Podejście zachłanne, a programowanie dynamiczne

Algorytm zachłanny pobiera po kolei elementy danych, za każdym razem wybierając taki, który wydaje się „najlepszy” w zakresie spełniania pewnych kryteriów i bez względu na wybory dokonane wcześniej lub potencjalnie możliwe do wykonania w przyszłości.

Podobnie jak programowanie dynamiczne również algorytmy zachłanne są często używane do rozwiązywania problemów optymalizacyjnych.

W przypadku programowania dynamicznego w celu podziału realizacji problemu na mniejsze realizacje wykorzystywana jest właściwość rekurencyjna.

Algorytm zachłanny znajduje rozwiązanie w wyniku podjęcia szeregu decyzji, z których każda wydaje się w danym momencie najlepsza. Oznacza to, że każdy dokonany wybór jest optymalny lokalnie. Podejście takie daje nadzieję, że osiągnie się rozwiązanie optymalne globalnie, choć nie zawsze tak się stanie.

Przykładem realizacji zachłannej problemu jest wydawanie reszty jak najmniejsza ilością monet. Realizacja mogłaby wyglądać nastepująco:

```
while (są jeszcze monety i realizacja pozostaje
       nierozwiązana) {
       pobierz największą dostępna monetę;
                                    //procedura
                                    //wyboru
      if (dodanie monety powoduje przekroczenie
          kwoty reszty)           //sprawdzenie
                                  //wykonalności
          odrzuć monetę;
      else
          dodaj monetę do reszty;
```

```
        if (całkowita wartość reszty jest równa
            należnej kwocie)          //sprawdzenie
                                      //rozwiązania
            realizacja rozwiązania;
}
```

Algorytm zachłanny nie gwarantuje otrzymania optymalnego rozwiązania. Zawsze musimy określić, czy dzieje się tak w przypadku określonego algorytmu zachłannego.

Algorytm zachłanny rozpoczyna się od zbioru pustego, do którego po kolei dodawane są elementy – aż do momentu, gdy zbiór będzie reprezentował rozwiązanie realizacji problemu.

Każdy przebieg składa się z następujących etapów:

- ***Procedura wyboru*****,** służąca do wybrania kolejnego elementu dodawanego do zbioru.Wybór jest dokonywany zgodnie z kryterium zachłannym, które w danym momencie spełnia pewien warunek optymalny lokalnie.

- ***Sprawdzenie wykonalności*****,** służące do określenia, czy ów zbiór jest akceptowalny, dzięki sprawdzeniu, czy istnieje możliwość uzupełnienia go w taki sposób, który zapewni osiagnięcie rozwiązania realizacji problemu.

- ***Sprawdzenie rozwiązania*****,** służące do określenia, czy ów zbiór stanowi rozwiązanie realizacji problemu.

Pakowanie najcenniejszego plecaka

Problem: zapakować do plecaka o ograniczonej pojemności możliwie najbardziej wartościowych rzeczy.

Dane: n rzeczy (towarów, produktów) R_1, R_2, ..., R_n , każda w nieograniczonej ilości; rzecz R_i , waży (lub zajmuje miejsce o wielkości) w_i jednostek oraz ma wartość p_i . Dana jest także maksymalna pojemność plecaka wynosząca W jednostek.

Wynik: Ilości q_1, q_2 , ..., q_n , poszczególnych rzeczy (mogą być zerowe), których całkowita waga/pojemność nie przekracza W i których sumaryczna wartość jest największa wśród wypełnień plecaka rzeczami o wadze/pojemności nie przekraczającej W .

Matematycznie odpowiada to znalezieniu liczb q_1, q_2 , ..., q_n , dla których wartość sumy (całkowita wartość plecaka)

$$p_1 q_1 + p_2 q_2 + ... + p_n q_n$$

jest największa i jednocześnie spełniony jest warunek (pojemność plecaka W nie zostaje przekroczona)

$$w_1 q_1 + w_2 q_2 + ... + w_n q_n \leq W$$

Liczby q_1, q_2 , ..., q_n są nieujemnymi liczbami całkowitymi.

