Wstęp do Programowania potok funkcyjny

Marcin Kubica

2010/2011

Outline

Zastosowania backtrackingu

- Problemy, do których można zastosować zasadę „dziel i rządź”.
- Wiele sposobów podziału problemu na podproblemy.
- Rozwiązanie składa się z szeregu „kroków”.
  Na każdym etapie możliwych jest wiele „kroków” do wyboru.
- Rekurencyjne rozpatrywanie wszystkich możliwości.
- Problemy optymalizacyjne:
  Dodatkowo szukamy rozwiązania o najmniejszym koszcie.
- Jeżeli okazuje się, że wcześniejsze „kroki” nie prowadzą do rozwiązania, lub prowadzą jedynie do gorszych rozwiązań od już znanych, to wycofujemy się.

Prosty generyczny backtracking

- Funktor, który na podstawie instancji problemu znajduje jedno lub wszystkie jego rozwiązania.
- Instancja problemu:
  - **konfiguracja** $=$ dane $+$ częściowe rozwiązanie,
  - konfiguracja **końcowa** — pełne rozwiązanie problemu,
  - **wynik** — wyłuskanie wyniku z konfiguracji,
  - **iterator** — wywołuje zadaną procedurę dla wszystkich konfiguracji osiągalnych w jednym kroku.

Instancja problemu

```
module type BT_PROBLEM = sig
    type config     (* Typ rozpatrywanych konfiguracji. *)

    type result                  (* Typ szukanych wyników. *)

    val final : config -> bool
                        (* Czy rozwiązanie jest kompletne? *)

    val extract : config -> result
                  (* Wyłuskuje z konfiguracji rozwiązanie. *)
              (* Kopia dla imperatywnych struktur danych. *)

    val iter : (config -> unit) -> config -> unit
                     (* Wywołaj daną procedurę, dla każdej *)
               (* konfiguracji osiągalnej w jednym kroku. *)
  end;;
```

Mechanizm znajdujący jedno lub wszystkie rozwiązania

```
module type BT_SOLVER = functor (Problem : BT_PROBLEM) ->
  sig
    exception NoSolution                    (* Brak rozwiązań. *)

    val onesolution : Problem.config -> Problem.result
            (* Procedura znajdująca jedno rozwiązanie. *)

    val solutions : Problem.config -> Problem.result list
         (* Procedura znajdująca wszystkie rozwiązania. *)
  end;;
```

Implementacja

```
module Bt_Solver : BT_SOLVER =
  functor (Problem : BT_PROBLEM) -> struct
    exception NoSolution

    exception Solution of Problem.result
                (* Sygnalizuje znalezienie rozwiązania. *)

    let onesolution s =
                          (* Znajduje jedno rozwiązanie. *)
        ⋮

    let solutions s =
                       (* Znajduje wszystkie rozwiązania. *)
        ⋮
  end;;
```

Jedno rozwiązanie

```
let onesolution s =
  let rec backtrack s =
    begin
      if Problem.final s then
        raise (Solution (Problem.extract s));
      Problem.iter backtrack s
    end
  in
    try (backtrack s; raise NoSolution)
    with Solution x -> x
```

Wszystkie rozwiązania

```
let solutions s =
  let wynik = ref []
  in
    let rec backtrack s =
      begin
        if Problem.final s then
          wynik := (Problem.extract s)::!wynik;
        Problem.iter backtrack s
      end
    in begin
      backtrack s;
     !wynik
    end
```

Ograniczenia

To rozwiązanie ma pewne ograniczenia:

- Nie pasuje do problemów optymalizacyjnych.
- Brak obcinania gałęzi (rekurencji).
- Nie pasuje do problemów, które rozbija się na wiele mniejszych podproblemów.

Problem ośmiu hetmanów

Example

- Jak ustawić n hetmanów na szachownicy $n \times n$ tak, aby żadne dwa się nie atakowały.
- W każdym wierszu i każdej kolumnie musi stać dokładnie jeden hetman.
- Ustawiamy hetmany w kolejnych kolumnach, sprawdzając czy się nie atakują.
- Interesuje nas jeden wynik — pierwszy znaleziony.

