A={1,2,3,4,5} p=31542 p' = 32145

Algorytm:

Załóżmy, ze p będzie permutacją przedstawioną w tablicy tab[1..n]. W celu znalezienia kolejnej permutacji, wyszukujemy najdłuższy rosnący ciąg elementów w p, począwszy od strony prawej do lewej, który oznaczamy (tab[k],tab[k+1], ..., tab[n]). W przykładzie ciągiem tym będzie (5,4,2). Następny element tab[k-1] w permutacji p jest mniejszy od tab[k], czyli możliwe jest wygenerowanie następnej permutacji. Aby to zrobić zamieniamy elementy tab[k-1] z tab[n], a następnie sortujemy rosnąco elementy od pozycji k do n, tak by tab[k] < tab[k+1] < ... < tab[n]. W ten sposób powstaje nam kolejna (w porządku leksykograficznym) permutacja p'. W najbardziej niekorzystnym przypadku będziemy sortowali n elementów, a jak wiadomo wszystkich permutacji jest n!, dlatego stosując do sortowania algorytm *quicksort* o złożoności obliczeniowej O(n*log(n)) otrzymamy ostateczną złożoność obliczeniową algorytmu znajdowania kolejnych permutacji: O(n!*n*log(n)).

Rozdział V Funkcje rekursywne

V. 1 Funkcje rekursywnie prymitywne

Funkcjami rekursywnie prymitywnymi nazywamy te funkcje, które analizuje bezpośrednio komputer. Konstrukcja układów scalonych pozwala na wygenerowanie podstawowych funkcji, czyli tzw. **funkcji bazowych**, wśród których wyróżniamy:

- funkcję **stałą** z N w N, której wartość stale jest równa 0.
- funkcję **następnika** oznaczaną *suc*, która ze zmienną wejściową x wiąże wartość x+1.
- funkcję projekcji pr_p^i przestrzeni N^p w N zdefiniowaną jako $pr_p^i(x_1,...x_p) = x_i$ dla i=1, 2, ..., p.

Definiujemy *f* jako **funkcję złożoną** z funkcji g_1, g_2, ..., g_p odwzorowujących przestrzeń N^n w N oraz funkcji *h* z przestrzeni N^p w N w następujący sposób:

$f(x_1, ..., x_n) = h[g_1(x_1, ..., x_n), ..., g_p(x_1, ..., x_n)]$

Schemat rekurencji prymitywnej wiąże z dwiema funkcjami *g* oraz *h* mającymi odpowiednio p oraz p+2 argumentów, funkcję p+1 argumentową *f* zdefiniowaną w sposób następujący:

- $f(x_1, ..., x_p,0) = g(x_1, \ldots, x_p)$
- $f(x_1, \ldots, x_p, y+1) = h[x_1, \ldots, x_p, y, f(x_1, \ldots, x_p,y)]$

W powyższym przypadku funkcja f została zdefiniowana przez **rekurencję** przy użyciu funkcji inicjującej g oraz funkcji h będącej etapem rekurencji.

Przykłady:

Przy użyciu rekurencji i złożenia możemy, korzystając z funkcji projekcji i następnika skonstruować funkcję:

Dodawania (a+b) oznaczmy jako +(a,b):

$$+(x,0) = pr_1^1(x)$$
$$+(x,y+1) = pr_3^3(x,y,+(x,y)) + 1$$

Mnożenia (a*b) oznaczamy jako *(a,b):

$$*(x,0) = C_0$$
$$*(x,y+1) = pr_3^3(x,y, *(x,y)) + x$$

gdzie C_0 jest funkcją stale równą 0.

Poprzednika liczby:

$$pred(0) = C_0$$
$$pred(k+1) = pr_2^1(k, pred(k))$$

gdzie $C_1 = succ(C_0)$

Zera, która dla zera przyjmuje wartość 1, a dla każdej innej liczby 0:

$$zero(0) = C_1$$
$$zero(k+1) = pred[pr_2^2(k, zero(k))]$$

Odejmowania symetrycznego:

$$n \div 0 = 0$$
$$n \div (k+1) = pred(pr_3^3(n, k, n \div k))$$

Definicje:

- zbiór T funkcji jest domknięty ze względu na proces konstrukcji, jeśli każda funkcja f zdefiniowana przy pomocy funkcji z T przy pomocy tego procesu jest także w T.
- zbiór **funkcji rekursywnie prymitywnych** jest najmniejszym spośród zbiorów funkcyjnych zawierających funkcje bazowe i jest on domknięty przez złożenie i rekurencję prymitywną.
- podzbiór A przestrzeni N^p nazywamy **rekursywnie prymitywnym**, jeśli jego **funkcja charakterystyczna** jest rekursywnie prymitywna.

Funkcją charakterystyczną podzbioru A przestrzeni N jest:

$$\chi_A(x) = \begin{cases} 1 \text{ gdy } x \in A \\ 0 \text{ w przeciwnym przypadku} \end{cases}$$

- Relacja R oparta na ciągach długości p jest **rekursywnie prymitywna**, jeśli zbiór $\{(x_1, ..., x_p) \in N^p : R(x_1, ..., x_p)\}$ jest rekursywnie prymitywny.

Aby pokazać, że funkcja jest rekursywnie prymitywna wystarczy sprawdzić czy może ona zostać otrzymana począwszy od funkcji bazowych przy wykorzystaniu złożeń i schematu rekurencji prymitywnej.

Przykład podzbiorów rekursywnie prymitywnych:

Każdy jednoelementowy podzbiór (np. {m}) zbioru liczb naturalnych jest rekursywnie prymitywny, gdyż możemy dla niego skonstruować następującą funkcję charakterystyczną, która będąc złożeniem funkcji prymitywnych, także będzie prymitywna:

$\chi_{\{m\}}(x) = zero\{ +(n \div x, x \div n)\}$

Widzimy, że jeśli $n \neq x$, wtedy funkcja + zwróci wartość większą od 0, co w wyniku działania funkcji zero da nam wartość 0. Jeśli $n = x$, wówczas funkcja charakterystyczna zwróci 1.

V. 2 Konstrukcja funkcji rekursywnie prymitywnych

Funkcje rekursywnie prymitywne możemy tworzyć na kilka sposobów:

Procesem najczęściej używanym w programowaniu jest **definiowanie przez przypadki**. Zakładamy, że mamy dane dwie p parametrowe funkcje rekursywnie prymitywne, oraz podzbiór rekursywnie prymitywny A przestrzeni N^p. Zdefiniowana poniżej funkcja h:

$$h(x_1, ..., x_p) = \begin{cases} f(x_1, ..., x_p) \text{ gdy } (x_1, ..., x_p) \in A \\ g(x_1, ..., x_p) \text{ w przeciwnym przypadku} \end{cases}$$

jest rekursywnie prymitywna.

W rzeczywistości jeśli weźmiemy χ_A oraz $\chi_{C(A)}$, funkcje charakterystyczne odpowiednio zbiorów A oraz jego dopełnienia C(A), to funkcja h może się wyrażać jako $h = f \bullet \chi_A + g \bullet \chi_{C(A)}$. Biorąc pod uwagę fakt że jest to suma, której składniki są iloczynami funkcji rekursywnie prymitywnych dostajemy funkcję rekursywnie prymitywną h.

Funkcją rekursywnie prymitywną jest suma i iloczyn graniczny wartości funkcji rekursywnie prymitywnych. Dla danej p+1 parametrycznej funkcji rekursywnie prymitywnej f, funkcje g oraz h zdefiniowane jako:

$$g(x_1, ..., x_p, y) = \sum_{t=0}^{y} f(x_1, \ldots, x_p, t)$$

$$h(x_1, ..., x_p, y) = \prod_{t=0}^{y} f(x_1, \ldots, x_p, t)$$

są także rekursywnie prymitywne. Dla przykładu pokażemy, że g może być zdefiniowana przez rekurencję:

$g(x_1, ..., x_p, 0) = f(x_1, ..., x_p, 0)$

$g(x_1, \ldots, x_p, y+1) = g(x_1, \ldots, x_p, y) + f(x_1, \ldots, x_p, y+1)$

Zbiór relacji rekursywnie prymitywnych jest domknięty przez **kwantyfikację ograniczoną**, tzn. jeśli A jest podzbiorem rekursywnie prymitywnym przestrzeni N^{p+1}, wówczas rekursywnie prymitywnymi są także zbiory B i C zdefiniowane jako:

$B = \{(x_1, ..., x_p, z) : \exists\, t \le z \;\; (x_1, ..., x_p, t) \in A\}$

$C = \{(x_1, ..., x_p, z) : \forall\, t \le z \;\; (x_1, ..., x_p, t) \in A\}$

ponieważ możemy dla nich zdefiniować funkcje charakterystyczne w następujący sposób:

$$\lambda_B(x1, ..., xp, z) = sg(\sum_{t=0}^{z} \chi_A(x_1, ..., x_p, t))$$

$$\lambda_C(x1, ..., xp, z) = \prod_{t=0}^{z} \chi_A(x_1, ..., x_p, t)$$

gdzie funkcja sg(x), taka że $sg(0) = 0$ i $sg(x) = 1$ gdy $x \neq 0$ jest także rekursywnie prymitywna, bowiem definiujemy ją jako:

$sg(x) = 1 \div zero(x)$

Zatem powyższe funkcje charakterystyczne zbiorów B i C są odpowiednio sg z sumy granicznej funkcji rekursywnie prymitywnych oraz iloczynu granicznego funkcji rekursywnie prymitywnych.

Do zdefiniowania funkcji rekursywnie prymitywnych możemy także użyć **schematu minimalizacji ograniczonej**. Niech zbiór rekursywnie prymitywny A będzie podzbiorem przestrzeni N^{p+1}. Zdefiniowana poniżej funkcja f jest także rekursywnie prymitywna:

$$f(x_1, ..., x_p, z) = \begin{cases} t \text{ ,najmniejsza wartość} \le z \\ \text{spełniająca } f(x_1, ..., x_p, t) \in A \\ \hline 0 \text{ w przeciwnym przypadku} \end{cases}$$

Jest to funkcja oznaczana niekiedy jako:

$f(x_1, ..., x_p, z) = \mu t \le z \;\; (x_1, ..., x_p, t) \in A$

Pokażemy, że funkcja f może być zdefiniowana przez rekurencję przy pomocy poprzedniego schematu, definicji przez przypadki oraz sumy ograniczonej:

$f(x_1, ..., x_p, 0) = 0$

$f(x_1, \ldots, x_p, z+1) = f(x_1, \ldots, x_p, z)$ gdy $\sum_{t=0}^{z} \chi_A(x_1, ..., x_p, t) \geq 1$

- ponieważ funkcja charakterystyczna relacji $\geq$ jest także rekursywnie prymitywna
- w przeciwnym wypadku ,gdy zachodzi $(x_1, ..., x_p, z+1) \in A$ mamy

$f(x_1, \ldots, x_p, z+1) = z+1$

- w każdym innym przypadku jest

$f(x_1, \ldots, x_p, z+1) = 0$

Przykłady:

Zbiory skończone

Udowodniliśmy już, że każdy jednoelementowy podzbiór przestrzeni N jest rekursywnie prymitywny. Teraz korzystając z sumy ograniczonej udowodnimy, że każdy skończony podzbiór przestrzeni N jest rekursywnie prymitywny. Mamy więc dany $A = \{m_1, ..., m_n\} \subset N$. Dla każdego m_i i=1..n mamy już zdefiniowaną funkcję charakterystyczną $\chi_A{}^i$. Dla zbioru A określamy więc funkcję charakterystyczną, jako:

$$\chi_A(x) = \sum_{i=0}^{n} \chi_A{}^i(x)$$

będącą funkcją rekursywnie prymitywną.

Podobnie dowodzimy, że jednoelementowy podzbiór A przestrzeni N^p jest rekursywnie prymitywny. Ostatecznie dla skończonego podzbioru A przestrzeni N^p twierdzimy, że jest on rekursywnie prymitywny przez zdefiniowanie dla niego funkcji charakterystycznej. Niech $A = \{c_1, ..., c_m\} \subset N^p$ gdzie kolejne $c_i = (x_{i1}, ... ,x_{ip})$. Dla każdego c_i mamy funkcję charakterystyczną $\lambda_A{}^i$, stąd poszukiwana funkcja charakterystyczna (dla wejściowego ciągu c) będzie miała postać:

$$\Psi_A(c) = \sum_{i=0}^{m} \lambda_A{}^i(c)$$

co można również zapisać jako:

$$\Psi_A(x_1, \ldots, x_p) = \sum_{i=0}^{m} \{ \prod_{j=1}^{p} \chi_A{}^j (pr_p{}^j (x_1, \ldots, x_p)) \}$$

Zbiory nieskończone

Udowodnimy, że dla danej liczby naturalnej $m > 1$, zbiór potęgowy $A=\{1, m, m^2, m^3 ...\}$ jest rekursywnie prymitywny. Na początku zauważmy, że zachodzi zależność $x < m^x$. Zdefiniujemy funkcję charakterystyczną częściową F, która będzie „akceptowała" skończoną ilość potęg m, tzn.:

$F(x,k) = \chi\{1,m^1,m^2, ..., m^k\}$ (x)

Konstruujemy funkcję

$$\chi(a,x)= \begin{cases} 1 \text{ gdy } x=a \\ 0 \text{ w przeciwnym przypadku} \end{cases}$$

w następujący sposób:

$\chi(a,x) = zero(+(a \div x, x \div a))$

jest to funkcja rekursywnie prymitywna zależna od 2 argumentów.