Wersja decyzyjna problemu plecakowego polega na założeniu, że wielkości q_i mogą przyjmować wartości 1- gdy rzecz i dokładamy do plecaka lub 0 - gdy jej nie bierzemy (***problem plecakowy 0-1***). Wersja ta nie jest ograniczeniem, bo każdy egzemplarz danej rzeczy można nazwać inaczej i traktować osobno.

Problem możemy rozwiązać zarówno przez działanie zachłanne jak i programowanie dynamiczne.

1. Algorytm zachłanny dla ogólnego problemu plecakowego.

Przy pakowaniu plecaka ścierają się ze sobą trzy kryteria wyboru kolejnych rzeczy do zapakowania:

- wybierać najcenniejsze rzeczy czyli w kolejności nierosnących wartości p_i,
- wybierać rzeczy zajmujące najmniej miejsca, czyli w kolejności niemalejących wag w_i,
- wybierać rzeczy najcenniejsze w stosunku do swojej wagi, czyli w kolejności nierosnących wartości ilorazu p_i/w_i ; iloraz ten można uznać za jednostkową wartość rzeczy i

W metodzie zachłannej zawartość plecaka kompletowana jest krok po kroku dokonując wyboru zgodnie z jednym z powyższych kryteriów. Strategia ta nie zawsze prowadzi do optymalnego rozwiązania (najbardziej wartościowego wypełnienia plecaka).

Przykład:

i	1	2	3	4	5	6	W
p_i	6	4	5	7	10	2	
w_i	6	2	3	2	3	1	23

p_i/w_i | 6/6 | 4/2 | 5/3 | 7/2 | 10/3 | 2/1|

Według kryterium 1 bierzemy: nr 5 (7 szt.) + nr 4 (1 szt.) = **77**
Według kryterium 2 bierzemy: nr 6 (23 szt.) = **46**
Według kryterium 3 bierzemy: nr 4 (11 szt.) + nr 6 (1 szt.) = **79**

Bardzo łatwo zauważyć, że kryterium 3 prowadzi do najbardziej optymalnego rozwiązania.

Bardzo łatwo też wskazać lepsze rozwiązanie:

nr 4 (10 szt.) + nr 5 (1szt.) = **80**

Metoda zachłanna prowadzi do rozwiązań przybliżonych w stosunku do rozwiązania optymalnego.

W algorytmie ograniczymy się do kryterium 3 tzn. założymy, że rzeczy przewidziane do zabrania uporządkowane są zgodnie z:

$$p_1/w_1 \geq p_2/w_2 \geq \ldots \geq p_n/w_n$$

Algorytm:

Problem: zapakować do plecaka o ograniczonej pojemności możliwie najbardziej wartościowych rzeczy.

Dane: n rzeczy, każda w nieograniczonej ilości, ważących w_i i mających wartość p_i (i=1,2,...,n); rzeczy zostały ponumerowane tak, że spełniona jest powyższa nierówność. Dana jest też maksymalna pojemność plecaka W.

Wynik: ilości $q_1, q_2, \ldots, q_n$, poszczególnych rzeczy (mogą być zerowe), których całkowita pojemność nie przekracza W.

Krok 1. Dla kolejnych rzeczy $i=1,2,..,n$ uporządkowanych zgodnie z kryterium 3 (nierówność powyżej) , wykonać krok 2.

Krok 2. Określić największą wartość q_i, spełniającą nierówność $w_i q_i \leq W$. Przyjąć $W = W - w_i q_i$.

Krok 3. Znaleziony ładunek plecaka ma wartość

$$p_1 q_1 + p_2 q_2 + \ldots + p_n q_n$$

2. Programowanie dynamiczne dla ogólnego problemu plecakowego.

W programowaniu tym nie potrafimy przewidzieć, z których rozwiązań pod-problemów będziemy korzystać i dlatego rozwiązujemy wszystkie mniejsze problemy.

Dla problemu plecakowego mniejsze problemy odpowiadają mniejszej liczbie rodzajów pakowanych rzeczy (n) i mniejszej pojemności plecaka W .

Rozwiązania pod-problemów umieszcza się w tablicy P i skojarzonej z nią tablicy Q.