Problem ośmiu hetmanów

Example

```
module Hetman = struct
```

- Pozycje ustawionych hetmanów na szachownicy:

```
type result = (int * int) list
```

- Konfiguracja: wielkość planszy, liczba początkowych kolumn do wypełnienia i pozycje hetmanów:

```
type config = int * int * result
```

- Pusta plansza rozmiaru n:

```
let empty n = (n, n, [])
```

- Wyłuskanie pozycji hetmanów:

```
let extract (_, _, l) = l
```

- Czy rozwiązanie jest gotowe:

```
let final (_, k, _) = k=0
```

Problem ośmiu hetmanów

Example

- Czy dwa hetmany się atakują?

```
let atakuje (x1, y1) (x2, y2) =
  x1=x2 || y1=y2 || x1+y1=x2+y2 || x1-y1=x2-y2
```

- Czy można dostawić hetmana `h` do `het`?

```
let mozna_dostawic h het =
  fold_left (fun ok x -> ok && not (atakuje h x))
            true het
```

- Dostawienie hetmana w kolumnie `k`:

```
let rec ints a b =
  if a > b then [] else a :: ints (a+1) b

let iter p (n, k, l) =
  let r = filter (fun i -> mozna_dostawic (k, i) l)
                 (ints 1 n)
  in List.iter (fun i -> p (n, k-1, (k,i)::l)) r


end;;
```

Problem ośmiu hetmanów

Example

- Hetmany są ustawiane w kolejnych kolumnach od prawej do lewej.
- Rozwiązanie jest gotowe, gdy w każdej kolumnie stoi hetman.
- Kolejnym kroku ustawiamy hetmana w ostatniej pustej kolumnie.
- Jeśli tylko ustawione hetmany atakują się, przerywamy przeszukiwanie.

Problem ośmiu hetmanów

Example

- Procedury rozwiązujące problem:

```
module H = Bt_Solver (Hetman);;

let hetman n = H.onesolution (Hetman.empty n);;

let hetmany n = H.solutions (Hetman.empty n);;
```

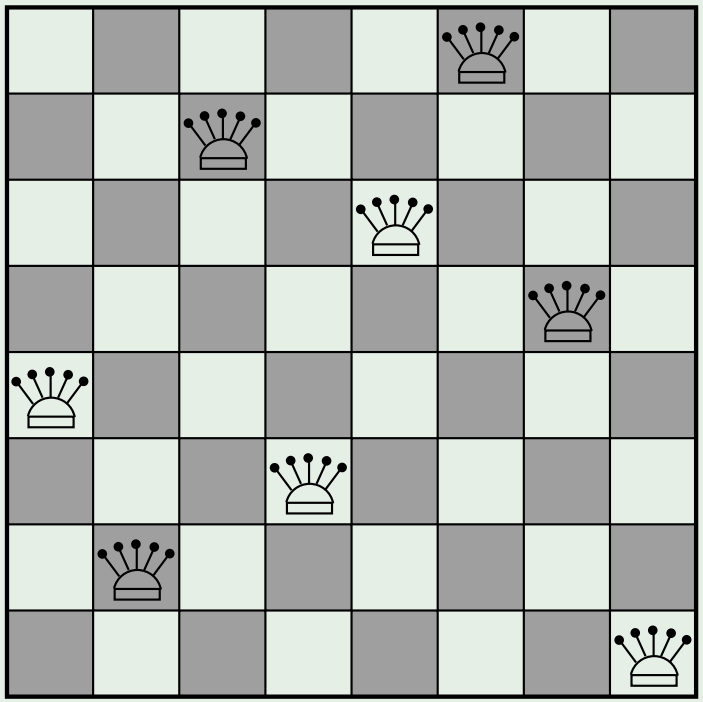
- ... i ich zastosowanie:

```
hetman 4;;
- : Hetman.result = [(1, 3); (2, 1); (3, 4); (4, 2)]

hetmany 4;;
- : Hetman.result list =
                    [[(1, 2); (2, 4); (3, 1); (4, 3)];
                     [(1, 3); (2, 1); (3, 4); (4, 2)]]
```

Problem ośmiu hetmanów

Example

```
hetman 8;;
- : Hetman.result = [(1, 4); (2, 2); (3, 7); (4, 3);
                     (5, 6); (6, 8); (7, 5); (8, 1)]
```



Sudoku

Example

- Kwadrat złożony z $n \times n$ mniejszych kwadratów $n \times n$.
- Sudoku należy wypełnić liczbami od 1 do n^2, tak żeby: w każdej kolumnie, każdym wierszu i każdym mniejszym kwadracie każda liczba występowała dokładnie raz.
- Część liczb jest już na swoich miejscach.