Funkcję F zdefiniujemy rekurencyjnie tak, by w nieskończoności akceptowała ona wszystkie potęgi m.

$F(x,0) = \chi(C_1(x),x)$

Krok inicjujący zależny jest tylko od x.

$F(x,k+1) = pr_3^3(x,k,F(x,k)) + \chi(exp_m(succ(k)), x)$

Krok rekurencyjny zależny jest tylko od x oraz k

Ostatecznie funkcja charakterystyczna dla zbioru A przyjmie postać:

$\Im(x) = F(x,x)$

ponieważ element $x < m^x$ będzie mógł być zaakceptowany przez jedną z funkcji F.

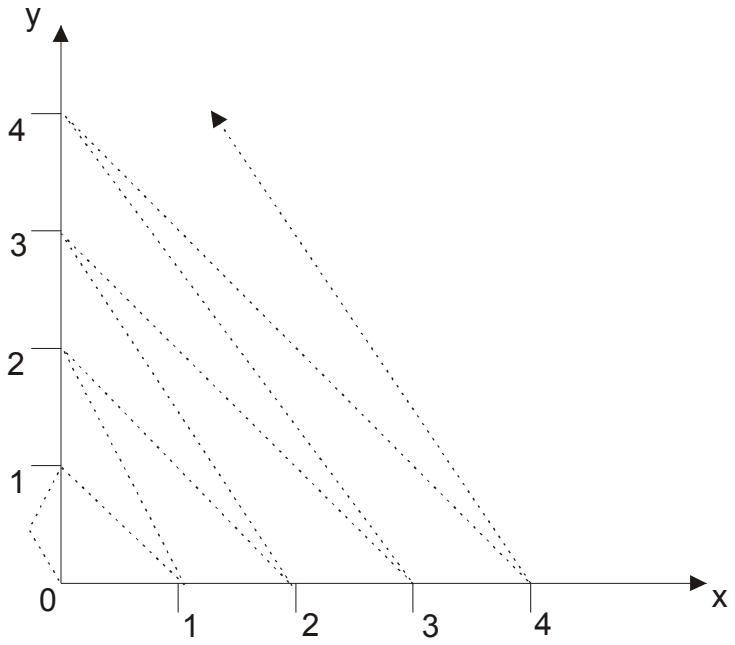
V. 3 Kodowanie ciągów

Umiejętność zredukowania problemu opierającego się na dwóch, lub skończonej ilości zmiennych do problemu jednej zmiennej jest w informatyce bardzo użyteczna. Redukcja taka pozwala na zapisanie skończonego ciągu liczb przy pomocy jednej liczby oraz odwrotnie: przywrócenie ciągu liczb na podstawie jednej liczby. W tym rozdziale udowodnimy istnienie bijekcji pomiędzy przestrzenią N a N^p .

Zaczniemy od znalezienia odwzorowania wzajemnie jednoznacznego pomiędzy przestrzenią N i N^2. Rozpatrzmy funkcję:

$\propto_2(x,y) = (x+y)(x+y+1)/2 + x$

jest to funkcja rekursywnie prymitywna. Odpowiada ona procesowi numerowania punktów płaszczyzny o współrzędnych naturalnych kolejnymi liczbami naturalnymi, co przedstawia rysunek nr 34.



Rysunek 34 Pokrycie płaszczyzny kolejnymi liczbami naturalnymi

Udowodnimy na początku, że dla dowolnych dwóch różnych punktów: (a,b) i (c,d) funkcja przyjmuje różne wartości:

Weźmy przypadek, gdy a+b = c+d, czyli obydwa punkty leżą na jakiejś prostej y=-x+w , w≥0 wówczas gdyby:

$\propto_2(a,b) = \propto_2(c,d)$

to:

$(a+b)(a+b+1)/2 + a = (c+d)(c+d+1)/2 + c$

czyli:

a=c oraz b=d, zatem punkty są takie same – sprzeczność.

Weźmy przypadek, gdy $a+b < c+d$ i oznaczmy $a+b=r_1$ $c+d = r_2$.

$\propto_2(a,b) = (a+b)(a+b+1)/2 + a = r_1(r_1+1)/2 + a \le r_1(r_1+1)/2 + {}_r1 = 1+2+ .. +r_1 + r_1 < 1+2+\ldots+r_2 \le r_2(r_2+1)/2 +c = (c+d)(c+d+1)/2 + c = \propto_2(c,d)$

ostatecznie $\propto_2(a,b) < \propto_2(c,d)$

podobny rezultat dostajemy, gdy a+b > c+d , czyli funkcja $\propto_2$ jest różnowartościowa.

Pozostaje nam jeszcze udowodnić, że dla każdej liczby naturalnej n zdołamy znaleźć parę liczb naturalnych x i y, aby spełnione było równanie:

$\propto_2(x,y) = n$

Dla każdej liczby n możemy tak dobrać liczby r_1 i r_2, by n znalazło się w środku przedziału $[1+2+ ... + r_1, 1+2+ ... +r_2]$, innymi słowy zachodzi:

$r_1(r_1+1) \leq n \leq r_2(r_2+1)$,

możemy więc dobrać takie δ by było spełnione równanie:

$r_1(r_1+1) + \delta = n$.

Aby ze zmiennych r_1 i δ przejść do poszukiwanych x i y wystarczy rozwiązać układ równań:

$$\begin{cases} x + y = r_1 \\ x = \delta \end{cases}$$

co daje nam poszukiwaną parę (x,y).

Możemy teraz naszą bijekcję rozszerzyć, by odwzorowywała przestrzeń N^p w N. W tym celu definiujemy przez rekurencję $\propto_p$ dla p>2 jako:

$\propto_{p+1}(x_1, ..., x_p, x_{p+1}) = \propto_p(x_1, \ldots, x_{p-1}, \propto_2(x_p, x_{p+1}))$

Tak zdefiniowana funkcja jest bijekcją, co dowieść możemy indukcyjnie.

Pierwszy krok indukcyjny jest spełniony, gdyż bijekcję N^2 w N funkcji $\propto_2(x,y)$ już udowodniliśmy.

Załóżmy teraz, że $\propto_p$ jest bijekcją i spróbujmy dowieść, że $\propto_{p+1}$ jest bijekcją.

Iniekcja: dowód nie-wprost. Weźmy dwa różne ciągi liczb: $c_1=(x_1,x_2, ..., x_p, x_{p+1})$ oraz $c_2=(y_1, y_2, ..., y_p, y_{p+1})$. Jeśli $\propto_{p+1}(c_1) = \propto_{p+1}(c_2)$ wówczas :

$\propto_p(x_1, \ldots, x_{p-1}, \propto_2(x_p, x_{p+1})) = \propto_p(y_1, \ldots, y_{p-1}, \propto_2(y_p, y_{p+1}))$

Wykorzystując fakt, iż $\propto_p$ jest bijekcją otrzymujemy, że $c_1=c_2$ co jest sprzeczne z założeniem, czyli $\propto_{p+1}$ jest iniekcją:

Suriekcja: wynika prosto z definicji rekurencji. Dla dowolnej liczby naturalnej „n” z założenia indukcyjnego istnieją takie $x_1, x_2, ..., x_{p-1}, \delta$, że zachodzi:

$n = \propto_p(x_1, \ldots, x_{p-1}, \delta)$

oraz dla dowolnego δ mamy istnienie takich x_p i x_{p+1}, że ostatecznie:

$n = \propto_p(x_1, \ldots, x_{p-1}, \propto_2(x_p, x_{p+1}))$, czyli

$n = \propto_{p+1}(x_1, \ldots, x_p, x_{p+1}))$ co kończy dowód.

V. 4 Rekurencja nie prymitywna

Funkcje rekursywnie prymitywne pozwalają nam zaspokoić większość potrzeb przy definiowaniu funkcji w programowaniu, niemniej jednak niektóre programy nie obliczają tych funkcji, ponieważ nie zawsze się kończą: w rzeczywistości obliczają one funkcje zdefiniowane częściowo.

Funkcje obliczalne, głównie definiowalne i nie rekursywne prymitywnie są wyjątkowo rzadkie i w praktyce nie używane. Za przykład posłuży nam **funkcja Ackermann'a**. Poniższa funkcja oznaczona przez A jest trudniejszym wariantem tej, która została zdefiniowana przez Ackermann'a.

$A(0, x) = x + 2$ dla każdego x
$A(1,0) = 0$ oraz $A(y, 0) = 1$ dla każdego $y \geq 2$
$A(y + 1, x + 1) = A(y, A(y+1, x))$ dla każdego x oraz y.

Dla każdej wartości n, funkcja wiążąca z x wartość A(n,x) jest oznaczana jako A_n. Funkcja ta jest zdefiniowana przez rekurencję:
$A_0(x) = x + 2$, $A_1(x) = 2x$, $A_2(x) = 2^x$ oraz dla każdego $n > 2$:

$A_n(0) = 1$
$A_n(x+1) = A_{n-1}(A_n(x))$.

Dla każdej zmiennej n, funkcja A_n jest więc rekursywnie prymitywna. Aby wykazać, że funkcja A nie jest rekursywnie prymitywna musimy przeanalizować kolejno:
dla każdego $n > 1$ oraz każdego x, $A(n,x) > x$
kolejne funkcje A_n są ściśle rosnące
jeśli $n \geq 2$, wtedy $A(n, x) \geq A(n-1, x)$
jeśli $x \geq 4$, wtedy $A(n+1, x) \geq A(n, x+1)$
jeśli $n_1, n_2, ..., n_p$ są zmiennymi, wówczas istnieje taka zmienna m, że dla każdego x zachodzi:

$$\sum_{i=1}^{p} A(n_i , x) \leq A(m , x)$$

jeśli f jest funkcją rekursywnie prymitywną p zmiennych, wtedy istnieje takie m, że dla każdego n1, ..., np zachodzi:

$$f(n_1, ..., n_p) \leq A(m , \sum_{i=1}^{p} n_i)$$

patrząc na funkcję g zdefiniowaną jako $g(n) = A(n,n)$ wnioskujemy, że funkcja A nie jest rekursywnie prymitywna.

V. 5 Rodzaje funkcje rekursywnych

Aby zdać sobie sprawę z wszystkich funkcji obliczalnych, konieczne jest zdefiniowanie klasy większej niż klasa funkcji rekursywnie prymitywnych. Może to zostać osiągnięte dzięki nowemu schematowi konstrukcji nazywanemu **schematem minimalizacji**.

Wprowadzenie do tego schematu pociąga za sobą konieczność skupienia się na funkcjach częściowych. Dla danego podzbioru A przestrzeni N^{p+1} chcemy zdefiniować funkcję p argumentową, która z ciągiem liczb $(x_1, x_2, ..., x_p)$ wiąże najmniejszą wartość „z", taką że $(x_1, x_2, ..., x_p, z) \in A$. Jeśli wystąpi przypadek, że dla każdej wartości zmiennej „z" ciąg $(x_1, x_2, ..., x_p, z) \notin A$ w przeciwieństwie do minimalizacji ograniczonej, schemat minimalizacji nie zatrzymuje się.

Definicje:

Funkcją częściową z przestrzeni N^p w N nazywamy parę (D,f) gdzie D jest podzbiorem przestrzeni N^p ,a f funkcją ze zbioru D w N. D nazywamy dziedziną definicji funkcji f. W przypadku, gdy $D = N^p$, f nazywamy funkcją **totalną**.

Jeżeli $(x_1, ..., x_p) \in D$, wówczas f jest zdefiniowana dla $(x_1, ..., x_p)$ co oznaczamy $f(x_1, ..., x_p)\downarrow$. Jeśli $(x_1, ..., x_p) \notin D$, wtedy f nie jest zdefiniowana dla $(x_1, ..., x_p)$.

Należy zauważyć, że dwie funkcje częściowe są równe, jeżeli posiadają taki sam zbiór definicji i jeśli są identyczne na jego dziedzinie, trzeba więc przy każdej nowo tworzonej funkcji dokładnie określić jego dziedzinę definicji:

1. Dziedzina funkcji f , złożonej z „p" funkcji n - argumentowych $g_1, .., g_p$ oraz p – argumentowej funkcji h jest definiowana przez warunki:

$f(x_1, ..., x_p)\downarrow$

wtedy i tylko wtedy gdy:

$g_1(x_1, ..., x_p)\downarrow$

.....

