# Programowanie Funkcyjne 2018

Lista zadań nr 6 dla grupy TWI: programowanie imperatywne

Na zajęcia 29 listopada 2018

W poniższych zadaniach rozważamy algorytmy przeszukiwania wyczerpującego znajdujące ścieżki Hamiltona dla skoczka szachowego poruszającego się po uogólnionej szachownicy rozmiaru $m \times n$. Z pola $(i, j)$ skoczek szachowy może przejść na pole $(i + \Delta i, j + \Delta j)$ jeśli $\Delta i, \Delta j \neq 0$, $|\Delta i| + |\Delta j| = 3$ oraz docelowe pole mieści się na szachownicy.

**Zadanie 1 (3 pkt).** Najprostsza metoda poszukiwania ścieżki Hamiltona dla skoczka polega na reprezentowaniu ścieżki skoczka za pomocą listy odwiedzonych pól w kolejności od bieżącego do startowego. Jeśli skoczek startuje w polu $(i_0, j_0)$, to na początku obliczeń lista ma wartość $[(i_0, j_0)]$. Jeśli w trakcie obliczeń droga skoczka jest opisana listą $[(i_k, j_k), \ldots, (i_0, j_0)]$, to możemy ją (zarówno drogę, jak i listę) przedłużyć o jeden krok w następujący sposób: wyznaczamy listę od 2 do 8 pól (mniej niż 8, jeśli skoczek znajduje się przy brzegu szachownicy), na które skoczek może przejść w jednym kroku. Odrzucamy te z pól, które skoczek już odwiedził (tzn. które znajdują się już w budowanej liście). Dla każdego z pozostałych pól przedłużamy ścieżkę o to pole (dodajemy je do głowy listy) i rekurencyjnie uruchamiamy algorytm na przedłużonej liście. Rekursja kończy się gdy wykonamy $m \cdot n - 1$ kroków. Zbudowana lista po odwróceniu jest wówczas ścieżką Hamiltona. Zaprogramuj powyższy algorytm w postaci funkcji

```
knight_list : board_size -> position -> position list Streams.t
```

gdzie

```
type board_size = int * int
type position = int * int
```

zaś `Streams.t` jest typem leniwych strumieni z poprzedniej listy. Wartością wyrażenia

```
knight_list (m,n) (i,j)
```

jest strumień ścieżek Hamiltona dla skoczka startującego z pola `(i,j)` na szachownicy rozmiaru `(m,n)`. Dzięki wykorzystaniu strumieni możemy z łatwością zatrzymać obliczenie po znalezieniu pojedynczego rozwiązania i wznawiać je w celu znalezienia kolejnych.

**Zadanie 2 (6 pkt).** Aby algorytm przeszukiwania wyczerpującego działał szybko, obliczenie wykonywane wielokrotnie podczas przeszukiwania należy zoptymalizować. W algorytmie z poprzedniego zadania w każdym kroku od nowa wyznaczamy listę pól, na które skoczek może przejść. To samo obliczenie jest więc wykonywane miliony razy. Bardziej optymalne rozwiązanie polega na utworzeniu w pamięci grafu, którego wierzchołkami są pola szachownicy, a skierowane krawędzie odpowiadają pojedynczym ruchom skoczka. Druga, ważniejsza optymalizacja, polega na przyspieszeniu sprawdzenia, czy dane pole było już odwiedzone. Zamiast przeszukiwać w tym celu listę odwiedzonych wierzchołków, możemy z każdym wierzchołkiem związać zmienną mutowalną, która będzie zawierać informację, że pole nie było jeszcze odwiedzone lub wskazywać na pole, z którego skoczek na nie przeszedł. Niech więc:

```
type field = { coord : position;
               next : field list Lazy.t;
               mutable prev : step }
and step = Taken of field | Free | Start
```

Zaprogramuj funkcję

```
knight_board : board_size -> position -> field
```

która buduje opisany wyżej graf i zwraca pole startowe. Ponieważ graf przejść skoczka zawiera cykle, to pole `field.next` musimy wyliczać leniwie. Pola `prev` wszystkich pól szachownicy z wyjątkiem startowego powinny zawierać wartość `Free`. Zaprogramuj następnie funkcję

```
knight_graph : board_size -> position -> position list Streams.t
```

która wykorzysta reprezentację szachownicy zbudowaną przez funkcję `knight_board` do znalezienia drogi skoczka.

**Zadanie 3 (3 pkt).** W tym zadaniu uogólnioną szachownicę będziemy reprezentować w postaci dwuwymiarowej tablicy:

```
type step = Taken of position | Free | Start
type board = step array array
```

Zaprogramuj funkcję

```
knight_array : board_size -> position -> position list Streams.t
```

wykorzystującą taką reprezentację stanu obliczeń.

**Zadanie 4 (1 pkt).** Skompiluj programy z zadań 1–3 do kodu rodzimego i porównaj ich efektywność (szybkość znalezienia pierwszego rozwiązania).

Drugim znanym zadaniem związanym z przeszukiwaniem wyczerpującym i szachownicą jest problem rozstawienia $n$ hetmanów na szachownicy $n \times n$ w taki sposób, by się wzajemnie nie szachowały. Skoro hetmany szachują cały wiersz i całą kolumnę w której stoją, to w każdym wierszu i w każdej kolumnie powinien stać dokładnie jeden hetman. Zatem rozstawienie $n$ hetmanów można opisać za pomocą permutacji liczb $1, \ldots, n$. Jednak nie każda permutacja jest dobra — także na każdej przekątnej powinien stać dokładnie jeden hetman.

**Zadanie 5 (2 pkt).** Ustawienia hetmanów w kolejnych kolumnach opisujemy za pomocą listy długości $n$ liczb całkowitych z przedziału $1, \ldots, n$. Krok indukcyjny jest następujący: jeśli mamy listę $[w_k, \ldots, w_n]$ opisującą ustawienia hetmanów w kolumnach $k, \ldots, n$, to spośród liczb $i = 1, \ldots, n$ odrzucamy takie, że hetman ustawiony na polu $(k-1, i)$ szachowałby któregoś z dotychczas ustawionych (na polach $(k, w_k), \ldots, (n, w_n)$). Ustawiamy hetmana w jednym z pozostałych, dopuszczalnych wierszy $i$ (tj. dodajemy liczbę $i$ do listy $[w_k, \ldots, w_n]$) i wywołujemy nasz algorytm rekurencyjnie. Obliczenie kończy się, gdy budowana lista ma długość $n$. Zaprogramuj funkcję

```
queens_list : int -> int list Streams.t
```

generującą poprawne rozstawienia hetmanów na szachownicy podanego rozmiaru i wykorzystującą powyższy algorytm.

**Zadanie 6 (4 pkt).** Sprawdzenie, czy kolejny hetman może być ustawiony w wierszu $i$ w poprzednim zadaniu wymaga przejrzenia listy $[w_k, \ldots, w_n]$. Jest to czasochłonne. Klasyczne rozwiązanie polega na przechowywaniu informacji o ustawionych hetmanach w trzech tablicach rozmiaru $n$ opisujących ustawienie hetmanów w wierszach oraz na wznoszących i opadających przekątnych. Zaprogramuj funkcję

```
queens_array : int -> int list Streams.t
```

wykorzystującą typ `int array` do zaimplementowania opisanego wyżej algorytmu.

**Zadanie 7 (1 pkt).** Skompiluj programy z zadań 5–6 do kodu rodzimego i porównaj ich efektywność (szybkość znalezienia pierwszego rozwiązania).