FAKULTA APLIKOVANÝCH VĚD
ZÁPADOČESKÉ UNIVERZITY
V PLZNI

KIV/PRO

Výpočet konvexního obalu ve 2D

03. 12. 2021 Lukáš Varga

Obsah

1 Úvod 3

1.1 Zadání 3

1.2 Řešený problém 3

2 Existující metody 3

2.1 Jarvis March 4

2.2 Graham Scan 4

2.3 Chan’s Algorithm 4

3 Zvolené řešení 4

3.1 Jarvis March 5

3.2 Graham Scan 6

4 Experimenty a výsledky 7

4.1 Porovnání časů 7

4.2 Zhodnocení výsledků 8

5 Závěr 9

Literatura 9

1 Úvod

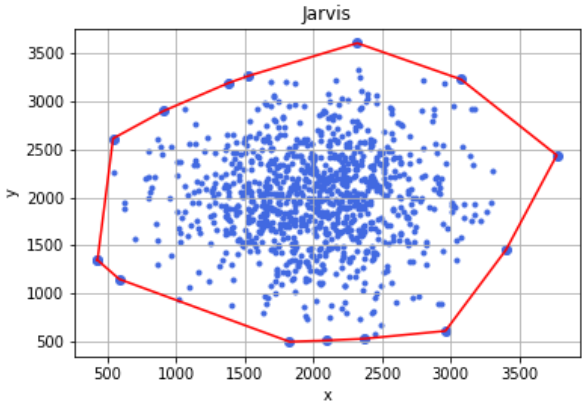
1.1 Zadání

Prostudujte pro zvolený problém existující metody řešení. Vyberte jednu z nich nebo navrhněte vlastní, implementujte a ověřte na experimentech. Postup a výsledky popište ve zprávě.

1.2 Řešený problém

Ve svém práci se budu zabývat porovnáním dvou algoritmů pro hledání konvexního obalu a také jejich testováním na vstupních náhodně generovaných datech.

Mějme množinu S obsahující n bodů v d-dimenzionálním prostoru. Naším úkolem je najít nejmenší možný mnohoúhelník (mnohostěn) takový, že obsahuje všechny body množiny S [1]. Pokud se nám toto podaří, právě jsme nalezli konvexní obal či konvexní obálku těchto bodů. Obal je tedy tvořen hraničními body množiny S, jako vidíme na Obr. 1.



Obr. 1: Konvexní obal nad body v rovině

Nalezení konvexního obalu množiny bodů je jedním z elementárních problémů výpočetní geometrie, stejně jako je třídění elementárním problémem v kombinatorických algoritmech. Konstrukce obalu zachycuje hrubou představu o tvaru a rozsahu množiny bodů.

2 Existující metody

Mezi hlavní algoritmické metody hledání konvexního obalu patří následující tři algoritmy - *Jarvis March*, *Graham Scan* a *Chan's Algorithm.* V následujících kapitolách se pokusím nastínit jejich princip, výhody a nevýhody.

2.1 Jarvis March

Tento algoritmus, někdy také označovaný jako *Gift Wrapping Algorithm*, je implementačně nejjednodušším algoritmem z výše uvedených. Velkou předností této metody je také jeho převoditelnost do *n*-dimenzí [1]. Jeho principem je postupné procházení bodů a přidávání těch bodů, které postupně obalí danou množinu.

Jarvis March má výpočetní složitost odpovídající $O(h*n)$, kde n odpovídá počtu bodů v množině a h značí počet výsledných bodů konvexního obalu [2]. Z toho vyplývá, že máme-li $h << n$, tak algoritmus má poměrně obstojnou složitost vůči ostatním zmíněným algoritmům. V opačném případě ale můžeme dojít ke složitosti $O(n^2)$.

2.2 Graham Scan

Další uvedený algoritmus pohlíží na daný problém podobně, ale rafinovaněji. Algoritmus si seřadí body podle úhlu a poté přidává body, pokud je výsledný úhel poslední přidané trojice stále konvexní. V opačném případě body odebírá.