$g_p(x_1, ..., x_p)\downarrow$

$h(g_1(x_1, ..., x_p), ..., g_p(x_1, ..., x_p))\downarrow$

2. Dziedzina funkcji f definiowanej przez rekurencję prymitywną przy pomocy p – argumentowej funkcji g oraz p+2 argumentowej funkcji h jest definiowana przez rekurencję:

$f(x_1, ..., x_p, 0)\downarrow$ wtw $g(x_1, ..., x_p)\downarrow$

$f(x_1, ..., x_p, y+1)\downarrow$ wtw $h(x_1, ..., x_p, y, f(x_1, ..., x_p, y))\downarrow$

Schemat minimalizacji pozwala na skonstruowanie nowych funkcji częściowych:

Weźmy p+1 argumentową funkcję g. Mówimy, że funkcja częściowa f jest zdefiniowana przez minimalizację począwszy od g, gdy:

- istnieje co najmniej jedna zmienna z , taka że $g(x_1,...,x_p, z) = 0$ i jeśli dla każdego $z'<z$ $g(x_1,...,x_p, z') \neq 0$,wtedy $f(x_1,...,x_p) = z$;

w przeciwnym przypadku, $f(x_1,...,x_p)$ nie jest zdefiniowana.

Definicje:

Zbiór **funkcji rekursywnych częściowych** jest najmniejszym zbiorem funkcji zawierającym funkcje bazowe i domknięty przez złożenie, rekurencje prymitywną i minimalizację.

Podzbiór przestrzeni N^p nazywamy rekursywnym, jeśli jego funkcja charakterystyczna jest rekursywna.

Funkcje rekursywne częściowe są przeliczalne w takim sensie, że dla każdej z nich istnieje algorytm, który albo zatrzyma się pod koniec ograniczonego czasu zwracając wartość funkcji, jeśli jest ona zdefiniowana albo w przeciwnym przypadku nie zatrzyma się.

V. 6 Teza Church'a

Spróbujmy odwrócić rozpatrywane pytanie: czy wszystkie funkcje obliczalne są rekursywne ? Na to pytanie można odpowiedzieć tylko dzięki dokładnej definicji natury funkcji obliczalnej. **Teza Church'a** (1936) utrzymuje, że funkcje obliczalne są dokładnie funkcjami rekursywnymi. Doświadczenia pokazują, że za każdym razem, gdy funkcja jest zdefiniowana przy pomocy efektywnej procedury (algorytmu) , procedura ta dostarcza środków, by dowieść, że funkcja jest w rezultacie rekursywna.

Niemniej jednak, teza Church'a nie mówi nic o reprezentacji algorytmu związanego z tą funkcją rekursywną. Z punktu widzenia informatyka, wielkości takie jak czas oraz wielkość pamięci potrzebnej do obliczeń zależą od modelu rozpatrywanej maszyny. Z drugiej strony, wszystkie modele przyzwoitych maszyn definiują taką samą klasę funkcji: funkcji rekursywnie częściowych.

Rozdział VI Rekursywność i obliczalność

W tym rozdziale zaobserwujemy równoważność pomiędzy własnościami funkcji rekursywnych częściowych i funkcjami obliczalnymi przy pomocy maszyny Turniga. Przedstawiony zostanie także język λ-term.

Będziemy korzystali z maszyn Turinga posiadających dwie taśmy oraz poniższe właściwości:

- zbiorem symboli używanych jest $\Sigma = \{0,1, b\}$,gdzie symbol b oznacza „blanc" (pusty)
- zbiór stanów Q jest skończony i zawiera dwa stany wyróżnione: stan początkowy q_0 i stan końcowy q_1.
- funkcja przejścia δ jest odwzorowaniem zbioru $Q \times \Sigma^2$ w zbiór $Q \times \Sigma^2 \times \{L, S, R\}^2$, gdzie L reprezentuje przejście w lewo, R reprezentuje przejście w prawo, a S brak ruchu głowicy czytającej.

Weźmy f, p – argumentową funkcję częściową oraz M – maszynę Turinga. W tym rozdziale będziemy badali funkcje M obliczalne – zdefiniowane przy pomocy maszyny Turinga, innymi słowy Turing-obliczalne.

Standardowa reprezentacja zmiennych w maszynie jest reprezentacją binarną. Od tej chwili będziemy posługiwali się inną notacją, która uprości demonstrację: zmienna o wartości 0 jest reprezentowana przez 0, natomiast zmienna „n" jest reprezentowana przez n pozycji, na których znajdują się wartości 1.

W rozpatrywanych przez nas maszynach Turniga pierwsza taśma reprezentuje ciąg danych wejściowych, natomiast druga taśma ciąg będący rezultatem funkcji reprezentowanej przez maszynę.

VI. 1 Funkcje rekursywne i Turing – obliczalne

Na początku udowodnimy, że funkcje bazowe są Turing – obliczalne.

- Funkcja stale równa 0 jest obliczalna na następującej maszynie Turniga:

$$\delta(q_0, 0, b) = (q_0, 0, 0, R, R)$$
$$\delta(q_0, 1, b) = (q_0, 1, 0, R, R)$$
$$\delta(q_0, b, b) = (q_1, b, b, S, S)$$

- Funkcja następnika jest realizowana na maszynie Turniga, której funkcja przejścia spełnia warunki:

$$\delta(q_0, 1, b) = (q_0, 1, 1, R, R)$$
$$\delta(q_0, b, b) = (q_1, b, 1, S, S)$$

- Funkcja projekcji pr_p^i $(1 \leq i \leq p)$ jest obliczalna na maszynie posiadającej p+1 stanów $q_0, q_1, q_2, ..., q_p$. Wejściem jest ciąg wartości oddzielonych symbolem b. Przykładowo, funkcja pr_2^2 jest obliczalna na 3 – stanowej q_0, q_1, q_2 maszynie, której funkcja przejścia spełnia:

$$\delta(q_0, 1, b) = (q_0, 1, b, R, S)$$
$$\delta(q_0, 0, b) = (q_0, 0, b, R, S)$$
$$\delta(q_0, b, b) = (q_2, b, b, R, S)$$
$$\delta(q_2, 1, b) = (q_2, 1, 1, R, R)$$
$$\delta(q_2, 0, b) = (q_2, 0, 0, R, R)$$
$$\delta(q_2, b, b) = (q_1, b, b, S, S)$$

Pozostaje nam dowieść, że zbiór funkcji częściowych, obliczalnych na maszynie Turniga jest domknięty przez złożenie, rekurencję prymitywną i minimalizację. Rezultat ten możemy osiągnąć kolejno utożsamiając maszynę z każdym procesem konstrukcji funkcji rekursywnej.

VI. 2 Rekursywność funkcji Turing – obliczalnych

Poniższe obliczenia zapewniają, że klasa funkcji rekursywnych częściowych zamyka się razem z klasą funkcji Turnig – obliczalnych. Przedstawiona demonstracja używa kodowania zbioru operacji wykonanych przez maszynę w trakcie obliczeń.

Wniosek 1:

Każda funkcja częściowa, obliczalna przez maszynę Turniga jest rekursywna.

Weźmy f, funkcję częściową, obliczalną przy pomocy maszyny Turniga M, która posiada 2 taśmy i m stanów. Aby pokazać, że funkcja f jest rekursywna musimy na początku zakodować sytuację maszyny przy pomocy zmiennej wejściowej t oraz pokazać, że kod jest funkcją rekursywnie prymitywną, zależną od t i od warunków początkowych. Każdy stan maszyny „q_i" jest kodowany przez wartość „i", symbol pusty (blanc) przez wartość 0, natomiast symbol „0" przez wartość 2.

Definicje:

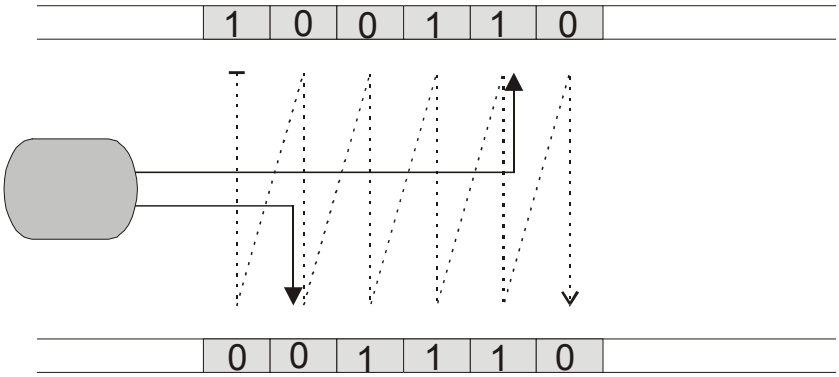
Konfiguracja maszyny M do danego momentu „t" jest nieskończonym ciągiem $C(t) = (s_0, ..., s_i, ...)$ symboli zapisanych do tego momentu przez dwie taśmy maszyny M. Ciąg ten jest uzyskany przez konkatenację ciągów σ_0, σ_1, σ_2, ..., σ_j ..., gdzie dla każdej wartości „j" , σ_j jest ciągiem symboli zapisanych w komórkach o numerze j. Ten nieskończony ciąg posiada tylko jedną, skończona ilość znaków nie pustych.

Sytuacja maszyny do momentu „t" jest ciągiem $(e, k_1, k_2, C(t))$, gdzie „e" jest kodem stanu maszyny do momentu „t" , k_1 oraz k_2 są numerami komórek, przed którymi do tego momentu znajdują się głowice czytające, a C(t) jest konfiguracją maszyny.

Konfiguracja C(t) może zostać zakodowana przez wartość: $\Gamma(C) = \Sigma_{i\geq 0}\ s_i \bullet 3^i$. Funkcje dzielnika „q" i reszty z dzielenia „r" pozwalają na odzyskanie z powyższego kodu symboli zapisanych w komórce o numerze „u" dla taśmy o numerze „v": $r(q(\Gamma(C), 3^{2(u-1)+v-1}), 3)$. Sytuacja maszyny do momentu „t" może więc zostać zakodowana przez wartość:

$\Gamma(S) = \propto_4(e, k_1, k_2, \Gamma(C))$.

Na rysunku nr 35 przedstawiony jest sposób kodowania maszyny Turniga. Zapisana jest uporządkowana sekwencja C(t) = 1,0,0,0,0,1,1,1,1,1,0,0, k_1=5 k_2=2.



Rysunek 35 Kodowanie maszyny Turinga

Lemat 1

Istnieje funkcja rekursywnie prymitywna g, która dostarcza kod sytuacji maszyny do momentu t+1, na podstawie kodu sytuacji maszyny z momentu t.

Dowód:

Przejście zmiennej opisującej w stan następny odbywa się przy pomocy funkcji przejścia. Funkcja, która pozwala wyrazić konfigurację maszyny do momentu „t+1" przy pomocy konfiguracji do momentu „t" może być zdefiniowana w sposób rekursywnie prymitywny w przypadku odnoszącym się do zbioru definicji funkcji przejścia.

Lemat 2

Funkcja „Sit" , która dostarcza kod sytuacji maszyny do momentu „t" na podstawie początkowej konfiguracji danych jest rekursywnie prymitywna.

Dowód:

Funkcja „Sit" jest zdefiniowana przez rekurencję. Jej wartość do momentu t = 0 jest uzyskiwana w sposób rekursywnie prymitywny począwszy od konfiguracji początkowej. Przejście od momentu „t" do momentu „t+1" jest realizowane przy pomocy funkcji g.

W dalszej części identyfikujemy kod związany z $Sit(t,x_1, x_2, ..., x_p)$ oraz czteroelementową sekwencję (e, k_1, k_2, C(t)). W szczególności $\pi_4^1(Sit(t,x_1, x_2, ..., x_p))$ jest stanem „e". Demonstracja powyższej własności jest stosunkowo prosta.

Dowód:

Czas obliczania wartości $f(x_1, x_2, ..., x_p)$ jest zadany przez:

$$T(x_1, x_2, ..., x_p) = \mu t (\pi_4^1(Sit(t,x_1, x_2, ..., x_p)) = 1),$$

gdyż maszyna osiąga tylko jeden stan końcowy q_1.

Znając sytuację do momentu $T(x_1, x_2, ..., x_p)$ możliwe jest zliczenie ilości wystąpień symbolu 1 na drugiej taśmie, która jest równa wartości $f(x_1, x_2,..., x_p)$:

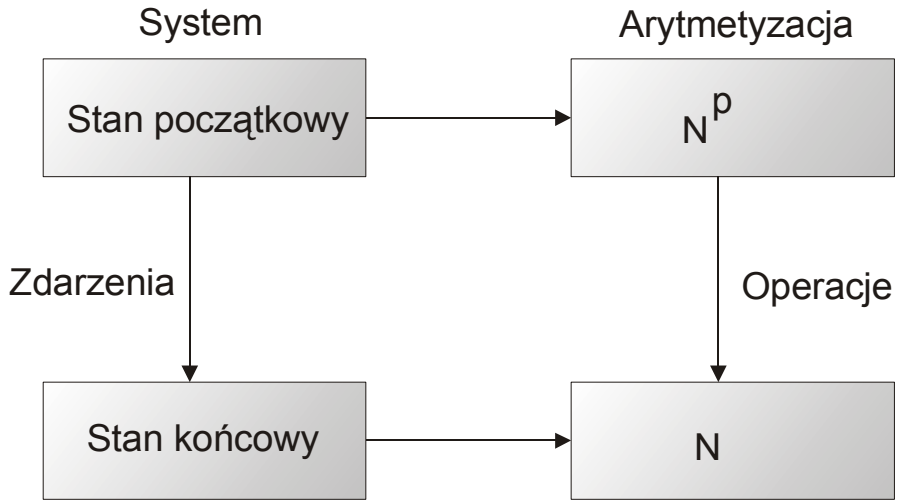
$$f(x_1, x_2,..., x_p) = \mu y(r(q(\pi_4^4(Sit(T(x_1, x_2, ..., x_p), x_1, ..., x_p)), 3^{2y+1}), 3) = 0)$$

Ta ostatnia funkcja oblicza przy pomocy funkcji q (dzielnika) i r (reszty) ilość „1" na drugiej taśmie.

