KIV/PRO

Zpráva zabývající se problémem konvexní obálka z článku Skiena, S. S. (2008). The Algorithm Design Manual(2nd ed.). Springer. [1]

Obsah

Zadání

Definice problému: Nalézt nejmenší možný polygon (konvexní obálku), který obsahuje všechny body množiny S.

Input: Množina S obsahující n bodů v d-dimenzionálním prostoru.

Output: Množina O obsahující body tvořící konvexní obálku množiny S.

Důležitost problematiky: Jedná se o jednu z nejzákladnějších úloh ve výpočetní geometrii. Díky konvexnímu obalu můžeme získat pojem o tvaru nebo rozsahu dat. Řešení úlohy poskytuje základ k řešení několika dalších složitějších úloh ve výpočetní geometrii. Často se úloha vyskytuje ve fázi předzpracování dat v geometrických algoritmech. Například pro nalezení 2 bodů, které se od sebe nacházeji v nejdelší vzdálenosti. Je jasné, že taková dvojice leží na konvexním obalu. Proto se udělá předzpracovaní dat, kde se zjistí množina bodů patřící do této obálky. Nasledně se z množiny pro každý bod určí vzdálenost do ostatních a aktualizuje se nejdelší vzdálenost.

Důležité předpoklady k výběru optimálního algoritmu

Dimenze prostoru: Existuje několik algoritmů, které pracují v dobré složitosti O(nlogn) v případě, že body jsou v 2 nebo 3 dimenziálním prostoru. S postupně zvyšující se dimenzí v algoritmech přestávají platit určité předpoklady, jejichž nepřítomnost způsobí pád algoritmu. Je potřeba také rozlišovat, jestli chceme pouze body konvexní obálky, případně tvar, kterým může být například v 3 dimenzionálním prostoru krychle. Typicky platí čím vyšší je dimenze, tím více je algoritmus složitý.

Typ vstupu: Je třeba rozlišovat vstup, kdy máme množinu bodů (hledáme konvexní obal) a kdy poloroviny (hledáme průnik polorovin). Obě 2 úlohy lze řešit stejným algoritmem a je potřeba provést duální transformaci, neboli jednu úlohu lze převést na druhou a opačně. Musíme si však dát pozor na případ, kdy při hledání průniku polorovin není dán bod uvnitř průniku. Pak problém může být neřešitelný (jelikož poloroviny se nepřekrývají). Pokud jsou vstupní data body, není problém neřešitelný, protože body vždy definují konvexní obálku (i když by se mohlo jednat o jediný bod nebo degenerovaný tvar).

Kolik bodů očekáváme na obálce: Při randomizované generaci bodů se obvykle velká většina bodů bude nacházet uvnitř obálky. Pokud tvoříme obálku v 2 dimenzionálním prostoru, je možné využít vlastnosti, že body nejvíce vlevo, vpravo, nahoře a dole budou na obálce. Díky těmto bodům můžeme tvořit regiony. Všechny body unvitř regionů se nacházejí uvnitř obálky a tyto body tak můžeme vyloučit. Ideálně pak algoritmus pro nalzení obálky pouštíme přes pár bodů. Zvýšením dimenze se efektivita tohoto principu snižuje. Této vlastnosti využívá například Quick Hull algoritmus.

Nalezení tvaru množiny bodů: Obal je pouze hrubým odhadem tvaru množiny bodů (chybí například informace o konkávnosti bodů). Například obaly množin bodu přípomínající G a O by byly nerozlišitelné. Pro přesnější informaci tvaru bodů lze využít alpha-shapes, které umožní zachovat informace o konkávnosti tvaru.

Existující metody

Gift Wraping: algoritmus použijeme pro zkonstruování konvexní obálky vyšší dimenze. Začíná se od nejnižšího bodu. Například 3. dimenzionální polyhedron se skládá ze stěn (obdobně facet ve vyšších dimenzích), které jsou spojeny 1 dimenzionálními hranami. Obecně algoritmus funguje tak, že nadje první facet pro nejnižší bod, následně hledáním do šířky nalezne další facets. Každá hrana, která je hranicí facet musí být také součástí hranice navazujícího facet. Postupným procházením n bodů množiny jsme schopni určit, jaké body definují následující facet pro aktualní hranu. Postupně tedy facets body obalujeme, dokud se nedostaneme k prvnímu inicializačnímu bodu. Efektivnost tohoto algoritmu spočívá v tom, že každou hranu projdeme pouze jednou. Složitost je O ($n\varphi_{d-1} + \varphi_{d-2} \lg \varphi_{d-2}$), kde φ_{d-1} je počet facets a φ_{d-2} počet hran konvexního obalu. Složitost nejhoršího případu může být $O(n^{(d/2)+1})$, právě když je konvexní obal příliš složitý.

Výhody: Velmi jednoduchý pro 2D prostory, jelikož hrany se stanou body a facety jsou hrany. Složitost ve 2D je pak O(nh), kde h je počet bodů na obálce. Z předchozí věty vyplývá, že alogritmus bude vhodný pro body, kde h bude velmi malé. Obvykle jednodušší implementace.

Nevýhody: Složitejší ve vyšších dimenzích. Pokud se h bude blížit n, tak nemusí být nejlepší volbou.

Graham Scan: Na začátku se zvolí bod ležící na obalu (některý z extremních bodů, většinou se jedná o ten nejlevější). Zbytek bodů se prochází dle úhlového pořadí. Začíná se s původním bodem a 1. bodem z úhlového pořadí. Následně se přídávají body v protisměru hodinových ručiček. Pokud úhel mezi zkoušeným bodem a poslední hranou je menší než 180 stupňů, tak tento bod přidáme do obálky. Pokud je naopak úhel větší, tak se řetězec vrcholů začínající od posledního bodu obálky vymaže, aby se zachovala konvexnost. Časová složitost je O(nlogn). Nejdražší operací je řazení bodů dle úhlu. Také se dá využít pro nalezení polygonu, kde se žádná hrana nepřekřižuje. Zde se postupuje tak, že se netestuje úhel, ale body jsou rovnou propojeny tak, jak jdou za sebou v úhlovém pořadí od počátečního bodu.

