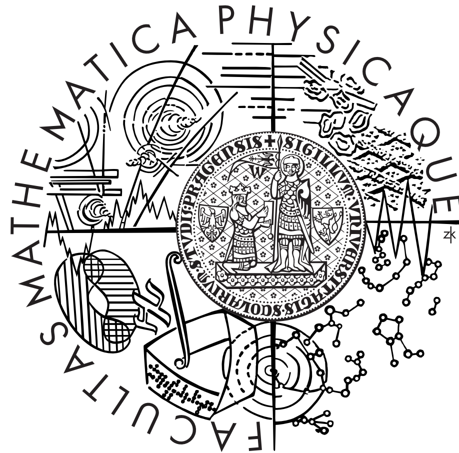
Univerzita Karlova v Praze

Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Bc. Tomáš Pavlík

Skládání obdélníků

Informatický ústav Univerzity Karlovy

Vedoucí diplomové práce:	doc. Mgr. Robert Šámal, Ph.D.
Studijní program:	Matematika
Studijní obor:	matematické metody informační bezpečnosti

Praha 2015

Poděkování.

Rád bych zde poděkoval svému vedoucímu doc. Mgr. Robertu Šámalovi, Ph.D. za cenné rady a připomínky, kterými přispěl k vypracování této práce. Dále bych rád poděkoval Katedře aplikované matematiky za poskytnutí výpočetní síly.

Prohlašuji, že jsem tuto diplomovou práci vypracoval(a) samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V .... dne .... Podpis autora

Název práce: Skládání obdélníků

Autor: Bc. Tomáš Pavlík

Katedra: Informatický ústav Univerzity Karlovy

Vedoucí diplomové práce: doc. Mgr. Robert Šámal, Ph.D.

Abstrakt: Tato diplomová práce se zabývá otevřeným problémem skládání obdélníků. Je možné naskládat obdélníky rozměrů $\frac{1}{n} \times \frac{1}{n+1}$ do jednotkového čtverce? Cílem práce je podrobná analýza tohoto problému a s ním spojeného algoritmu. Pozornost bude zaměřena hlavně na implementaci tohoto algoritmu a na studii jeho fungování.

Klíčová slova: skládání, obdélník, algoritmus, kombinatorická geometrie

Title: Packing rectangles

Author: Bc. Tomáš Pavlík

Department: Computer Science Institute of Charles University

Supervisor: doc. Mgr. Robert Šámal, Ph.D.

Abstract: This thesis studies the open problem of packing rectangles. Is it possible to pack rectangles with dimensions $\frac{1}{n} \times \frac{1}{n+1}$ into a unit square? The aim of this thesis is analysis of the problem and the related algorithm. Attention will be focused mainly on the implementation of this algorithm and on study of its functioning.

Keywords: packing, rectangle, algorithm, combinatorial geometry

Obsah

1 Úvod **2**

2 Problém **3**

2.1 Zobecnění 3

2.2 Formalizace 4

3 Algoritmus **6**

3.1 Implementace 8

3.2 Datové typy reprezentace 9

4 Analýza **12**

4.1 Indikátory 13

5 Výsledky **18**

6 Podobné problémy **19**

7 Doslov **20**

Seznam použité literatury **21**

Přílohy **22**

1. Úvod

Existuje mnoho problémů podobného znění. Máme množinu konvexních těles $C_1, C_2, \dots \in \mathbb{R}^d$ a snažíme se je poskládat do nějakého tělesa $C \in \mathbb{R}^d$ tak, aby se vnitřní části každých dvou těles neprotínali, dále této vlastnosti budeme říkat, že se tělesa nepřekrývají. Problémy jsou pak typu jestli vůbec takové C existuje, jestli je konečné a popřípadě jak malé může být. My se zde zaměříme na tři otevřené problémy, které si jsou ve svém principu velmi podobné.

Problém (1). *Mohou být obdélníky $\frac{1}{i} \times \frac{1}{i+1}$ ($i = 1, 2, \dots$) poskládány do čtverce o straně 1?*

Problém (2). *Mohou být čtverce se stranami $\frac{1}{i}$ ($i = 2, 3, \dots$) poskládány do obdélníku o obsahu $\frac{\pi^2}{6} - 1$?*

Problém (3). *Mohou být čtverce se stranami $\frac{1}{2i+1}$ ($i = 1, 2, \dots$) poskládány do obdélníku o obsahu $\frac{\pi^2}{8} - 1$?*

Všechny tři problémy jsou založeny na známých algebraických identitách. Víme tedy, že součet obsahů vkládaných obdélníků je roven obsahu cílového obdélníku. A právě tato vlastnost dělá výše popsané problémy těžkými. Zmíníme některé dosavadní úspěchy při řešení těchto úloh. V prvním problému Jennings [1], [2] dokázal, že se všechny obdélníky poskládají do čtverce o straně $\frac{204}{203}$. Bálint [3] dokázal, že pokud mohou být obdélníky $R_1, R_2, \dots, R_{k-1}$ vloženy do jednotkového čtverce, kde k je dělitelné 4, pak se všechny dají vložit do obdélníku $1 \times (1 + \frac{6}{5k})$. Předpoklad dokázal ručně ověřit až do $k = 500$ a tedy jeho nový odhad byl $1 \times 1,0024$. Ohledně třetího problému dokázal Jennings [1], že se čtverce dají poskládat do obdélníku $(\frac{1}{3} + \frac{1}{5}) \times (\frac{1}{3} + \frac{1}{9})$, jehož obsah přesahuje požadovaných $\pi^2/8 - 1$ jen o méně než $\frac{1}{299}$.

Paulhus [4] navrhl algoritmus, který postupně vložil prvních 10^9 obdélníků ve všech třech úlohách a odhadl jak musíme zvětšit původní obdélník, aby se tam vešli i všechny zbylé obdélníky. V těchto třech problémech dospěl postupně k těmto výsledkům:

$$1 \times \left(1 + \frac{1}{10^9 + 1}\right)$$

$$\frac{1}{2} \times \left(2\left(\frac{\pi^2}{6} - 1\right) + 1.606553066 \cdot 10^{-9}\right)$$

$$\frac{1}{3} \times \left(3\left(\frac{\pi^2}{8} - 1\right) + 1.344586785 \cdot 10^{-9}\right)$$

Naším cílem bude prozkoumat tento algoritmus, navrhnout jeho implementaci, ověřit předchozí výsledky a popřípadě vylepšit dosavadní odhady.

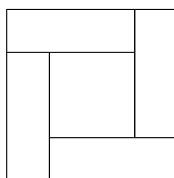
2. Problém

2.1 Zobecnění

Všechny tři předešlé problémy můžeme zobecnit tak, že máme zadaný jeden velký obdélník a posloupnost menších obdélníků, jejichž součet obsahů je stejný jako obsah velkého obdélníku. Výstupem je pak umístění jednotlivých obdélníků tak, aby se nepřekrývaly.

K tomuto problému můžeme přistupovat tak, že ho rozdělíme na více menších problémů. Velký obdélník rozdělíme podle přímky, která je rovnoběžná s jednou ze stran, a posloupnost obdélníků rozdělíme na dvě disjunktní posloupnosti, kde opět součet obsahů se rovná obsahu příslušné části. Pro vyřešení by stačilo, kdybychom každou takovou situaci uměli buď znovu rozdělit na menší, nebo kdyby posloupnost obsahovala pouze jeden obdélník a ten by se shodoval s velkým příslušným obdélníkem.

Je vhodné zmínit, že se nejedná o ekvivalenci, tedy existuje situace, které se nedá vyřešit tímto způsobem. Jednoduchý protipříklad poskytne následující obrázek 2.1.



Obrázek 2.1: protipříklad

Dále ukážeme, že je úloha v nějakém smyslu jednodušší, pokud se obdélníky v posloupnosti zmenšují pomalu.

Definice 2.1. Posloupnost $a = (a_i)_{i=1}^{\infty}$, $a_i \in \mathbb{R}_0^+$ nazveme *pomalu klesající* pokud je klesající, její součet je konečný a zároveň $\forall i \in \mathbb{N} : a_i \leq \sum_{j=i+1}^{\infty} a_j$.

Lemma 2.2. *Nechť $a = (a_i)_{i=1}^{\infty}$ je pomalu klesající posloupnost, kde $s = \sum_{i=1}^{\infty} a_i$. Pak pro každé $s' \in \mathbb{R}_0^+, s' \leq s$ existuje podposloupnost jejíž součet je s'.*

Důkaz. Budeme postupovat indukcí tak, že postupně o každém prvku rozhodneme, zda patří do vybrané posloupnosti nebo ne. Prvek a_i do posloupnosti patří právě když jeho přidáním součet podposloupnosti nepřekročí s'. Částečné součty této posloupnosti jsou rostoucí a omezené s', tedy konvergují k nějakému s''. Pro spor nechť $s'' < s'$. Zvolme j nejmenší možné tak, že a_j není ve vybrané posloupnosti a zároveň $a_j > s' - s''$. Všechny menší prvky posloupnosti pak musí být ve vybrané posloupnosti z důvodu metody konstrukce a jejich součet je alespoň a_j. Při rozhodování o prvku a_j, jestli patří do podposloupnosti, tedy zbývalo do součtu s' více než a_j a tento prvek by měl být ve vybrané posloupnosti. To je spor s volbou j a tedy $s'' = s'$. □

Toto lemma můžeme v našem problému využít následovně. Nechť obsahy vkládaných obdélníků tvoří pomalu klesající posloupnost, a uvažujme libovolný řez velkého obdélníku. Pak umíme rozdělit vkládané obdélníky na dvě disjunktní

posloupnosti tak, aby součet obsahů byl roven obsahu příslušné části velkého obdélníku. Pomalu klesající posloupnosti nám tedy dávají více prostoru tím, že řez můžeme vést libovolně.

