Temat: Technika powrotów (ang. backtracking).
Schemat algorytmu i przykłady zastosowania.

Przypomnienie

Porównanie czasów realizacji algorytmu wykładniczego na dwóch komputerach o różnej szybkości wykonania operacji elementarnej

Rozmiar n	20	50	100	200
Czas działania $\left(2^n/10^6\right)$	1,04 s	35,7 lat	$4\cdot10^{14}$ wieków	$5\cdot10^{44}$ wieków
Czas działania $\left(2^n/10^9\right)$	0,001 s	13 dni	$4\cdot10^{11}$ wieków	$5\cdot10^{41}$ wieków

Problemy łatwo rozwiązalne to takie, dla których można podać algorytm o wielomianowej złożoności czasowej, czyli taki, którego koszt jest $O(n^k)$ (dla pewnego stałego k).

Problemy o złożoności wyższej niż wielomianowa, czyli takiej, że nie da się jej ograniczyć przez $O(n^k)$, dla dowolnego k, nazywamy trudno rozwiązalnymi.

Wśród problemów trudno rozwiązalnych wyróżnia się:

- Klasę problemów, dla których, można udowodnić, że nie istnieje algorytm wielomianowy.

- Klasę problemów, dla których, jak do tej pory nie udało się podać dowodu na brak wielomianowego rozwiązania.

Strategia powrotów, to strategia pozwalająca na skonstruowanie optymalnego rozwiązania ponad-wielomianowego, w przypadku, gdy nie potrafimy podać rozwiązania wielomianowego albo wiemy, że takie rozwiązanie nie istnieje.

1. Schemat algorytmu typu backtracking

Przyjmijmy, że dana jest przestrzeń stanów, przy czym stan jest sytuacją stanowiącą rozwiązanie problemu albo mogącą prowadzić do rozwiązania oraz sposób przechodzenia z jednego stanu w drugi.

Aby rozwiązać taki problem, musimy przeszukać przestrzeń stanów, przechodząc z jednego stanu w drugi, aż znajdziemy się w stanie określającym rozwiązanie problemu. Ponieważ dla każdego stanu może istnieć wiele dopuszczalnych ruchów, czyli wiele stanów, do których można dojść, możemy wybrać złe posunięcie. Jeżeli wykonamy zły ruch i znajdziemy się w sytuacji bez wyjścia (nie osiągając poprawnego rozwiązania i nie mając więcej dopuszczalnych posunięć do wykonania), musimy cofnąć ostatni ruch i spróbować zrobić inny. Jeżeli cofnięcie ostatniego ruchu w dalszym ciągu nie prowadzi do rozwiązania, to cofamy ruch przedostatni i próbujemy dalej.
Ta metoda nosi nazwę metody powrotów.

Metoda powrotów wymaga zapamiętania wszystkich wykonanych ruchów czy też wszystkich odwiedzonych stanów, aby możliwe było cofanie przesunięć.

Naturalną techniką kodowania algorytmów opartych na strategii powrotów jest rekurencja.

Schemat algorytmu z powrotami

Oznaczenia: $<x_1, ..., x_n>$ - szukane rozwiązanie

A_k - zbiór możliwych rozszerzeń rozwiązania częściowego $<x_1, ..., x_{k-1}>$

P – funkcja (algorytm), która dla danego rozwiązania częściowego $<x_1, ..., x_i>$ przyjmuje wartość 0, gdy ciąg $<x_1, ..., x_i>$ nie daje się rozszerzyć do rozwiązania całkowitego, a 1 w przeciwnym przypadku.

k – aktualna długość rozwiązania częściowego

BackTracking(k)

```
for y ∈ A_k
  if (P(x_1, x_2, ..., x_{k-1}, y) == 1)
    {
      x_k = y;
      if „⟨x_1, x_2, ..., x_k⟩ jest rozwiązaniem całkowitym"
      "wypisz rozwiązanie ⟨x_1, x_2, ..., x_k⟩
      else BackTracking(k+1);
      A_k = A_k ∪ {y};
    }
```