Výhody: Dobře funguje pro případy s velkým počtem bodů na obalu.

Nevýhody: Náročnější implementace. Může vyžadovat úpravy u degenerovaných tvarů (například když jsou body velmi blízko u sebe).

Informace byly čerpány z článku [1].

QuickHull: využívá principu rozděl a panuj. Výhodou algoritmu je, že jeho princip zůstává nezměněný napříč různými dimenzemi. D-dimenzionální obálka je reprezentována extrémními vrcholy (vrcholy, které tvoří hranici obálky) a (d-1)-dimenzionálními facets. Každá facet obsahuje seznam vrcholů, které ji tvoří, a seznam sousedních facets. Facets jsou sousední, pokud sdílejí stejnou ridge (například v 3D hranu). Ridge je průnik vrcholů 2 sousedních facets. Následně se využívají geometrické operace, které nadrovinou procházející d body rozdělí prostor na dvě části: jednu s body, které jsou uvnitř obálky (na "negativní" straně, neboli uvnitř facet), a druhou s body, které jsou nad facet (na "pozitivní" straně). Poté se vybere bod, který je nejdále od aktuální facet. Tento bod může být klíčovým vrcholem nové facet. Z tohoto bodu se zjistí, které facet jsou nad tímto bodem (tedy které jsou "viditelné" z tohoto bodu, tj. které jsou na "pozitivní" straně prostoru). Tyto facets se přidají k obálce. Tento proces se opakuje, přičemž každý nový bod rozšiřuje obálku tím, že vytváří nové facets, které tvoří její hranice.

Výhody: Pro 2D a 3D prostory poměrně jednoduchá implementace. Časová složitost v těchto prostorech bývá v průměru O(nlogn).

Nevýhody: Zjištění viditelných facets z bodu může být ve vyšších dimenzích složitá a výpočetně náročná operace. Nejhorší případy ve 2 a 3 dimenzionálních prostorech mají $O(n^2)$.

Informace byly čerpány z článků [2] [3] [4].

Zvolené řešení (Quick Hull ve 2D prostoru)

„ implementace je inspirována pseudokodem z [3]“

Popis Quick Hull alogritmu pro 2D prostory:

1. Najít extrémní body na x-ové souřadnici. Tyto body přidat do konvexního obalu. Pokud je více bodů na extrémní x-ové souřadnici, tak z nich vybrat ty s největší a nejmenší y-ovou souřadnicí.
2. Spojit extrémní body úsečkou, tím rozdělit prostor bodů na 2 části.
3. V každé části najít bod nacházející se nejdále od úsečky. Tento bod spojit s krajními body úsečky.
4. Body uvnitř trojúhelníku leží uvnitř konvexní obálky, tím je můžeme vyřadit z nadále procházených bodů.
5. Rekurzivně opakovat předchozí 2 kroky na vnější strany bodů trojúhelníků.
6. Končí se v případě, kdy už nejsou žádné body k zpracování.

Pseudo-kód v Javě:

```
Main function
    //Nageneruejeme si body tak aby splnovali predpoklady pro QuickHull (nesmi byt 3 body kolinearni, vice nebo
    // rovno 2 bodu a nesmime mit duplicitni body)
    List S = vytvorBody(); //Input
    List O = quickHull(S); //Output
end function

quickHull(List S) function
    List O; //obalka
    List seznamMnozinBodu;
    Pole extremy = najdiExtremniBody(S);
    O.add(extremy);
    S.remove(extremy);
    //rozpulime mnozinu useckou
    seznamMnozinBodu = rozdelMnozinuUseckou(S, extremy[0], extremy[1]);
    //Rekurzivne budeme hledat body do obalky
    najdiBodObalky(seznamMnozinBodu.getFirst(), extremy[0], extremy[1], O);
    najdiBodObalky(seznamMnozinBodu.getLast(), extremy[1], extremy[0], O);
    return O;
end function
```