Nic nám ale nezaručuje, že výsledné posloupnosti budou opět pomalu klesající. Naopak každou pomalu klesající posloupnost můžeme vždy rozdělit na dvě se stejným součtem obsahů a vybrat tu, která obsahuje jeden předem daný prvek. Opakovanou aplikací se nám zmenšuje součet celé posloupnosti, ale jeden prvek je stále konstantní. Tímto způsobem tedy časem dostaneme posloupnost, která není pomalu klesající.

V našem případě ale musíme brát v úvahu i rozměry obdélníků, nejen obsahy. Problém nastává, kdy sice součet obsahů souhlasí, ale rozměr některého vkládaného obdélníku je větší než rozměry velkého obdélníku. Ale ani to není dostačující podmínka, jak ukazuje následující protipříklad.

Uvažujme posloupnost obdélníků $(1, \frac{1}{2^i})$. Součet jejich obsahů je 1 a do obdélníku $(1, 1)$ jdou vložit. Vezměme ale obdélník $(1 + \epsilon, 1 - \delta)$, jehož obsah necht' je také roven 1. Pro dostatečně malé ϵ, se pak obdélníky musí překrývat.

Otevřenou otázkou tedy zůstává, jestli existuje nějaká podmínka, která by zaručila, že posloupnost obdélníků lze vložit do nějakého obdélníku. Ta bude určitě muset zahrnovat i rozměry velkého obdélníku, a ne jen vlastnosti vkládaných obdélníků.

2.2 Formalizace

Definice 2.3. *Obdélníkem* budeme rozumět uspořádanou dvojici nezáporných reálných čísel (w, h), kde w je šířka obdélníku a h je jeho výška.

Definice 2.4. *Umístěným obdélníkem* rozumíme uspořádanou čtveřici nezáporných reálných čísel (x, y, w, h), jehož vrcholy se nacházejí v bodech (x, y), $(x, y + h)$, $(x + w, y)$ a $(x + w, y + h)$. Tedy jde o obdélník (w, h) jehož levý dolní vrchol je v bodě $[x, y]$.

Dva umístěné obdélníky se *neprotínají*, pokud nemají žádný společný vnitřní bod.

Definice 2.5. Umístěním posloupnosti obdélníků (w'_i, h'_i) do obdélníku (W, H) rozumíme posloupnost umístěných obdélníků $R = (r_i)_{i=1}^{\infty}$, tak že $r_i = (x_i, y_i, w_i, h_i)$ a zároveň:

- $(w_i = w'_i \land h_i = h'_i) \lor (w_i = h'_i \land h_i = w'_i)$ pro všechna i
- $x_i \geq 0 \land y_1 \geq 0 \land x_i + w_i \leq W \land y_i + h_i \leq H$ pro všechna i
- Pro všechna $i \neq j$ se obdélníky r_i a r_j neprotínají.

Problém. *Existuje umístění posloupnosti obdélníků* $((\frac{1}{i}, \frac{1}{i+1}))_{i=1}^{\infty}$ *do obdélníku* $(1, 1)$*?*

Následující dvě věty nám říkají, že pro vyřešení tohoto problému stačí, když se jen limitně přiblížíme. Obecnější verzi dokazuje G. Martin [6] ve své práci poněkud složitě. Uvedeme proto zde jednodušší a kratší důkaz.

Věta 2.6. *Nechť existuje umístění posloupnosti obdélníků R do obdélníku $(x+\epsilon_1, y+\epsilon_2)$ pro každé $\epsilon_1 \geq 0$ a $\epsilon_2 \geq 0$, pak existuje umístění R do obdélníku (x, y).*

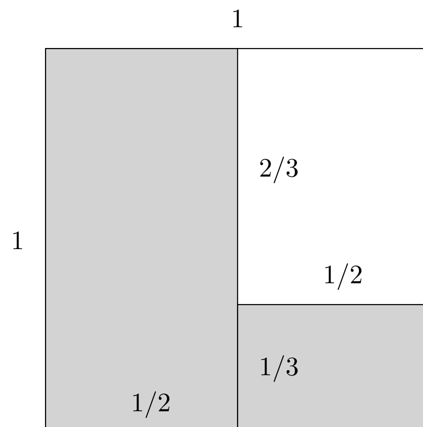
Důkaz. Uvažujme posloupnost umístění S_i do obdélníku $(x+1/i, y+1/i)$. Z těchto umístění vytvoříme posloupnost a z té budeme pro každé k vybírat podposloupnost takovou, kde prvních k umístěních obdélníků bude konvergovat. Tyto body konvergence pak budou hledaná umístění v obdélníku (x, y). Indukcí nechť máme posloupnost, kde prvních $k-1$ umístěných obdélníků konverguje. Nejdříve vybereme nekonečnou podposloupnost, kde k-tý obdélník má stejnou orientaci. Jelikož každé umístění jsou dvě čísla v kompaktním prostoru, můžeme navíc vybrat podposloupnost takovou, že umístění k-tého obdélníku konverguje. Je snadné nahlédnout, že takto nalezený obdélník se nachází uvnitř obdélníku (x, y) a zároveň se neprotíná s předešlými $k-1$ umístěnými obdélníky. □

Věta 2.7. *Nechť existuje umístění posloupnosti obdélníků R do obdélníku s obsahem $S+\epsilon$ pro každé $\epsilon \geq 0$, pak existuje umístění R do nějakého obdélníku s obsahem S.*

Důkaz. Uvažujme množinu kratších stran z obdélníků v R. Tato množina je určitě shora omezená, protože jinak by součet obsahů nebyl konečný, takže umíme najít její horní závoru x_0. Uvažujme posloupnost obdélníků (x_i, y_i) s obsahem $S_i = S + 1/i$. Víme, že $x_i \geq x_0$ a zároveň $x_i \leq S_i/x_0$, tedy x_i je v kompaktním prostoru a umíme vybrat podposloupnost takovou, že x_i konverguje k nějakému x. Je jasné, že v této posloupnosti také y_i konverguje k nějakému y a že, $x \cdot y = S$. To budou tedy rozměry hledaného obdélníku a zbytek plyne z předchozí věty. □

3. Algoritmus

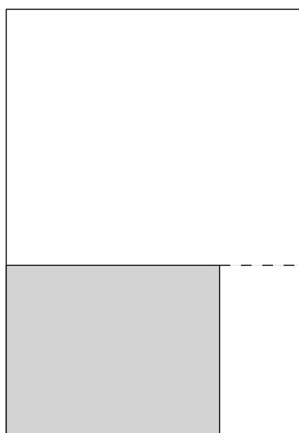
Marc M. Paulhus [4] ve své práci navrhl algoritmus na skládání obdélníků. Začneme tak, že vložíme obdélníky $(1, \frac{1}{2})$ a $(\frac{1}{2}, \frac{1}{3})$ podle obrázku.



Obrázek 3.1: první dva obdélníky

Dále budeme postupovat tak, že pro každý vkládaný obdélník nalezneme vhodný nepoužitý obdélník, do kterého ho vložíme, a zbylou oblast rozdělíme na dva obdélníky (Obr. 3.1). Budeme se řídit těmito pravidly.

- Pravidlo 1: Vlož následující obdélník do rohu nejmenšího ze zbývajících obdélníků. Pokud mají dva obdélníky stejnou šířku, vyber ten s nejmenší možnou výškou.
- Pravidlo 2: Zbytek rozděl úsečkou z volného vrcholu malého obdélníku směrem k delší straně velkého obdélníku.