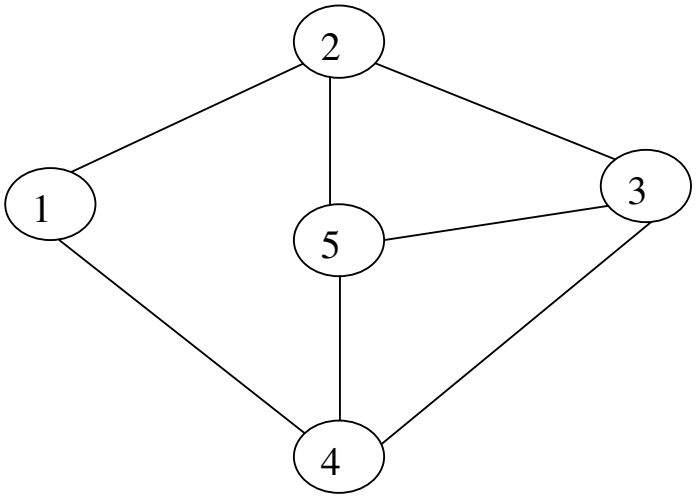
2. Przykłady zastosowania schematu powrotów

a) Problem znalezienia drogi (cyklu) Hamiltona w grafie

Droga (cykl) Hamiltona w grafie to droga (cykl) przechodząca przez wszystkie wierzchołki grafu dokładnie raz.

Nie istnieje jednoznaczne kryterium, które pozwala, tak jak w przypadku drogi (cyklu) Eulera stwierdzić, czy dany graf ma drogę (cykl) Hamiltona.

Przykład



Powyższy graf ma wiele cykli Hamiltona, np.: 1-2-3-5-4-1

Algorytm naiwny

Stosujemy pełny przegląd wszystkich możliwości, tj. generujemy wszystkie możliwe permutacje zbioru n–1 elementowego, gdzie n jest liczbą wierzchołków grafu i sprawdzamy, czy określają one cykl (drogę) Hamiltona.

Koszt algorytmu naiwnego

Zakładając, że szukamy wszystkich cykli z ustalonego wierzchołka startowego, koszt algorytmu naiwnego jest równy

$$T(n) = (n-1)! \cdot n$$

Rozwiązanie problemu znajdowania wszystkich cykli Hamiltona oparte na strategii powrotów

G – graf o *n* wierzchołkach ponumerowanych od 1 do *n* ,
s – wierzchołek startowy
X – tablica z wierzchołkami analizowanego rozwiązania częściowego
DOP – tablica logiczna rejestrująca możliwość wykorzystania wierzchołków w cyklu; *DOP*[*i*]=1, gdy wierzchołek może być wykorzystany w aktualnie generowanym rozwiązaniu częściowym, *DOP*[*i*]=0, gdy wierzchołek *i* został już wykorzystany w generowanym cyklu.
k – długość rozwiązania częściowego

Hamilton(*k*)

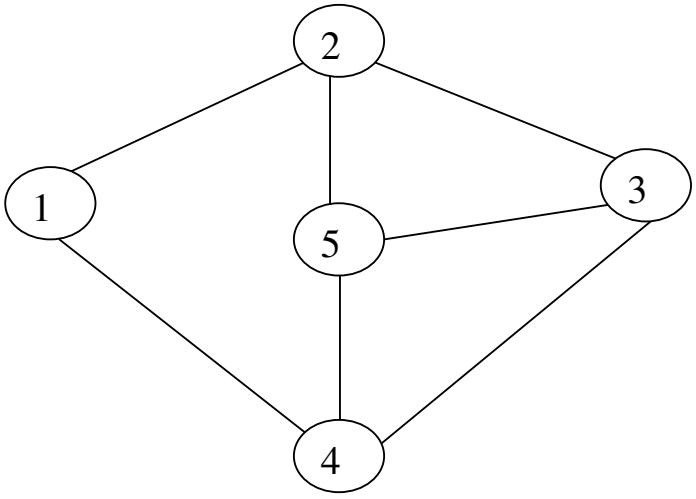
```
y=X[k-1];
for  (v∈ V && (v, y) ∈ E)
  {
    if (k = = n +1 && v == s) "X zawiera cykl Hamiltona";
    else  if DOP[v])
            {
               X[k]=v;
               DOP[v]=0;
               Hamilton(k +1);
               DOP[v]=1;
             }
   }
```

