Gymnázium Evolution Jižní Město

Eric Dusart

Polytop maximální dimenze a minimálního obvodu s vrcholy v dané množině bodů

Ročníková práce

Školitel práce:

Adam Klepáč

Školní rok: 2023/2024

Poděkování

Chtěl bych poděkovat Adamu Klepáčovi za odborné vedení mé ročníkové práce, cenné rady, tipy, inspiraci, ochotu a za všechny konzultační hodiny, které mi velice pomohly zkompletovat tuto práci.

Prohlášení

Prohlašuji, že jsem tuto ročníkovou práci vypracoval samostatně a s použitím uvedené literatury a pramenů.

V Praze dne 26. dubna 2024 Eric Dusart

Abstrakt

Cílem práce je najít způsob, jak nalézt polytop maximální dimenze s minimálním obvodem z množiny bodů v libovolné dimenzi. Tento matematický problém rozdělíme do tří variant: problém v 1D, ve 2D, a obecně v nD.

V 1D budeme hledat minimální vzdálenost mezi dvěma po sobě jdoucími čísly v seřazené množině bodů. Ve 2D úlohu převedeme na graf, ve kterém budeme hledat cyklus s minimální váhou a délkou tři. Lehce nám to zkomplikuje kontrolování, jestli polytop má maximální dimenzi. Proto musíme ověřit, zda body neleží na přímce. V nD budu hledat polytop tvořený $n + 1$ body. Zde bude ale mnohem těžší zkontrolovat, jestli polytop má maximální dimenzi.

V praktické části naprogramuji algoritmy z teoretické části, změřím kolik času potřebují v závislosti na počtu bodů, a nakonec je mezi sebou porovnám.

Klíčová slova: polytop, obvod, algoritmus

Abstract

The aim of this work is to determine how a polytope with maximal dimension but minimal perimeter in a set of points in $\mathbb{R}^n$. We will divide this problem into three variants: 1D, 2D, and general nD case.

In 1D, we will search for a minimal distance between two consecutive numbers from a sorted set of points. In 2D, we will create a graph with nodes representing points and weighted edges representing segments between each pair of points. We will then search for a cycle with minimal weight and length three. The problem gets slightly complicated because we need to check whether the dimension of the polytope is maximal, so we need to verify that the points do not lie on a line. In nD, I will search for a polytope formed by $n + 1$ points. However, it will be much harder to check whether the polytope has maximum dimension.

In the practical part, I will develop a program for each algorithm from the theoretical part, measure how much time they need depending on the number of points, and finally compare them with each other.

Keywords: polytope, perimeter, algorithm

Obsah

Použitá notace

Symbol	Význam
$\mathbb{R}^+$	$\{x \in \mathbb{R} \mid x \geq 0\}$.
$\mathbb{N}$	$\{x \in \mathbb{Z} \mid x \geq 1\}$.
n	$n \in \mathbb{N}$ značí dimenzi.
$\binom{n}{k}$	Kombinační číslo: $\frac{n!}{k!(n-k)!}$.
$p(a, b, c)$	Obvod trojúhelníku a, b, c.
$\Delta(a, b, c)$	Trojúhelník a, b, c.
$d(a, b)$	Vzdálenost bodu a od b.
$w(a, \ldots, n)$	Váha cesty z a do n.
min, max	Funkce minimum a maximum.
$\#V$	Velikost množiny V.
${}_n x_k$	n-tá souřadnice k-tého bodu

Základní definice a tvrzení

Definice 1 (Polytop). *Polytop* dimenze $n \in \mathbb{N}$ je uzavřená podmnožina $P \subseteq \mathbb{R}^n$ definovaná induktivně:

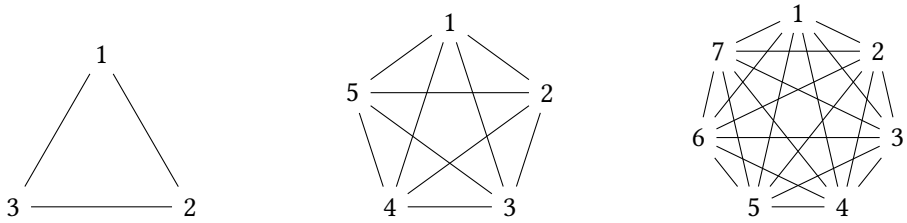
- Polytop dimenze 1 je úsečka.
- Polytop dimenze n je slepením polytopů dimenze $n-1$, jež spolu mohou sdílet stěny libovolné dimenze, kde *stěnou* polytopu rozumíme jeho libovolnou podmnožinu jsoucí rovněž polytopem. Zároveň neexistuje nadrovina (podprostor dimenze $n-1$), která by obsahovala všechny jeho vrcholy. [Ada24]

Definice 2 (Bod). Bod je uspořádaná n-tice $({}_1x, {}_2x, \ldots, {}_nx) \in \mathbb{R}^n$. Ve 2D budu používat značení $a = (a_x, a_y)$.

Definice 3 (Vzdálenost). Zobrazení $d : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^+$ nám určí vzdálenost dvou bodů $u, v \in \mathbb{R}^n$ podle předpisu $d(u,v) \coloneqq \sqrt{({}_1v - {}_1u)^2 + ({}_2v - {}_2u)^2 + \ldots + ({}_nv - {}_nu)^2}$.

Definice 4 (Ohodnocený graf). $G = (V, E, w)$ je ohodnocený graf, kde V je množina vrcholů, E je množina dvouprvkových podmnožin $E \subseteq \binom{V}{2}$ a w je libovolné zobrazení $E \to \mathbb{R}^+$, které hranám přiřazuje jejich váhu.

Definice 5 (Úplný ohodnocený graf). Úplný ohodnocený graf $G = (V, E, w)$ má každé dva vrcholy spojeny hranou, neboli $E = \binom{V}{2}$. Takový graf můžeme také zapsat jako $K_n \coloneqq (V, \binom{V}{2}, w)$.



Obrázek 1: Úplné grafy K_3, K_5 a K_7

Definice 6 (Podgraf). Graf $H = (V', E')$ je podgraf grafu $G = (V, E)$, pokud $V' \subseteq V$ a $E' \subseteq E \cap \binom{V'}{2}$.

Definice 7 (Cesta). Cestou v grafu nazveme posloupnost **různých** vrcholů $v_1, \ldots, v_n$, pokud $\forall i \in \{1, \ldots, n-1\}$ platí $\{v_i, v_{i+1}\} \in E$.

Definice 8 (Váha cesty). Pokud cestu tvoří posloupnost vrcholů $v_1, \ldots, v_n$, tak váha cesty je rovna

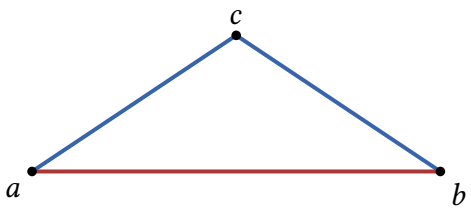
$$w(v_1, \ldots, v_n) = \sum_{i=1}^{n-1} w(v_i, v_{i+1}).$$

Definice 9 (Cyklus). Cyklus je posloupnost vrcholů $v_1, \ldots, v_n, v_1$, kde $v_1, \ldots, v_n$ je cesta a $\{v_1, v_n\}$ je hrana v množině hran E.

Definice 10 (Váha cyklu). Pokud cyklus tvoří posloupnost vrcholů $v_1, \ldots, v_n$, tak váha cyklu je rovna $w(v_1, \ldots, v_n, v_1) = d(v_1, v_n) + w(v_1, v_n)$.

Definice 11 (Soused). V grafu $G = (V, E, w)$ je vrchol u soused vrcholu v, pokud hrana $\{u, v\} \in E$.