Tablica P dla danych z przykładu:

i \ *j*	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
1	0	0	0	0	0	6	6	6	6	6	6	12	12	12	12	12	12	18	18	18	18	18	18
2	0	4	4	8	8	12	12	16	16	20	20	24	24	28	28	32	32	36	36	40	40	44	44
3	0	4	5	8	9	12	13	16	17	20	21	24	25	28	**29**	32	33	36	37	40	41	44	45
4	0	7	7	14	14	21	21	28	28	35	35	42	42	49	49	56	56	63	63	70	70	77	77
5	0	7	10	14	17	21	24	28	31	35	38	42	45	49	52	56	59	63	66	70	73	77	80
6	2	7	10	14	17	21	24	28	31	35	38	42	45	49	52	56	59	63	66	70	73	77	**80**

i - numeracja kolejnych rzeczy wykorzystywanych przy pakowaniu
j - numeracja kolejnych pojemności plecaka

$P_{i,j}$ - wartość optymalnego wypełnienia plecaka o pojemności j rzeczami, których numery mieszczą się między *1*, a *i*

Kolejność rozpatrywania rzeczy nie ma znaczenia (przyjęto kolejność jak w tabeli danych).

Algorytm programowania dynamicznego jest sposobem wypełniania pól tablicy P, a pola te są wypełniane wierszami.

Wypełnienie pierwszego wiersza jest proste bo dysponujemy tylko rzeczami nr 1 i możemy ich wziąć 1, 2 lub 3 sztuki zależnie od pojemności plecaka.

Przy wypełnianiu kolejnych wierszy korzystamy z już wypełnionych wierszy zgodnie z **zasadą optymalności Bellmana**:
na każdym kroku należy podejmować najlepszą decyzję z uwzględnieniem stanu wynikającego z poprzednich decyzji.

W naszym algorytmie:

jeśli pojemność plecaka ***j*** jest taka, że zmieści się w nim rzecz nr 2, czyli $j \geq w_2$, to wybieramy większą z dwóch wielkości ***P[1][j]*** i $P[2][j-w_2] + p_2$ gdzie

P[1][j]- wartość najlepszego wypełnienia plecaka o pojemności *j* rzeczami nr 1,

$P[2][j-w_2] + p_2$ - wartość wypełnienia plecaka, składającego się z jednej sztuki rzeczy nr 2 oraz optymalnego wypełnienia reszty przestrzeni w plecaku (czyli $j-w_2$) rzeczami nr 1 lub 2.

Czyli ogólnie:

$$P[i][j] = \begin{cases} maximum(P[i-1][j], P[i][j-w_i]+p_i) & \text{jeżeli } w_i \leq j \\ P[i-1][j] & \text{jeżeli } w_i > j \end{cases}$$

Zgodnie z powyższym przepisem można zauważyć, że:
1) dla wiersza 2 (rzeczy nr 1 i 2) - pakujemy tylko rzeczy nr 2
2) dla wiersza 3 (rzeczy nr 1,2,3) - dla parzystych *j* bierzemy tylko rzeczy nr 2, dla nieparzystych jedną sztukę rzeczy nr 2 zamieniamy na rzecz nr 3

3) dla wiersza 4 (rzeczy nr 1,2,3,4) - pakujemy tylko rzeczy nr 4 (najlepszy stosunek wartości do wagi)

4) dla wiersza 5 (rzeczy nr 1,2,3,4,5) - dla parzystych *j* bierzemy tylko rzeczy nr 4, a dla nieparzystych jedną sztukę rzeczy nr 4 zamieniamy na rzecz nr 5

5) dla wiersza 6 (rzeczy 1,2,3,4,5,6) - jak w wierszu 5 (zmiana tylko dla *j*=1)

Aby wiedzieć jaki zestaw rzeczy tworzy poszczególne wypełnienie tworzymy tablicę *Q* skojarzoną z *P*. Wartością *Q[i][j]* jest numer rzeczy, która ostatnia trafiła do plecaka o pojemności *j*, gdy dysponujemy rzeczami o numerach od *1* do *i*.