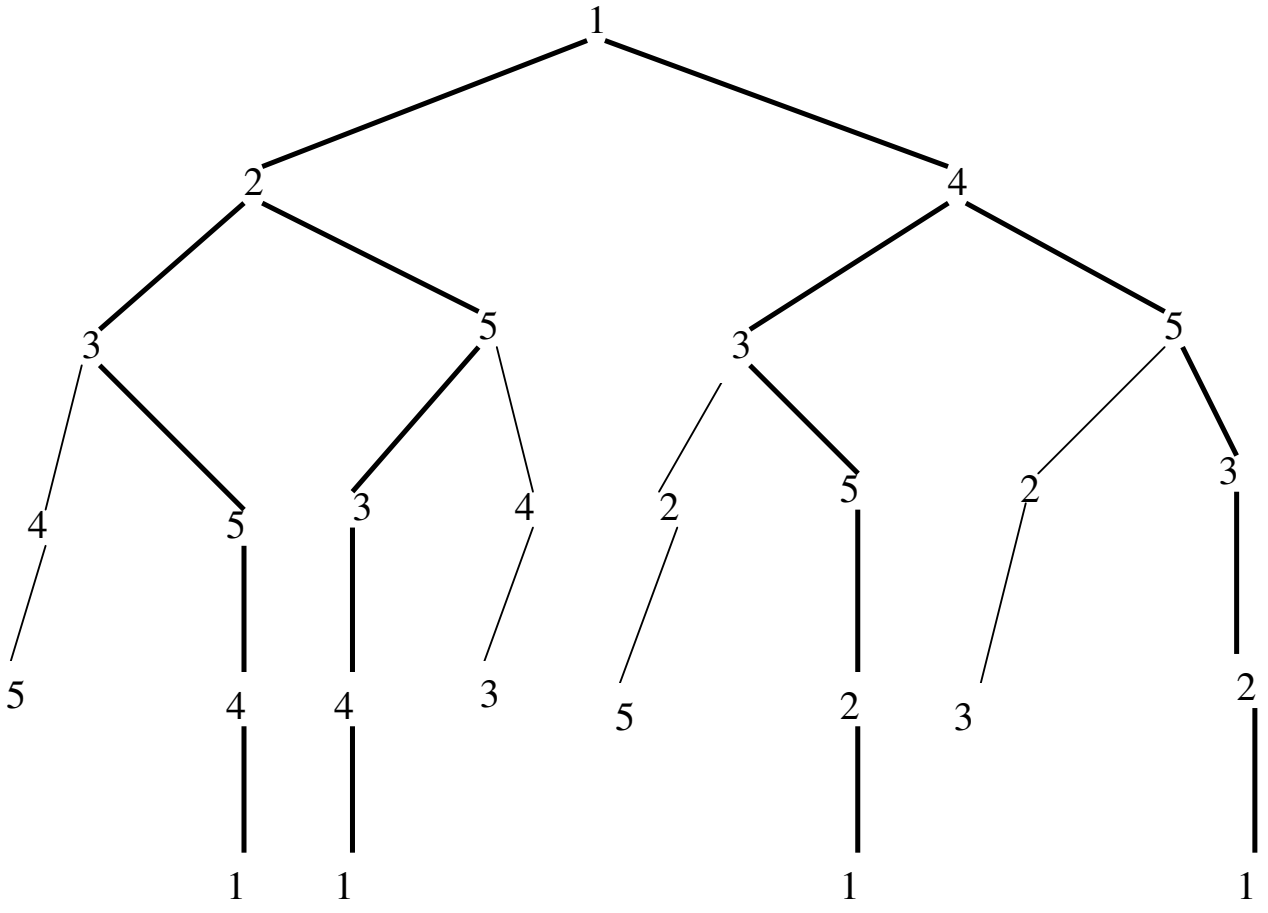
Wywołanie funkcji Hamilton

```
for (i=1; i<=n; i++) do DOP[i]=1;
 X[1]=s;
```

DOP[*s*]=0;
Hamilton(2);

Przykład





Złożoność czasowa algorytmu generowania cykli Hamiltona

Drzewo możliwych "rozszerzeń" rozwiązania częściowego jest drzewem wielokierunkowym o wysokości równej co najwyżej *n*. Jeżeli za operację elementarną przyjmiemy każde wywołanie rekurencyjne procedury Hamilton, to koszt algorytmu jest równy liczbie węzłów w drzewie. Można pokazać, że liczba ta jest $O(2^n)$.

b) Problem ustawienia hetmanów na szachownicy

Problem ten rozważany był po raz pierwszy przez Gaussa w 1850 roku i polega na znalezieniu takiego ustawienia ośmiu hetmanów na szachownicy, aby żaden z hetmanów nie szachował żadnego innego (hetman szachuje inne figury w tym samym wierszu i kolumnie, w której stoi i dodatkowo po przekątnych). Szachownica w problemie Gaussa miała wymiary 8x8.