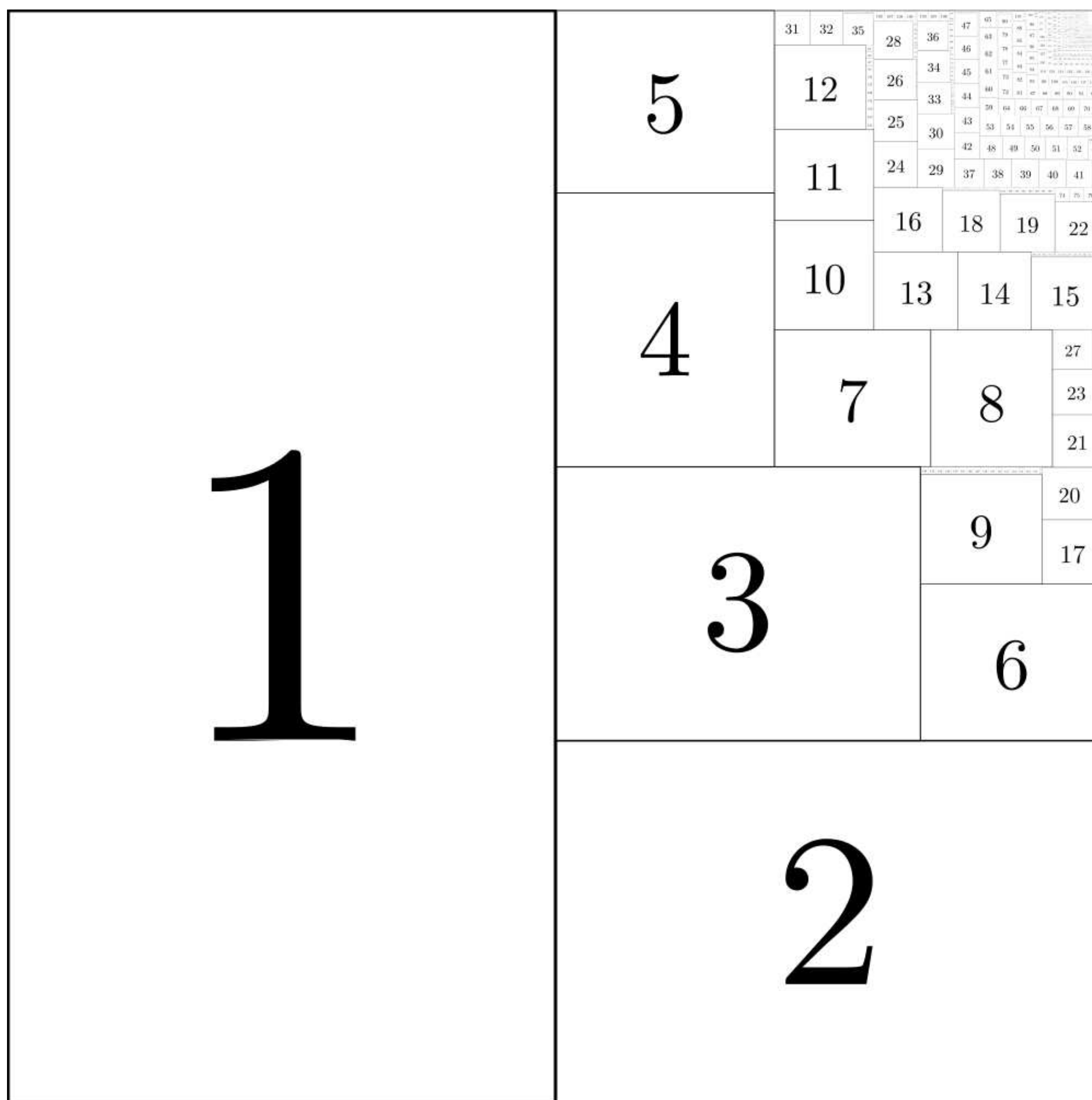
Obrázek 3.2: vložení obdélníku

Pozorování:

- Je jedno do kterého z rohů budeme obdélník umisťovat. Na obrázcích to bude vždy levý dolní roh.

- U zbývajících obdélníků si pamatujeme jen jejich rozměry, a ne polohu. Pro tento algoritmus je konkrétní poloha nedůležitá.
- V algoritmu není specifikováno, jak je orientovaný vkládaný obdélník. V naší implementaci budeme používat ještě jedno pravidlo: Pokud to lze, tak vložíme obdélník na šířku - tedy bude se dotýkat krátká strana malého obdélníku s delší stranou velkého obdélníku (Obr. 3.2).

Rozdělením na obdélníky sice ztrácíme volnost, ale lépe se pracuje s obdélníky, než s obecnými pravoúhelníky. Stačí si pamatovat jen dva rozměry a vkládaní obdélníku do jiného obdélníku je přímočaré. Navíc to vypadá, že tato metoda funguje, proto není nutné používat mnohoúhelníky.



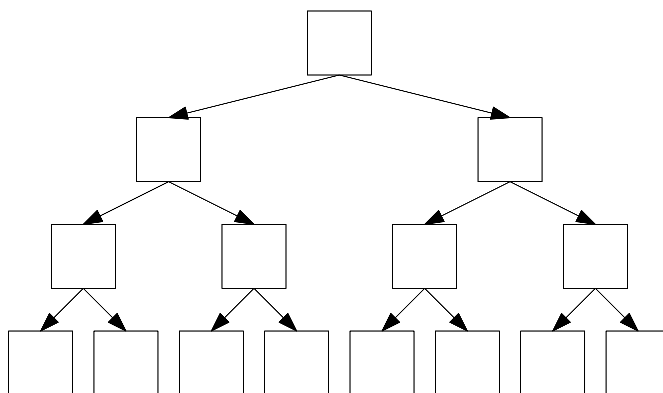
Obrázek 3.3: Prvních 1000 obdélníků

3.1 Implementace

I když Marc M. Paulhus [4] ve své práci popisuje algoritmus na vkládání obdélníků a také své výsledky. Neuvádí, jak daný algoritmus v praxi implementoval. Samotná implementace přitom nabízí dva problémy. Totiž jak reprezentovat samotné rozměry obdélníků a do jaké vhodné datové struktury zbývající obdélníky ukládat.

Pro zvolení správné datové struktury potřebujeme vědět kolik obdélníků budeme chtít ukládat, a jaké operace od této struktury vyžadujeme. Obdélníků bude přibližně stejný počet jako je počet dosavadních kroků, protože v každém kroku odebereme jeden obdélník a přidáme místo něj dva menší. Občas se nám může stát, že přidáme jen jeden nebo žádný, ale tyto počty jsou zanedbatelné. Požadované operace jsou pak vyhledávání vhodného obdélníku, což je tedy nejmenší takový, který je větší než vkládaný obdélník, následně pak jeho odebírání a vložení nově vzniklých. Pro zvolení správné datové struktury potřebujeme vědět kolik obdélníků budeme chtít ukládat, a jaké operace od této struktury vyžadujeme. Obdélníků bude přibližně stejný počet jako je počet dosavadních kroků, protože v každém kroku odebereme jeden obdélník a přidáme místo něj dva menší. Občas se nám může stát, že přidáme jen jeden nebo žádný, ale tyto počty jsou zanedbatelné. Požadované operace jsou pak vyhledávání vhodného obdélníku, což je tedy nejmenší takový, který je větší než vkládaný obdélník, následně pak jeho odebírání a vložení nově vzniklých.

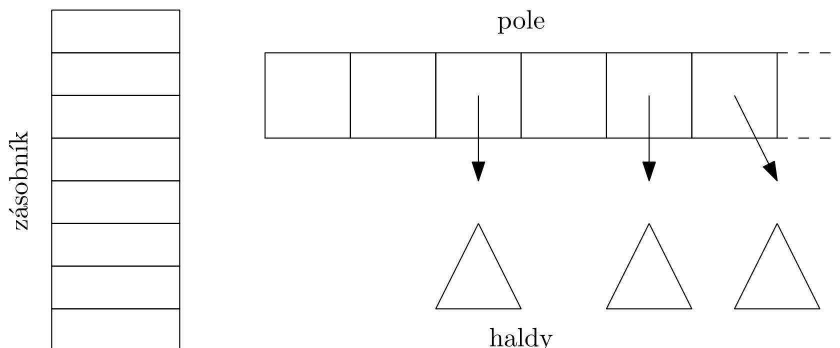
Dobrou datovou strukturou by mohl být binární vyhledávací strom, jehož vyvažovaná verze má všechny potřebné operace v $O(\log n)$, kde n je počet prvků. To nám dává časový odhad $O(n \cdot \log n)$ pro prvních n prvků. Prostorová složitost je $O(n)$. Tuto datovou strukturu nejspíše použil Paulhus, ale nemůžeme to vědět jistě. My ale zavedeme vlastní datovou strukturu, která nám umožní dopočítat se dále.



Obrázek 3.4: binární vyhledávací strom

Pro lepší datovou strukturu využijeme toho, že pro každý nově vzniklý obdélník si můžeme v konstantním čase spočítat, který největší vkládaný obdélník se do něj vejde. Obdélníky pak můžeme dávat do pole, kde index bude číslo onoho největšího vkládaného obdélníku. V každém kroku se pak můžeme podívat v konstantním čase do buňky se správným indexem a vybrat nejmenší obdélník, který se použije. Zbytek obdélníků v této buňce přesuneme do zásobníku, ve kterém budeme ukládat obdélníky, které jsou větší než současný vkládaný obdélník. Navíc

budou v tomto zásobníku seřazené od největších po nejmenší. Pokud by v nalezené buňce žádný obdélník nebyl, pak použijeme nejmenší ze zásobníku. Obdélníky v jedné buňce můžeme ukládat do haldy, což nám umožní vkládání a odebíraní v logaritmickém čase vzhledem k počtu prvků v buňce. Každý krok bude mít tedy časovou složitost jen $O(\log k)$, kde k je největší počet obdélníků v jedné buňce pole. Nevýhodou bude ale prostorová složitost. Pole bude muset být tak velké, aby bylo možné uložit i ty nejmenší obdélníky, a jejich rozměry můžou být o mnoho řádů menší než rozměry vkládaného obdélníku. Pokud například vložíme stranu délky $1/n$ do strany $1/(n+1)$, pak výsledný obdélník bude v buňce s číslem řádově n^2. Tedy prostorová složitost bude až $O(n^2)$.



