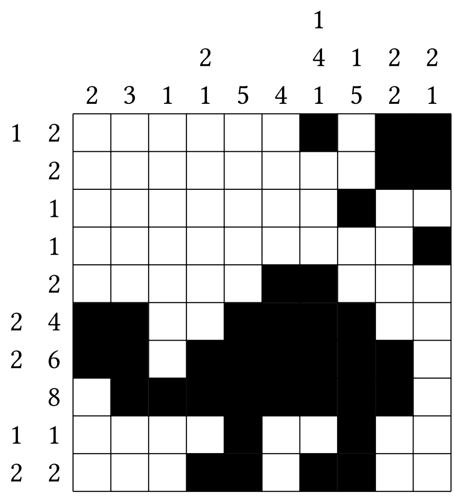
Zadanie nr 1 na pracownię

Obrazki logiczne

Obrazki logiczne (znane też jako: malowanie liczbami, nonogramy, picross) to rodzaj łamigłówki, której celem jest odkrycie zakodowanego rysunku przez zamalowywanie kratek prostokątnej siatki zgodnie z określonymi regułami. Na lewo od każdego wiersza oraz powyżej każdej kolumny siatki znajduje się ciąg cyfr oznaczający długości ciągłych bloków zamalowanych kratek znajdujących się w zadanym wierszu lub kolumnie. Bloki muszą wystąpić w tej samej kolejności, w jakiej są wymienione, muszą też być oddzielone co najmniej jedną niezamalowaną kratką.

Oto przykładowy obrazek logiczny, razem z rozwiązaniem:



Jak rozwiązuje się obrazki logiczne? Osoby rozwiązujące takie łamigłówki robią to przez dedukcję: oznaczają na siatce, które kratki muszą być zamalowane, a które niezamalowane. Wykonując takie oznaczenia, stopniowo odsłaniają się kolejne możliwe do wykonania dedukcje. W tym zadaniu nie będziemy implementować takiego podejścia – zamiast tego wykorzystamy dużo prostszą metodę, znaną jako przeszukiwanie z nawrotami.

Przeszukiwanie z nawrotami

Ideą przeszukiwania z nawrotami jest stopniowe generowanie wszystkich możliwych kandydatów na rozwiązanie w taki sposób, że jeśli wygenerowany kandydat nie jest możliwym rozwiązaniem, to algorytm nawraca do takiego punktu, w którym można zmienić podjętą przez algorytm decyzję i wygenerować kolejnego kandydata w inny sposób.

W języku OCaml wygodną metodą implementacji przeszukiwania z nawrotami jest użycie list. Funkcja przeszukująca zwraca wtedy listę rozwiązań znalezionych przez przeszukiwanie (lista pusta reprezentuje brak rozwiązań). Bardzo użyteczna jest do tego funkcja `List.concat_map`, zachowującą się jak połączenie funkcji `List.concat` i `List.map`:

```
# let concat_map f xs = List.concat (List.map f xs)
val concat_map : ('a -> 'b list) -> 'a list -> 'b list *)
```

Funkcję `concat_map f xs` można rozumieć, z punktu widzenia przeszukiwania z nawrotami, jako wybranie jednego elementu z listy `xs`, po czym kontynuowanie przeszukiwania przez aplikację funkcji `f` do wybranego elementu. Wynikami przeszukiwania są wszystkie wyniki przeszukiwania przez `f` dla każdego elementu listy `xs`. Przykładowo, poniższy kod znajduje wszystkie takie liczby z przedziału $[1, n]$, których iloczynem jest m:

```
let rec choose m n =
  if m > n then [] else m :: choose (m+1) n

let two_num_product n m =
  List.concat_map (fun a ->
    List.concat_map (fun b ->
      if a * b = m then [a, b] else [])
    (choose a n))
  (choose 1 n)
```

Taki zapis ma istotną wadę: wielokrotne użycie `concat_map` prowadzi do wielokrotnego zagnieżdżania funkcji, a w efekcie – do dużej liczby pojawiających się w kodzie źródłowym wcięć i nawiasów. Zapis taki można jednak skrócić, definiując własny operator wiązania `let*` w następujący sposób:

```
# let ( let* ) xs ys = List.concat_map ys xs
val ( let* ) : 'a list -> ('a -> 'b list) -> 'b list
```