Tvrzení 1 (Trojúhelníková nerovnost). *Trojúhelníková nerovnost říká, že pro každé tři různé body a, b, c platí $d(a, b) + d(c, b) \geq d(a, b)$, neboli vzdálenost mezi dvěma body je vždy menší nebo rovna součtu vzdáleností mezi těmito body a třetím bodem.*

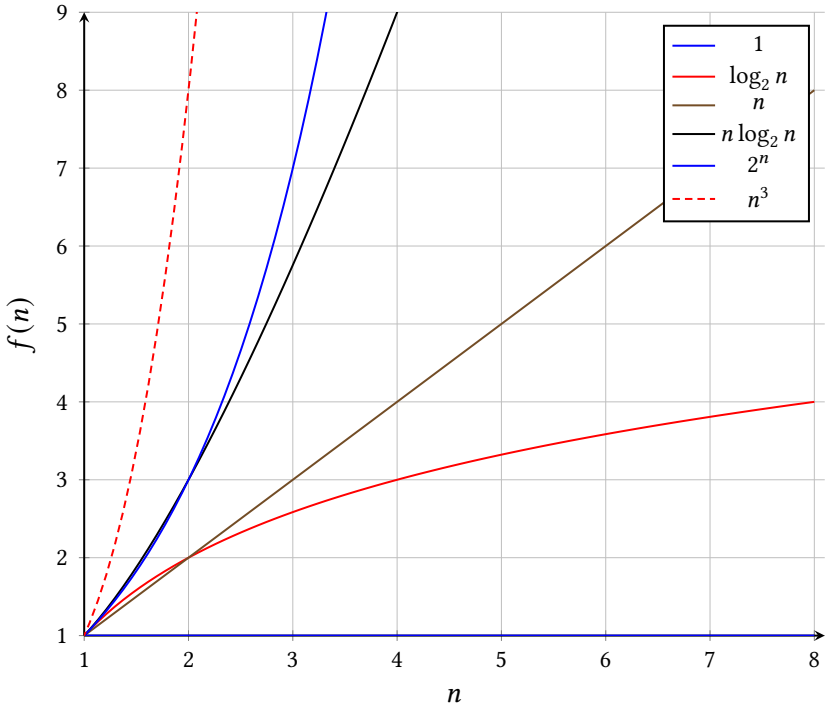


Obrázek 2: Trojúhelníková nerovnost

Definice 12 (Obvod polytopu). Definujeme zobrazení $p : \mathbb{R} \to \mathbb{R}^+$ podle následujícího předpisu:

$$p(\mathcal{P}) = \sum_{i=1}^{n} \sum_{j=i+1}^{n+1} d(x_i, x_j).$$

Definice 13 (Landauova notace). Vizte například [Bae19]. Velmi krátce řečeno: notace $O(\ldots)$ představuje počet kroků, které algoritmus musí udělat až na násobení konstantou nebo přičtení konstanty. O algoritmu můžeme říct, že má časovou složitost například $O(n^2)$, kde n je délka vstupu.



Obrázek 3: Grafy funkcí, které jsou častými případy časové náročnosti algoritmů.

Úvod

Hledání polytopu maximální dimenze s minimálním obvodem. To je problém, který budu v této práci zkoumat. Hlavní otázkou je, jak takový polytop najít v množině bodů, která je v prostoru jakékoliv dimenze. Dimenzi budu mít v celé práci označenou písmenem $n \in \mathbb{N}$.

Zadání problému není zas tak složité. Najít polytop maximální dimenze s minimálním obvodem. Může to znít složitě, ale je to pouze hledání nějakého počtu bodů, které jsou u sebe „v jistém smyslu“ nejblíž a tvoří objekt, který má maximální dimenzi. Například ve 3D, polytop maximální dimenze s minimálním obvodem je čtyřstěn (zvaný také trojboký jehlan).

Tyto polytopy budu hledat různými algoritmy, které by v průměru měly být lepší, než zkoušení všech možností. Algoritmy by měly vyřešit všechny případy (například, když body leží v jedné nadrovině), měly by skončit v rozumném čase v závislosti na počtu bodů a musí být korektní. Problém si rozdělím na 1D, 2D a nD. V 1D je problém velmi jednoduchý, jde pouze o množinu čísel, ve které najdeme nejkratší úsečku. Ve 2D budeme hledat trojúhelník s minimálním obvodem. Budeme muset ale zkontrolovat, že body opravdu tvoří trojúhelník, protože by se mohlo stát, že leží na přímce. V nD budu hledat polytopy maximální dimenze s minimálním obvodem a budu kontrolovat, jestli body, které tvoří polytop, neleží v nadrovině.