Tento algoritmus se může pochlubit výpočetní složitostí $O(n*log(n))$, jelikož jeho „bottleneck“ je právě ona operace řazení, kterou lze obecně nejlépe řešit právě v supralineární složitosti. V nejhorším případě je tedy optimálnější než předchozí algoritmus. V nejlepší případě, pokud budou body tvořit polygon, můžeme dokonce dosáhnout na výpočetní složitost $O(n)$. Nevýhodou tohoto algoritmu je fakt, že ho lze aplikovat pouze ve dvoudimenzionálním prostoru [2].

2.3 Chan’s Algorithm

Poslední a taky nejmladší algoritmus kombinuje výhody obou předchozích algoritmů. [5] Pokusím se nastínit jeho princip. Nejprve si množinu bodů rozdělíme na několik clusterů, u kterých najdeme konvexní obal pomocí *Graham Scan* algoritmu. Poté vezmeme výsledné body konvexních obalů clusterů a provedeme s nimi algoritmus Jarvis March. Takto získáme konvexní obal všech bodů dané množiny.

Výpočetní složitost tohoto algoritmu je $O(n\ log(h))$, kde n odpovídá počtu bodů v množině a h značí počet výsledných bodů konvexního obalu [2]. Získaná výpočetní složitost je tedy lepší, než jakou mají předchozí dva algoritmy. Mezi další výhodu tohoto algoritmu patří, že je proveditelný ve 2D a 3D. Jako nevýhodu bych označil složitější implementaci.

3 Zvolené řešení

Jak už jsem zmínil v úvodu, pro svou práci jsem vybral dva algoritmy. V této kapitole se pokusím u každého z nich detailněji přiblížit princip fungování i za pomoci pseudokódů.

3.1 Jarvis March

Smysl této metody si můžeme představit následovně [3]. Na vstupu máme množinu bodů ve 2D prostoru. Nejprve si zvolíme bod s nejmenší hodnotou *x*. Následující bod nalezneme tak, že metodou podobné *brutální síle* procházíme všechny ostatní body a vybereme ten, který má s výchozím bodem nejmenší úhel proti směru hodinových ručiček. Jako výchozí bod si zvolíme nově přidaný bod a opakujeme procházení zbývajících bodů. Pokud jsou bodu kolineární, přidáme poslední nalezený bod. Takto proti směru hodinových ručiček se snažíme obalit všechny ostatní body, jako když balíme vánoční dárek do balicího papíru. Skončíme až v momentě, kdy opět dojdeme k prvnímu zvolenému bodů.

Jako implementaci jsem použil kód, který naleznete v literatuře [3]. Kód je mírně upraven obalením samotných metod do jedné třídy pro snazší a přehlednější práci při následném testování. Pro jednoduchost jsem v pseudokódu neuvedl dílčí metody výpočtů a překopírování bodů do pole z výsledných indexů obalu.

Jarvis March

[VSTUP:] *points* // Množina bodů *points* ve 2D
[VÝSTUP:] *result* // Množina indexů bodů *result*, jež jsou konvexním obalem množiny *points*

```
def convexHull(points):
    n = length(points)
    // Osetreni aby byly zadany alespon 3 body
    if n < 3:
        return null
    // Nalezeni nejlevejsiho bodu (min x-souradnice)
    l = Left_index(points)
    hull = []
    p = l
    q = 0
    // Dokud se nedostaneme k indexu puvodniho bodu "l"
    while(True):
        hull.append(p)
        // Modulo resi pripadne preteceni pole u nasledujiciho bodu
        q = (p + 1) % n
        // Prochazime body "i" a hledame ty, u kterych je uhel tvoreny
        body "p", "i", "q" nejmensi mozny
        for i in range(n):
            // 0 --> p, q and r are collinear
            // 1 --> Clockwise
            // 2 --> Counterclockwise
            if(orientation(points[p], points[i], points[q]) == 2):
                // Pridame posledni nalezeny counterclockwise turn
                q = i
        // Novy bod "q" -> hull (pri dalsim pruchodu)
        p = q
        // Narazili jsme na prvni bod -> konec
        if(p == l):
            break
    // Vracime pole indexu s prvky konvexniho obalu
    return hull
```