Obrázek 3.5: vlastní datová struktura 1. verze

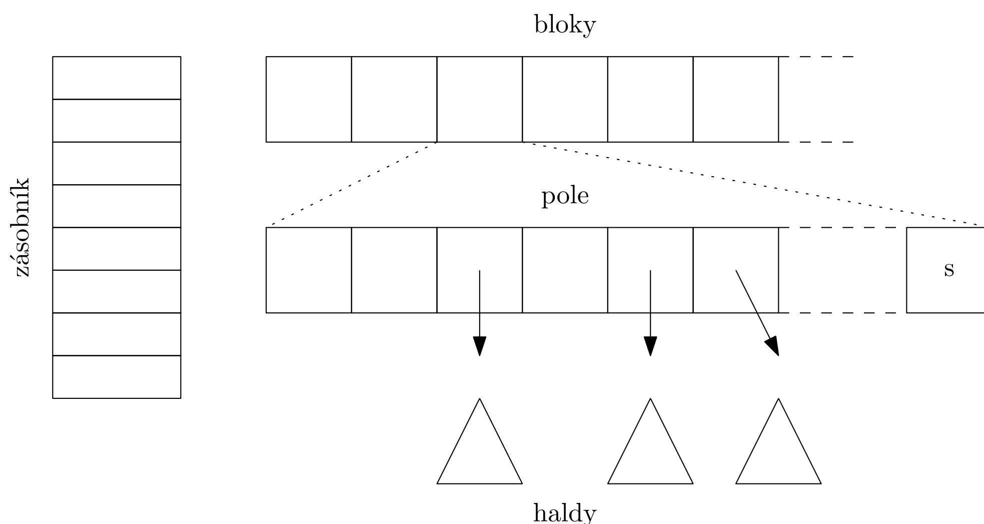
Pole bude muset být velké a začátek pole se už později nebude vůbec používat. To by zlepšilo dynamicky alokované cyklické pole. Ale i tak ve většině programovacích jazyků je problém s polem, jehož velikost přesahuje 32-bitový integer.

Vylepšíme tedy naši datovou strukturu následujícím způsobem. Buňky pole shlukneme do bloků o velikosti s. Nové obdélníky budeme ukládat jen do těchto bloků. Využijeme toho, že vyhledávaní obdélníků probíhá v blocích postupně, takže vždy potřebujeme jen jeden. Můžeme tedy všechny obdélníky z požadovaného bloku zkopírovat do pole o velikosti s. Toto pole budeme používat stejně jako v předchozím případě. Když dojedeme až na konec, pole bude prázdné a budeme ho tak moci použít na rozbalení dalšího bloku. Děláme vlastně to, že pozdržíme vložení obdélníků až než to opravdu potřebujeme, a to nám umožní recyklovat jedno pole konstantní délky s. Prostorová složitost je pak jen velikost samotných obdélníků plus velikost pole plus počet bloků, tedy přibližně $n+s+n/s$, což je pro vhodnou volbu s rovno $n + 2\sqrt{n}$.

3.2 Datové typy reprezentace

Pro přesné počítaní potřebujeme rozměry obdélníků reprezentovat jako čísla s nekonečnou přesností. Takový datový typ nám nabízí například knihovna Boost jménem cpp_rational. Tento datový typ ukládá čísla jako zlomky v základním tvaru, kde čitatel i jmenovatel jsou celá čísla s libovolnou přesností.

Druhou možností je reprezentovat délky jako celé číslo a to konkrétně 64-bitový integer bez znaménka. Jeho maximální hodnotu budeme chápat jako číslo jedna a nula zůstane nulou. Zbylé hodnoty lineárně rozprostřeme mezi nulou a jedničkou. To si můžeme představit tak, že číslo převedeme na zlomek, jehož jmenovatel je pevně daný, a uložíme jen jeho čitatel.



Obrázek 3.6: vlastní datová struktura 2. verze

Bohužel ne každé číslo lze vyjádřit jako zlomek s daným jmenovatelem, proto se při výpočtu musíme uchýlit k zaokrouhlování. Všechny délky vkládaných obdélníků tedy zaokrouhlíme nahoru k nejbližšímu požadovanému zlomku. Toto zaokrouhlování nahoru nám pak zaručuje, že i nezaokrouhlené obdélníky jdou vložit.

Nevýhodou je, že zaokrouhlováním přicházíme o místo, které někdy určitě bude chybět. Takový algoritmus se pak někdy určitě zastaví. Kdy se tak stane, můžeme přibližně odhadnout.

Nechť n je společný jmenovatel, jeho hodnota je přibližně 2^{64}. Předpokládejme, že u každého obdélníku zvětšíme oba rozměry o polovinu nejmenší jednotky v našem počítaní, což je $1/n$. Obsah i-tého obdélníku se tedy zvětší o $(\frac{1}{i} + \frac{1}{i+1})/2n$. Nyní se ptáme kolik těchto přebytků musíme posčítat, aby se to rovnalo zbylému prostoru. Hledáme tedy k takové, že

$$\sum_{i=1}^{k}(\frac{1}{i} + \frac{1}{i+1}) \cdot \frac{1}{2n} \approx \frac{1}{k}$$

$$\frac{1}{2n} \cdot \sum_{i=1}^{k} \frac{2}{i} \approx \frac{1}{k}$$

$$\frac{\ln(k)}{n} \approx \frac{1}{k}$$

$$k \approx e^{W(n)} \approx 4.5 \times 10^{17}$$

Náš odhad k je tedy dostatečně velký, protože více než 10^{17} kroků stejně nezvládneme udělat z časových i prostorových důvodů.

Toto počítání v celých číslech nám dává výhodu, že všechny operace jsou v konstantním čase a všechny rozměry zabírají konstantně prostoru. U čísel s nekonečnou přesností závisí čas i prostor na velikosti jmenovatele, který v našem případě roste velmi rychle.

W značí Lambertovu W funkci

Další varianta je datový typ double. Ten má sice velikou přesnost v hodnotách kolem nuly, ale kvůli jeho zaokrouhlování na obě strany je pro náš algoritmus nepoužitelný.

V naší implementaci budeme tedy používat vlastní, dříve popsanou, datovou strukturu a čísla budeme reprezentovat celými čísly. Tato kombinace bude dávat nejlepší výsledky. Konkrétní zdrojový kód je přiložen na konci práce.

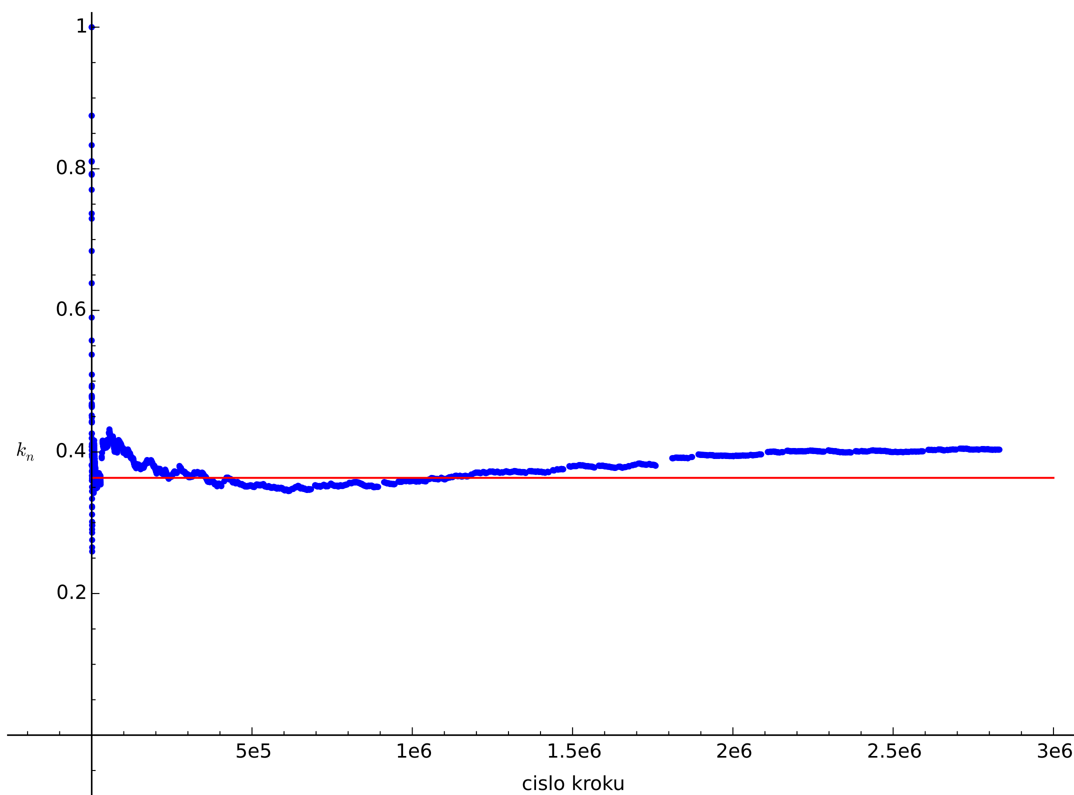
4. Analýza

Nyní se přesuneme od implementace k analýze samotného algoritmu. Budeme tedy zjišťovat, jak vypadá stav v průběhu algoritmu. Zde stavem myslíme složení uložených obdélníků.

