# Backtracking

Bartosz Chomiński

## 1 Wstęp

Ile jest różnych ustawień ośmiu hetmanów na szachownicy, w których żaden hetman nie bije żadnego innego?

Najprostsze możliwe rozwiązanie tego problemu to oczywiście sprawdzenie każdego możliwego ułożenia hetmanów i zliczenie działających „po jednym". Zasadniczy problem związany z tym podejściem to liczba kombinacji do sprawdzenia. Będzie ich

$$\binom{64}{8} = \frac{64 \cdot 63 \cdot 62 \cdot 61 \cdot 60 \cdot 59 \cdot 58 \cdot 57}{8 \cdot 7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1} \approx 4.4 \cdot 10^9,$$

zatem nawet na najszybszym domowym komputerze uruchomienie programu opartego na tej metodzie zajmie co najmniej kilka minut.

Możemy nieco ulepszyć poprzedni pomysł. Skoro dwa hetmany w jednej kolumnie szachownicy na pewno się biją, a więc jakiekolwiek ułożenie hetmanów z dwoma w tej samej kolumnie się nie liczy, to ograniczmy się tylko do ustawień, w których w każdej kolumnie jest dokładnie jeden hetman. Tym razem ustawień będzie

$$8^8 \approx 1.6 \cdot 10^7,$$

a więc ponad dwieście razy mniej, niż w poprzednim pomyśle. To jednak dalej niekomfortowo dużo kombinacji i nadal mamy spore pole do poprawy.

## 2 Pomocna obserwacja

Zauważmy, że możemy odrzucić jakieś ustawienie hetmanów tylko na podstawie kilku początkowych pozycji hetmanów. Przykładowo, jeżeli pierwszy i drugi hetman są w tym samym wierszu, to pozycje kolejnych hetmanów nie mają już znaczenia – zawsze na podstawie tego, że dwa pierwsze hetmany się biją, odrzucimy całe ustawienie i nie zliczymy go do wyniku. Skoro tak, to moglibyśmy spróbować generować ustawienia hetmanów „leniwie" – jeśli w jakimś ustawieniu (niekoniecznie wszystkich ośmiu) hetmanów na szachownicy będą dwa bijące się, to nie będziemy sprawdzać żadnych ustawień, które powstają przez dostawienie hetmanów do tego trefnego ustawienia.

## 3 Ogólny schemat

„Leniwe" podejście opisane wyżej można w jego ogólności skonkretyzować, co czynimy poniższym pseudokodem:

```
vector<Decyzja> decyzje;

void próbuj(Stan obecny_stan){
  if (jest_zły(obecny_stan)) {
      return;
  }

  if (jest_końcowy(obecny_stan)) {
      zapisz(obecny_stan)
      return;
  }

  for (Decyzja d : decyzje_z(obecny_stan)) {
      decyzje.push_back(d);
      próbuj(obecny_stan + d);
      decyzje.pop_back();
  }
}
```

Przeszukiwanie uruchamiamy wywołując procedurę `próbuj(pusty_stan)`, a następnie program podejmuje kolejne decyzje rekurencyjnie i na bieżąco odrzuca te decyzje, które prowadzą do złych ustawień.

Metoda pokazana w powyższym pseudokodzie nazywa się *backtracking* (pol. przeszukiwanie z nawrotami).

W przykładzie z problemem ośmiu hetmanów `Decyzja` to pozycja kolejnego hetmana na planszy, a `Stan` to układ hetmanów na szachownicy.

## 4 Uogólnienie obserwacji

Kluczowa własność problemu ośmiu hetmanów, która pozwoliła nam skorzystać z backtrackingu to *lo-*

*kalna koniunktywność warunku.* Konkretniej: warunek, który musi spełniać plansza, żeby była dobra, to koniunkcja kilku innych warunków, z których każdy zależy od dwóch decyzji, a nie od całej planszy:

hetmany 1 i 2 się nie biją **i**
hetmany 1 i 3 się nie biją **i**
⋮
hetmany 7 i 8 się nie biją.

Taka konstrukcja warunku pozwala odrzucać złe częściowe ustawienia szybciej, niż dopiero po wygenerowaniu pełnego ustawienia, ponieważ jeśli którykolwiek z warunków koniunkcji jest niespełniony, to ona cała też nie. Analogiczną własność warunku mają też takie problemy jak Sudoku, krzyżówki czy problem wieży z Hanoi.

## 5 Podziały liczb

Rozważmy problem podziałów liczb, czyli wyznaczenia wszystkich sposobów, na jakie można podzielić liczbę naturalną na całkowite dodatnie części z dokładnością do kolejności. Przykładowo, dla $N = 4$ wszystkie podziały to

$$4,\ 3+1,\ 2+2,\ 2+1+1,\ 1+1+1+1,$$

zatem dla $N = 4$ oczekujemy odpowiedzi 5. Na początek, dla ustalenia uwagi, przyjmijmy, że będą nas interesować tylko podziały o składnikach posortowanych nierosnąco ($2+1+1$ tak, ale $1+2+1$ nie). To pozwoli nam łatwiej zapewnić warunek unikatowości podziałów co do kolejności, bo kolejność będzie już ustalona.

Zauważmy, że do tego problemu również można przyłożyć metodę opisaną wyżej: decyzją będzie wybór kolejnego składnika podziału, a stanem będzie dotychczasowy podział. Warunki do spełnienia będą dwa: składniki podziału mają być uporządkowane nierosnąco oraz suma składników ma być równa docelowej liczbie. Pierwszy z warunków to tak naprawdę koniunkcja wielu innych warunków, takich jak „pierwszy składnik podziału jest większy lub równy drugiemu składnikowi podziału" i tego typu warunki sprawdzamy już w chwili generowania podziałów (tak jak warunki niebicia się w problemie ośmiu hetmanów).

To rozwiązanie, w wersji zliczającej wszystkie różne podziały, można zapisać następująco, tym razem już w kodzie C++:

```
int N;
vector<int> podzial;
int suma = 0;
int wynik = 0;

void probuj(){
  if (suma > N) {
    return;
  }

  if (suma == N) {
    wynik += 1;
    return;
  }

  int s_limit = N - suma;

  if (!podzial.empty()) {
    s_limit = min(s_limit, podzial.back());
  }

  for (int i = 1; i <= s_limit; i++) {
    podzial.push_back(i);
    suma += i;
    probuj()
    podzial.pop_back();
    suma -= i;
  }

}
```

## 6 Zobacz też


- O.-J. Dahl, E. W. Dijkstra, C. A. R. Hoare Structured Programming, Academic Press, London, 1972 ISBN 0-12-200550-3, pp. 72–82.
- `https://queens.cspea.co.uk/index.html`
- `https://oeis.org/A319283`
- `https://en.wikipedia.org/wiki/Shunting_yard_algorithm`