Przykład

Pola szachowane przez figurę hetmana. Pole na którym stoi hetman jest oznaczone literą H, a pola szachowane przez hetmana są wypełnione literą S.

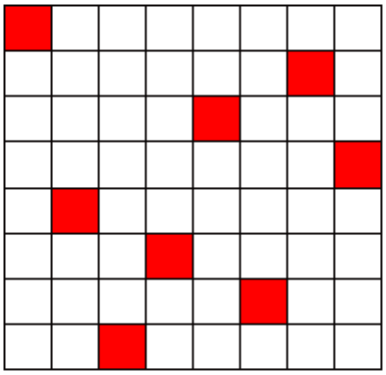
	S		S		S		
		S	S	S			
S	S	S	H	S	S	S	S
		S	S	S			
	S		S		S		
S			S			S	
			S				S
			S				

Idea algorytmu

Próbujemy umieścić pierwszego hetmana na szachownicy, potem drugiego, ale tak, by nie mógł zbić pierwszego, następnie trzeciego tak, aby nie był w konflikcie z dwoma już postawionymi i tak dalej, dopóki nie umieścimy wszystkich. Jeśli na przykład szósty hetman nie może być umieszczony na żadnej pozycji nie kolidującej z wcześniejszymi, to wykonujemy powrót i wybieramy inną pozycję dla piątego hetmana i próbujemy znowu ustawić szóstego. Jeśli to nie pomaga, to ponownie przestawiamy piątego. Jeśli wszystkie możliwości dla piątego hetmana zostały wyczerpane, przestawiamy czwartego i proces zaczyna się od nowa. Algorytm szuka wszystkich możliwych rozwiązań. Dla szachownicy 8 na 8 jest ich aż 92.

Przykład

Przykładowe rozwiązanie problemu ustawienia ośmiu hetmanów.



Pseudokod algorytmu rozwiązującego problem ustawienia hetmanów

n - rozmiar szachownicy; liczba pól w pionie i poziomie i jednocześnie liczba hetmanów do ustawienia

PutQueen(*r*)

r – numer wiersza, w którym próbujemy ustawić hetmana

```
 for (c = 0;c< n; c++)
    if  („pozycja [r, c] nie jest atakowana”)
        {
           „umieść hetmana na pozycji [r, c]”;
           if  (r < n-1)  PutQueen(r+1);
           else „wypisz rozwiązanie”;
           „usuń hetmana z pozycji [r, c]”;
        }
```

Optymalna implementacja powyższego pseudokodu powinna w krótkim czasie realizować następujące operacje:

- sprawdzenie, czy pozycja [*r*, *c*] nie jest atakowana,
- umieść hetmana na pozycji [*r*, *c*],
- usuń hetmana z pozycji [*r*, *c*].

Zauważmy, że:

- Indeksy wszystkich pól na "lewych przekątnych" spełniają warunek: $r + c = nl$, gdzie *nl* jest numerem lewej przekątnej. *nl* przyjmuje wartość od 0 do $2n-1$.

- Indeksy wszystkich pól na "prawych przekątnych" spełniają warunek: $r - c = nr$, gdzie *nr* jest numerem prawej przekątnej. *nr* przyjmuje wartość od $-n+1$ do $n-1$.

Przykład

Dla uproszczenia przyjrzyjmy się szachownicy 4 na 4. Wiersze i kolumny szachownicy ponumerujemy liczbami od 0 do 3.

0, 0	0, 1	0, 2	0, 3
1, 0	1, 1	1, 2	1, 3
2, 0	2, 1	2, 2	2, 3
3, 0	3, 1	3, 2	3, 3

- Struktura danych dla wszystkich lewych przekątnych to jednowymiarowa tablica logiczna *LeftDiagonal*. Jeżeli *LeftDiagonal*[*i*]=0, to lewa przekątna o numerze *i* jest szachowana.