Nechť jsme v n-tém kroku algoritmu. Vkládaný obdélník je tedy $(\frac{1}{n}, \frac{1}{n+1})$ a máme uloženy nějaké obdélníky. Největšímu z nich říkáme *hlavní čtverec*.

Při každém vkládání obdélníku máme více možností, kam ho dát. Pravidla algoritmu nám říkají, že si máme vybrat ten nejmenší z nich. To znamená, že největší z nich se nebude používat tak často, a tedy že bude zůstávat spíše velký. Navíc při vložení n-tého obdélníku se z něj uřízne jen pás šířky $1/n$, což je poměrově k jeho velikosti nepatrná část. Navíc je algoritmus navrhnutý tak, že se pás uřízne podél jeho kratší strany, což znamená, že si hlavní čtverec zachovává přibližně čtvercovitý tvar i když je to obecně obdélník. Při dalších odhadech budeme často předpokládat, že je to čtverec (pro usnadnění výpočtů).

Čím větší je hlavní čtverec v poměru ke zbytku obdélníků, tím více máme volnosti v dalších krocích, protože každý jednotlivý vkládaný obdélník do něj můžeme vložit, když se nevejde nikam jinam. Označme k_n poměr mezi obsahem hlavního čtverce a součtem obsahů všech uložených obdélníků. Součet obsahů je ale roven $1/n$, tedy $k_n = S_n \cdot n$, kde S_n je obsah hlavního čtverce v n-tém kroku. Hodnotu k_n můžeme odhadnout z měření. Z grafu (Obr. 4.1) je patrné, že se k_n pohybuje kolem $1/3$ a při přesnějším počítání z posledního spočítaného kroku (řádově 10^{13}), dostaneme konstantu $\alpha = 0.3478$.



Obrázek 4.1: Graf závislosti k_n na n

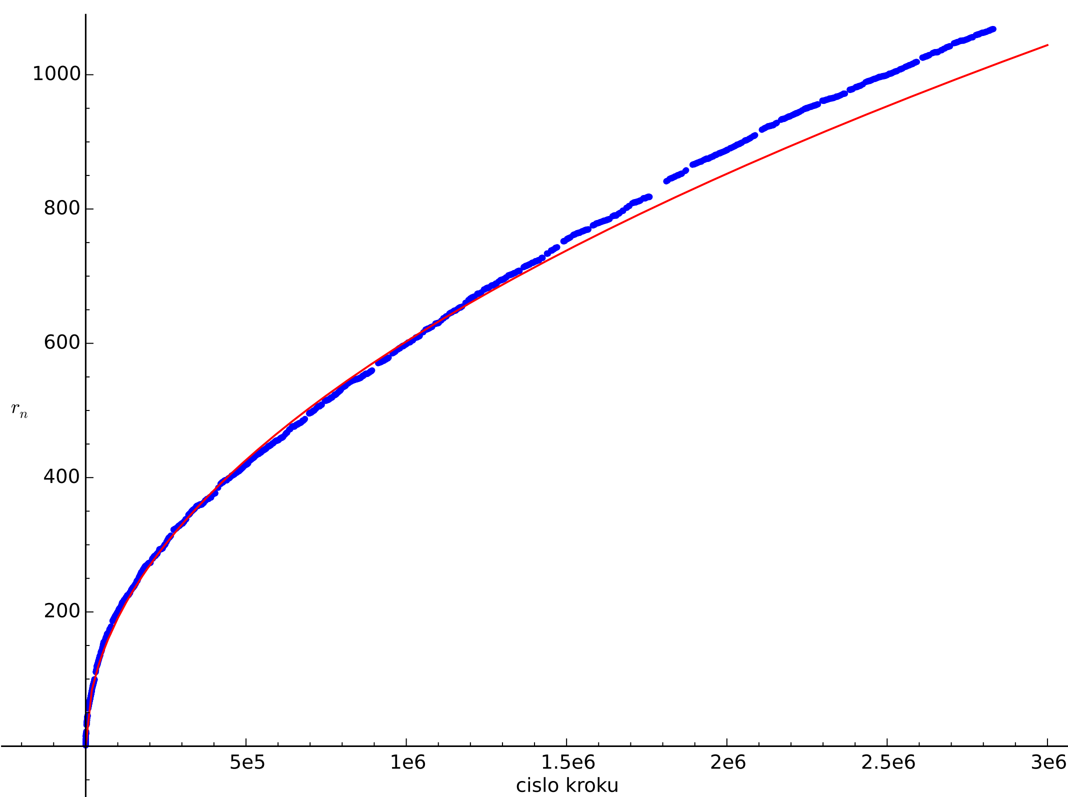
4.1 Indikátory

Pro vyřešení celého problému je potřeba dokázat, že se algoritmus nikdy nezastaví. V průběhu budeme pozorovat určité hodnoty, které nám napoví, jak si zatím vedeme. Budeme jim říkat indikátory. Jedním z nich je právě k_n. Pokud bychom dokázali, že k_n je stále přibližně α, nebo že neklesá pod nějakou konstantu, pak by se algoritmus nikdy nezastavil, protože vkládané obdélníky by stále bylo kam dávat.

Druhý podobný indikátor je poměr kratších stran mezi hlavním čtvercem a vkládaným obdélníkem, označme ho r_n. Z něho můžeme odvodit, že dalších alespoň r_n^2 obdélníků půjde vložit do hlavního čtverce. Dokud bude tato hodnota větší než jedna, víme, že algoritmus neskončí. Jelikož hlavní čtverec je přibližně čtverec, je jeho strana $a \approx \sqrt{S_n}$. Strana vkládaného obdélníku je rovna $1/n$. Můžeme teď vyjádřit vztah r_n ke k_n:

$$r_n = \frac{a}{1/n} \approx \sqrt{S_n} \cdot n = \sqrt{\frac{k_n}{n}} \cdot n = \sqrt{k_n \cdot n}$$

Hodnoty r_n můžeme také experimentálně změřit a porovnat s předpokládanou funkcí $\sqrt{\alpha \cdot n}$ (Obr. 4.2).



Obrázek 4.2: Graf závislosti r_n na n

Oba indikátory se vážou k velikosti hlavního čtverce. Abychom ale věděli, jak se v čase chová, musíme zkoumat zbylé obdélníky, protože na nich závisí to, jak často musíme hlavní čtverec zmenšit a o kolik.

Uvažujme situaci, kdy $k_n = \alpha$ a všechny obdélníky kromě hlavního jsou menší než vkládaný obdélník. Budeme sledovat algoritmus až do doby, kdy nastane opět

situace, kdy budeme vkládat do hlavního čtverce. Tento úsek nazveme hlavní krok a je jasné, že kroky algoritmu můžeme rozdělit do hlavních kroků. Počet jednotlivých kroků v hlavním kroku budeme označovat jako *délku hlavního kroku.*

Nechť pro jednoduchost je hlavní čtverec opravdu čtverec a jeho strana je tedy $a = \sqrt{S_n} = \sqrt{\alpha/n}$. Vkládáme do něj obdélník s šířkou $1/n$. Uřízneme tedy ze čtverce pás o šířce $1/n$ a délce a. Odhadněme, kolik dalších vkládaných obdélníků můžeme do tohoto pásu vložit.