VI. 3 Abstrakcja funkcjonalna

Jakakolwiek byłaby dziedzina aplikacji, użytkowanie komputera sprowadza się za każdym razem do obliczania wartości funkcji. W rzeczywistości dane na wejściu są kodowane do postaci ciągu bitów interpretowanych jako sekwencje wejściowe, a na wyjściu rezultaty ponownie są kodowane i interpretowane. Etapy pośrednie realizują wszystkie istotne obliczenia.

Taka interpretacja oznacza proces arytmetyzacji, który jest szeroko używany w realizacji procesów modelowania i symulacji. Przepływ danych ilustruje rysunek nr 36.



Rysunek 36 Proces arytmetyzacji

Powyższy schemat pokazuje wagę jaką kładzie się na analizowanie w informatyce funkcji, w szczególności z przestrzeni N^p w N. Rozpatrując te ostatnie, matematycy i programiści wiedzą, że mogą one być stworzone przy wykorzystaniu zaledwie kilku funkcji prymitywnych. Przyjrzymy się teraz niektórym procedurom konstrukcji, które pomogą nam zrozumieć matematykę algorytmiczną.

Przypomnijmy, że funkcja jest regułą wiążącą obiekty, która pozwala określić wartość dla każdego zadanego jej argumentu dziedziny. Użyteczne jest zdefiniowanie tej reguły przez wyrażenie zależności pomiędzy argumentem i jego wartością. Rozumiemy przez to, np. przypisanie $x \rightarrow x^2$ lub $f: x \rightarrow f(x)$, gdzie f jest właściwym oznaczeniem reguły.

Łącząc ze sobą te dwa rodzaje zapisu dochodzimy do wyrażeń typu: „dana jest funkcja $f(x) = x^2$” , w których dostrzegamy dwuznaczność. Nawet jeżeli potrafimy dokładnie rozróżnić funkcję od jej wartości, to jak powinno się liczyć wartość wyrażenia F(f(x+1)) ? Czy należałoby liczyć na początku g(x)=f(x+1) , a następnie F(g(x)) ? Czy też może h(x) = F(f(x)) na początku, a później h(x+1) ? Wystarczy jeszcze podać operator pochodnej $D(x^2) = 2x$, by przekonać się jakie niezrozumiałości mogą zaistnieć podczas działania algorytmu.

By wyeliminować wszystkie niezrozumiałości należy zdefiniować koncepcję funkcji biorąc pod uwagę:

- samą funkcję : jako obiekt
- obiekty, do których funkcja się odnosi
- uzyskane wartości

To rozróżnienie nosi nazwę abstrakcji funkcjonalnej. Przed zapisaniem mechanizmu tego rozróżnienia, który będzie bazował również na klasycznej notacji funkcyjnej, ustalmy podstawowe zasady.

Załóżmy, że dana jest zmienna x, która może (ale nie musi) wystąpić w obiekcie E nazywanym wyrażeniem. Niech λ będzie symbolem wyróżnionym. Rozpatrzmy obiekt (λx. E) : jest on funkcją. Gdy do tego obiektu wstawiamy wartość „a” otrzymujemy wyrażenie (λx. E)a . Wartość (λx. E)a obliczamy podstawiając w wyrażeniu „E” za „x” wartość „a”.

W praktyce, funkcja f notowana jako f: $x \rightarrow E(x)$ będzie się nazywała (λx.E(x)) co jest λ notacją f.

Komentarz:

1. Każde wyrażenie typu 2*5 jest liczbą, „A” jest znakiem, natomiast (λx.E) jest funkcją
2. Można interpretować obiekt (λx. E) jako rezultat zastosowania reguły oznaczanej jako λ do pary (x,E). Z tego powodu, symbol λ jest nazywany konstruktorem funkcjonalnym. Pomimo tej interpretacji należy pamiętać, że jako notacja funkcyjna, (λx. E) formuje obiekt nierozdzielny.

3. Można sobie wyobrazić, że istnieje reguła oznaczana przez @, która dla każdej pary ((λx. E),a) tworzy odpowiednio (λx. E)a , której wartość jest rezultatem podstawienia w wyrażeniu E za „x" elementu „a". Ściślej mówiąc rozpatrzmy zbiór funkcji częściowych X w Y oznaczony przez (X→Y). Reguła @ może być interpretowana jako funkcja częściowa ze zbioru (X→Y) × X na zbiór Y, która z każdą parą (f,x) utożsamia f(x). Z tego powodu reguła @ jest naturalnie nazywana aplikacją. Korzystnie jest zamienić @ przez konkatenację pisząc fx zamiast @(f,x) . To wyjaśnia dlaczego używa się niekiedy notacji fx dla wyrażenia wartości funkcji f na x.
4. Podstawienie wymaga prawidłowego sformalizowania. Zmienna x służy tylko do wskazania pozycji, gdzie ma miejsce podstawienie. Sama nazwa zmiennej x nie gra roli. W konsekwencji możemy, co jest bardzo wskazane, zmienić nazwę zmiennej, do której podstawiamy wartość, jeżeli może wystąpić błędne zrozumienie formuły. Opisaną sytuację przedstawiają dwa przykłady:
   - dwie funkcje (λx. (2*x+1)) i (λt. (2*t+1)) są równe
   - nie zapisuje się (λx. (2*x+1))x , lecz (λt. (2*t+1))x .

Przykłady:

1. Rozpatrzmy teraz wyrażenie x^2+x+1. Jeśli rozpatrzymy x jako zmienną, wtedy (λx. (x^2+x+1)) jest funkcją oznaczaną zazwyczaj jako f:x→x^2+x+1; odwrotnie: λ - notacją funkcji g: x→2x+3 jest (λx. 2x+3)).
2. Weźmy wyrażenie 3x-xy+5. Jeśli chcemy zrobić z 3x-xy+5 funkcję (częściową) zależną od y (dla ustalonego y) , wtedy zapisujemy (λy. (3x-xy+5)) co odpowiada notacji klasycznej f_x: y→3x-xy+5. Począwszy od (λy. (3x-xy+5)) napiszmy (λx. (λy. (3x-xy+5))), co odpowiada zapisowi funkcji 2 zmiennych f: (x,y) → 3x-xy+5; odwrotnie możemy interpretować wyrażenie (λy. (λx. (3x-xy+5))) jako λ - notację funkcji g: (y,x) → 3x-xy+5 . Zaczęliśmy właśnie rozważania funkcji wielu zmiennych. W tym przypadku λ - notacja funkcji f: (x_1, x_2, ..., x_n) → E(x_1, x_2, ..., x_n) będzie przyjmowała postać: λx_1. (λx_2. (... (λx_n. E(x_1, x_2, ..., x_n)) ...)
3. Zanotujmy operator wyrażający pochodną: D(λx. x^2) = (λx. 2x)
4. Niech E będzie wyrażeniem zawierającym x, a T operatorem zdefiniowanym jako T(λx. E(x)) = (λx. E(x+1)) . Weźmy funkcję F; dwiema możliwymi interpretacjami F(f (x + 1)) będą: F(T(λx. f(x))) oraz T(F(λx. f(x))) .

5. Kiedy reguła aplikacji zapisuje się przez konkatenację, dobrze jest wyróżnić multiplikację. Przykładowo:

C::= (λx. (x^2+2*x+1))
S::= (λx. (λy. (x+y)))
P::= (λx. (λy. (x*y)))
Mamy odpowiednio C3 = 3^2+2*3 +1 = 16
S2 = (λx. (λy. (x+y)))2 = (λy. (2+y)); (S2)5 = (λt. (2+t))5 = 7
Jeśli f jest funkcją jednej zmiennej i ma jeden argument, wtedy:
P(Fa) = (λy. (Fa*y)),

W szczególności:
P(C3) = (λy. (16*y)) ; P((S1)x) = (λy. ((1+x) *y))

VI. 4 Język λ – term

Formalizm przeznaczony do opisania mechanizmów przedstawionych poniżej został wprowadzony w pracach A. Church'a, który jako pierwszy użył języka formalnego nazwanego λ-calcul. Język ten jest używany wtedy, gdy chcemy opisywać funkcje, stąd Church użył greckiej litery λ ,która symbolizuje rzymską literę L, by w ten sposób zasugerować, że jego system formalny jest językiem.

Jak to zwykle bywa, dany jest nieuporządkowany zbiór Var symboli zwanych zmiennymi oraz cztery symbole specjalne: „ λ ” „.” „(” „)” . Język λ - term jest generowany przez gramatykę:

<lambda-term> ::= <Var> | (<lambda-term><lambda-term>)
| (λ <Var> . <lambda-term>)

Pierwsza reguła ustala zbiór *atomów*. Dwie następne reguły formalizują odpowiednio *aplikację* i *abstrakcję*. Można stosunkowo łatwo pokazać, że powyższa gramatyka nie jest dwuznaczna.

Komentarz:

1. Każdy λ - term reprezentuje funkcję jednej zmiennej, której argumenty i wartości mogą same być funkcjami.
2. Term oznaczony jako FX jest nazywany *aplikacją* (lub *a-wyrażeniem*). Reprezentuje on połączenie operatora F z operandem X.

3. Term postaci (λx_1. (λx_2. (... (λx_n. T) ...))), gdzie (x_1, x_2, ..., x_n) jest listą zmiennych, a T jest termem nazywamy abstrakcją (lub *λ-wyrażeniem*). Reprezentuje on funkcję zmiennej X=(x_1, x_2, ..., x_n). Mówi się, że X jest listą *parametrów*, a T *ciałem* abstrakcji. Uzgodnimy ponadto, że term (λx_1. (λx_2. (... (λx_n. T) ...))) będzie zapisywany jako $\lambda x_1 \lambda x_2 \lambda x_n$.T .
4. W celu uproszczenia zapisu, zaadoptujemy dwie inne konwencje:
   - dla a-wyrażeń, nawiasowanie będzie występować tylko od lewej do prawej
   - dla λ-wyrażeń, ich ciało będzie się rozszerzać możliwie jak najdalej w prawą stronę

Przykład:

Załóżmy, że mamy dwie zmienne: x i y. Dla przedstawionych term

x; y; (xy); (λx. (xy)); (y(λx. (xy)); (λx. (λy. (y(λx. (xy)))))

możemy zastosować uproszczony zapis, otrzymując:

xy; λx. xy; y(λx.xy); λxy. y(λx. xy)

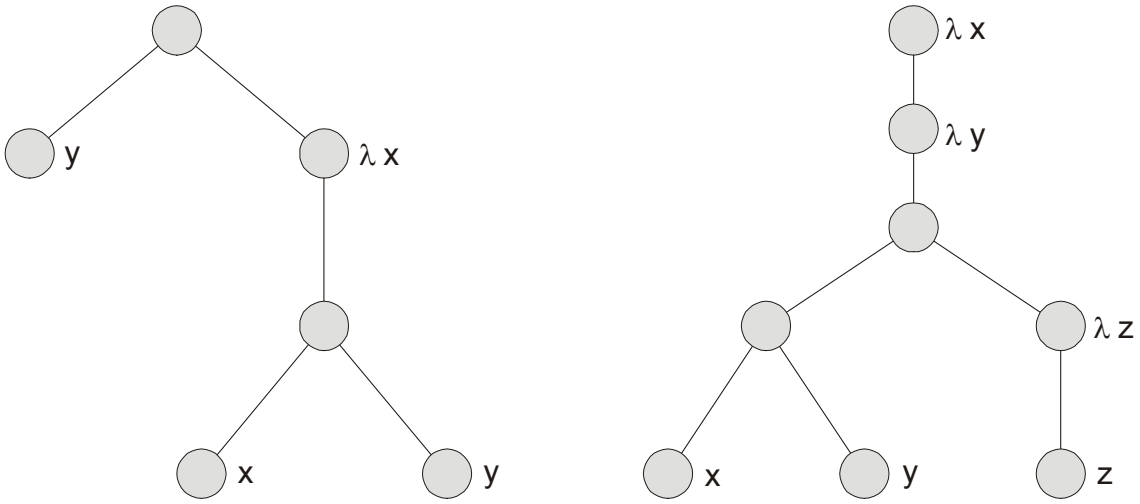
VI. 5 Obliczenia na λ – termach

Formalnie, każda forma jest tworzona dzięki dwóm regułom, którymi są abstrakcja i aplikacja. Każdą formę możemy reprezentować przez drzewo binarne i w zależności od stopnia $d^+(s)$ każdego wierzchołka s, wyróżniamy następujące przypadki:

- $d^+(s) = 0$: s jest zmienną
- $d^+(s) = 1$: s reprezentuje abstrakcję
- $d^+(s) = 2$: s reprezentuje aplikację.