Algoritmy, na které přijdu, dokážu, že jsou korektní. To znamená, že skončí a jsou správné (dělají to, co chceme).

V praktické části tyto algoritmy naprogramuji v programovacím jazyce Python, který má výhodu, že není těžký na pochopení a je ideální na počítání různých věcí v matematice.

Část I

Teoretická část

Kapitola 1

Problém v 1D

Problém v 1D je velice jednoduchý. Vstupem bude množina čísel $V \subset \mathbb{R}$ a výstupem bude nejkratší úsečka $\{a, b\}$, kde $a, b \in V$. Nejprve množinu čísel seřadíme pomocí algoritmu Quicksort (vizte [Hoa62]) a poté vybereme minimální vzdálenost ze všech dvou po sobě jdoucích čísel z množiny V.

1.1 Algoritmus

1. Nejprve body seřadíme pomocí algoritmu Quicksort.
2. Projdeme všechny body $x_1 \dots x_n \in V$ a spočítáme všechny vzdálenosti mezi po sobě jdoucími body: $d(x_i, x_{i+1})$, kde $i \in \{1, ..., \#V - 1\}$. Tuto vzdálenost uložíme, pokud je menší, než ta doposud uložená.
3. Po tom, kdy projdeme všechny body, je uložená minimální vzdálenost řešením problému.

Tvrzení 2. *Algoritmus na hledání nejkratší úsečky je korektní.*

Důkaz. Algoritmus skončí, protože prochází konečnou množinu bodů a je korektní, protože spočítá všechny vzdálenosti po sobě jdoucích bodů a vybere tu minimální. □

Poznámka 1. Pseudokód je popis jednotlivých kroků v algoritmu s použitím základní logiky programovacích jazyků. Následuje algoritmus výše popsaný v pseudokódu.

Algoritmus 1: Algoritmus na hledání úsečky s minimální délkou.

input : množina čísel $V \subset \mathbb{R}$.
output: $a, b \in V, d(a, b)$

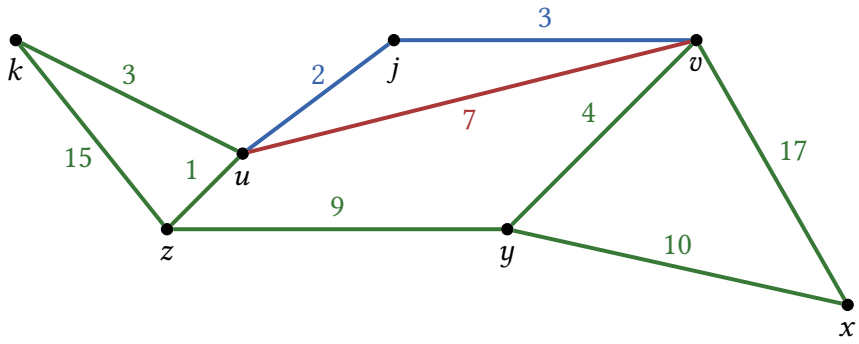
```
1 points ← quicksort(V);
2 shortest_distance ← ∞ ;
3 closest_points ← ∅ ;
4 for i ∈ {1, ..., #V − 1} do
5     if d(x_i, x_{i+1}) < shortest_distance then
6         shortest_distance ← d(x_i, x_{i+1}) ;
7         closest_points ← (x_i, x_{i+1}) ;
8 return closest_points, shortest_distance
```

Kapitola 2

Problém ve 2D

Ve 2D budu hledat trojúhelník s minimálním obvodem. Nejprve mě napadl algoritmus, který úlohu převede na grafovou úlohu. Funguje tak, že postupně vybere každou hranu, odebere ji z grafu a z jednoho konce hrany spustí Dijkstrův algoritmus [BS99], který najde nejkratší cestu do druhého konce hrany (například algoritmus odebere hranu $\{u, v\}$ a algoritmus spustí Dijkstrův algoritmus z bodu u do bodu v). Z trojúhelníkové nerovnosti víme, že taková cesta povede právě přes jeden bod. Trojúhelníková nerovnost v našem grafu platí, protože je převzat z roviny a ohodnocení hran jsou vzdálenosti mezi body. Kdyby náš graf nebyl převzatý z roviny, mohla by nastat situace na obrázku 2.1.