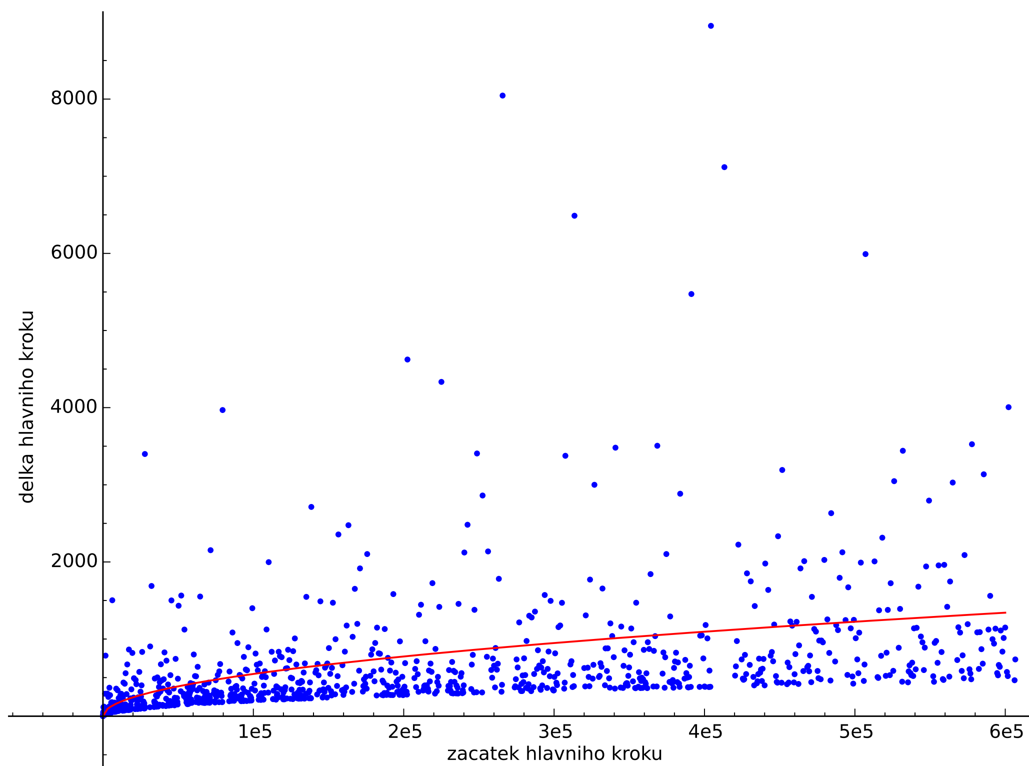
$$\frac{a}{1/n} = a \cdot n = \sqrt{\frac{\alpha}{n}} \cdot n = \sqrt{\alpha \cdot n}$$

Ne všechny tyto obdélníky ale nutně vložíme sem, protože se zmenšujícími se rozměry vkládaných obdélníků se některý může zmenšit dost na to, aby se vešel do obdélníku, který je menší. Můžeme zkusit odhadnout kolik obdélníků musíme vložit jinam než do tohoto pásu tak, aby koeficient k zůstal po hlavním kroku roven α. Pokud totiž budeme vkládat jen do pásu, pak se koeficient k bude nutně zmenšovat až k nule. Chceme tedy do hlavního čtverce, potažmo pásu, vložit poměrově α obsahu, což znamená přibližně α vložených obdélníků. Pokud tedy do pásu vložíme $\sqrt{\alpha \cdot n}$ obdélníků, pak mimo něj jich potřebujeme vložit přibližně $\frac{1-\alpha}{\alpha} \cdot \sqrt{\alpha \cdot n}$.

Je jasné, že čím jsou hlavní kroky delší, tím méně se zmenšuje hlavní čtverec a tím lepší je pro nás situace. Naopak pokud jsou kroky krátké, pak se hlavní čtverec rychle zmenšuje a blížíme se konci. Předešlou úvahou jsme odhadli, že aby byl koeficient k stále α, pak by měl hlavní krok mít délku

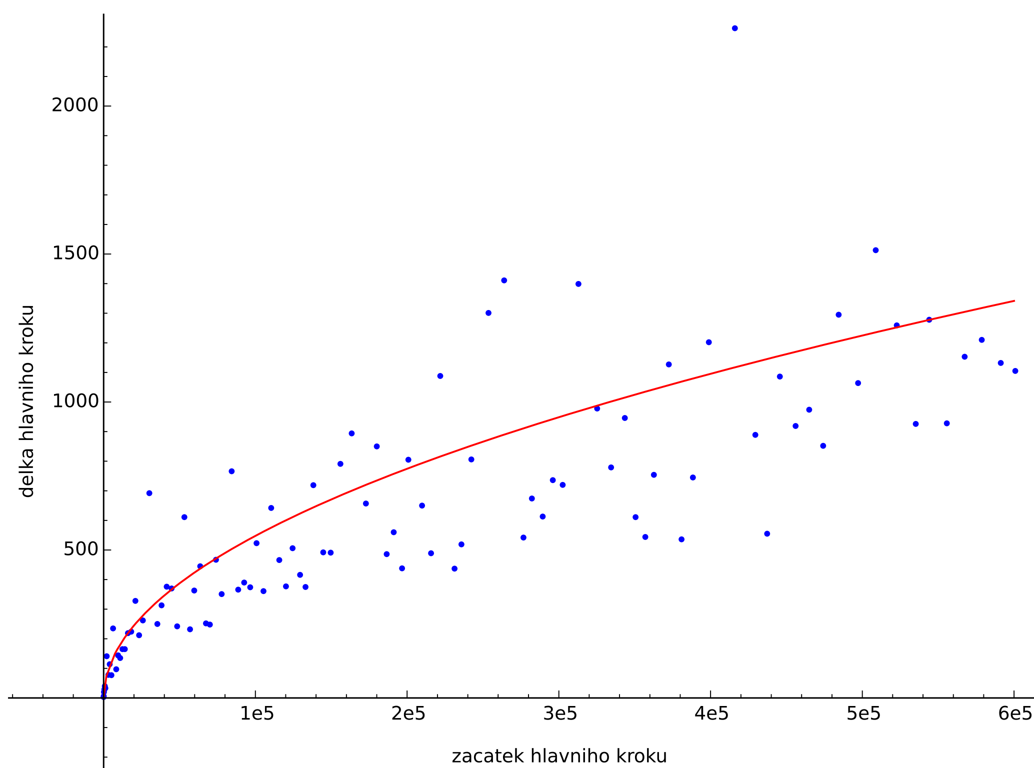
$$\left(1 + \frac{1-\alpha}{\alpha}\right) \cdot \sqrt{\alpha \cdot n} = \frac{1}{\alpha} \cdot \sqrt{\alpha \cdot n} = \sqrt{\frac{n}{\alpha}}.$$

Náš odhad můžeme porovnat se skutečnými výsledky z běhu programu. Následující graf (Obr. 4.3) zobrazuje údaje z prvních 1000 hlavních kroků a červenou linkou je zobrazen náš odhad.



Obrázek 4.3: Graf závislosti délky hlavního kroku na n

Z obrázku vidíme, že odhad přibližně sedí. Hodnoty se ale hodně mění, pro lepší zřetelnosti můžeme deset hodnot sdružit do jedné zprůměrované hodnoty.



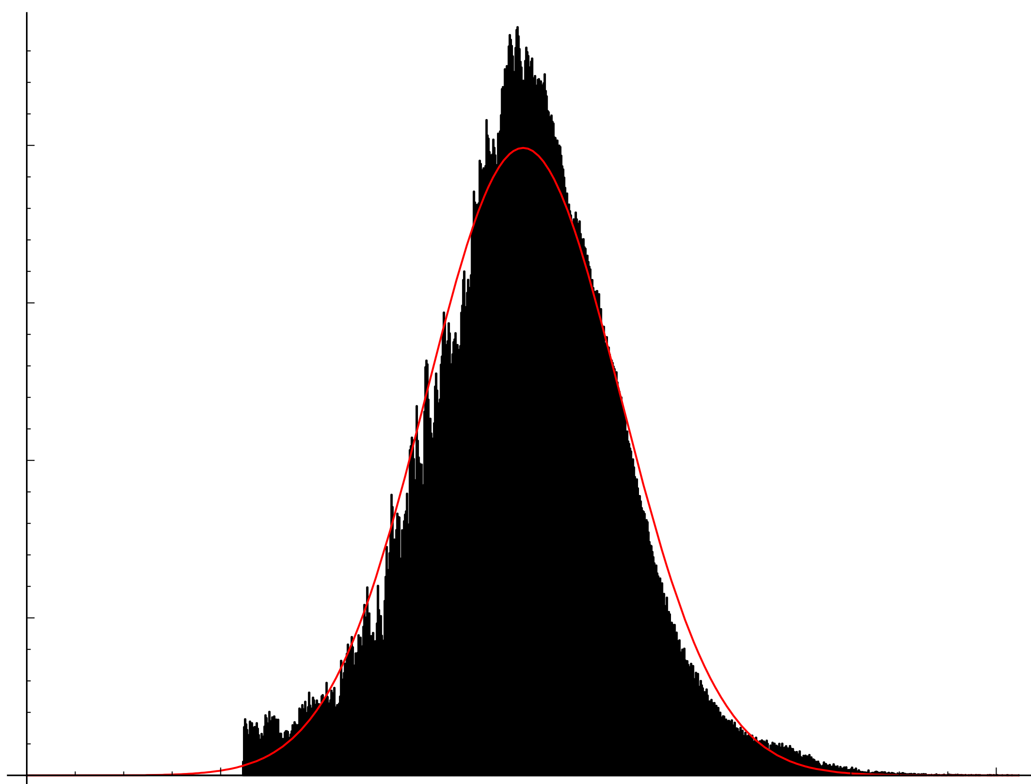
Obrázek 4.4: Graf závislosti délky hlavního kroku na n se sdruženými hodnotami

Další věc, kterou můžeme studovat, je statistika zbylých obdélníků. U nich je důležité, jak je velká jejich kratší strana. Pokud by je měly všechny obdélníky příliš krátké, pak by se algoritmus brzo zastavil. Navíc můžeme zavést pojem *kapacita obdélníku*.