Przykład:

Respektując powyższe przypadki, termy: y(λx. xy) oraz λxy. xy(λz. y) są reprezentowane przez następujące drzewa:



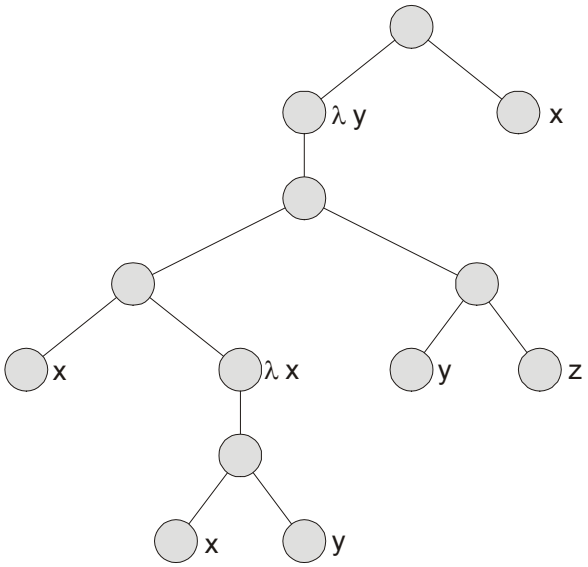
Rysunek 37 Drzewa reprezentujące termy

Począwszy od definicji termy , można zdefiniować indukcyjnie zbiór podzbiorów termy.

Graficznie pod – termy danej termy T są reprezentowane przez poddrzewa drzewa syntaktycznego T. Zmienna x, która figuruje na liście parametrów abstrakcji i znajduje się pod – termem T jest nazywana *połączoną*. *Wystąpienie* x jest połączone, jeśli znajduje się ono w ciele abstrakcji, której lista parametrów zawiera x. Wystąpienie zmiennej jest nazywane *wolnym*, jeśli nie jest ono połączone. Zmienna jest wolna, jeśli pozwala ona na wolne wystąpienie.

Zmienna połączona gra rolę parametru formalnego, który służy do wskazania miejsca w ciele abstrakcji, stąd jego nazwa nie grali. Ponadto dozwolone jest zmienianie nazwy zmiennej połączonej. Mówiąc językiem formalnym: niech x będzie zmienną połączoną formy T. Istnieje pod-drzewo minimalne (w sensie inkluzji) posiadające jako korzeń λx, które reprezentuje term postaci λx. M . Niech y będzie zmienną nie występującą w M. Zamiana x na y w λx. M pociąga za sobą podmianę wszystkich x na y. W ten sposób można zawsze podejrzewać, że w każdej termie, co więcej, w zbiorze term, nazwy zmiennych połączonych są różne od zmiennych wolnych.

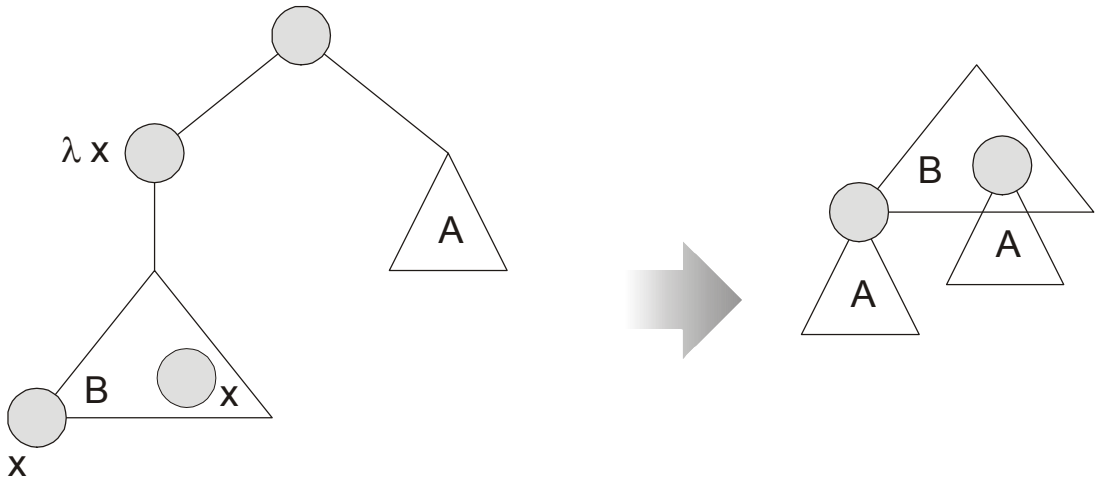
Przykład:
Na rysunku nr 38 przedstawiony jest term: (λy. x(λx. xy)(yz))x



Rysunek 38 Zmienne wolne i połączone

Zmienne x i y są połączone; x i z posiadają wystąpienia wolne. Aby zapobiec kolizji, wystarczy zmienić nazwę zmiennej połączonej x. Otrzymujemy wtedy (λy. x(λt. ty)(yz))x , co po zamianie y na u można zapisać jako: (λu. x(λt. tu)(uz))x.

Jeśli B jest termem zawierającym zmienną wolną x, wtedy λx. B reprezentuje funkcję, której wartość dla A, (λx. B)A otrzymuje się przez podstawienie A do wszystkich wystąpień x w B; graficznie - podłącza się korzeń drzewa reprezentującego A do wszystkich liści etykietowanych przez x w drzewie reprezentującym B, co przedstawia rysunek nr 39.



Rysunek 39 Zmiana nazw zmiennych w drzewie

Dokonana zamiana nazywa się skurczeniem wyrażenia (λx. B)A . W rezultacie (λx. xy)A kurczy się do Ay. Skurczenie może być interpretowane jako reguła ponownego przepisania. Sekwencja skurczeń nazywa się redukcją. Skurczenie oznacza się symbolem $\rightarrow$.

Obliczenia na λ termach składają się więc z sekwencji podmian zmiennych połączonych oraz skurczeń.

Przykład:

Obliczmy (λy. x((λx. xy)(xyz)))(xy). Dla uniknięcia wszelkich nieporozumień zamieńmy nazwy zmiennych połączonych x (na u) oraz y (na v). Otrzymujemy (λv. x((λu. uv)(xvz)))(xy) . Pod – term (λu. uv)(xvz) redukuje się do (xvz)v. Term początkowy postaci (λv. x((xvz)v))(xy) redukuje się więc do x((x(xy)z)(xy)).

Rozdział VII Równania rekurencyjne

VII. 1 Zastosowanie równań rekurencyjnych

Nim przedstawimy niektóre ciekawe rodzaje równań rekurencyjnych i pokażemy jak je rozwiązać, należy zwrócić uwagę na ich wszechstronne zastosowanie w informatyce. Mówiąc o sortowaniu mamy często do czynienia z algorytmami rekurencyjnymi i aby móc je ze sobą porównać, trzeba umieć oszacować czas obliczeń dla wszystkich zagnieżdżonych wywołań rekurencyjnych. Przykładowo dla algorytmu MERGESORT, który w dużym przybliżeniu wyglądał następująco:

MERGESORT(1..n) = MERGESORT(1..n/2) *merge* MERGESORT(n/2 +1, n)

czas obliczeń dla n danych wejściowych był rzędu O(k*n*logn). By uzyskać tą wielkość należało rozwiązać równanie rekurencyjne: f(n) = k + 2*f(n/2).

Ten rozdział będzie poświęcony zarówno rozwiązywaniu tego typu równań, jak i równań trudniejszych, gdzie wzór rekurencyjny zależy od wielu różnych czynników.

Rozpoczniemy jednak od pokazania, że rozwiązanie równania rekurencyjnego, czyli znalezienie ogólnego wzoru na n-ty wyraz funkcji przydaje się również do innych problemów. Weźmy zbiór n elementowy. Wiemy, że ilość różnych (względem pozycji występowania) ułożeń elementów tego zbioru, to ilość permutacji na tym zbiorze, co wyraża się wzorem n!. Wiemy również, że wyniku zastosowania permutacji, nie zawsze każdy element zbioru zmieni swoją pozycje. Niech operacja, w wyniku której każdy element zbioru zmieni swoją pozycję będzie nazwana *zaburzeniem* zbioru. Interesuje nas zarówno to, ile będzie różnych zaburzeń zbioru n elementowego, jak również to, jaki procent permutacji zbioru stanowią jego zaburzenia.

Zaczniemy od skonstruowania wzoru rekurencyjnego d_n przedstawiającego ilość zaburzeń w zbiorze n elementowym. Załóżmy, że elementami zbioru są: $(a_1, a_2, a_3, ..., a_n)$.

W każdym zaburzeniu tego zbioru, element a_1 musi znaleźć się na innej pozycji. Element a_1 możemy z elementem a_i $2 \le i \le n$ zamienić na n-1 sposobów.

- Jeśli po takiej zamianie element a_i znajdzie się na pozycji 1, wtedy pozostaje nam znaleźć ilość zaburzeń zbioru $(a_2, ..., a_{i-1}, a_{i+1}, ..., a_n)$, czyli problem zredukowaliśmy do mniejszego (gdy i=n wtedy pozostaje nam znaleźć ilość zaburzeń zbioru $(a_2, ..., a_{n-1})$. Ogólnie należy znaleźć d_{n-2}.

- Może się zdarzyć takie zaburzenie zbioru (a_1, a_2, a_3, ..., a_n), w którym na pozycji 1 nie będzie stał a_i. Ilość tego typu zaburzeń, to ilość zaburzeń zbioru (a_i, a_2, ..., a_{i-1}, a_{i+1}, ..., a_n), czyli problem znów staje się mniejszy (gdy i=n wtedy pozostaje nam znaleźć ilość zaburzeń zbioru (a_n, a_2..., a_{n-1}). Ogólnie należy znaleźć d_{n-1}.

Łatwo zauważyć, że w zależności od położenia elementu a_i po zamianie z a_1 (na pozycji 1, lub pozycjach dalszych), odpowiednie zaburzenia zbiorów będą różne. Ostatecznie otrzymujemy rekurencyjną formułę na ilość zaburzeń zbioru n elementowego, która wyraża się wzorem:

$d_n = (n-1)\,(d_{n-1} + d_{n-2})$ dla $n \geq 3$.

Taki wzór pozwala nam na stosunkowo łatwe wyznaczenie kolejnych ilości zaburzeń zbioru: $d_1 = 0$, $d_2 = 1$ (otrzymujemy z obserwacji) i kolejno $d_3 = 2$, $d_4 = 9$, $d_5 = 44$ itd. Dla przedstawionego zbioru można spróbować znaleźć wzór ogólny, który będzie miał postać:

$$d_n = n!\left(1 - \frac{1}{1!} + \frac{1}{2!} - \ldots + (-1)^n \frac{1}{n!}\right)$$

Mimo iż korzystając z tego wzoru trudniej jest obliczyć ilość zaburzeń dla kolejnych wartości n (wzór rekurencyjny w tym przypadku jest bardziej efektywny), posiadanie ogólnej formuły na wielkość d_n przydaje się do oszacowania stosunku ilości zaburzeń w zbiorze do ilości permutacji na tym zbiorze. Jeśli pamiętamy, że:

$$e^{-1} = \left(1 - \frac{1}{1!} + \frac{1}{2!} - \ldots + (-1)^n \frac{1}{n!}\right)$$

to od razu wyznaczymy, że $d_n/p_n \approx 0.367...$

VII. 2 Równania rekurencyjne liniowe

W ogólnym przypadku równania rekurencyjnego liniowego, którego rozwiązaniem jest wyznaczenie wszystkich wartości ciągu u_n, dla $n \geq 0$ spotykamy się z sytuacją, że znanych jest k początkowych wartości u(n), oraz znana nam jest formuła wyznaczająca wartość u(n+k) na podstawie wartości u(n), u(n+1) ... u(n+k-1). Można to zapisać w następujący sposób:

$u_0 = c_0$, $u_1 = c_1$, ..., $u_{k-1} = c_{k-1}$

$u_{n+k} + a_1 u_{n+k-1} + a_2 u_{n+k-2} + \ldots + a_k u_n = 0$ dla $n \geq 0$

gdzie c_0, c_1, …, c_{k-1} oraz a_1, a_2, …, a_k są stałymi. W zależności od parametru k mamy do czynienia z równaniami rekurencyjnymi liniowymi k-tego stopnia.

Pod koniec tego rozdziału zajmiemy się przypadkiem, gdy k>2, teraz jednak skupimy się na równaniach 2-go stopnia.