Tablica Q

i \ *j*	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
1	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	0	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
3	0	**2**	3	**2**	3	**2**	3	**2**	3	**2**	3	**2**	3	2	**3**	2	3	2	3	2	3	2	3
4	0	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4
5	0	4	5	4	5	4	5	4	5	4	5	4	5	4	5	4	5	4	5	4	5	4	5
6	6	**4**	5	**4**	5	**4**	5	**4**	5	**4**	5	**4**	5	**4**	5	**4**	5	**4**	5	**4**	5	4	**5**

Aby znaleźć skład plecaka startujemy z pola $Q_{6,23}$ (rzecz nr 5) i przenosimy się do kolumny mniejszej o wagę rzeczy z pozycji $Q_{6,23}$ (rzecz nr 5 waży 3) i dalej podobnie.

Algorytm:

Dane: n rzeczy, każda w nieograniczonej ilości, ważących w_i i mających wartość p_i $(i=1,2,...,n)$, oraz maksymalna pojemność plecaka W

Wynik: tablica wartości $P_{i,j}$ najlepszych upakowań plecaka o pojemności j, rzeczami rodzajów od 1 do i, dla $i=1,2,...,n$ oraz $j=1,2,...,W$; skojarzona tablica rodzajów rzeczy Q[i][j], dołożonych w ostatnim dopakowaniu plecaka.

Krok 1. { Ustalenie wartości początkowych w tablicach P i Q rozszerzonych, dla ujednolicenia obliczeń, o wiersze i kolumny zerowe.}

Dla $j=1,2,...,W$ przypisz $P[0][j]= 0$, $Q[0][j] = 0$.
Dla $i=1,2,...,n$ przypisz $P[i][0]= 0$, $Q[i][0]= 0$.

Krok 2. Dla kolejnych rzeczy $i=1,2,...,n$ wykonaj krok 3.

Krok 3. Dla kolejnych pojemności plecaka $j=1,2,...,W$ wykonaj krok 4.

Krok 4. Jeśli $j \geq w_i$, {Czyli gdy pojemność plecaka jest wystarczająca, by pomieścić rzecz i } oraz $P[i-1][j] < P[i][j-w_i]+ p_i$, to przypisz $P[i][j] = P[i][j-w_i]+ p_i$ oraz $Q[i][j] = i$,
a w przeciwnym razie pozostaw wartości z poprzedniego wiersza, czyli przypisz $P[i][j] = P[i-1][j]$ oraz $Q[i][j] = Q[i-1][j]$

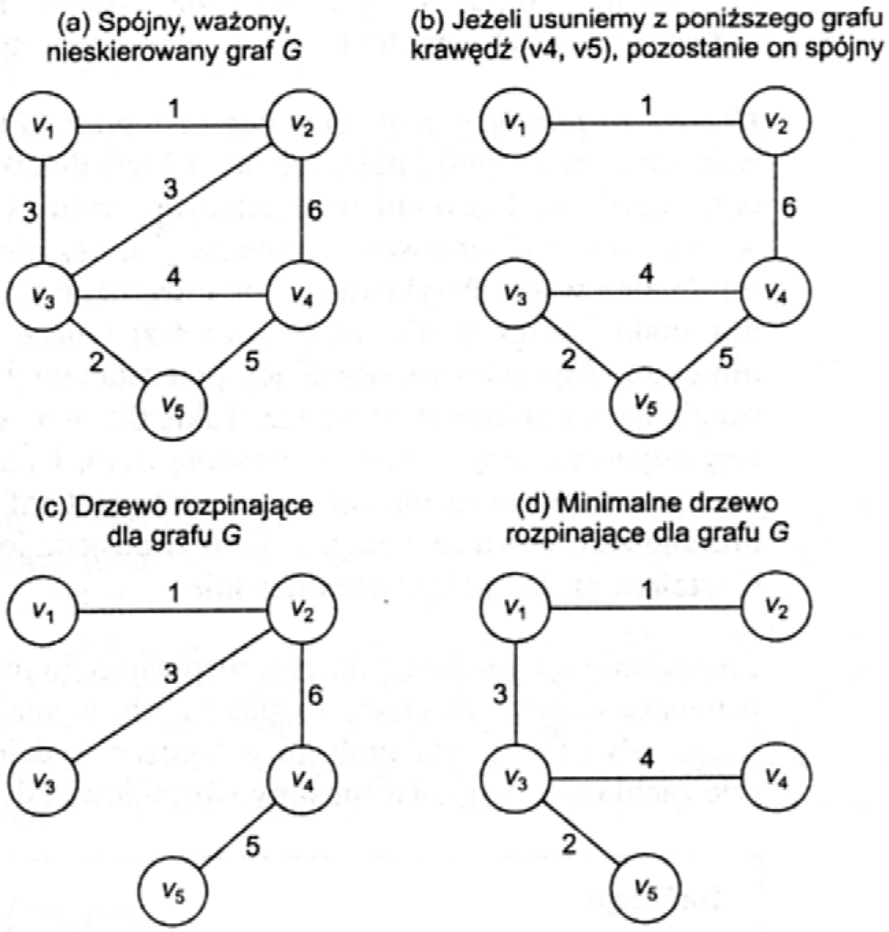
Liczba obliczonych wpisów w tablicy to $n*W$.