3.2 Graham Scan

Nyní je na čase představit si implementaci druhého zmiňovaného algoritmu na hledání konvexního obalu [6]. Nejprve si opět zvolíme krajní bod například ten s nejmenší hodnotou *x*. Tento bod přidáme do zásobníku. Dále projedeme všechny ostatní body a seřadíme si je podle úhlů, které svírají s původním bodem proti směru hodinových ručiček. Následně do zásobníku přidáváme body podle tohoto seřazení. Jakmile je v zásobníku 3 a více začneme po každém přidání rovněž testovat, zda-li poslední trojice bodů vytváří levotočivý úhel. Na to použijeme výsledek vektorového součinu daných po sobě jdoucích dvojic. Pokud je hodnota vektorového součinu kladná, přidáním nového bodu se zachová konvexnost množiny doposud přidaných bodů v zásobníku. V opačném případě bychom však vytvářeli konkávní úhel. To nechceme, a proto odstraníme prostřední bod a pokračujeme s přidáním následujícího bodu do zásobníku [4]. Případné kolineární v tomto případě body jsou zachované. Takto postupujeme, dokud neprojdeme všechny body původní množiny.

Při implementaci jsem použil kód, který najdete v literatuře [4]. Jelikož autor kódů pracuje s body jakožto s listem dvojic a já ve svém programu používám *class Point* [3], musel jsem před samotným výpočtem změnit strukturu dat. Aby se tato operace nepromítnula do časového srovnání, měřím pouze logický vnitřek metody bez měření vedlejších operací. V pseudokódu jsem opět vynechal dílčí metody.

Graham Scan

[VSTUP:] *points* // Množina bodů *points* ve 2D
[VÝSTUP:] *results* // Množina bodů *result*, jež jsou konvexním obalem množiny *points*

```
def convexHull(points):
    hull = []
    // Nalezeni nejpravejsiho bodu, v pripade shody min y-hodnota
    points.sort(key=lambda x:[x[0],x[1]])
    // Pridani nejpravejsiho bodu do mnoziny konvexniho obalu
    // Zaroven bod odstranime ze zasobinku metodou "pop"
    start = points.pop(0)
    hull.append(start)
    // Serazeni zbyvajicich bodu podle uhlu svirajiciho s prvnim bodem
    points sort(key=lambda p: (get_slope(p,start), -p[1],p[0]))
    // Prochazime vsechny body
    for pt in points:
        // Docasne pridani dalsiho bodu z mnoziny vsech bodu do obalu
        hull.append(pt)
        // Jestlize hull ma aspon 3 body testujeme konvexnost pomoci
        // vektoroveho soucinu tri aktualnich bodu
        while ((len(hull) > 2)
            and (get_cross_product(hull[-3],hull[-2],hull[-1]) < 0):
                // Pokud uhel neni konvexni, odstranime prostredni bod
                // Jinak bod nechame a pokracujeme dalsi iteraci
                hull.pop(-2)
    // Vracime pole s body konvexniho obalu
    return hull
```

4 Experimenty a výsledky

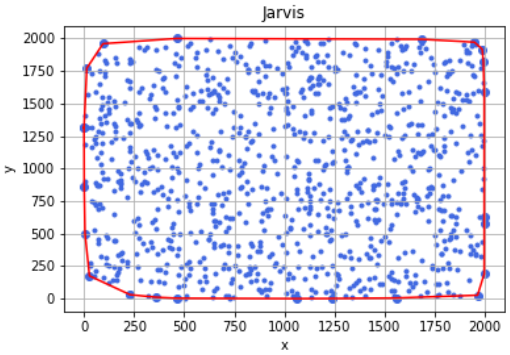
Program jsem napsal v jazyce Python verze 3, kde jsem pro vývoj použil cloudové prostředí *Colab (Google Colaboratory)* [7]. Tato služba již obsahuje nakonfigurované Python prostředí ze základními knihovnami. K programu není klasický zdrojový kód či spustitelný soubor, jako tomu je například u Javy či C++, nýbrž se spouští jednotlivé po sobě jdoucí skripty v tzv. *Notebooku*. Tento dokument je rozdělen na jednotlivé sekce s buňkami kódu. U každé buňky je okomentované, za co přesně daná část skriptu zodpovídá.