Równania 2-go stopnia mają tą przewagę, że można bez znajomości szeregów potęgowych stosować dla nich następujące twierdzenie:

Jeśli (u_n) jest ciągiem spełniającym równanie rekurencyjne:

$u_0 = c_0, \qquad u_1 = c_1$

$u_{n+2} + a_1u_{n+1} + a_2u_n = 0 \quad$ dla $n \geq 0$ $u_{n+1} + 4u_{n+1} - 5u_n = 0$

oraz α, β są rozwiązaniami pomocniczego równania kwadratowego:

$t^2 + a_1t + a_2 = 0$

to:

- Jeśli $\alpha \neq \beta$, to istnieją stałe A, B takie, że:
  $u_n = A\alpha^n + B\beta^n \quad$ dla $n \geq 0$
- Jeśli $\alpha = \beta$, to istnieją stałe C, D takie, że
  $u_n = (C^n + D)\alpha^n \quad$ dla $n \geq 0$

Wartości tych stałych wyznaczane są dzięki parametrom c_0, c_1.

Dowód przeprowadzimy dla przypadku pierwszego, tzn. dla $\alpha \neq \beta$ gdyż dla przypadku $\alpha = \beta$ wygląda on analogicznie. Skorzystamy z indukcji matematycznej. Zaczniemy od n = 0 i n =1. Twierdzenie mówi, że dla tych wartości zachodzi:

$c_0 = A\alpha^0 + B\beta^0 \qquad$ oraz $\qquad c_1 = A\alpha^1 + B\beta^1$

Zatem jeśli stałe A, B przyjmą wartości:

$$A = \frac{c_1 - c_0\beta}{\beta - \alpha}, \qquad B = \frac{c_1 - c_0\alpha}{\alpha - \beta}$$

twierdzenie będzie spełnione dla n = 0 oraz n = 1.

Niech stałe A i B pozostaną ustalone w pierwszym kroku indukcji. Teraz pozostaje pokazać, że jeżeli twierdzenie jest prawdziwe dla wszystkich u_m $0 \leq m \leq n+1$, jest też prawdziwe dla u_{n+2}.

$u_{n+2} = - a_1u_{n+1} - a_2u_n \quad = -a_1(A\alpha^{n+1} + B\beta^{n+1}) - a_2(A\alpha^n + B\beta^n)$

$= -A\alpha^n(a_1\alpha + a_2) - B\beta^n(a_1\beta + a_2) =$

(α, β są rozwiązaniami równania pomocniczego $t^2 + a_1t + a_2 = 0$)

$= -A\alpha^n(-\alpha^2) - B\beta^n(-\beta^2)$

$= A\alpha^{n+2} + B\beta^{n+2}$

Zobaczmy jak wykorzystuje się przedstawione twierdzenie w praktyce:

Załóżmy, że mamy do czynienia z następującym równaniem rekurencyjnym:

$u_0 = 0, \qquad u_1 = 2$

$u_{n+2} + 4u_{n+1} - 5u_n = 0$

Musimy najpierw wyznaczyć rozwiązania równania pomocniczego:

$t^2 + 4t - 5 = 0$, stąd $\alpha = 1$ $\beta = -5$

Wykorzystując równanie $u_n = A\alpha^n + B\beta^n$ dla n=0 i n=1 mamy:

$0 = A + B$ oraz

$2 = A + -5B$

,stąd wyliczamy stałe A oraz B:

$A = \frac{1}{3}$, $B = -\frac{1}{3}$

i ostatecznie podajemy wzór na n-ty wyraz ciągu (u_n)

$u_n = \frac{1}{3} + \frac{1}{3}(-5)^n$

Czasem, gdy pierwiastkami pomocniczego równania kwadratowego są liczby zespolone, wygodnie jest do przedstawienia wzoru na n-ty wyraz ciągu użyć twierdzenia De Moivre'a. Weźmy dla przykładu równanie:

$u_0 = 2$, $u_1 = 0$,

$u_{n+2} + u_n = 0$ dla $n \geq 0$

Jego równanie pomocnicze posiada rozwiązania w zbiorze liczb zespolonych:

$\alpha = i$ $\beta = -i$

stąd wzór na n-ty wyraz ciągu u_n ma postać:

$u_n = i^n + (-i)^n$

Korzystając z twierdzenia De Moivre'a przyjmie ono postać:

$u_n = 2\cos(\frac{1}{2} n \pi)$

VII. 3 Rekurencyjne przepołowienie

Mówiąc o problemach związanych z sortowaniem często mamy do czynienia z sytuacją, gdy rekurencyjny algorytm dzieli problem na dwa, dwukrotnie mniejsze problemy. Ten rodzaj rekurencji jest powszechnie stosowanym rekurencyjnym przepołowieniem. Podział problemu na m, m-krotnie mniejszych problemów jest tylko rozwinięciem schematu dla m=2, na którym teraz się skupimy. Ogólna postać rekurencyjnego przepołowienia, to:

$u_{2n} = P*u_n + Q(n)$

gdzie P jest stałą, a Q jest funkcją zależną od n. Taka rekurencja służy do prostego wyznaczenia wyrazów u_n o indeksach będących kolejnymi potęgami liczby 2. Jest ona często utożsamiana z metodą „dziel i zwyciężaj".

Jako przykład znalezienia ogólnego wzoru na n-ty wyraz ciągu u_n przeanalizujemy problem wyszukiwania maksymalnego i minimalnego elementu w danym zbiorze. Jeśli zbiór ma n elementów, to możemy zastosować najprostszy algorytm, który porównuje ze sobą kolejne pary liczb. Taki algorytm

musi dwukrotnie (osobno dla minimum i maksimum) przejść po wszystkich kolejnych parach zbioru, stąd podczas pracy wykona on 2n-2 porównań.

Pokażemy teraz alternatywny algorytm poczynając od przypadku, kiedy zbiory mają wielkość będącą potęgą 2. By dla zbioru 2^k elementowego znaleźć wartości $\min(1..2^k)$ oraz $\max(1..2^k)$, algorytm wywoła siebie samego dla zbiorów $(1..2^{k-1})$ oraz $(2^{k-1}+1..2^k)$, a następnie wykona dwa porównania: $\max(1..2^{k-1})$ z $\max(2^{k-1}+1..2^k)$ oraz $\min(1..2^{k-1})$ z $\min(2^{k-1}+1..2^k)$. Stąd rekurencyjny wzór na ilość porównań będzie miał postać:

$f(n) = 2*f(n/2) + 2$

Jeśli n jest potęgą 2, to dążymy do znalezienia rozwiązania równania:

$f(2^k) = 2*\ f(2^{k-1}) + 2$

Przyjmijmy tymczasowo oznaczenie $f(t) = f(2^k)$, $f(t-1) = f(2^{k-1})$, ..., $f(1) = f(2^1)$, $f(0) = f(2^0)$. Gdy zbiór ma 2 elementy, wówczas musimy wykonać 2 porównania, stąd $f(0) = 2$. Mamy więc:

$$\begin{aligned} f(2^k) &= 2*\ f(t-1) + 2 = 2*[2*\ f(t-2) + 2] + 2 = 2*[2*\ [2*\ f(t-3) + 2] + 2] + 2 \\ &= 2^{t-2}*\ f(0) + 2^1 + 2^2 + \ldots + 2^{t-1} \\ &= 2^{t-1} + 2*(1-2^{\,t-1})/(1-2) \\ &= 2^{t-1} + 2^t - 2 \\ &= \frac{3}{2}2^t - 2 \end{aligned}$$

$t = \log_2(2^k)$,czyli ostatecznie: $f(2^k) = \frac{3}{2}2^k - 2$

Teraz udowodnimy poprawność tego wzoru. Dla $k = 1$, $f(2^1) = \frac{3}{2}2^1 - 2 = 1$. Załóżmy, że wzór jest prawdziwy dla k. Chcemy udowodnić jego poprawność dla k+1.

$f(2^{k+1}) = 2*\ f(2^k) + 2 = 2*\ (\frac{3}{2}2^k - 2) + 2 = \frac{3}{2}2^{k+1} - 2$ c. b. d. o.

Mając udowodnioną poprawność wzoru dla zbiorów, których ilość elementów jest potęgą 2, możemy udowodnić jego poprawność dla każdego zbioru o parzystej liczbie elementów. W tym celu stosujemy podział zbioru wyjściowego S_0 na kolejne (od największego możliwego) zbiory o liczbie elementów będącą potęgą 2. Mówiąc ściślej, jeśli $\#S_0 = n$, to pierwszy wyodrębniony zbiór ma 2^m elementów, przy czym, 2^m jest największą potęgą 2 nie większą niż n. Drugi wyodrębniony zbiór będzie miał $n-2^m$ elementów. Tą strategię stosujemy do momentu, aż zbiór wejściowy w całości podzielimy na zbiory mające ilość elementów, będącą potęgą 2. Przy scalaniu każdych dwóch zbiorów będziemy potrzebowali 2 operacji porównania. Dla przykładu weźmy f(26):

$f(26) = f(16) + f(10) + 2$, $f(10) = f(8) + f(2)$

Wstawiając do powyższych równości obliczony wcześniej wzór rekurencyjny dostajemy:

$$f(10) = (\frac{3}{2} * 8 - 2) + (\frac{3}{2} * 2 - 2) = (\frac{3}{2} * 10 - 2)$$

$$f(26) = (\frac{3}{2} * 16 - 2) + (\frac{3}{2} * 10 - 2) = (\frac{3}{2} * 26 - 2)$$

Łatwo zauważyć, że dla n nieparzystego $f(n) = \left\lceil \frac{3}{2} \right| n - 2$

VII. 4 Równania rekurencyjne liniowe dowolnego stopnia

Na początku tego rozdziału przedstawiony został sposób rozwiązywania dowolnego równania rekurencyjnego liniowego 2 go stopnia. W tej sekcji przedstawimy aparat matematyczny pozwalający na rozwiązanie dowolnego równania liniowego k- tego stopnia. Zaczniemy od przedstawienia szeregów potęgowych, ich własności i związku z funkcjami rekurencyjnymi.

W dużym uproszczeniu szereg potęgowy możemy traktować jako niekończący się wielomian, powstały np. w wyniku podzielenia przez siebie dwóch wielomianów skończonych:

$(1-x)^{-1} = 1 + x + x^2 + x^3 + x^4 ...$

Współczynniki szeregu potęgowymi oznaczane są stałymi, np.:

$U(x) = u_0 + u_1x + u_2x^2 + u_3x^3 + ...$

Dla szeregów potęgowych zachodzi ważne twierdzenie:

Założenie: F – ciało,

$U(x) \in F[x]$ (pierścień wielomianów o współczynnikach z ciała F

Teza:

$U(x)$ jest odwracalny $\leftrightarrow$ $u_0 \neq 0$

Szereg potęgowy możemy utożsamiać z ciągiem jego współrzędnych np.

$U(x) \leftrightarrow (u_0, u_1, u_2, u_3, ...)$

By taki ciąg współrzędnych przesunąć w prawo o k pozycji należy wykonać mnożenie:

$U(x)*x^k \leftrightarrow (0, 0, ..., 0, u_0, u_1, u_2, u_3, ...)$ - ilość początkowych zer, to k.

By taki ciąg współrzędnych przesunąć w lewo o k pozycji, trzeba najpierw odjąć od odpowiadającego mu szeregu potęgowego pierwszych k wyrazów, a następnie podzielić go przez x^k:

$$[U(x) - (u_0 + u_1x + u_2x^2 + ... + u_{k-1}x^{k-1})]*(1/x^k) \leftrightarrow (u_k, u_{k+1}, u_{k+2}, u_{k+3}, ...)$$

Przy rozwiązywaniu równań rekurencyjnych bardzo przydatny okazuje się szereg $(1-x)^{-1}$, którego współczynniki są równe (1, 1, 1, 1, ...). Można też wykonać operację $(1-\alpha x)^{-1}$, by przekonać się, że:

$$(1-\alpha x)^{-1} = 1 + \alpha x + \alpha^2x^2 + \alpha^3x^3 + \alpha^4x^4 ...$$

Dla szeregu potęgowego U(x) zachodzi również:

$$xU(x)' \leftrightarrow u_1x + 2u_2x^2 + 3u_3x^3 + ... + ku_kx^k$$

stąd łatwo zauważyć, że $[(1-\alpha x)^{-1}]' \leftrightarrow (1, 2, 3, 4, 5, ...)$

Umiejętność identyfikowania różnych szeregów potęgowych i rozkładania ich na szeregi, które zostały zademonstrowane, jest podstawą do rozwiązywania dowolnego równania liniowego k- tego stopnia.

Każde równanie rekurencyjne jest ciągiem wartości (u_0, u_1, u_2, ...), które odtąd będziemy identyfikowali z odpowiednim szeregiem potęgowym. Szereg potęgowy, którego współrzędne są wartościami równania rekurencyjnego będziemy nazywali **funkcją tworzącą** równania rekurencyjnego.

