*Lubomír Sedlář, 359719*

# 1 Růst čtverečků – hledání minimálních řešení hrubou silou

Úkolem dále popsaného algoritmu je hledat všechna minimální řešení s $k = n$ čtverečky na ploše o straně velikosti $n$. Algoritmus implementuji v jazyce C.

Základní myšlenkou je vygenerování všech možných kombinací, jak lze umístit $k$ čtverečků na danou plochu. Následně z těchto rozestavení vyberu ta, která po dopočtení přidaných čtverečků vedou k zabarvení celé plochy. V této fázi budu stav herní plochy reprezentovat jako pole $n^2$ pravdivostních hodnot. Index 0 označuje levý horní roh, $n - 1$ pravý horní roh, první pozice druhého řádku bude $n$ a čtvereček úplně vpravo dole bude pod indexem $n^2 - 1$.

Vzhledem k tomu, že na ploše o straně $n$ je možné $n$ čtverců rozestavit $\binom{n^2}{n}$ způsoby, už pro poměrně malá $n$ je třeba projít velice mnoho možných řešení. Proto před pokusem o vyřešení provedu několik jednodušších testů, které pomohou identifikovat rozestavení, která nemohou vést k zabarvení celé plochy.

Pro generování a následné testy budu stav hry reprezentovat jedním polem o $n$ prvcích typu `int`, což budou indexy v duchu předešlé reprezentace. Práce s kratším polem je rychlejší a pro potřeby testování jednodušší.

## 1.1 Generování kombinací

Pro vytváření všech možných kombinací použiji algoritmus popsaný na stránce [1], přepsaný do jazyka C (výpis 1). Tento algoritmus nejprve připraví pole o velikosti $n$ naplněné přirozenými čísly od 1 do $n - 1$. Potom ve smyčce tvoří další kombinace a každou předá funkci `process`, která kombinaci analyzuje.

V každé iteraci najde v poli ten prvek, který je nejblíže konci pole a nenabývá maximální hodnoty (poslední prvek může být nejvýše $n^2 - 1$, předposlední $n^2 - 2$ atd.), zvětší jeho hodnotu o jedna a všechny následující nastaví na jejich minimální hodnoty (pokud byl změněn prvek třetí od konce na hodnotu $a$, potom předposlední bude $a + 1$ a poslední $a + 2$).

V této podobě vnější cyklus nevytvoří všechny kombinace, ale pouze ty, kde první prvek je menší než $n$. Důvod dále rozeberu v v části 1.3 Filtrování.

## 1.2 Řešení

V této části využívám tři funkce. První funkce (výpis 2) očekává jako parametry velikost herní plochy a matici pravdivostních hodnot reprezentující tuto plochu. Zjišťuje, jestli je celá plocha zabarvená pomocí logické operace `or` provedené nad všemi prvky pole, tedy jakékoli `false` (bílý čtvereček) v matici udělá `false` na výstupu. Tato funkce má kvadratickou složitost vzhledem k parametru $n$.

Druhá funkce (výpis 3) je pomocná, jako argumenty vyžaduje index v matici, velikost hrací plochy a samotnou matici. Spočítá vybarvené sousedy čtverečku na pozici `pos`. Nestačí se pouze podívat na sousední čtverečky, v případě pozic vlevo a vpravo je třeba ověřit, že leží na stejném řádku (dávají stejný výsledek po celočíselném dělení velikostí herní plochy). V případě směrů nahoru a dolů je třeba ověřit, že indexem nebudeme přistupovat ven z matice.

Poslední funkce (výpis 4) v této fázi přebírá jako argumenty velikost herní plochy a pole reprezentující zadání, které je třeba vyřešit. Funkce je složena z hlavního `while` cyklu, který se opakuje, dokud se matice mění. V každé jeho iteraci se projdou všechna pole matice a pokud je nalezeno takové, které není vyplněné a má alespoň dva sousedy, vyplní se a do příznaku `changed` se poznačí, že je třeba provést další iteraci `while` cyklu.

Tato funkce je v celém postupu výpočetně nejnáročnější, neboť s velikostí hrací plochy kvadraticky roste i velikost pole, které je nutno projít, a také se zvyšuje počet průchodů nutných k vyřešení.