Minimalne drzewo rozpinające

Załóżmy, że planista przestrzenny chce połączyć określone miasta drogami w taki sposób, aby było możliwe dojechanie z dowolnego z tych miast do dowolnego innego. **Dążymy do zbudowania najkrótszej sieci dróg.** Do rozwiązania tego problemu niezbędne jest poznanie zagadnień z zakresu teorii grafów.

Graf jest ***nieskierowany*****,** gdy jego krawędzie nie posiadają kierunku. Mówimy wówczas po prostu, że krawędź jest między dwoma wierzchołkami.
Droga w grafie nieskierowanym jest sekwencją wierzchołków, taką że każdy wierzchołek i jego następnik łączy krawędź. Krawędzie nie mają kierunku, więc droga z wierzchołka ***u*** do wierzchołka ***v*** istnieje wtedy i tylko wtedy, gdy istnieje droga z ***v*** do ***u***.

Graf nieskierowany jest nazywany ***spójnym***, kiedy między każdą parą wierzchołków istnieje droga. Grafy z rysunku poniżej są spójne.



W grafie nieskierowanym droga wiodąca z wierzchołka do niego samego, zawierająca co najmniej 3 wierzchołki, wśród których wszystkie wierzchołki pośrednie są różne, jest nazywana ***cyklem prostym.*** Graf nieskierowany nie zawierający żadnych cykli prostych jest określany mianem ***acyklicznego*** (grafy (a), (b) są cykliczne).

Drzewo jest acyklicznym spójnym grafem nieskierowanym (grafy (c), (d) są drzewami). Funkcjonuje też pojęcie ***drzewo korzeniowe*** – to drzewo posiadające jeden wierzchołek, określony jako korzeń.

Szerokie zastosowanie ma problem usuwania krawędzi ze spójnego, ważonego grafu nieskierowanego G w celu utworzenia takiego podgrafu, że wszystkie wierzchołki pozostają połączone, a suma ich wag jest najmniejsza. Podgraf o minimalnej wadze musi być drzewem, ponieważ gdyby tak nie było, zawierałby cykl prosty, więc usunięcie krawędzi tego cyklu prowadziłoby do grafu o mniejszej wadze.

Drzewo rozpinające grafu G to spójny podgraf, który zawiera wszystkie wierzchołki należące do G i jest drzewem ((c) i (d) są drzewami rozpinającymi). Spójny podgraf o minimalnej wadze musi być drzewem rozpinającym, ale nie każde drzewo rozpinające ma minimalną wagę. Algorytm rozwiązujący przedstawiony wcześniej problem musi tworzyć drzewo rozpinające o minimalnej wadze. Takie drzewo nosi nazwę ***minimalnego drzewa rozpinającego*** ((d) jest takim drzewem).

Znalezienie minimalnego drzewa rozpinajacego metodą siłową wymaga czasu gorszego niż wykładniczy. Chcemy rozwiązać to bardziej wydajnie wykorzystując podejście zachłanne.