najdiBodObalky(List body, Bod zacatek, Bod konec, List O) function

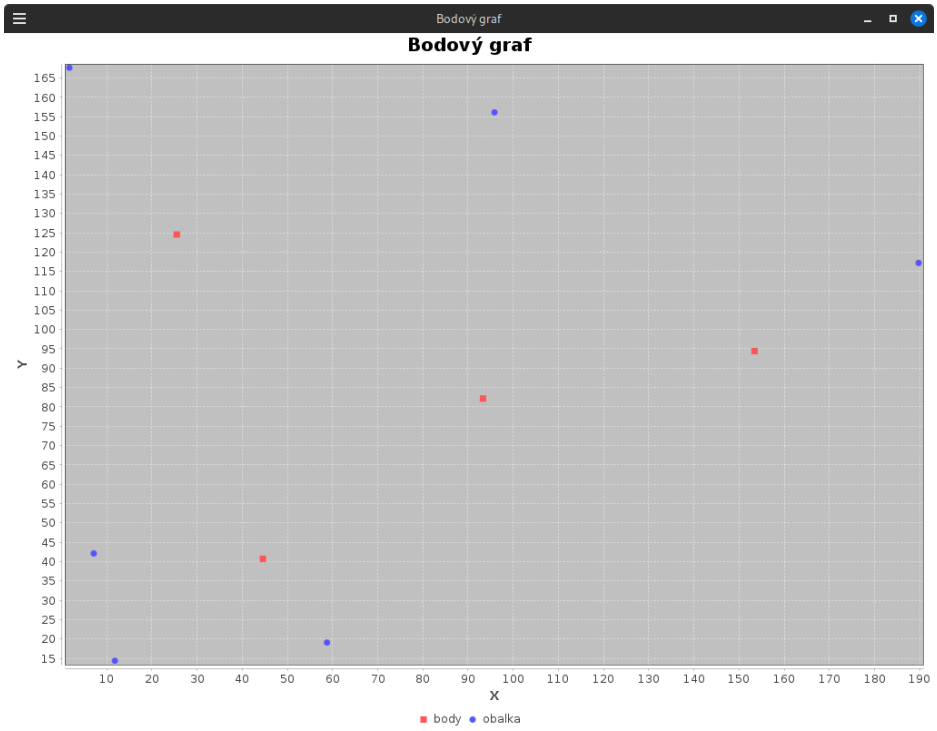
```
    // Pokud je predana mnozina prazdna → trivialni pripad rekurze
    if (S.isEmpty()) return;
    //najdeme bod nejdale od usecky
    Bod extremniBod = najdiBodNejdaleOdUsecky(body, zacatek, konec);
    //pridame bod do obalky
    O.add(extremniBod);
    //odstranime extremni bod z bodu
    body.remove(extremniBod);
     //ziskame 2 mnoziny bodu → mnozina bodu nachazejici se na pravo od orientovane
    // usecky zacatek, extremniBod
    // a mnozinu bodu nachazejici se na pravo od orientovane usecky extremniBod, konec
    // k jejich ziskani se v metode  spojBodyARozdelMnozinu() vyuziva metodu
    // rozdelMnozinuUseckou()
    mnoziny = spojBodyARozdelMnozinu(body, zacatek, konec, extremniBod);
    // spustime rekurzi
    najdiBodObalky(mnoziny.getFirst(), zacatek, extremniBod, O);
    najdiBodObalky(mnoziny.getLast(), extremniBod, konec, O);
end function
```

Poznámky k pseudo kódu:

Body nacházející se nejdále od úseček se vypočítávají tak, že se najde obecná rovnice přímky mezi body začátek a konec. Body se postupně dosazují do této rovnice a v každém kroku se pomocí vzorce pro vzdálenost aktualizuje maximální vzdálenost.

Rozdělení množiny bodů úsečkou na dvě množiny (levou a pravou) se počítá tak, že se pro každý bod spočítá jeho orientace vůči přímce. Orientace bodu se získá výpočtem determinantu pomocí Sarrusova pravidla. Orientace bodů menší než 0 značí, že body leží vpravo, větší než 0 vlevo a rovna 0 na přímce.