Pak jsem si uvědomil, že tento algoritmus je stejný jako ten, který vám představím v sekci 2.3. Dijkstrův algoritmus tím pádem nepoužiji, protože na tento problém je zbytečně komplikovaný. Pro zájemce je Dijkstrův algoritmus popsán v příloze A. Pro nový algoritmus je třeba dokázat tvrzení 3 o podobnosti trojúhelníků, díky kterému zjistíme, zda body neleží na přímce.



Obrázek 2.1: V obecném grafu nemusí platit trojúhelníková nerovnost.

2.1 Podobnost trojúhelníků

Velmi důležitou částí algoritmu je kontrola, zda body neleží na přímce, a jestli skutečně tvoří trojúhelník. K zkontrolování máme následující tvrzení.

Tvrzení 3 (Podobnost trojúhelníků a body na jedné přímce). *Jsou dány tři body ležící v* $\mathbb{R}^2$*:* $a = (a_x, a_y)$, $b = (b_x, b_y)$, $c = (c_x, c_y)$. *Tyto body jsou na přímce, pokud platí, že trojúhelníky* $\Delta((a_x, a_y), (b_x, a_y), (b_x, b_y))$ *a* $\Delta((a_x, a_y), (c_x, a_y), (c_x, c_y))$ *jsou podobné, proto platí rovnice:* $(c_y - a_y)(b_x - a_x) = (b_y - a_y)(c_x - a_x)$.

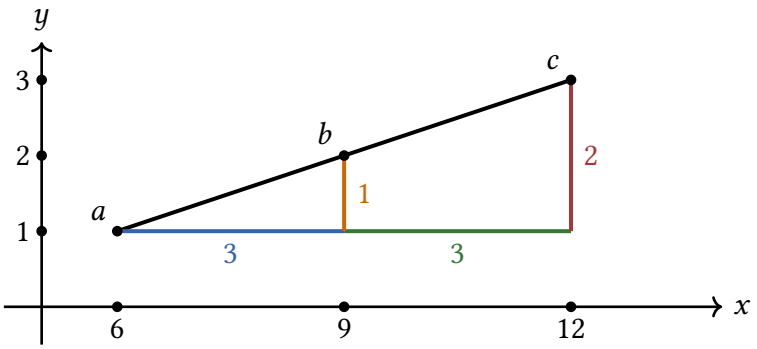
Důkaz. Předpokládejme, že existuje lineární funkce $f(x) = mx+k$, na které leží všechny tři body. Po dosazení bodů do rovnice nám vznikne soustava tří rovnic o dvou neznámých.

$$a_y = ma_x + k$$
$$b_y = mb_x + k$$
$$c_y = mc_x + k$$

Úpravami dostaneme

$$a_y - c_y = m(a_x - c_x)$$
$$b_y - a_y = m(b_x - a_x)$$

A tím pádem $m = (a_y - c_y)/(a_x - c_x)$ a zároveň $m = (b_y - a_y)/(b_x - a_x)$. Dále můžeme sestavit rovnici $(a_y - c_y)/(a_x - c_x) = (b_y - a_y)/(b_x - a_x)$. Nakonec můžeme rovnici upravit do tvaru $(a_y - c_y)(b_x - a_x) = (b_y - a_y)(a_x - c_x)$. Tímto je tvrzení 3 dokázáno. □



Obrázek 2.2: Příklad podobnosti trojúhelníků

2.2 Algoritmus na hledání nejkratší cesty

Vstupem je bod a a cílem je bod b. Dijkstrův algoritmus tuto úlohu řeší, ale v úplném grafu jde použít jiný algoritmus, který je stejně časově náročný. Můžeme vyzkoušet všechny cesty, které vedou právě přes jeden bod a uložit si tu s minimální váhou. V algoritmu 2 je tento algoritmus funkce *path*.