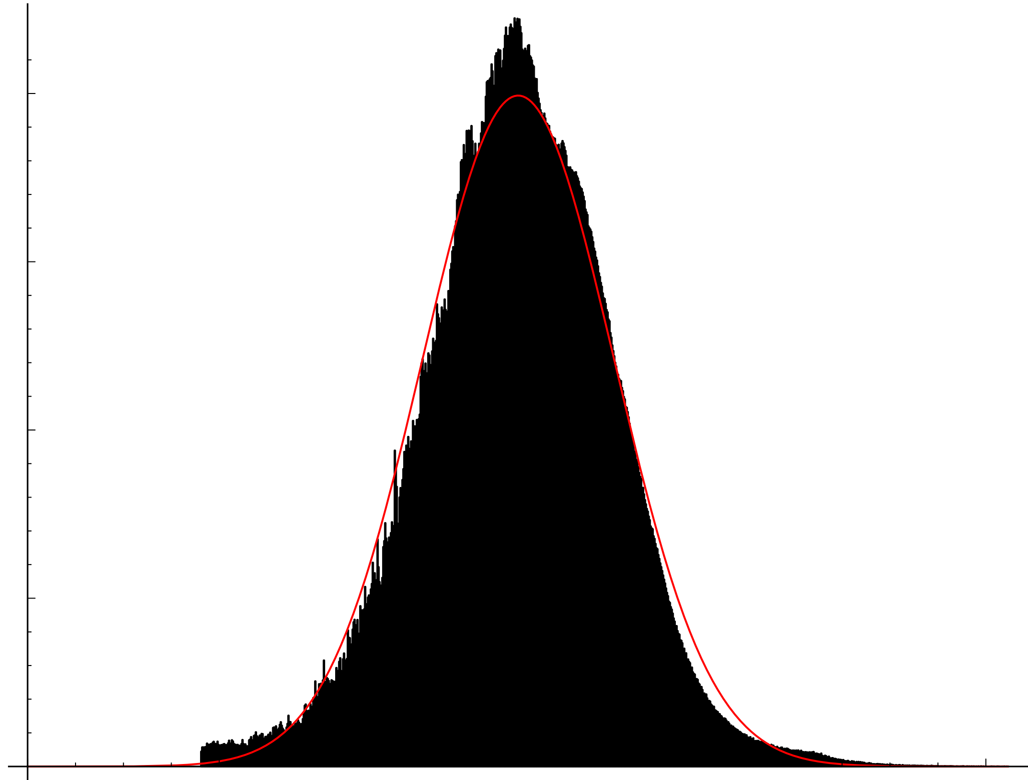
Definice 4.1. *Kapacitou obdélníku* $w \times l$ rozumíme hodnotu $\lfloor \frac{l}{w} \rfloor$, kde w je kratší strana obdélníku.

Tato hodnota vyjadřuje kolikrát za sebou můžeme tento obdélník použít. Pokud tedy zrovna vkládáme obdélník s delší stranou l, pak součet kapacit všech zbývajících obdélníků, které mají s kratší stranou menší než l, vyjadřuje počet kroků, při kterých se algoritmus určitě nezastaví. Tato hondnota bude náš další indikátor c. Zajímá nás jeho hodnota v n-tém kroce, tedy $l = 1/n$, a značíme ji c_n. Chtěli bychom dokázat, že je vždy alespoň jedna. Problém je, že tato hodnota kolísá. Konkrétně na začátku každého hlavního kroku je přesně rovna r_n, protože jediný obdélník s delší stranou je hlavní čtverec. A v ostatních případech je vetší než r_n.

Můžeme se ale v n-tém kroku podívat, jak vypadá distribuce kapacit vzhledem ke kratší straně zbylých obdélníků (Obr. 4.5 a 4.6). Ukazuje se, že pokud osu x přeškálujeme jako logaritmus z převrácené hodnoty kratší strany, pak se kapacity chovají přibližně jako normální rozdělení.



Obrázek 4.5: Znormovaný graf kapacit po 10^6 krocích
osa x: $log\left(\frac{1}{w}\right)$, osa y: kapacita



Obrázek 4.6: Znormovaný graf kapacit po 10^7 krocích
osa x: $log\left(\frac{1}{w}\right)$, osa y: kapacita

Vidíme, že vlevo mají grafy zub. Chybějící část odpovídá obdélníkům, které by byly větší než vkádaný obdélník, a ty jsou už všechny použité. V každém kroku odebereme jednu kapacitu zleva (odpovídající vložení obdélníku) a přidáme kapacity které odpovídají malému, nově vzniklému, obdélníku. Tím se nám střední hodnota posouvá doprava.

5. Výsledky

Popsaný algoritmus jsme nechali běžet přibližně 3 týdny. Za tuto dobu se mu podařilo úspěšně vložit $N = 2,67 \cdot 10^{13}$ obdélníků. Na konci měl hlavní čtverec stranu $a = 1,16677 \cdot 10^{-7}$. Abychom vytvořili celkový odhad na velký obdélník, budeme potřebovat jedno užitečné lemma.

Lemma 5.1. *Všechny čtverce o straně $1/i$, kde $i \geq n$ se dají vložit do obdélníku se stranami l a w pokud platí*

$$n \geq \frac{1+l}{w \cdot l}$$

Důkaz. Čtverce budeme postupně vkládat do řad kratších než l. Nechť $n_0 = n$. První řada bude obsahovat čtverce od n_0 do $n_1 - 1$, druhá řada bude obsahovat čtverce od n_1 do $n_2 - 1$ a i-tá řada bude obsahovat čtverce od n_{i-1} do $n_i - 1$. Položme $n_{i+1} = n_i \lfloor 1 + l \rfloor$ pro všechna $i \geq 0$. Platí tedy, že $n_i \leq n_0 (1+l)^i$ pro všechna $i \geq 0$. Pak

$$\sum_{k=n_i}^{n_{i+1}-1} \frac{1}{k} \leq \frac{n_{i+1} - n_i}{n_i} \leq l,$$

a proto se opravdu čtverce od n_{i-1} do $n_i - 1$ dají vložit i-tého řádku. A proto se všechny čtverce od n do nekonečna dají poskládat do danného obdélníku $w \times l$ pokud

$$w \geq \sum_{i=0}^{\infty} \frac{1}{n_i} \geq \frac{1}{n_0} \sum_{i=0}^{\infty} \left(\frac{1}{1+l} \right)^i = \frac{1+l}{n_0 \cdot l}.$$

□

Prvních N obdélníků tedy už máme vložených. Dalších N můžeme vložit do obdélníku $1 \times \frac{1}{N}$ a pak dalších $2N$ do obdélníku $1 \times \frac{1}{2N}$. Zbydou nám obdélníky od $4 \cdot N = 1,068 \cdot 10^{14}$ dále. Použitím předešlého lemmatu pro $w = h = a$ zjistíme, že ty můžeme vložit do hlavního čtverce. To nám dává jako celkový výsledek obdélník se stranami $1 \times (1 + 5.62 \cdot 10^{-12})$. Můžeme použít Bálintovo tvrzení [3], že pokud N je dělitelné 4, pak výsledný obdélník má strany $1 \times \left(1 + \frac{6}{5N}\right)$. Tím dostaneme o trochu lepší odhad $1 \times (1 + 4.49 \cdot 10^{-12})$. To je o tři řády lepší výsledek než doposud nejlepší $1 \times \left(1 + \frac{1}{10^9+1}\right)$.

6. Podobné problémy

Problém s podobným zadáním, ale jednoduchým řešením se objevil v roce 1974 v matematické soutěži *Kürschák Competition.*

Příklad. S_n je čtverec o straně $1/n$. Najděte nejmenší k takové, že čtverce $S_1, S_2, \ldots$ mohou být vloženy do čtverce o straně k bez překrytí.

Důkaz. Pokud zvolíme $k < 3/2$ tak se čtverce S_1 a S_2 budou určitě překrývat. Nyní dokážeme, že $k = 3/2$ vyhovuje. Do obdélníku o stranách $3/2$ a 1 vložíme čtverce S_1, S_2, S_3. Do zbývajícího pruhu $\frac{1}{2} \times \frac{3}{2}$ stačí vložit čtverce $S_4, S_5, S_6, \ldots$. Označme $M_i = \{S_j \mid 2^i \leq j < 2^{i+1}\}$. Pokud každý takový čtverec S_j shora odhadneme čtvercem S_i je jasné, že celá M_i lze poskládat do obdélníku $\frac{1}{2} \times \frac{1}{2^{i-1}}$. Tedy všechny požadované $M_2, M_3, M_4, \ldots$ lze vložit do $\frac{1}{2} \times 1$. □

Pro skládaní vícerozměrných krychlí platí následující věta, kterou dokázali společně A. Meir a L. Moser [5].