Definicja

Graf nieskierowany G składa się ze skończonego zbioru V wierzchołków oraz zbioru E par wierzchołków ze zbioru V czyli krawędzi grafu. Graf G oznaczamy:

$$G = (V, E)$$

Dla grafu (a):

$$V = \{\nu_1,\nu_2,\nu_3,\nu_4,\nu_5\}$$
$$E = \{(\nu_1,\nu_2),(\nu_1,\nu_3),(\nu_2,\nu_3),(\nu_2,\nu_4),(\nu_3,\nu_4),(\nu_3,\nu_5),(\nu_4,\nu_5)\}$$

Drzewo rozpinające *T* dla grafu *G* zawiera te same wierzchołki *V*, co graf *G*, jednak zbiór krawędzi drzewa *T* jest podzbiorem *F* zbioru *E*. Drzewo rozpinające możemy oznaczyć jako $T = (V, F)$. Problem polega na znalezieniu podzbioru *F* zbioru *E*, takiego aby $T = (V, F)$ było minimalnym drzewem rozpinającym grafu *G*.

Wysokopoziomowy algorytm zachłanny realizujący to zadanie mogłby wyglądać:

```
F=∅;   //Inicjalizacja zbioru krawędzi
while (realizacja nie została rozwiązana) {
    wybierz krawedz zgodnie z warunkiem
    optymalnym lokalnie;
    // procedura wyboru
    if(dodanie krawędzi do F nie powoduje
       powstania cyklu) // spr. wykonalnosci
       dodaj ją;

    if(T=(V,F) jest drzewem rozpinajacym)
       realizacja jest rozwiazana;
}
```

Oczywiście „warunek optymalny lokalnie” może być inny w różnych problemach i w rożnych algorytmach rozwiązania.

Dwa najbardziej znane algorytmy realizujące to zadanie to algorytm Prima i algorytm Kruskala.

Algorytm Dijkstry

Algorytm Dijkstry służy określeniu najkrótszych dróg z pojedyńczego źródła do wszystkich innych wierzchołków w ważonym grafie skierowanym. Wzorcem dla niego jest algorytm Prima.

Podobny problem został rozwiązany przez algorytm Floyda (programowanie dynamiczne) – algorytm klasy $O(n^3)$. Algorytm Dijkstry jest klasy $O(n^2)$.

Wykorzystujemy podzbiór krawędzi *F* oraz podzbiór wierzchołków *Y*.

- w algorytmie tym inicjalizujemy zbiór *Y* w ten sposób, aby zawierał tylko wierzchołek, którego najkrótsze drogi chcemy określić. Możemy przyjąć, że wierzchołkiem tym jest ν_1. Inicjalizujemy zbiór krawędzi *F* jako pusty.

- najpierw wybieramy wierzchołek ν, który jest najbliższy ν_1 , dodajemy go do zbioru *Y* oraz dodajemy do zbioru *F* krawędź $<\nu_1, \nu>$ (tzn. krawędź skierowaną z ν_1 do ν). Krawędź ta jest najkrótszą drogą z ν_1 do ν.

- następnie sprawdzamy drogi z ν_1 do wierzchołków należących do zbioru *V-Y*, które dopuszczają tylko wierzchołki ze zbioru *Y* jako wierzchołki pośrednie. Najkrótsza z tych dróg jest najkrótszą drogą.

- wierzchołek leżący na końcu takiej drogi zostaje dodany do zbioru *Y*, zaś krawędź (należąca do drogi), która łączy ten wierzchołek z innym jest dodawana do zbioru *F*.