2.3 Algoritmus

Nechť V je množina bodů v rovině a pro každé dva body $u, v \in V$ označme $d(u, v)$ jejich vzájemnou vzdálenost. Množinu hran označíme $E = \binom{V}{2}$ a váhu, neboli ohodnocení, nám určuje zobrazení w dané předpisem $w(\{u, v\}) \coloneqq d(u, v)\ \forall (u, v) \in E$. Nyní můžeme definovat graf $G = (V, E, w)$. Tímto je příprava hotova. Následuje hledání cyklu délky tři s minimální váhou.

Náhodně vybereme jednu hranu $\{u, v\} \in E$ a odebereme ji z množiny hran E. V grafu bez hrany $\{u, v\}$ potřebujeme najít cestu s minimální váhou mezi body u a v. K tomuto použijeme algoritmus na hledání nejkratší cesty, který nám vrátí cestu délky

dva. Cesta povede právě přes jeden vrchol, protože z trojúhelníkové nerovnosti je jasné, že pokud by cesta vedla přes více vrcholů, byla by delší. Bude se skládat ze dvou hran: $\{u, j\}$ a $\{j, v\}$.

Teď musíme zkontrolovat, zda tato cesta s hranou $\{u, v\}$ opravdu tvoří trojúhelník, protože se může stát, že body u, j, v leží na jedné přímce. K tomu využijeme podobnost trojúhelníků, z které vyplývá, že body se nacházejí na přímce právě tehdy, pokud rovnice $(v_y - u_y)(j_x - u_x) = (j_y - u_y)(v_x - u_x)$ je pravdivá.

Poznámka 2 (jiný způsob). Předchozí způsob funguje perfektně pro počítače, ale pro člověka je v některých případech zbytečně komplikovaný. Tyto případy nastávají, když se body nacházejí na horizontálních nebo vertikálních přímkách. To jde poznat tak, že $u_y = j_y = v_y$ nebo $u_x = j_x = v_x$. V případě, že se body nacházejí na horizontální přímce, musíme odebrat delší hranu. To uděláme tak, že si body uspořádáme (a přejmenujeme) tak, že $a_x < b_x < c_x$, nebo a_y, b_y, c_y (podle toho, jestli jsou na horizontální nebo vertikální přímce), kde $\{a, b, c\} = \{u, j, v\}$. Z grafu pak musíme odebrat hranu $\{a, c\}$.

Pokud tyto body tvoří trojúhelník, pak tvoří cyklus délky tři. Bude-li tento cyklus bude mít váhu menší než ten, který jsme doposud našli, uložíme jej a vrátíme hranu $\{u, v\}$ do množiny hran E. Tento postup opakujeme, dokud nevyzkoušíme všechny hrany. Výsledkem bude trojúhelník s minimálním obvodem.

2.4 Důkaz algoritmu na hledání cyklu délky tři

Abychom mohli dokázat korektnost algoritmu, musíme dokázat, že algoritmus skončí a že je správný, to znamená, že dělá přesně, co chceme.

Tvrzení 4. *Algoritmus na hledání minimální cesty (v sekci 2.2) je korektní.*

Důkaz. Algoritmus je konečný, protože prochází konečnou množinu vrcholů, a je správný, protože ze všech cest z a do b vybere tu s minimální váhou. □

Tvrzení 5. *Algoritmus na hledání cyklu délky tři je korektní.*

Důkaz. Konečnost algoritmu je zřejmá; jediným cyklem v algoritmu je procházení všech stran. Jelikož je množina hran konečná a víme, že algoritmus na hledání cesty s minimální váhou je korektní, můžeme říci, že náš algoritmus skončí.