Ukázka výsledku běhu



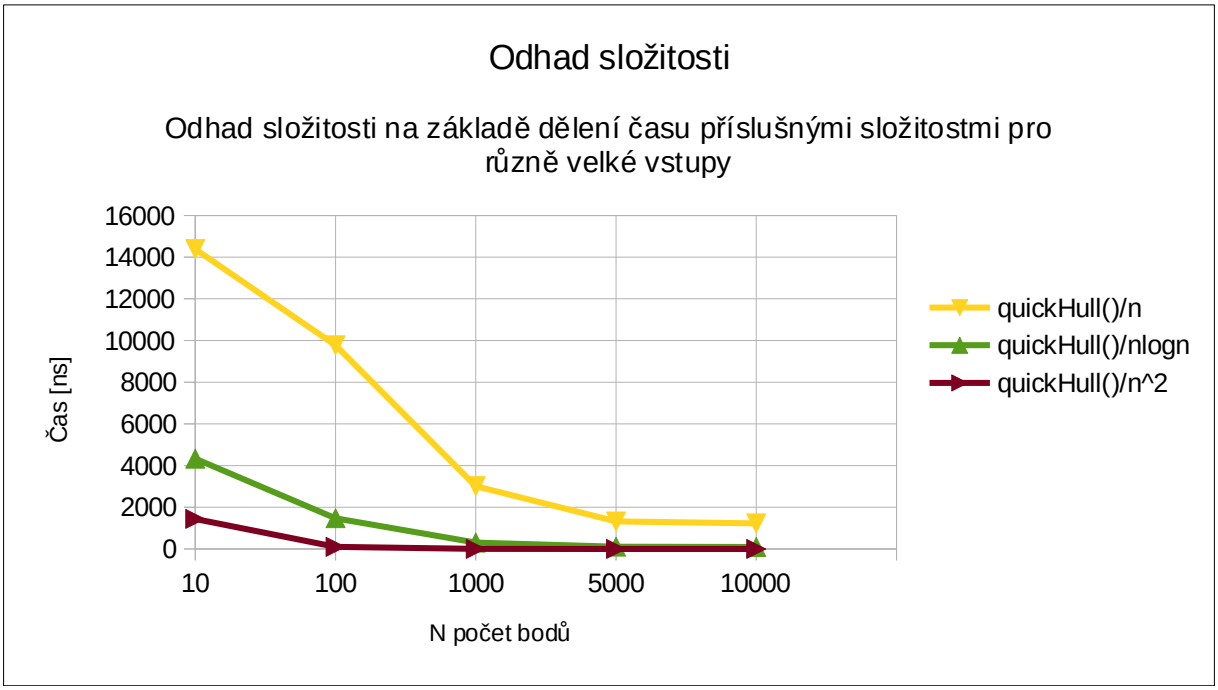
Experimenty a výsledky

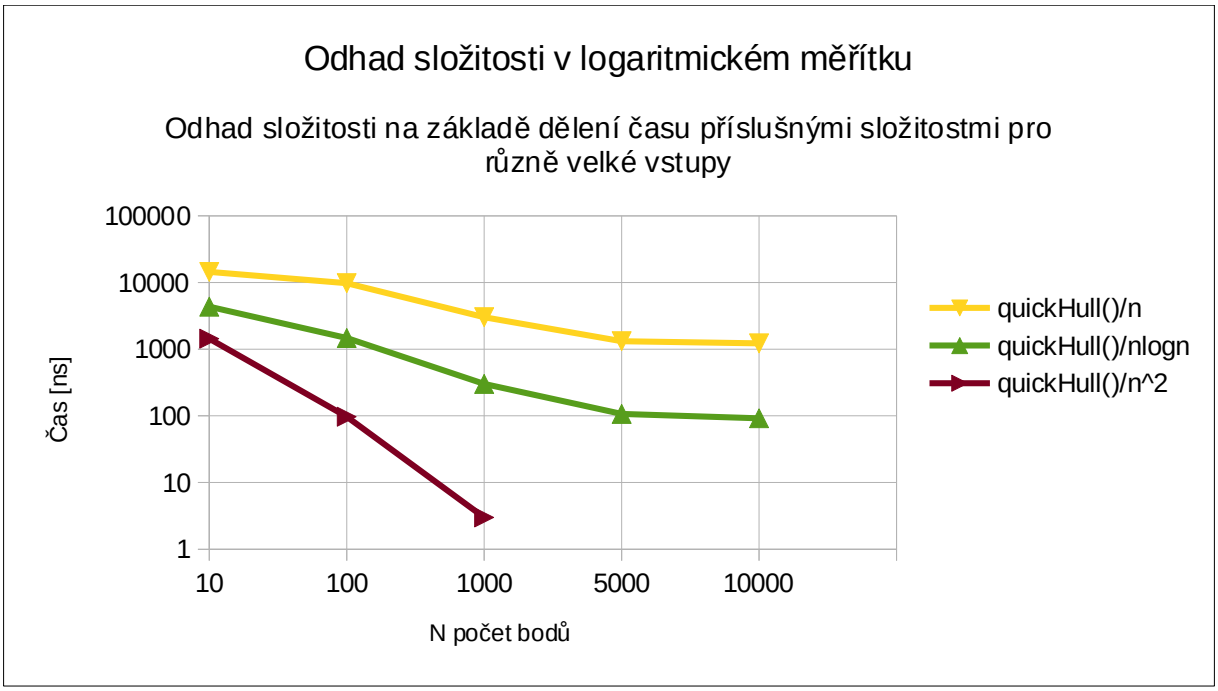
Implementovat algoritmus QuckHull jsem se rozhodl v jazyce java. Ve své práci jsem použil knihovny → java.util.ArrayList (pro dynamické seznamy), java.util.Random (pro generování náhodných čísel pro body), pro knotrolu výsledků → java.awt.* a javax.swing.* (pro GUI komponenty, jako je JFrame, BorderLayout atd.), JfreeChart pro vykreslování grafů (scatter plot) → ChartFactory, ChartPanel, JFreeChart, XYSeriesCollection, XYSeries (pro vytváření datových sérií bodů) a automaticky importované balíčky např java.lang (konrétně Math). Pro porovnání s ostatními algoritmy jsem si vypůjčil implementace algoritmů Jarvis scan [5] a Graham scan [6].

Odhad složitosti implementovaného řešení na náhodně generovaných prvků splňující předpoklady

n	QuickHull() v [ns]	quickHull()/O(n)	quickHull()/ O(nlogn)	quickHull()/ O(n^2)
10	143872	14387	4330	1438
100	977075	9770	1470	97
1000	3013499	3013	302	3
5000	6600000	1320	107	0 „ nula kvůli n které je moc velké → malé číslo/moc velké číslo„
10000	12236082	1223	92	0 „ nula kvůli n které je moc velké → malé číslo/moc velké číslo„

Z výchozí tabulky je vidět, že nejvíce se implementace blíží časové složitosti O(nlogn), jelikož se s postupně zvyšujícím n ustává hodnota vyjadřující podíl čas / časová složitost, což je vidět na grafech níže. Obecně by quickHull algoritmus (viz. Kapitola existující metody) měl fungovat ve složitosti O(nlogn).


Odhad složitosti
Odhad složitosti na základě dělení času příslušnými složitostmi pro různě velké vstupy
16000
14000
12000
10000
8000
6000
4000
2000
0
Čas [ns]
10
100
1000
5000
10000
N počet bodů
quickHull()/n
quickHull()/nlogn
quickHull()/n^2


Odhad složitosti v logaritmickém měřítku
Odhad složitosti na základě dělení času příslušnými složitostmi pro různě velké vstupy
100000
10000
1000
100
10
1
Čas [ns]
10
100
1000
5000
10000
N počet bodů
quickHull()/n
quickHull()/nlogn
quickHull()/n^2