4						6		5
		7	9	6			1	
	6		4			8		7
	8	3			6			
	1						5	
			2			4	6	
7		6			9		2	
	5			2	4	7		
8		9						6

Sudoku

Example

- Sudoku — dwuwymiarowa tablica $n^2 \times n^2$ liczb.
- Puste miejsca są wypełnione zerami.

```
module Sudoku_Framework = struct
  open List

  type sudoku = int array array

  let size (s:sudoku) =
    truncate(sqrt (float (Array.length s)))

  ⋮
```

Sudoku

Example

- Produkt kartezjański list indeksów:

```
let pairs l1 l2 =
  flatten (map (fun i-> map (fun j -> (i,j)) l1) l2)
```

- Kolejne indeksy w sudoku rozmiaru n:

```
let indeksy s =
  let n = size s
  in ints 0 (n * n - 1)
```

- Lista par indeksów jednego małego „kwadratu”:

```
let kwadrat s =
  let n = size s
  in let s = ints 0 (n-1)
     in pairs s s
```

Sudoku

Example

- Lista cyfr, którymi wypełniamy sudoku:

```
let cyfry s =
  let n = size s
  in ints 1 (n * n)
```

- Tworzy listę indeksów do pustych miejsc w sudoku:

```
let brakujace (s : sudoku) =
  filter
    (fun (i,j) -> s.(i).(j) = 0)
    (pairs (indeksy s) (indeksy s))
```

Sudoku

Example

- Na pustym polu możemy wstawić tylko taką liczbę, która nie występuje w danej kolumnie, wierszu i mniejszym kwadracie:

```
let valid_value (s : sudoku) i j k =
  let valid_column i =
    for_all (fun x -> s.(i).(x) <> k || x = j)
            (indeksy s)
  and valid_row j = ...
  and valid_square i j = ...
  in (valid_column i) && (valid_row j) &&
     (valid_square i j)
```

- Lista liczb, które można wstawić na polu (i, j):

```
let valid_values (s : sudoku) i j =
  filter (valid_value s i j) (cyfry s)
```

Sudoku

Example

- Połączenie backtrackingu z wnioskowaniem:
  - Na podstawie wypełnionych pól staramy się wypełniać kolejne.
  - Gdy się nie da — backtracking.
- Implementacja w postaci funktora:
  - Parametrem funktora jest procedura wnioskująca.
  - Wywnioskowane wartości są wstawiane do tablicy.
  - Wynik: lista wywnioskowanych pól + lista pustych pól.

```
module type SUDOKU_REASONING = sig
    type config = Sudoku_Framework.sudoku * (int * int) list

    val reason : config -> (int * int) list * (int * int) list
  end;;
```

Sudoku

Example

Funktor przekształca procedurę wnioskowania w instancję problemu, który jest rozwiązywany za pomocą backtrackingu.

```
module Sudoku_BT (R : SUDOKU_REASONING) = struct
  open Sudoku_Framework

  module Problem = struct
    type result = sudoku
    type config = R.config
    let final (s, l) = l = []

    let copy_sudoku s =
      Array.init (Array.length s)
                 (fun i-> Array.copy s.(i))

    let extract (s, l) = copy_sudoku s
```

Sudoku

Example

Wypełniane jest puste pole o najmniejszej liczbie możliwości.

```
let iter p (s, l) =
  if not (final (s, l)) then
    let (wst, l1) = R.reason (s, l)
    in begin
      if not (final (s, l1)) then begin
        let sorted_moves = ... postaci (length v, (i, j), v)
        in
          let (_, (i, j), v) = List.hd sorted_moves
          and t = List.map (fun (_, p, _) -> p) (List.tl sorted_moves)
          in List.iter
               (fun k ->
                   s.(i).(j) <- k;
                   p (s, t);
                   s.(i).(j) <- 0)
               v
      end else p (s, l1);
      List.iter (fun (i,j) -> s.(i).(j) <- 0) wst
    end
```

Sudoku

Example

Wypełniane jest puste pole o najmniejszej liczbie możliwości.

```
let sorted_moves =
  let cmp (x, _, _) (y, _, _) =
    if x < y then -1 else if x > y then 1 else 0
  in List.sort cmp
               (List.map
                 (fun (i, j) ->
                   let v = valid_values s i j
                   in (List.length v, (i, j), v))
                 l1)
in
...
```

Sudoku

Example

Korzyści z wypełniania pola o najmniejszej liczbie możliwości:

- Obcinanie wywołań rekurencyjnych:
  jeżeli na planszy jest pole, na którym nie może znajdować się żadna liczba.
- Proste wnioskowanie:
  jeżeli na planszy znajduje się pole, na którym może znajdować się tylko jedna liczba, to wstawiamy ją.
- Optymalizacja/heurystyka:
  Zmniejszenie liczby rozgałęzień rekurencji.