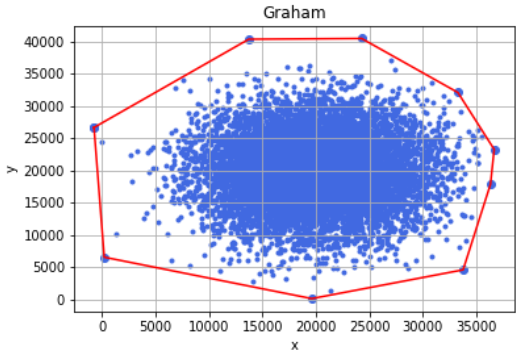
Program lze rovněž stáhnout z *Colab* jako soubor s příponou *.ipynb* a postupně spustit skripty podobným způsobem na lokálním PC pomocí nástroje *Jupyter Notebook* [8]. To již ale vyžaduje stažené a nakonfigurované Python prostředí s potřebnými knihovnami. Zmíněnou možnost jsem následně využil pro samotné testování a export výsledků. Níže jsou uvedené všechny knihovny, které jsem použil při vývoji:

```
import time // Mereni casu behu programu
import numpy as np // Generovani testovacich dat a prace s cisly
import pandas as pd // Vysledne tabulky a export do CSV
from matplotlib import pyplot as plt // Export vysledku do grafu
```

4.1 Porovnání časů

Pro porovnání časů trvání obou algoritmů (*Jarvis March* a *Graham Scan*) jsem použil náhodně generovaná čísla s celkem pěti různými hodnotami počtu bodů (1 000, 10 000, 25 000, 50 000, 100 000). Tyto datové sady jsem zároveň generoval za použití dvou různých statistických rozděleních, jmenovitě rovnoměrné (Obr. 2) a Gaussovo/normální (Obr. 3).





Obr. 2 (Vlevo): Jarvis March na 1 000 bodech v rovnoměrném rozdělení

Obr. 3 (Vpravo): Graham Scan na 10 000 bodech v Gaussovo rozdělení

Na následujících tabulkách (Tab. 1 a Tab. 2) můžete vidět srovnání časů běhu obou algoritmů na generovaných datech. Daný časový výsledek je uveden v sekundách. Výpočty běžely na PC s procesorem Intel Core i5-10300H s taktem 2.50 GHz, RAM 16 GB a Windows 11 64-bit:

Even Distribution [s]

points	Jarvis	Graham
1000	0.015490	0.002992
10000	0.210664	0.035804
25000	0.604400	0.085677
50000	1.331728	0.236573
100000	1.983256	0.552011

Gauss Distribution [s]

points	Jarvis	Graham
1000	0.017206	0.003990
10000	0.075772	0.039590
25000	0.361050	0.098114
50000	0.743473	0.253989
100000	1.060200	0.514099

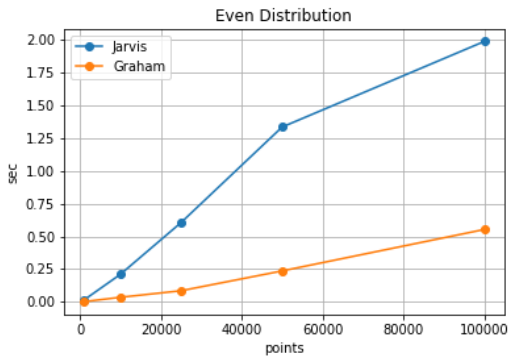
Tab. 1 (Vlevo): Porovnání časů obou algoritmu v rovnoměrném rozdělení [sekundy]

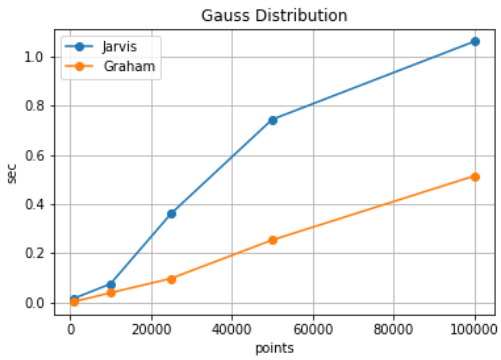
Tab. 2 (Vpravo): Porovnání časů obou algoritmu v Gaussovo rozdělení [sekundy]

Z výsledků v tabulkách je vidět, že z obou testovaných algoritmů je rychlejší *Graham Scan*. Platí to pro všechny testované velikosti množin bodů nezávisle na volbě rozdělení.