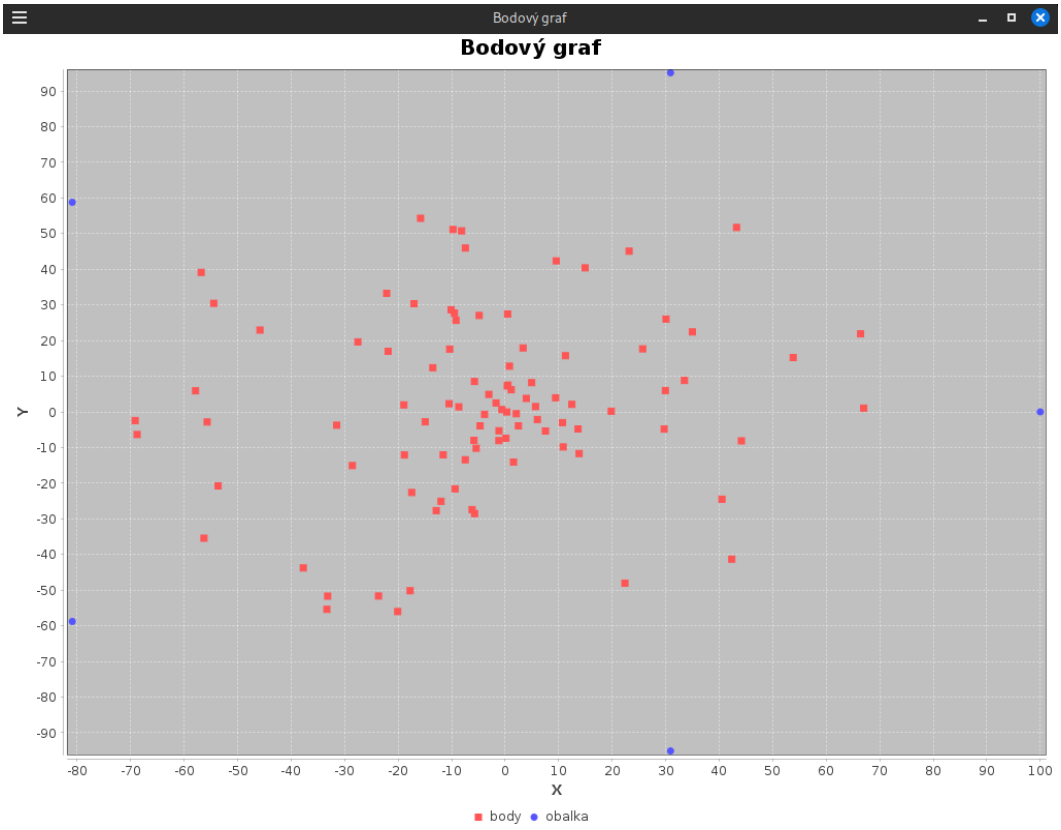
Porovnávací scénáře

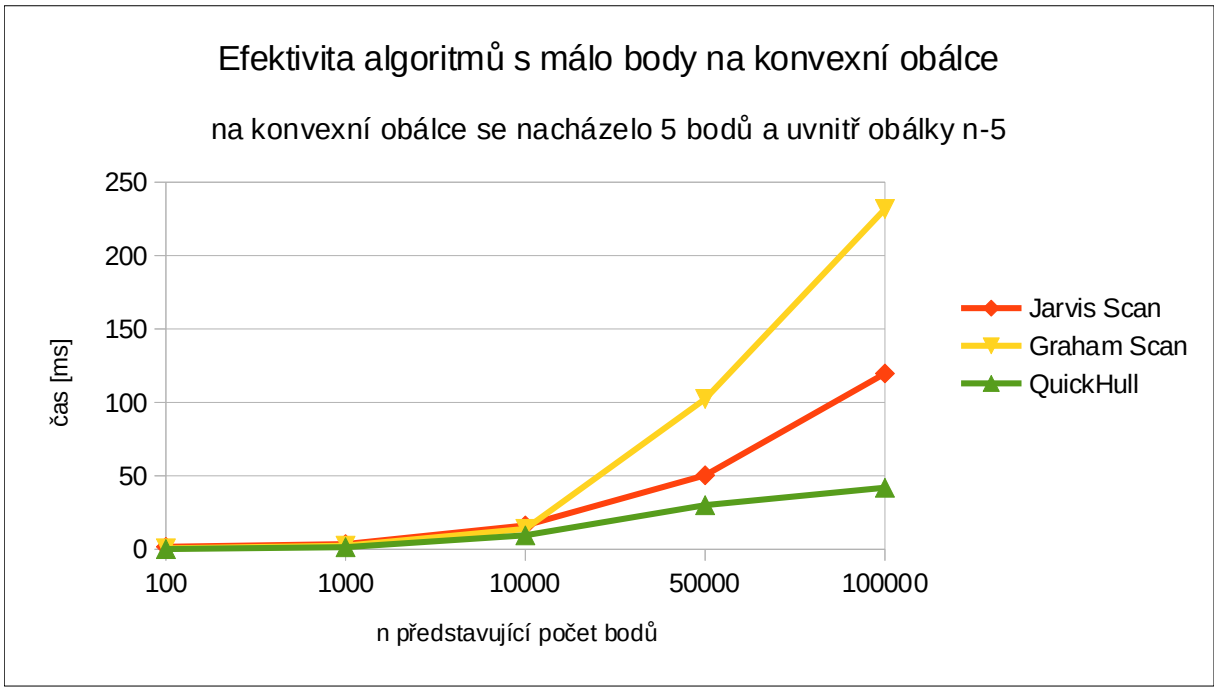
Většina bodů na konvexním obalu, mnoho bodů uvnitř a málo bodů na obalu, polovina na obalu a polovina uvnitř. Pro vytvoření scénářů využívám metodu vygenerujHull(), která generuje body na a do kružnice.