## 1.3 Filtrování

U některých vygenerovaných kombinací lze poměrně jednoduše poznat, že nemohou vést k zabarvení celé plochy. Postupně každou kombinaci zpracuji dvěma funkcemi, z nichž každá aplikuje jedno pravidlo a vrátí pravdivostní hodnotu označující kombinaci jako vadnou. Každá z těchto funkcí bude vyžadovat dva argumenty – velikost hrací plochy a pole indexů, na kterých se nachází černé čtverečky. Obě tyto funkce budou volány ze sdružující funkce ve výpisu 5.

První filtrační funkce (výpis 6) kontroluje, jestli se v posledním řádku a ve sloupci nejvíce vlevo a vpravo vyskytuje alespoň jeden čtvereček. První řádek není nutné kontrolovat, po prvním výskytu kombinace s prvním čtverečkem na druhém řádku se algoritmus generování zastaví.

K ověření posledního řádku stačí jednoduše porovnat poslední prvek s hodnotou $n \cdot (n + 1)$, což je první sloupec posledního řádku.

Kontrolu sloupců provádím cyklem procházejícím přes všechna pole. Pokud některý čtvereček leží vlevo (dává po celočíselném dělení `size` zbytek 0) nebo vpravo (zbytek `size` $- 1$), zapamatuji si tuto informaci. Pokud alespoň v jednom krajním sloupci není žádný čtvereček, zadání nemůže být řešením.

Druhá filtrační funkce (výpis 7) hledá dvojice čtverečků, které spolu přímo sousedí. Takové dvojice se ve správném řešení nemohou vyskytovat. Zjištění dvojic v horizontálním směru je triviální – projdu všechny prvky a zjistím, jestli existují takové dva po sobě následující prvky, které se liší o 1 a po vydělení `size` dávají stejný zbytek (tedy leží na stejném řádku).

Ve vertikálním směru je třeba projít přes všechny prvky dvakrát – pro každý zkusit najít takový, který se liší právě o `size`. Stačí se podívat pouze dolů, protože nezáleží na tom, který z dvojice sousedních čtverečků funkce najde.

## 1.4 Spojení předešlých funkcí

Ke spojení předešlých funkcí je třeba funkce `process` (výpis 8), která je volána nad každou vygenerovanou kombinací. Tato funkce nejprve ověří, že daná kombinace projde testovacími funkcemi, následně pole indexů převede na matici, kterou vyřeší a ověří, jestli se jedná o řešení.

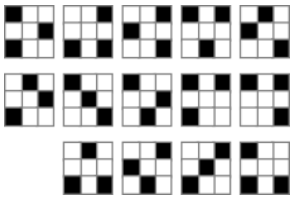
Nijak neřeším situaci, že některá rozestavení čtverečků se výsledcích objeví několikrát, pokaždé jinak otočená. Jednak se jedná o prakticky různá řešení, a za druhé je téměř nemožné sledovat, která řešení už jsem nalezl. To by totiž vyžadovalo ukládání všech řešení do nějaké datové struktury. Ať už zvolím jakoukoli, pokud do ní uložím stovky tisíc kombinací, určitě bude zásadně brzdit celý výpočet.

## 2 Výsledky

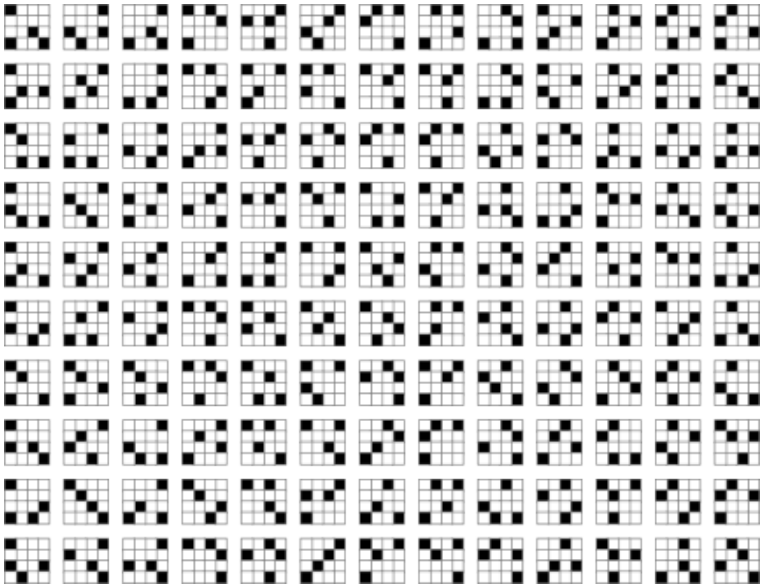
K měření doby výpočtu jsem použil program *time*. Naměřené časy jsou ale jen orientační, protože během výpočtu byl počítač běžným způsobem používaný.