4.2 Zhodnocení výsledků

Pokud si tyto výsledky zaneseme na graf (Obr. 4 a Obr. 5), vidíme, že spojnice časů u algoritmu *Jarvis March* je ve všech místech nad spojnicí časů algoritmu *Graham Scan*. Taky si můžeme všimnout, že čím větší je množina dat, tím větší je tento časový rozdíl v běhu obou algoritmů. Avšak po určitém počtu bodů ve vstupní množině se již sklon obou grafů začíná opět srovnávat. Všechny tyto úvahy jsou brány pouze pro vybrané dvě statistické rozdělení.





Obr. 4 (Vlevo): Poměr počtu testovaných bodů na čase běhu v rovnoměrném rozdělení

Obr. 5 (Vpravo): Poměr počtu testovaných bodů na čase běhu v Gaussovo rozdělení

5 Závěr

Analyzoval jsem dva převzaté algoritmy na hledání konvexního obalu v dvoudimenzionální množině bodů, viz kapitola 3.1 a 3.2. Následně jsem oba implementované algoritmy otestoval na vstupních náhodně generovaných datech různé velikosti, viz kapitola 4. Při generování jsem také bral v potaz dvě odlišná statistická rozdělení (Gaussovo a rovnoměrné).

V rámci experimentu jsem nejprve graficky znázornil výsledný konvexní obal obou algoritmů pomocí Pythonu a za použití několika jeho knihoven. Při testování rychlosti běhu algoritmů jsem zjistil očekávaný výsledek a to, že *Graham Scan* je časově efektivnější algoritmem oproti algoritmu Jarvis March. Nakonec jsem výsledky doby běhu opět vizualizoval a zhodnotil. Všechny obrázky použité v této práci byly vykresleny mou aplikací pomocí knihovny Matplotlib [9].

Literatura

[1] **S. Skiena** (2008). The Algorithm Design Manual. *Spring-Verlag, New York Berlin Heidelberg*. [Online] [cit. 03.12.2021] http://mimoza.marmara.edu.tr/~msakalli/cse706_12/SkienaTheAlgorithmDesignManual.pdf

[2] **P. Felkel** (2018). Convex Hull In 3 Dimensions. *FEL CTU Prague* [Online] [cit. 03.12.2021] https://cw.fel.cvut.cz/b181/_media/courses/cg/lectures/05-convexhull-3d.pdf

[3] **GeeksforGeeks.org** (2021). Convex Hull | Set 1 (Jarvis's Algorithm or Wrapping). *GeeksforGeeks.* [Online] [cit. 03.12.2021] https://www.geeksforgeeks.org/convex-hull-set-1-jarviss-algorithm-or-wrapping/?ref=lbp

[4] **C. Levengood** (2020). Convex hulls in Python: the Graham scan algorithm. *LVNGD Blog.* [Online] [cit. 03.12.2021] https://lvngd.com/blog/convex-hull-graham-scan-algorithm-python/

[5] **Leios Labs** (2016). Gift Wrapping Algorithm (Convex Hull). [Online] [cit. 03.12.2021] https://www.youtube.com/watch?v=ZnTiWcIznEQ&ab_channel=LeiosLabs

[6] **Stable Sort** (2020). Convex Hull Algorithm - Graham Scan and Jarvis March tutorial. [Online] [cit. 03.12.2021] https://www.youtube.com/watch?v=B2AJoQSZf4M&ab_channel=StableSort

[7] **Google** (2019). Google Colaboratory. [Online] [cit. 03.12.2021] https://colab.research.google.com/

[8] **Jupyter** (2014). Jupyter Notebook. [Online] [03.12.2021] https://jupyter.org/install

[9] **Matplotlib** (2012). Matplotlib: Visualization with Python [Online] [03.12.2021] https://matplotlib.org/