Taka definicja wprowadza do języka konstrukcję postaci `let* x = xs in ys`, równoważną wcześniej użytemu `concat_map (fun x -> ys)xs`. Wcześniejszy przykład można przy użyciu nowej konstrukcji zapisać następująco:

```
let two_num_product n m =
  let* a = choose 1 n in
  let* b = choose a n in
  if a * b = m then [a, b] else []
```

Można ten zapis rozumieć tak: aby znaleźć dwie liczby a oraz b przedziału $[1, n]$ których iloczynem jest m, należy wybrać liczbę a z przedziału $[1, n]$, następnie liczbę b z przedziału $[a, n]$, a na końcu sprawdzić, czy ich iloczyn jest równy m. Jeśli jest, wynikiem przeszukiwania jest para (a, b); w przeciwnym wypadku nie odnaleźliśmy rozwiązania i zwracamy pustą listę wyników.

Rozwiązywanie obrazków logicznych

Skupimy się teraz na problemie rozwiązywania obrazków logicznych używając przeszukiwania z nawrotami. Rozwiązanie analogiczne do wcześniej zaprezentowanej funkcji `two_num_product` – generujące wszystkie możliwe zamalowania pól, a następnie sprawdzające, które z tych zamalowań są rozwiązaniami – jest niestety niepraktyczne ze względów wydajnościowych. Liczba możliwych zamalowań siatki $n \times m$ kratek wynosi przecież 2^{nm}, które jest bardzo dużą liczbą nawet dla niewielkich wartości m oraz n. Będziemy więc musieli wykazać się sprytem i zmniejszyć rozmiar przeszukiwanej przestrzeni rozwiązań.

Duże przyspieszenie można osiągnąć, budując obrazek wiersz po wierszu, przy czym każdy dodawany wiersz musi spełniać specyfikację podaną w łamigłówce. Tak skonstruowany obrazek należy następnie zweryfikować, sprawdzając zgodność kolumn ze specyfikacją.

Specyfikacje obrazków będziemy definiować przy użyciu następującego typu:

```
type nonogram_spec = {rows: int list list; cols: int list list}
```

Listy `rows` oraz `cols` zawierają specyfikacje wierszy (od góry do dołu) oraz kolumn (od lewej do prawej) dla zadanej łamigłówki. Rozwiązaniami będą listy list wartości boolowskich: `true` oznacza kratkę zamalowaną, a `false` – niezamalowaną.

Należy zaimplementować następujące funkcje:

- `build_row : int list -> int -> bool list list`

  Funkcja ta, dla zadanej specyfikacji wiersza i jego długości, generuje listę wszystkich wierszy spełniających specyfikację.

- `build_candidate : int list list -> int -> bool list list list`

  Funkcja ta, dla zadanych specyfikacji wszystkich wierszy oraz długości wiersza, generuje wszystkich kandydatów na rozwiązanie zagadki skonstruowanych wyłącznie z poprawnych wierszy. Należy wykorzystać funkcję `build_row`.

- `verify_row : int list -> bool list -> bool`

  Dla zadanej specyfikacji wiersza oraz zamalowań kratek wiersza, sprawdza, czy wiersz spełnia zadaną specyfikację. Funkcja ta działa również dla kolumn.

- `verify_rows : int list list -> bool list list -> bool`

  Dla zadanych specyfikacji wszystkich wierszy oraz zamalowań kratek planszy, sprawdza, czy wiersze planszy spełniają specyfikacje. Należy wykorzystać funkcję `verify_row`. Po użyciu `transpose`, funkcja ta nadaje się również do sprawdzania kolumn.

- `transpose : 'a list list -> 'a list list`

  Funkcja ta przekształca zadaną listę list (zawierającą n wierszy długości m dla pewnych n i m) na listę m wierszy długości n. Intuicyjnie, operacja odpowiada odbiciu lustrzanemu „przez przekątną”: wiersze wyjściowej listy list odpowiadają kolumnom wejściowej, a kolumny – wierszom.

Przy użyciu tych funkcji można skonstruować listę wszystkich rozwiązań w następujący sposób:

```
let solve_nonogram nono =
  build_candidate (nono.rows) (List.length (nono.cols))
  |> List.filter (fun xss -> transpose xss |> verify_rows
      nono.cols)
```

Wykorzystaj szablon rozwiązania dostępny na SKOS. Plik z rozwiązaniem, nazwany `solution.ml`, zgłoś przez system Web-CAT (dostęp przez odnośnik na SKOS) do dnia 10 kwietnia 2024, godz. 6:00.