Rozwiążemy teraz równanie liniowe 2go stopnia korzystając z przedstawionej teorii:

Mamy dane równanie:

$$u_0 = 0 \qquad u_1 = 1$$
$$u_{n+2} - 5u_{n+1} + 6u_n = 0$$

którego rozwiązaniem będzie funkcja tworząca o współczynnikach (u_0, u_1, u_2, u_3, ...). Rozpiszmy naszą funkcję tworzącą wykorzystując powyższe równanie rekurencyjne:

$$\begin{aligned} U(x) &= u_0 + u_1x + u_2x^2 + u_3x^3 + ... \\ &= 0 + x + (5u_1 - 6u_0)x^2 + (5u_2 - 6u_1)x^3 + (5u_3 - 6u_2)x^4 + ... \\ &= x + 5x(0 + u_1x + u_2x^2 + ...) - 6x^2(u_0 + u_1x + u_2x^2 + ...) \\ &= x + 5xU(x) - 6x^2U(x) \end{aligned}$$

stąd $U(x) = x/(6x^2 - 5x + 1)$

co po znalezieniu rozwiązania równania $6x^2 - 5x + 1 = 0$ daje:

$$U(x) = x/((1-2x)(1-3x))$$

Mimo iż U(x) jest już dobrze wyrażoną funkcją tworzącą, nie jesteśmy w stanie odgadnąć jej współczynników będących zarazem rozwiązaniem równania rekurencyjnego. W tym celu musimy rozpisać otrzymany wynik na ułamki proste, a następnie tak dobrać stałe A i B, by funkcja tworząca pozostała bez zmian:

$$U(x) = x/((1-2x)(1-3x)) = A/(1-2x) + B/(1-3x)$$

Stąd $A = -1,\ B = 1$
Czyli ostatecznie $U(x) = 1/(1-3x) - 1/(1-2x)$
Skoro $(1-\alpha x)^{-1} = 1 + \alpha x + \alpha^2x^2 + \alpha^3x^3 + \alpha^4x^4 \ldots$
to $u_n = 3^n - 2^n$.

Formalizacja:

Funkcja tworząca dla równania rekurencyjnego k-tego stopnia ma postać:

$$U(x) = R(x) / (1 + a_1x + a_2x^2 + \ldots + a_kx^k) \qquad (*)$$

gdzie $a_1, a_2, \ldots$ są liczbami z równania rekurencyjnego $u_{n+k} + a_1u_{n+k-1} + \ldots + a_ku_n = 0$

Dla takiej funkcji tworzącej definiuje się równanie pomocnicze:

$$t^k + a_1t^{k-1} + a_2t^{k-2} + \ldots + a_k = 0 \qquad (**)$$

Równanie (**) ma w zbiorze liczb zespolonych k pierwiastków. Wśród nich wyróżniamy pierwiastki $\alpha_1, \alpha_2, \ldots, \alpha_s$ z odpowiednimi krotnościami $m_1, m_2, \ldots, m_s$, przy czym suma wszystkich krotności wynosi k. Równanie (**) może być więc zapisane w następujący sposób:

$$(t- \alpha_1)^{m1} * (t- \alpha_2)^{m2} * \ldots * (t- \alpha_s)^{ms} = 0$$

co po podstawieniu za $t \rightarrow 1/x$ i pomnożeniu przez x^k daje:

$$(1- \alpha_1x)^{m1} * (1- \alpha_2x)^{m2} * \ldots * (1- \alpha_sx)^{ms} = 0$$

stąd funkcja tworząca może być zapisana jako:

$$U(x) = R(x)/ ((1- \alpha_1x)^{m1} * (1- \alpha_2x)^{m2} * \ldots * (1- \alpha_sx)^{ms})$$

Teraz pozostaje już tylko rozłożyć funkcję tworzącą na czynniki, które pomogą nam zidentyfikować wartości równania rekurencyjnego.

$$U(x) = (\ldots)/(1- \alpha_1x)^{m1} + (\ldots)/(1- \alpha_2x)^{m2} + \ldots + (\ldots)/(1- \alpha_sx)^{ms}$$

Gdyby krotność każdego rozwiązania równania była równa 1, to rozkład byłby trywialny, a dobór odpowiednich czynników natychmiastowy. Gdy jednak krotności są różne od 1, należy przeprowadzić następującą redukcję:

$$\frac{(\ldots)}{(1-\alpha_i x)^m} = \sum_{j=1}^{m} \frac{C_{i,j}}{(1-\alpha_i x)^j}$$

gdzie $C_{i,j}$ są odpowiednimi stałymi.

Jako przykład weźmy następujące równanie rekurencyjne:

$u_0 = 0, \quad u_1 = -9, \quad u_2 = 1, \quad u_3 = 21$

$$u_{n+4} - 5u_{n+3} + 6u_{n+2} + 4u_{n+1} - 8u_n = 0$$

którego równanie pomocnicze ma postać:

$t^4 – 5t^3 + 6t^2 + 4t – 8 = 0$

lub $(t - 2)^3(t + 1) = 0$

Każdemu pierwiastku tego równania będzie w rozwiązaniu równania rekurencyjnego odpowiadał wielomian stopnia o jeden mniejszego niż krotność pierwiastka:

$u_n = (An^2 + Bn + C)*2^n + D*(-1)^n$

Po rozwiązaniu zestawu równań wyznaczającego wartości A, B, C, D:

$u0 = 0 = C + D$

$u1 = -9 = 2A + 2B + 2C – D$

$u2 = -1 = 16A +8B +4C + D$

$u3 = 21 = 72A + 24B + 8C – D$

otrzymujemy końcowy wzór na n- ty element równania rekurencyjnego:

$u_n = (n^2 - n - 3)*2^n + 3*(-1)^n$

Na koniec przedstawimy jeszcze równanie rekurencyjne liniowe, które posiada czynnik wolny i wyznaczymy jego funkcję tworzącą:

$u_0 = 0, \qquad u_1 = 1$

$u_{n+2} – u_{n+1} – 6u_n = \mathbf{n}$

$U(x) = u_0 + u_1x + u_2x^2 + u_3x^3 + ...$

$= x + (u_1 + 6u_0 + 0)x^2 + (u_2 + 6u_1+1)x^3 + (u_3 + 6u_2 + 2)x^4 + ...$

$= x + (u_1x^2 + u_2x^3 + u_3x^4 +...) + 6(u_0x^2 + u_1x^3 + u_2x^4 + ...) + + (x^3 +2x^4 + 3x^5 +...)$

$= x + xU(x) + 6x^2U(x) + x^3(1 + 2x + 3x^2 + ...)$

$= x + xU(x) +6x^2U(x) + x^3(1-x)^{-2}$

czyli funkcja tworząca ma postać:

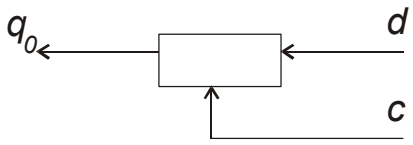
$U(x) = (x^3(1 - x)^{-2} + x)/(1 – x – 6x^2)$

Wyznaczenie kolejnych wyrazów odpowiadającego jej równania rekurencyjnego może być dobrym ćwiczeniem utrwalającym przedstawioną teorię.

Rozdział VIII Przykładowe procesy rekurencyjne

VIII. 1 Stan rejestru przesuwającego

Załóżmy, że dany jest przerzutnik pokazany na poniższym rysunku:

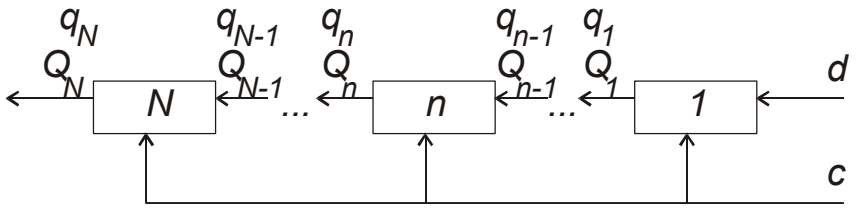


d napięcie danych
c impuls zapisu
q_0 napięcie stanu przerzutnika

Rysunek 40 Przerzutnik danych

Pod wpływem impulsu c do przerzutnika zapisywana jest dana d, tzn. stan q przyjmuje wartość d.

Rejestr przesuwający jest szeregowym zestawem przerzutników pokazanym na poniższym rysunku.



Q_n stan n- tego przerzutnika

Rysunek 41 Rejestr przesuwający

W rejestrze tym pod wpływem impulsu zapisu c stan Q_{n-1} zostaje zapisany do następnego przerzutnika.
Oznaczamy stan początkowy rejestru przez:

$$Q^0 = [\, q_N{}^0, \ldots, q_n{}^0, \ldots, q_1{}^0 \,]$$

Po pierwszym impulsie zapisu stan rejestru przyjmuje postać:

$$Q^1 = [\, q_N{}^1, \ldots, q_n{}^1 \quad , \ldots, q_n{}^1 \,]$$

przy czym

$q_1^1 = d$
$q_2^1 = q_1^0$
.....
$q_n^1 = q_{n-1}^0$
.....
$q_N^1 = q_{N-1}^0$

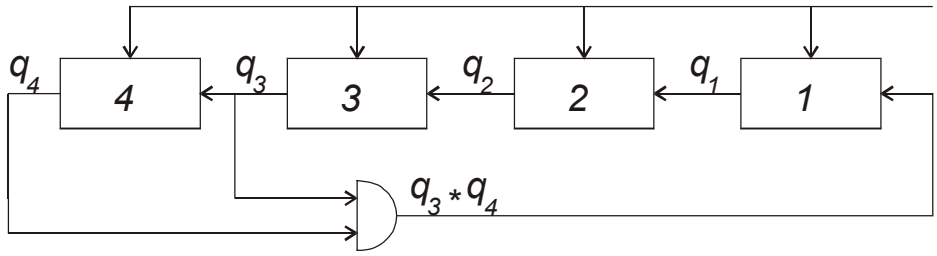
W analogiczny sposób modyfikowany jest stan rejestru przesuwającego po kolejnych impulsach zapisu. W rejestrach przesuwających ze sprzężeniem zwrotnym sygnał d jest funkcją stanu rejestru:

$$d = f(q_N,...,q_n ,..., q_1)$$

Funkcję tę realizuje układ bramek logicznych. Tak więc stany rejestru opisują równania rekurencyjne:

$$q_1^k = f(q_1^{k-1}, ..., q_n^{k-1}, \ldots, q_N^{k-1})$$
$$q_n^k = q_{n-1}^{k-1} \qquad n=2, ..., N$$

Na podstawie danego stanu początkowego Q^o można wygenerować graf stanów Q^k, k = 1,...,K.



Rysunek 42 Przykładowa modyfikacja rejestru przesuwającego

Stan początkowy rejestru:

$$Q^0 = [q_4^0, q_3^0 , q_2^0 , q_1^0] = [1101]$$

Kolejne stany tworzą ciąg przedstawiony na rysunku

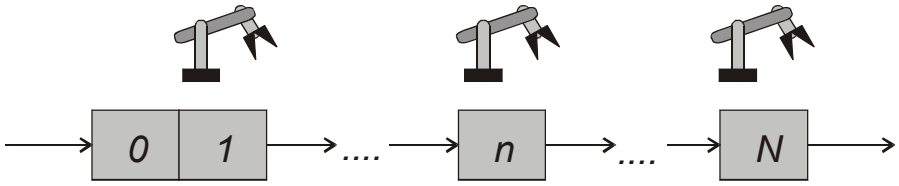
$Q^0 = [1101]$
$Q^1 = [1011]$
$Q^2 = [0110]$
$Q^3 = [1100]$
$Q^4 = [1001]$
$Q^5 = [0010]$

$Q^6 = [\ 0100\ ]$
$Q^7 = [\ 1000\ ]$
$Q^8 = [\ 0000\ ]$

Po 8 impulsach rejestr znajdzie się w ustalonym stanie [0, 0, 0, 0, 1]. W analogiczny sposób można podstawić zmiany stanu rejestru przesuwającego dowolnego układu realizującego funkcję f.

VIII. 2 Stan procesu montażu

Załóżmy, że dana jest linia montażowa pokazana na poniższym rysunku



Rysunek 43 Linia montażowa

Linia składa się z N stacji montażowych. Operacje na stacjach są wykonywane przez roboty przemysłowe. Stacja ni 0 służy do sprowadzania nowych obiektów do montażu.

W trakcie cyklu każdy robot wykonuje operacje na obiekcie znajdującym się na jego stacji. Po zakończeniu operacji przez wszystkie roboty, każdy obiekt jest przesuwany (synchronicznie) na kolejną stację.

Na linii montowane są równocześnie obiekty M różnych wersji. Czasy montażu obiektów różnych wersji na stacjach dane są macierze:

$$T = [\ t_{m,\,n}\ ] \qquad \begin{matrix} m = 1,..,M \\ n = 1,...,N \end{matrix}$$

gdzie: $t_{m,n}$ – czas montażu obiektu m tej wersji na n-tej stacji
Czas każdego cyklu wyznaczamy jako

$$C^k = \max_{1 \le n \le N} C_n^{\,k}$$

gdzie: $C_n^{\,k} = t_{m,\,n}$ jeśli w k- tym takcie na n- tej stacji był montowany obiekt m- tej wersji.