1. testovací případ → Málo bodů na obalu

n	Jarvis Scan	Graham Scan	QuickHull
100	1,69	0,65	0,18
1000	3,54	2,45	1,45
10000	16,27	13,9	9,51
50000	50,39	102,34	30,01
100000	119,74	231,71	41,93

Ukázka výstupu pro n = 100



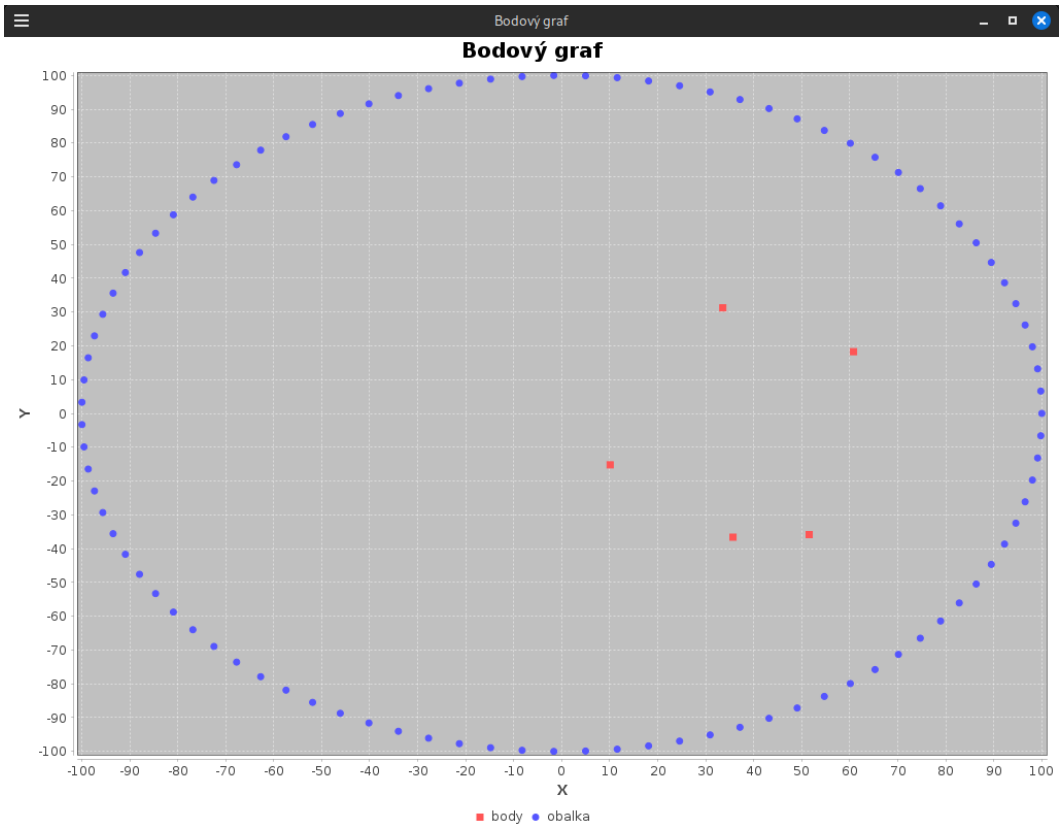


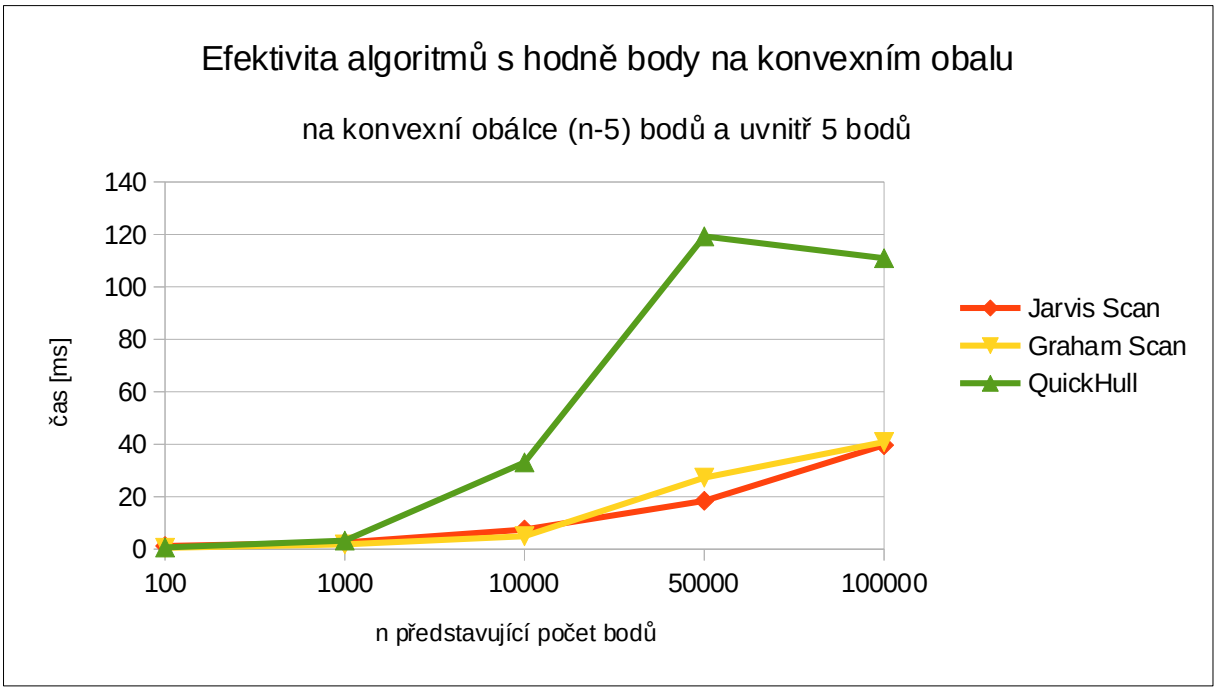
Komentář k výsledku grafu: V tomto případě si nejlépe vedl QuciHull. Což je vzhledem k obrázku logické. Drtivá většina bodů se nachází unitř obálky a pár bodů na obalu, což zapříčiní, že spousty bodů vyřadíme půlením prostorů bodů, když sestrojujeme přímky do nejvzdálenějších bodů. Tento trend by pokračoval i pro postupně zvyšující se n, což naznačuje i jen velmi jemný nárůst. Graham scan je spíše méně efektivní pro větší n. To je ale zarážející, jelikož jeho sort by měl mít časovou složitost O(nlogn) a měl by tak naopak fungovat mnohem lépe pro rostoucí n. Pravděpodobně by tak strmý nárůst křivky postupně ustal pro zvětšující se n. Nárůst Jarvis Scanu by dost pravděpodobně s rostoucím n nebyl nadále tak prudký, jelikož bodů na obalu je málo a jeho složitost je O(nh).