- Struktura danych dla wszystkich prawych przekątnych to jednowymiarowa tablica logiczna *RightDiagonal*. Jeżeli *RightDiagonal*[*i*]=0, to prawa przekątna o numerze *i* jest szachowana.

- Potrzebna będzie podobna tablica dla kolumn *Column*. Jeżeli *Column*[*i*]=0, to kolumna o numerze *i* jest szachowana.

Nie jest potrzebna tablica dla wierszy, ponieważ *i* - ty hetman "wędruje" wzdłuż wiersza *i*, a wszystkie hetmany o numerach mniejszych niż *i* zostały już ustawione w wierszach o numerach mniejszych niż *i*.

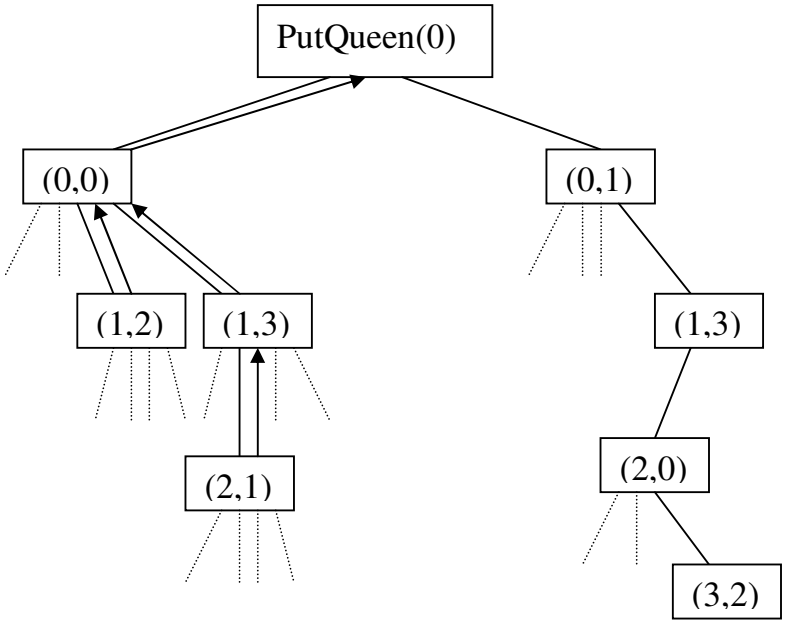
- Tablica *PositionInRow* to tablica jednowymiarowa, która przechowuje rozwiązania częściowe i całkowite problemu. Jeżeli *PositionInRow*[*i*]=*j*, to hetman z wiersza *i* jest ustawiony w kolumnie o numerze *j*.

Algorytm rozwiązujący problem ustawienia czterech hetmanów

n – rozmiar szachownicy
LeftDiagonal – tablica logiczna o indeksach od 0 do 2*n*-2
RightDiagonal - tablica logiczna o indeksach od -*n*+1 do *n*-1
Column – tablica logiczna o indeksach od 0 do *n*-1
PositionInRow – tablica wynikowa

```
PutQueen(r)
 for (c=0; c<n; c++)
      if  (Column[c] && LeftDiagonal[r+c]
          && RightDiagonal[r-c])
          {
             PositionInRow[r]=c;
             Column[c]=0;
             LeftDiagonal[r+c]=0;
             RightDiagonal[r-c]=0;
             if   (r < n-1)  PutQueen(r + 1);
             else  "wypisz wynik –PositionInRow";
             Column[c]=1;
             LeftDiagonal[r + c]=1;
             RightDiagonal[r - c]=1;
          };
```

Kroki prowadzące do pierwszego udanego ustawienia czterech hetmanów, wykonywane przez funkcję PutQueen



Zmiany w czterech tablicach wykorzystanych w funkcji PutQueen

PositionInRow	*Column*	*LeftDiagonal*	*RightDiagonal*
0 1 2 3	0 1 2 3	0 1 2 3 4 5 6	-3 -2 -1 0 1 2 3
0 2 - -	0 1 0 1	0 1 1 0 1 1 1	1 1 0 0 1 1 1
0 3 1 -	0 0 1 0	0 1 1 0 0 1 1	1 0 1 0 0 1 1
1 3 0 2	0 0 0 0	1 0 0 1 0 0 1	1 0 0 1 0 0 1