Věta 6.1. *Systém d-dimenzionálních krychlí se stranami $s_1 \geq s_2 \geq \ldots$ a celkovým objemem $V = \sum_{i=1}^{\infty}$ mohou být naskládány do kvádru $r_1 \times r_2 \times \cdots \times r_d$ pokud $s_1 \leq min\,\{r_1, r_2, \ldots, r_d\}$ a zároveň*

$$(r_1 - s_1)(r_2 - s_1) \ldots (r_d - s_1) \geq V - s_1^d.$$

Z této věty mimochodem plyne, že každý systém krychlí s celkovým objemem 1 lze vložit do krychle s objemem 2^{d-1}. K této větě existuje ještě duální, pokrývací věta.

Věta 6.2. *Systém d-dimenzionálních krychlí se stranami $s_1 \geq s_2 \geq \ldots$ a celkovým objemem $V = \sum_{i=1}^{\infty}$ mohou pokrýt kvádr $r_1 \times r_2 \times \cdots \times r_d$ pokud*

$$(r_1 + s_1)(r_2 + s_1) \ldots (r_d + s_1) \leq V + s_1^d.$$

7. Doslov

Hlavním cílem této práce bylo vyřešit alespoň jeden ze tří otevřených problému týkajících se skládání obdélníků. Zaměřili jsme se hlavně na první problém, protože s jeho vyřešením bychom mohli vyřešit i zbylé problémy. Popsali jsme zde konkrétní implementaci algoritmu, kde se každý krok provede téměř v konstantním čase. Dále jsme zkoumali jeho fungování a vylepšili dosavadní odhad u prvního z problémů o tři řády.

Připomeňme dále indikátor r_n, který se podle měření v prvních $2,67 \cdot 10^{13}$ krocích chová přibližně jako funkce $k \cdot \sqrt{n}$, kde k je konstanta. Abychom problém dokázali, stačilo by říci, že tato hodnota nikdy neklesne pod jedna. Z toho je možné usuzovat, že odpověď na zadanou otázku bude spíše ano a všechny obdélníky půjdou poskládat do jednotkového čtverce. Důkaz to však není.

Další práce, navazující na tuto, by mohly důkladněji studovat algoritmus, který vkládá obdélníky, nebo zkoumat jeho různé variace. V oblasti skládání konvexních těles zůstává stále ještě mnoho otevřených problémů, které je možné řešit.

Seznam použité literatury

[1] Jennings, D. *On packings of squares and rectangles.* Descrete Mathematics 138, 1995

[2] Jennings, D. *On packings unequal rectangles in the unit square.* Journal of Combinatorial Theory, Series A 64, 1994

[3] Bálint, Vojtěch. *Two packing problems.* Descrete Mathematics 178, 1998

[4] Paulhus, Marc M. *An Algorithm for Packing Squares.* Journal of Combinatorial Theory, Series A 82, 1997

[5] Meir, A., Moser, L. *On packing of squares and cubes.* Journal of Combinatorial Theory 5, 1968

[6] Martin, G. *Compactness theorems for geometric packings.* Journal of Combinatorial Theory, Series A 97, 2002

Přílohy

```
#include <iostream>
#include <queue>
#include <set>
#include <algorithm>
#include <cmath>

typedef unsigned long long ull;
const double PI = std::atan(1.0)*4;

using namespace std;

class Rectangle{
public:
        ull x, y;

        Rectangle(ull i, ull j){
                x = min(i, j); y = max(i, j);
        }

        ull smaller() const{
                return x;
        }

        ull larger() const{
                return y;
        }

        bool operator<(Rectangle other) const{
                if(smaller() != other.smaller()){return smaller() < other.smaller();}
                return larger() < other.larger();
        }
};

//defines ordering of rectangles
struct Order{
    bool operator()(Rectangle const& a, Rectangle const& b) const{
        return b < a;
    }
};

//general functions with rectangles
class Util{
public:
        //maximal value of side length. this equals to one.
        static const ull max_val = 1ll<<63;

        //get rectangle with index i
        static Rectangle getRect(long long i){
                return Rectangle((max_val + i - 1) / i, (max_val + i -1) / (i+1));
        }

        //get the fisrt large rectangle
        static Rectangle getFirst(){
                return Rectangle(max_val, max_val);
        }

        //return index of largest rectangle that can be inserted into r
        static long long largest_rect(Rectangle r){
                long long num = (max_val - 1 + r.smaller()) / r.smaller();
                long long num2 = (max_val - 1 + r.larger()) / r.larger();
                return std::max(num - 1, num2);
        }

        //insert r2 into r and return up to 2 remaining rectangles
        static vector <Rectangle> insert(Rectangle r, Rectangle r2){
                vector <Rectangle> vr;
                if(r.larger() <= r2.smaller() && r2.larger() >= r.smaller()){
                        if(r2.smaller() != r.larger())
                                vr.push_back(Rectangle(r.smaller(), r2.smaller() - r.larger()));
```

```
70                          if(r2.larger() != r.smaller())
71                              vr.push_back(Rectangle(r2.smaller(), r2.larger() - r.smaller()));
72                      }else if(r2.smaller() >= r.smaller() && r2.larger() >= r.larger()){
73                          if(r2.smaller() != r.smaller())
74                              vr.push_back(Rectangle(r.larger(), r2.smaller() - r.smaller()));
75                          if(r2.larger() != r.larger())
76                              vr.push_back(Rectangle(r2.smaller(), r2.larger() - r.larger()));
77                      }else{
78                          cout<<"Rectangle can not be inserted."<<endl;
79                          vr.push_back(Rectangle(0, 0));
80                      }
81                      if(vr.size()==2 && vr[0] < vr[1]){ swap(vr[0], vr[1]); }
82                      return vr;
83              }
84      };
85
86      //class that stores all remaining rectangles
87      class Rect_lib{
88      public:
89              vector <Rectangle> lifo;
90              vector <priority_queue<Rectangle, vector<Rectangle>, Order> > arr;
91              vector <vector<Rectangle> > block;
92              static const long long arr_size = 10000000;
93              static const long long block_size = 10000000;
94              long long arr_offset;
95              long long last_inserted;
96              long long count;
97              long long max_count;
98
99              Rect_lib(){
100                     last_inserted = 0;
101                     arr_offset = 0;
102                     count = 0;
103                     max_count = 0;
104                     block = vector<vector<Rectangle> >(block_size);
105                     arr = vector<priority_queue<Rectangle, vector<Rectangle>, Order> >(arr_size);
106             }
107
108             void add(Rectangle r){
109                     count++;
110                     if(count>max_count){max_count=count;}
111                     long long x = Util::largest_rect(r);
112                     if(x >= arr_size + arr_offset){
113                             if(x / arr_size < block_size){
114                                     block[x / arr_size].push_back(r);
115                             }else{count--;}
116                     }else if(x < arr_offset || x <= last_inserted){
117                             lifo.push_back(r);
118                     }else{
119                             arr[x - arr_offset].push(r);
120                     }
121             }
122
123             Rectangle lifo_out(){
124                     if(lifo.empty()){return Rectangle(0,0);}
125                     Rectangle r = lifo.back();
126                     lifo.erase(lifo.end()-1);
127                     return r;
128             }
129
130             Rectangle arr_out(int i){
131                     if(arr[i].empty()){ return lifo_out(); }
132                     Rectangle r = arr[i].top();
133                     arr[i].pop();
134                     return r;
135         }
136
137             void arr_move(int i){
138                     while(!arr[i].empty()){
139                             Rectangle r = arr[i].top();
140                             arr[i].pop();
141                             lifo.push_back(r);
142                     }
```

```
143         }
144 
145         Rectangle get(){
146             count--;
147             last_inserted++;
148             Rectangle r = Rectangle(0,0);
149             if(last_inserted < arr_offset + arr_size){
150                 r = arr_out(last_inserted - arr_offset);
151                 arr_move(last_inserted - arr_offset);
152             }
153             if(last_inserted == arr_offset + arr_size - 1){
154                 arr_offset += arr_size;
155                 for(int i = 0; i < block[arr_offset/arr_size].size(); i++){
156                     add(block[arr_offset/arr_size][i]);
157                 }
158                 block[arr_offset/arr_size].clear();
159             }
160             return r;
161         }
162 
163         void run(){
164             long long step = 1;
165             add(Util::getFirst());
166             while(true){
167                 if(step%10000000==0){
168                     cout<<"step: "<<step<<endl;
169                 }
170 
171                 Rectangle r = Util::getRect(step);
172                 Rectangle r2 = get();
173                 if(r2.x == 0 || r2.y == 0){
174                     cout<<"Rectangle not found. Step: "<<step<<endl;
175                     return;
176                 }
177 
178                 vector <Rectangle> vr = Util::insert(r, r2);
179                 for(int s = 0; s < vr.size(); s++){
180                     if(vr[s].smaller() < 0){
181                         cout<<"Rectangle do not fit. Step: "<<step<<endl;
182                         return;
183                     }
184                     add(vr[s]);
185                 }
186                 step++;
187             }
188         }
189 
190     };
191 
192     int main(){
193         Rect_lib rl = Rect_lib();
194         rl.run();
195         return 0;
196     }
```