2. testovací případ → Hodně bodů na obalu a málo bodů uvnitř

n	Jarvis Scan	Graham Scan	QuickHull
100	1,21	0,5	0,65
1000	2,53	1,81	3,28
10000	7,48	4,95	33,06
50000	18,44	27,26	119,27
100000	39,64	40,82	110,95

ukázka výstupu pro n 100



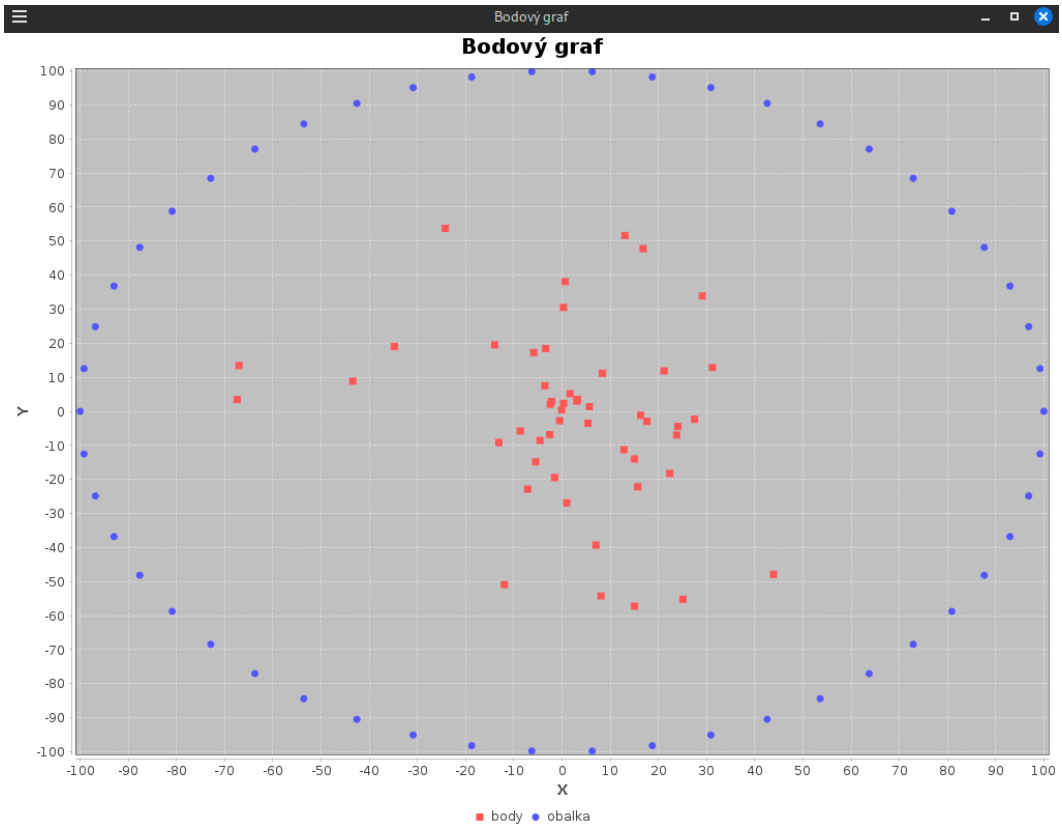


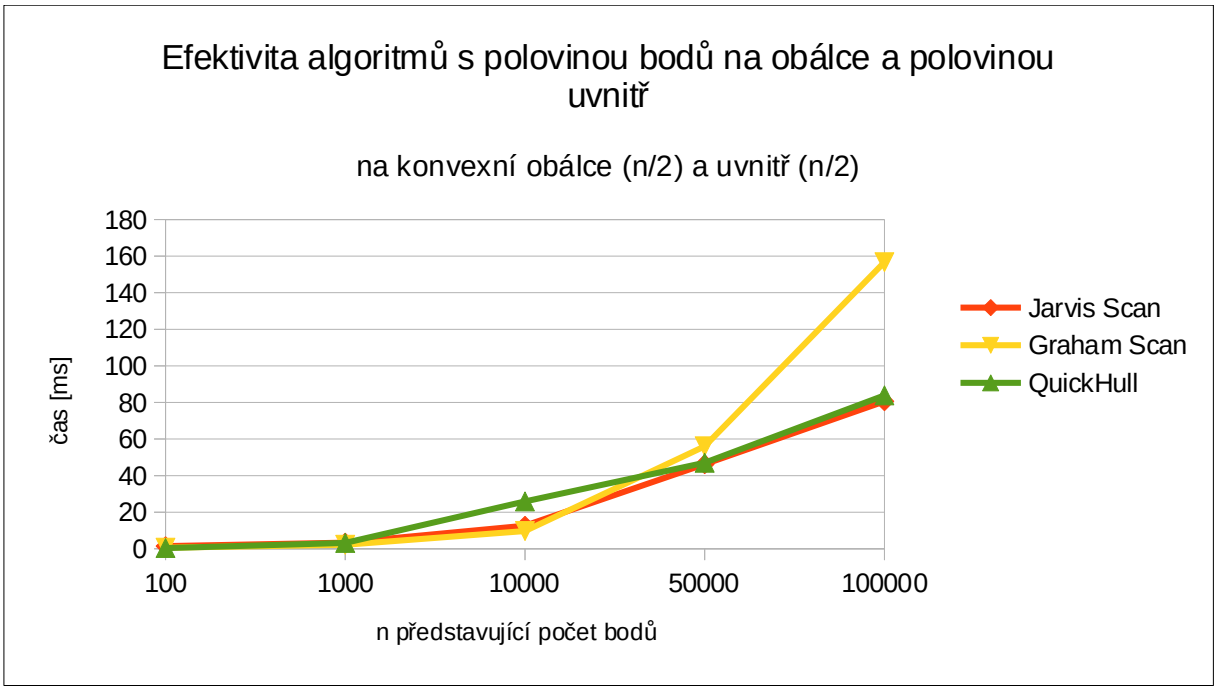
Komentář k výsledku grafu: Z výchozího grafu vychází nejhůře má implementace QuickHullem, čož by se dalo i očekávat vzhledem k tomu, že při půlení prostoru bodů se skoro vždy body rozdělí na polovinu. Zarážející je naopak, že z měření si velmi dobře vedl Jarvis Scan. Přitom jeho složitost je O(nh), kde h je počet bodů na obalu. Myslím si, že to je zapříčiněno hlavně zvolením malého n. Můžeme si totiž všimnout, že pro hodnoty n rovanjící se 50 000 a 100 000 začína poměrně strmě stoupat a myslím si, že tento trend by pokračoval s většími počty n. Nejlépe by si podle mě vedl Graham scan, což bych řekl, že je hlavně tím, že body jsou generovány do kruhu a jejich úhlové seřazení je tak z většiny uděláno předem. Z toho vyplývá, že je odstraněn hlavní problém algoritmu, což je řazení se složitostí O(nlogn). Dále na sebe navíc body dobře navazují a netvoří se tak případy, kdy by trojice bodů tvořila konkávní úhel.