Sudoku

Example

Rozwiązywanie sudoku za pomocą backtrackingu.

```
        :
      end

    module Solution = Bt_Solver (Problem)

    let sudoku s =
      Solution.solutions (s, brakujace s)

  end;;
```

Sudoku

Example

Prosty backtracking bez dodatkowego wnioskowania:

```
module NoReasoning =
  struct
    type config =
                Sudoku_Framework.sudoku * (int * int) list

    let reason (s,l) = ([],l)
  end;;

module Sudoku1 = Sudoku_BT (NoReasoning);;
```

Sudoku

Example

Backtracking z wnioskowaniem:

- Jeśli w danym wierszu/kolumnie/mniejszym kwadracie jest tylko jedno miejsce, w którym może być liczba, to można ją tam wstawić.
- Wnioskowanie o kolumnach wywołuje wnioskowanie o wierszach, a
  wnioskowanie o kwadratach wywołuje wnioskowanie o kolumnach.
- Dopóki cokolwiek się zmienia, powtarzamy wnioskowanie.

Sudoku

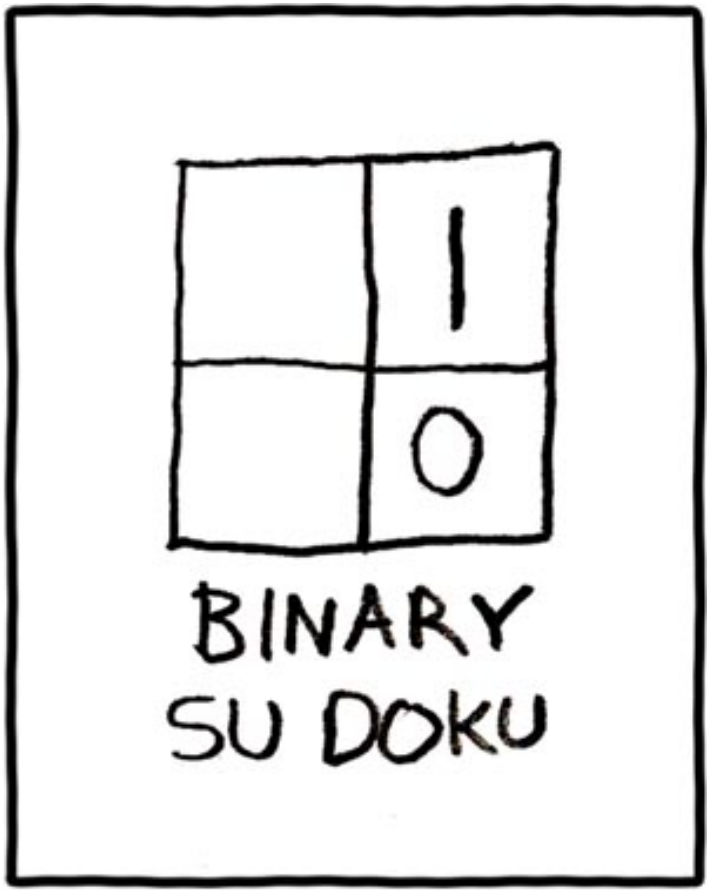
Example

```
module Reasoning = struct
    open Sudoku_Framework

    type config =  sudoku * (int * int) list

    let reason_row (s, l) = ...

    let reason_col (s, l) = ...

    let reason (s,l) = ...
  end;;

module Sudoku2 = Sudoku_BT (Reasoning);;
```

Optymalizacje

- Backtracking ma zwykle koszt wykładniczy.
  Ale jaki?
- Przeszukiwanie przestrzeni rozwiązań bez wielokrotnego rozpatrywania tych samych rozwiązań.
- Obcinanie gałęzi rekurencji — sprawdzamy czy częściowe rozwiązanie ma szansę na uzupełnienie do pełnego lub czy może być lepsze od już znalezionego.
- Kolejność rozpatrywania ruchów — najpierw wybieramy takie, które lepiej rokują.
  Wcześniej znajdujemy rozwiązanie, lub obcinamy więcej gałęzi.
- Zmniejszenie liczby rozgałęzień rekurencyjnych odpowiada zmniejszeniu podstawy potęgi w koszcie wykładniczym.

Deser



http://xkcd.com/74/