| $n$ | počet řešení | počet kombinací | doba výpočtu |
|---|---|---|---|
| 1 | 1 | 1 | zanedbatelná |
| 2 | 2 | 6 | zanedbatelná |
| 3 | 14 | 84 | zanedbatelná |
| 4 | 130 | 1 820 | zanedbatelná |
| 5 | 1 615 | 53 130 | zanedbatelná |
| 6 | 23 140 | 1 947 792 | 0,5 s |
| 7 | 383 820 | 85 900 584 | 35 s |
| 8 | 7 006 916 | 4 426 165 386 | 32 min |
| 9 | 140 537 609 | $2,6 \cdot 10^{11}$ | 37 hod |

Obrázek 1: Řešení pro $n = 2$



Obrázek 2: Řešení pro $n = 3$



Obrázek 3: Řešení pro $n = 4$

# 3 Zdrojový kód

**Výpis 1**

```
void generate(int n)
{
  int c[n];
  int i, j;

  for (i = 0; i < n; i++) c[i] = i;

  while (c[0] < n) {
    process(n, c);
    i = n - 1;
    while (c[i] == n*n - n + i) i--;
    c[i]++;
    for (j = i + 1; j < n; j++) c[j] = c[i] + j - i;
  }
  process(n, c);
}
```

**Výpis 2**

```
int is_solution(int n, int matrix[])
{
  int i;
  int solution = true;
  for (i = 0; i < n*n; i++) solution = solution && matrix[i];
  return solution;
}
```

**Výpis 3**

```
int get_neighbours(int pos, int size, int matrix[])
{
  int count = 0;
  if ((pos-1) / size == pos / size && matrix[pos-1]) count++; /* vlevo  */
  if ((pos+1) / size == pos / size && matrix[pos+1]) count++; /* vpravo */
  if (pos - size >= 0 && matrix[pos-size])           count++; /* nahoře */
  if (pos + size < size*size && matrix[pos+size])    count++; /* dole   */
  return count;
}
```

**Výpis 4**

```
void solve(int n, int c[])
{
  int changed = true;
  int i;

  while (changed) {
    changed = false;
    for (i = 0; i < n*n; i++) {
      if (!c[i] && get_neighbours(i, n, c) >= 2) {
        c[i] = true;
        changed = true;
      }
    }
  }
}
```

**Výpis 5**

```
int faulty(int size, int *combination)
{
  return bad_edges(size, combination) || adjacent_squares(size, combination);
}
```

**Výpis 6**

```
int bad_edges(int size, int *c)
{
  int left = false, right = false;
  int i;

  if (c[size-1] < size*(size-1)) return true;

  for (i = 0; i < size; i++) {
    if (!left)  left  = c[i] % size == 0;
    if (!right) right = c[i] % size == size - 1;
  }
  return !left || !right;
}
```

**Výpis 7**

```
int adjacent_squares(int size, int *c)
{
  int last, i, j;
  last = -2;
  for (i = 0; i < size; i++) {
    if (c[i] == 1 + last && c[i] / size == last / size) return true;
    last = c[i];
  }

  for (i = 0; i < size; i++) {
    for (j = i; j < size; j++) {
      if (c[i] == c[j] + size) return true;
    }
  }
  return false;
}
```

**Výpis 8**

```
void process(int n, int *c)
{
  int matrix[n*n];
  int i;

  if (faulty(n, c)) return;

  for (i = 0; i < n*n; i++) matrix[i] = false;
  for (i = 0; i < n; i++)   matrix[c[i]] = true;

  solve(n, matrix);
  if (is_solution(n, matrix)) solutions++;
}
```

# Reference

[1] David Burger: *Generating Combinations with Ruby and Javascript*, dostupné online na `http://david-burger.blogspot.com/2008/09/generating-combinations-in-ruby-and_21.html`