3. testovací případ → Přibližně polovina bodů na obalu a přibližně polovina bodů uvnitř

n	Jarvis Scan	Graham Scan	QuickHull
100	1,44	0,6	0,39
1000	3,38	2,15	3,21
10000	12,58	9,71	25,89
50000	46,22	56,04	47,02
100000	80,63	156,71	83,88

ukázka výstupu pro n 100





Komentář k výsledku grafu: Zde si na první pohled vede nejlépe Jarvis Scan. Nicméně tento trend by v dlouhodobém horizontu parametru n spíše nevydržel s tím, jak by se zvětšoval i počet bodů na obálce. Pokud by však z celkového počtu n bodů na obálce bylo velmi málo bodů (například 3), pak by nejspíše trend běhu alogoritmu pokračoval v lineárním růstu. Naopak u quickHull by měl nastolený trend přetrvat. Jelikož jak jsou body generovány, tak by měla mít eliminace bodů půlením prostorů lineární nárůst. Graham Scan opět působí jako nejhorší. Znovu bych ale předpokládal, že jeho přínos by se projevil až s velkým počtem n.

Závěr

Závěrem mohu shrnout, že Jarvisův algoritmus (Gift Wrapping) bude vhodný pro malé datové sady nebo při malém počtu bodů na obalu (h) s malým n (počtem bodů). Časová složitost je O(nh). Využijeme ho hlavně pro 2D prostory. Princip spočívá v tom, že postupně obaluje body tak, že vybírá bod s nejmenším úhlem k aktuálnímu bodu na obalu. Hlavní myšlenka Graham scanu spočívá v úhlovém setřídění bodů dle nejnižšího bodu (O(nlogn) operace) a následné kontroly konvexity (O(n) operace). Je méně citlivý na h než Jarvis Scan a bude také vhodnější pro větší n díky složitosti O(nlogn). Quick Sort je na principu rozděl a panuj, kdy vybírá extrémy z množiny bodů a ty spojí úsečkou, tím získá další dvě množiny a tento postup rekurzivně opakuje. Časová složitost je v průměru O(nlogn). Hodí se pro velká n a nebývá problém ho rozšířit pro vyšší dimenze oproti Gift Wrapping. Na některé případy nemusí být vhodný, například hodně bodů na kružnici, nejhorší případ může mít složitost $O(n^2)$. Díky rekurzi může třeba dojít k přetečení.

Zdroje a citace

1 Skiena, S. S. (2008). *The Algorithm Design Manual* (2nd ed.). Springer. Retrieved from http://mimoza.marmara.edu.tr/~msakalli/cse706_12/SkienaTheAlgorithmDesignManual.pdf

2 Barber, C. B., Dobkin, D. P., & Huhdanpaa, H. (1996). The Quickhull Algorithm for Convex Hulls. *ACM Transactions on Mathematical Software (TOMS)*, 22(4), 469–483. Retrieved from https://dl.acm.org/doi/pdf/10.1145/235815.235821

3 Wikipedia contributors. (n.d.). Quickhull. *Wikipedia*. Retrieved November 29, 2024, from https://en.wikipedia.org/wiki/Quickhull

4 Martínek, P. (2010). *Tvorba konvexní obálky v Java3D* (Bachelor's thesis). University of West Bohemia. Retrieved from http://graphics.zcu.cz/files/86_BP_2010_Martinek_Petr.pdf

5 Raulinio, R. (n.d.). Jarvis Algorithm Implementation. *GitHub Repository*. Retrieved November 29, 2024, from https://github.com/rraulinio/JarvisAlgorithm

6 Sanfoundry. (n.d.). Java Program to Implement Graham Scan Algorithm to Find Convex Hull. Retrieved November 29, 2024, from https://www.sanfoundry.com/java-program-implement-graham-scan-algorithm-find-convex-hull/