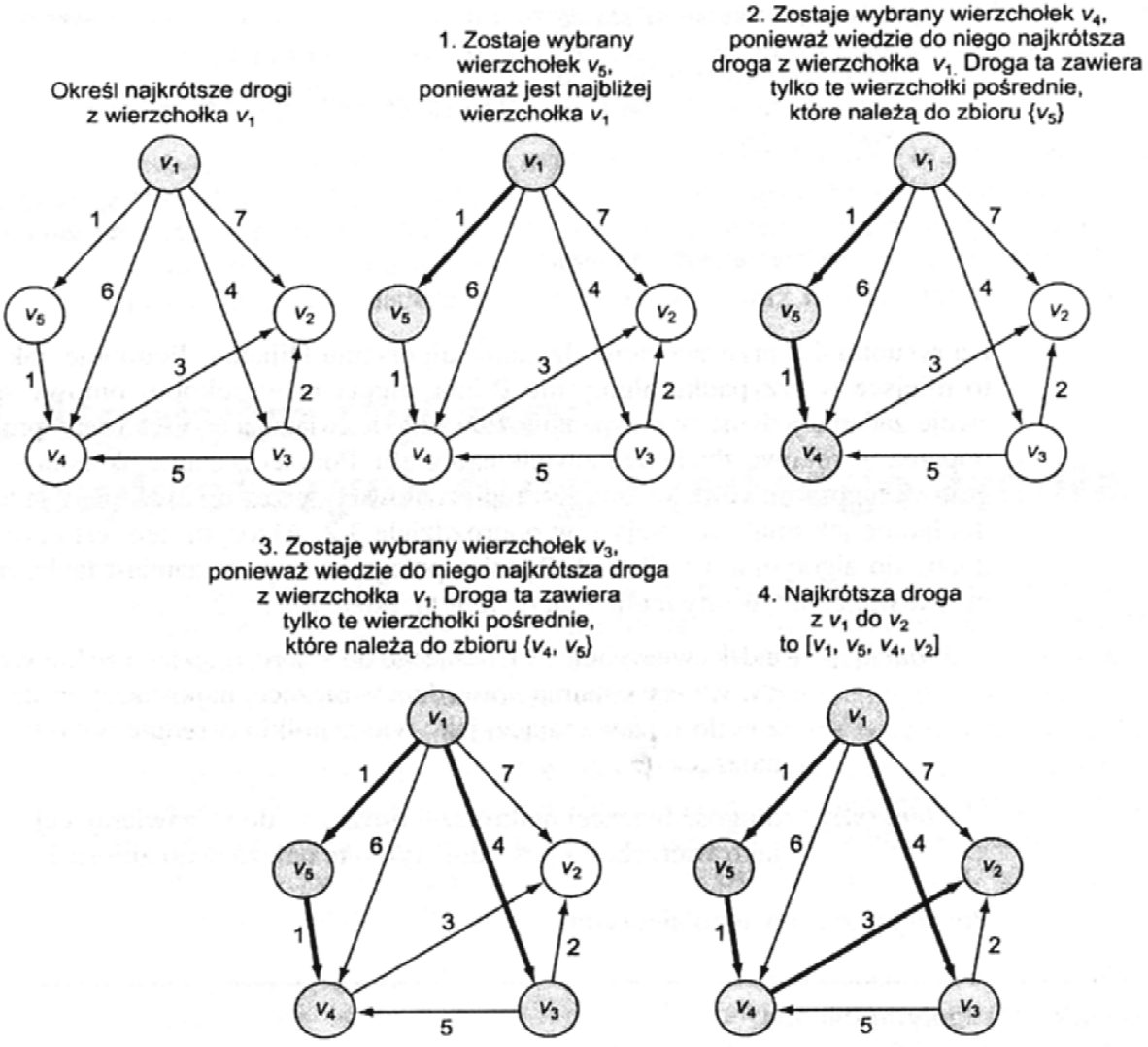
- procedura ta jest kontynuowana do momentu, w którym zbiór *Y* będzie równy *V* – zbiorowi wszystkich wierzchołków. W tym momencie *F* zawiera krawędzie najkrótszych dróg.

Ogólny algorytm można zapisać następująco:

```
Y = {ν1};    F = ∅;
while (realizacja nie została rozwiązana) {
  wybierz ze zbioru V-Y wierzchołek ν, //proc.
  mający najkrótszą drogę do ν1,       //wyboru
  używając jako wierzchołków pośrednich //  i
  tylko tych które należa do Y; // sprawdzenie
                                // wykonalności
  dodaj nowy wierzchołek ν do Y;

  dodaj do zbioru F nową krawędź (z najkrótszej
  drogi), do której należy wierzcholek ν;

  if(Y==V) realizacja jest rozwiązana;
}
```



Algorytm wysokopoziomowy (jak wyżej) spełnia swoje zadanie jedynie w przypadku człowieka, rozwiązującego realizację problemu poprzez wzrokowe zbadanie niewielkiego grafu.