Załóżmy, że zamówienia na obiekty dane są w wektorze;

$$Z = [\, Z_m \,] \qquad m = 1,...,M$$

gdzie: Z_m– liczba obiektów m- tej wersji

Problem polega na zmontowaniu zamówionych obiektów w najkrótszym czasie. Oznaczamy przez X^k wektor stanu linii montażowej po k- tym cyklu, (k = 0, 1, ..., K).
Stan ten definiujemy następująco;

$$X^k = [\, X_n^{\,k} \,] \qquad n = 0,1, \ldots , N$$

gdzie: $X_n^{\,k}$ - numer wersji obiektu znajdującego się na n- tej stacji po k- tym cyklu.

Stan początkowy X^0 jest dany. Załóżmy, że

$$X_0^{\,0} = 0$$
$$X_n^{\,0} > 0 \qquad n = 1, ...,N$$

To znaczy, że na każdej stacji montażowej znajduje się jakiś obiekt. Czas pierwszego cyklu wyznaczamy z warunku:

$$C^1 = \max_{1 \le n \le N} C_n^{\,1}$$

przy czym $C_n^{\,1} = t_{m,n}$
oraz $m = X_n^{\,0}$

W trakcie pierwszego (i każdego następnego)cyklu do stacji załadunkowej jest podawany obiekt wybranej wersji. Decyzja o obiekcie załadowanym do linii w k-tym cyklu będzie oznaczona przez d^k , przy czym:

$$d^k = \begin{cases} m, & \text{jeśli w k- tym cyklu do linii jest załadowany obiekt m – tej wersji} \\ 0, & \text{jeśli w k-tym cyklu do linii nie jest załadowany żaden obiekt} \end{cases}$$

Stan linii montażowej po takcie k jest zależny od stanu linii po poprzednim takcie oraz od decyzji w k- tym takcie. Rekurencyjne równania stanu mają postać:

$$X^k = f_x (X^{k-1} , d^k)$$

$$k = 1,..., K$$

W postaci jawnej mamy

$$X_n^k = X_{n-1}^{k-1}$$

$$n = 2,...,N$$

$$X_1^k = d^k$$

Ponadto obliczamy czas cyklu C^k, k = 1, ..., K.

W analogiczny sposób można przedstawić rekurencyjne równanie zamówień:

$$Z^k = f_z (Z^{k-1} , X^{k-1})$$

$$k =1, ..., K$$

W postaci jawnej mamy:

$$Z_m^k = \begin{cases} Z_m^{k-1} - 1 , & \text{dla } X_N^{k-1} = m \\ Z_m^{k-1} , & \text{dla } X_N^{k-1} \neq m \end{cases}$$

Rekurencyjne równania stanu linii montażowych oraz zamówień pozwalają symulować przebieg procesu – dla różnych decyzji d^k , k = 1, ..., K. Zatem generowane są trajektorie stanów;

$$(X^0 , Z^0) \rightarrow (X^1, Z^1) \rightarrow (X^k, Z^k) \rightarrow (X^K, Z^K)$$

na podstawie strategii decyzji

$$d^1 , d^2 , ..., d^k , ..., d^K$$

Warunkiem zakończenia procesu jest zrealizowanie zamówień tzn.

$$Z_m^k \leq 0$$

$$m = 1, ..., M$$

Stan początkowy X^0 może być dowolny. Stan końcowy X^k może być wektorem o zerowych elementach. Miarą jakości strategii decyzji jest czas realizacji zamówień:

$$Q = \sum_{k=1}^{k=K} C^k \rightarrow \min$$

Jak dotąd nie istnieje algorytm optymalnego rozwiązania sformułowanego problemu.

VIII. 3 Fundusz emerytalny

Załóżmy, że z danego funduszu emerytalnego należy dokonać określoną liczbę wypłat. Po tych wypłatach fundusz emerytalny zostanie wyczerpany. Problem polega na wyznaczeniu wartości wypłat.
Załóżmy, że dane są:

F_0 - początkowa wartość funduszu
N - liczba wypłat
r_n - stopa procentowa n- tego okresu wypłat, (n=1, ..., N)

Oznaczmy przez:

W_n - wypłata na zakończenie n- tego okresu
F_n - wartość funduszu po wypłacie W_n.

Fundusze F_n muszą spełniać warunki

$F_n > 0$ dla n=0, ..., N-1
oraz $F_N = 0$

Stosując zasadę równoważności kapitału dla kolejnych terminów n, n=1, ..., N otrzymamy równania:

$$F_0 A(0, 1) - W_1 = F_1$$
$$F_0 A(0, 2) - W_1 A(1, 2) - W_2 = F_2$$
$$F_0 A(0, 3) - W_1 A(1, 3) - W_2 A(2, 3) - W_3 = F_3$$
.....

$$F_0 A(0, n) - \sum_{i=1}^{i=n-1} W_i A(i, n) - W_n = F_n$$

.....

$$F_0 A(0, N) - \sum_{i=1}^{i=N-1} W_i A(i, N) - W_N = F_N$$

gdzie A(i, n) – czynnik akumulacji z i- tego na n- ty termin

Dla oprocentowania prostego:

$A(i, n) = 1 + r_{i+1} + ... + r_n$

Dla oprocentowania składanego:

$A(i, n) = (1 + r_{i+1})* \ldots *(1 + r_n)$

Dla wypłat przyjmuje się, że tworzą one:

- Postęp arytmetyczny
  $W_n = W_1 + (n-1)\Delta W$, gdzie ΔW – dane
- Postęp geometryczny
  $W_n = W_1 (1 + q)^{n-1}$ gdzie q – dane

Z układu powyższych równań można wyznaczyć ciąg wypłat W_n oraz funduszy F_n, n=1, ..., N.

VIII. 4 Fundusz amortyzacji

Amortyzacja tzw. środków trwałych pozwala firmom gromadzić fundusze na zakup nowego środka trwałego (np. samochodu) po całkowitym zużyciu poprzedniego. Istotne znaczenie ma fakt, że kwoty funduszu amortyzacji nie są objęte podatkiem dochodowym. Jednakże dla funduszu amortyzacji określane są górne limity:

$g_1, ..., g_n, \ldots, g_N$

które nie mogą być przekroczone.

Problem polega na tworzeniu funduszu amortyzacji poprzez wpłat:

$x_1, ..., x_n, ..., x_N$

tak, by wartości oprocentowanych wpłat były równe ustalonym limitom.

Załóżmy, że dane są stopy procentowe

$r_1, ..., r_n, ..., r_N$

w kolejnych latach.

Stosując zasadę równoważności kapitału wyznaczyć kolejno wpłaty $x_1, ..., x_n, ..., x_N$ z następujących rekurencyjnych równań:

$x_1 = g_1$

$x_1(1 + r_2) + x_2 = g_2$

.....

$x_1(1 + r_2)* ... *(1 + r_n) + x_2(1 + r_3)* ... *(1 + r_n) + \ldots + x_n = g_n$

……

$x_1(1 + r_2)* ... *(1 + r_N) + x_2(1 + r_3)* ... *(1 + r_N) + \ldots + x_N = g_N$

VIII. 5 Kredyty ratalne

Załóżmy, że kredyt P_0 ma być spłacony w N ratach:

- Kapitałowych K_n, (n=1, ..., N) oraz
- Odsetkowych I_n, (n=1, ..., N)

Kredyt może być spłacany w warunkach oprocentowania prostego lub składanego ze stopą stałą lub zmienną. Dane są stopy procentowe r_n, poszczególnych okresów, (n=1, ..., N).

Raty kredytu są wyznaczane z zasady równoważności kapitału, w postaci:

$$P_0\, A(0, N) = \sum_{n=1}^{n=N} (K_n + I_n)\, A(n, N) \qquad (1)$$

gdzie: A(n, N) – czynnik oprocentowania z n- tego terminu na N- ty termin

Czynniki oprocentowania mają postać:

- Dla oprocentowania prostego $A(n, N) = 1+r_{n+1}+...+r_N$ (2)
- Dla oprocentowania składanego $A(n, N) = (1+r_{n+1})*...*(1+r_N)$ (3)

Spłata kredytu w warunkach oprocentowania składanego lub prostego wymaga wyznaczenia kolejno:

- Rat kapitałowych K_n, (n=1, ..., N)
- Rat odsetkowych I_n, (n=1, ..., N)

Uwzględniając (2) w (1) otrzymamy rekurencyjny algorytm wyznaczania rat dla oprocentowania składanego w postaci:

Krok 1

Ustalić K_1 tak by $K_1 < P_0$

Obliczyć I_1 ze wzoru $I_1 = P_0 * r_1$

Obliczyć $P_1 = P_0 – K_1$

.....

Krok n (n=2, ..., N-1)

Ustalić K_n tak by $K_n < P_{n-1}$

Obliczyć I_n ze wzoru $I_n = P_{n-1} * r_n$

Obliczyć $P_n = P_{n-1} – K_n$

.....

Krok N

Ustalić K_n tak by $K_n = P_{N-1}$

Obliczyć I_N ze wzoru $I_N = P_{N-1} * r_N$

Sprawdzić, czy $P_N = P_{N-1} – K_N = 0$

Tak więc w n-tym terminie należy spłacić razem ratę całkowitą R_n, (n=1, ..., N), przy czym

$R_n = K_n + I_n$

Uwzględniając (3) w (1) otrzymamy rekurencyjny algorytm wyznaczania rat dla oprocentowania prostego, w postaci:

Krok 1

Ustalić K_1 tak by $K_1 < P_0$

Obliczyć I_1 ze wzoru $I_1 = (P_0 * r_1) / (1+r_2+...+r_N)$

Obliczyć $P_1 = P_0 – K_1$

.....

Krok n, (n=2, ..., N-1)

Ustalić K_n tak by $K_n < P_{n-1}$

Obliczyć I_n ze wzoru $I_n = (P_{n-1} * r_n) / (1+r_{n+1}+...+r_N)$

Obliczyć $P_n = P_{n-1} – K_n$

.....

Krok N

Ustalić K_N tak by $K_N = P_{N-1}$

Obliczyć I_N ze wzoru $I_N = P_{N-1} * r_N$

Sprawdzić czy $P_N = P_{N-1} – K_N = 0$

Analogicznie jak dla oprocentowania składanego w n- tym terminie należy spłacić ratę całkowitą R_n, (n=1, ..., N).

W przedstawionych wyżej algorytmach rekurencyjnych raty odsetkowe I_n są obliczane z różnych wzorów.

Zakończenie

W dzisiejszych czasach, gdy co kilka lat podwaja się prędkość obliczeniowa komputerów, a sieci komputerowe oferują coraz to większe przepustowości często zapomina się, że napisanie poprawnie działającego, szybkiego algorytmu także ma duże znaczenie.

Zespoły programistów piszą aplikacje coraz to bardziej skomplikowane, które jednak są najczęściej złożeniem innych mniejszych programów, niekoniecznie najszybszych i niezawodnych. Stąd programy komputerowe stają się zawodne i działają coraz wolniej, co zmusza producentów sprzętu do produkcji szybszych komputerów, które znów wpadają w ręce zespołów programistów..

Umiejętność wykorzystania klasycznych algorytmów informatycznych do rozwiązania skomplikowanych problemów jest tym, czego brakuje w dzisiejszej sztuce programowania. Nie należy zapominać, że dłuższy czas obliczeń danego algorytmu propaguje się, co przy rozbudowanych sieciach komputerowych wydłuża czas oczekiwania na konkretny rezultat.

Przedstawienie trudnych problemów w postaci grafów daje niekiedy możliwość innego spojrzenia na zagadnienie, co jest podstawą do znalezienia lepszego i bardziej pewnego algorytmu. Algorytmy rekurencyjne w połączeniu ze strukturami grafowymi doskonale nadają się do reprezentacji wielu problemów, z którymi możemy spotkać się na co dzień.

Analiza efektywności i poprawności rekurencji wraz umiejętnością prawidłowego jej wykorzystania jest elementem bardzo istotnym w dzisiejszej sztuce programowania.

Literatura

1) N. WIRTH, " Algorithms + Data Structures = Programs ", Prentice-Hall, 1976
2) G. BRASSARD, P. BRATLEY "Algorithmics: Theory and Practice" Prentice-Hall, 1988
3) N. H. XUONG, "Mathematiques Discretes et Informatique", Masson, Paris 1991
4) C. L. LIU. „Introduction to Combinatorial Mathematics " McGraw-Hill, 1968
5) T.H. CORMEN, C. E. LEISERSON, R. L. RIVEST „Introduction to Alghorithms", Massachusetts Institute of Technology, 1990
6) A.V. OHO, J.E. HOPCROFT, J.D. ULLMAN "Projektowanie i analiza algorytmów komputerowych" PWN, Warszawa 1983
7) J. J. F. CAVANAGH. "Digital Computer Arithmetic". McGraw-Hill, 1984.
8) DONALD E. KNUTH "Sorting and Searching", volume 3 of "The Art of Computer Programming" Addison-Wesley, 1973
9) N. BIGGS "Discrete Mathematics"