W algorytmie szczegółowym użyjemy dwóch tablic (dla $i = 2,\ldots,n$):

touch[*i*] = indeks wierzchołka ν należącego do zbioru *Y*, takiego że krawędź $<\nu, \nu_i>$ jest ostatnią krawędzią w bieżącej najkrótszej drodze z ν_1 do ν_i , zawierającej jako wierzchołki pośrednie tylko te należące do zbioru *Y*.

length[*i*] = długość bierzącej najkrótszej drogi z ν_1 do ν_i , zawierającej jako wierzchołki pośrednie tylko te należące do Y.

Algorytm Dijkstry

Problem: określić najkrótsze drogi z wierzchołka ν_1 do wszystkich innych wierzchołków w ważonym grafie skierowanym.

Dane wejściowe: liczba całkowita $n \geq 2$ oraz spójny, ważony graf skierowany, zawierający *n* wierzchołków. Graf jest reprezentowany przez tablicę dwywumiarową *W*, której wiersze i kolumny są indeksowane od 1 do *n* i gdzie element *W*[*i*][*j*] jest wagą krawędzi między wierzchołkiem *i*-tym, a *j*-tym.

Wynik: zbiór krawędzi *F*, zawierający krawędzie najkrótszych dróg.

```
void dijkstra (int n,
               const number W[][],
               set_of_edges& F)
{
 index i,vnear;
 number min;
 edge e;
 index touch[2..n];
 number length[2..n];
```

```
F = ∅;
for (i=2;i<=n;i++) {
     touch[i] = 1;
     length[i] = W[1][i];
}

repeat(n-1 razy) { // Dodajemy do Y wszystkie
                   // n-1 wierzchołków
min=∞;
 for (i=2;i<=n;i++) { // Sprawdzamy każdy
                      // wierzchołek pod kątem,
  if (0≤length[i]<min){  //tego czy należy do
                         //do najkrótszej drogi
      min = length[i];
      vnear = i;
  }

e = krawędź łącząca wierzchołek indeksowany
    przez vnear oraz touch[vnear];

dodaj e do F;

  for (i=2;i<=n;i++)
   // Dla każdego wierzchołka nie należącego
   // do zbioru Y, aktualizujemy długość jego
   // najkrótszej drogi

   if(length[vnear]+W[vnear][i]<length[i]) {
     length[i] = length[vnear]+W[vnear][i];
     touch[i] = vnear;
   }

length[vnear]= -1; // Dodajemy do zbioru Y
              // wierzcholek indeksowany vnear
}

}
```

Zakładamy tutaj, że istnieje droga z wierzchołka v_1 do każdego innego wierzchołka, więc zmiennej *vnear* w każdym przebiegu pętli `repeat` zostaje przypisana nowa wartość.
Gdyby tak nie było, algorytm w zapisanej postaci w końcu zacząłby wielokrotnie dodawać ostatnią krawędź aż do momentu, gdy zostałoby wykonanych n-1 przebiegów pętli `repeat`.

Algorytm określa tylko krawędzie w najkrotszych drogach, długości dróg można uzyskać na podstawie informacji o krawędziach lub po niewielkiej modyfikacji algorytmu zapamietywać w dodatkowej tablicy.

Złożoność czasowa w każdym przypadku.

W pętli `repeat` mamy do czynienia z dwiema pętlami, z których każda wykonuje n-1 przebiegów. Wykonanie instrukcji zawartych w każdej z nich można traktować jako jednokrotne wykonanie **operacji podstawowej**.

Rozmiarem danych wejściowych jest liczba wierzchołków n.

Złożoność czasowa wynosi zatem:

$$T(n) = 2(n\text{-}1)(n\text{-}1) \in O(n^2)$$