Správnost algoritmu dokážeme sporem. Budeme předpokládat, že existuje trojúhelník x, y, z, který má kratší obvod, než trojúhelník a, b, c, který našel algoritmus, neboli:

$$\exists\{x, y, z\} \subseteq V : p(x, y, z) < p(a, b, c) \mid a, b, c \leftarrow algoritmus.$$

Znamená to, že náš algoritmus vybral jednu stranu trojúhelníku s minimálním obvodem špatně, a tím vznikl trojúhelník s delším obvodem. Jelikož náš algoritmus prochází všechny hrany, tak tam nemohl vybrat špatnou hranu. Tím pádem špatnou hranu musel vybrat když vybíral zbylé dvě hrany. Ty ale vybral algoritmus na hledání cesty s minimální váhou, který je korektní. Tím vznikl spor a korektnost algoritmu je dokázána. □

2.5 Algoritmus v pseudokódu

Algoritmus 2: Algoritmus na hledání cyklu délky tři.

input : množina bodů V v rovině, kde každý bod je reprezentován jako dvojice souřadnic (v_x, v_y).
output: cyklus délky tři a jeho váha.

```
1  Function path(start, end):
2      min_path ← ∅;
3      for x ∈ V do
4          path ← w(start, x, end);
5          if w(min_path) > path then
6              min_path ← (start, x, end);
7      return min_path;
8  for u ∈ V do
9      for v ∈ V do
10         w({u, v}) ← d(u, v);
11 E ← (V choose 2);
12 G ← (V, E, w);
13 min_Δ ← ∅;
14 for {u, v} ∈ E do
15     E ← E \ {u, v};
16     {u, j, v} ← path(G, u, v);
17     if (v_y − u_y)(j_x − u_x) = (j_y − u_y)(v_x − u_x) then
18         if d(u, j) > d(u, v) then
19             E ← E \ {u, j};
20         else if d(j, v) > d(u, v) then
21             E ← E \ {v, j};
22         else
23             continue;
24     else if p(u, j, v) < p(min_Δ) then
25         min_Δ ← p(u, j, v);
26     E ← E ∪ {u, v};
27 return min_Δ, p(min_Δ);
```

Kapitola 3

Zobecnění na n dimenzí

3.1 Pomocné definice a tvrzení

Tyto definice nám umožní vyřešit problém zobecněný na n dimenzí.

Definice 14 (Nadrovina). Nadrovina je podprostor $\mathbb{R}^n$ dimenze $n-1$.

Definice 15 (Lineární kombinace). Bod $a_1x_1+\cdots+a_kx_k$, kde $a_1,\ldots,a_k \in \mathbb{R}$ je lineární kombinací bodů $x_1,\ldots,x_k \in \mathbb{R}^n$. [Jin05, s. 67]

Definice 16 (Lineární závislost). Množina bodů $x_1,\ldots,x_n \in \mathbb{R}$ se nazývá lineárně závislá, jestliže lze nějaký bod z této množiny vyjádřit jako lineární kombinaci ostatních bodů této množiny. [Jin05, s. 78]

Definice 17 (Afinní závislost). Body $x_1,\ldots,x_k \in \mathbb{R}^n$ jsou afinně závislé právě tehdy, když jejich rozdíly $x_2-x_1, x_3-x_1,\ldots,x_k-x_1$ jsou lineárně závislé. Geometricky to znamená, že body leží v jedné nadrovině. [Mat03, s. 4]

Tvrzení 6 (Determinant matice). *Determinant je zobrazení značené det, které posílá čtvercové matice na reálné číslo. Determinant jde interpretovat geometricky; lze chápat jako objem n-rozměrného rovnoběžnostěnu. Důležitá vlastnost determinantu je, že řádky nebo sloupce matice A jsou lineárně závislé právě tehdy, když* $det(A) = 0$. *[DJ19, s. 224; Mat03, s. 4] Pro hlubší pochopení determinantů vizte [Jin05, s. 164] nebo [DJ19, s. 187].*

3.2 Řešení problému v nD

Problém v n dimenzích bude komplikovanější a časově náročnější. Z množiny bodů $V = \{x_1,\ldots,x_k\} \in \mathbb{R}^n$ vytvoříme množinu všech $(n+1)$-prvkových podmnožin: $C = \binom{V}{n+1}$. Vybereme podmnožinu $\mathcal{P} = \{x_1,\ldots,x_{n+1}\} \in C$, jejíž body tvoří polytop s minimálním obvodem. Obvod polytopu spočítáme následovně:

Hlavní otázkou je, kolik hran opravdu tvoří obvod. (Například ve 3D tělesová úhlopříčka krychle netvoří obvod.) Je jich přesně $\binom{n+1}{2}$. Toto číslo pochází ze vzorce $\binom{n+1}{m+1}$ [Cox73, s. 120], který udává počet m-dimenzionálních podpolytopů v n-rozměrném polytopu ($\mathcal{P}_n$) za předpokladu, že $\mathcal{P}_n$ má minimální počet vrcholů. To je $n+1$. Když víme, že počet vnějších hran v polytopu je $\binom{n+1}{2}$, musíme zjistit, které hrany z celkové množiny hran to jsou. Jsou to všechny, protože náš polytop má přesně $n+1$ vrcholů. Proto můžeme místo $n+1$ v $\binom{n+1}{2}$ dosadit množinu vrcholů polytopu ($\mathcal{P} = \{x_1,\ldots,x_{n+1}\}$) a tím

dostaneme $\binom{\mathcal{P}}{2}$, což jsou všechny dvouprvkové podmnožiny z $\mathcal{P}$, neboli všechny hrany polytopu. Obvod tedy spočítáme následovně:

$$p(\mathcal{P}) = \sum_{i=1}^{n} \sum_{j=i+1}^{n+1} d(x_i, x_j).$$

Teď, když jsme vybrali polytop s minimálním obvodem, musíme ověřit, že jeho vrcholy opravdu tvoří polytop maximální dimenze (jako ve 2D, kde jsme ověřovali, jestli body neleží na přímce).

Polytop maximální dimenze se vyznačuje tím, že jeho body neleží ve stejné nadrovině. Z definice 17 víme, že pokud rozdíly bodů od jednoho fixního jsou lineárně závislé, tak body v jedné nadrovině leží. Z toho vyplývá, že chceme, aby body nebyly afinně závislé, neboli tvořily polytop maximální dimenze.

Poznámka 3. Značení ${}_nx_k$ znamená n-tá souřadnice k-tého bodu, například ${}_2x_3$ je druhá souřadnice třetího bodu, tudíž ${}_nx_k - {}_nx_1$ je rozdíl n-tých souřadnice bodu x_k a x_1.

Abychom zjistili, zda jsou body afinně závislé, musíme spočítat rozdíly bodů od jednoho fixního a zjistit, jestli jsou lineárně závislé. Od bodů $x_2, \ldots, x_{n+1} \in \mathbb{R}^n$ odečteme první bod x_1. Tím nám vzniknou rozdíly $x_2 - x_1, x_3 - x_1, \ldots, x_{n+1} - x_1$, které umístíme do matice A.

$$A = \begin{pmatrix} x_2 - x_1 \\ x_3 - x_1 \\ \vdots \\ x_{n+1} - x_1 \end{pmatrix} = \begin{pmatrix} {}_1x_2 - {}_1x_1 & {}_2x_2 - {}_2x_1 & \cdots & {}_nx_2 - {}_nx_1 \\ {}_1x_3 - {}_1x_1 & {}_2x_3 - {}_2x_1 & \cdots & {}_nx_3 - {}_nx_1 \\ \vdots & \vdots & \ddots & \vdots \\ {}_1x_{n+1} - {}_1x_1 & {}_2x_{n+1} - {}_2x_1 & \cdots & {}_nx_{n+1} - {}_nx_1 \end{pmatrix}$$

Teď, když máme matici, vypočítáme $det(A)$. Podle tvrzení 6, pokud vyjde determinant 0, znamená to, že pak rozdíly bodů jsou lineárně závislé. Z definice 17 ale víme, že pokud tyto rozdíly jsou lineárně závislé, pak body jsou afinně závislé, což znamená, že leží ve stejné nadrovině. Jestliže body leží ve stejné nadrovině, musíme tento polytop odebrat z množiny všech polytopů a zopakovat celý tenhle postup pro nový polytop s minimálním obvodem.

Tímto jsme vyřešili problém v n dimenzích, kde $n \in \mathbb{N}, n > 2$.

3.3 Algoritmus v nD

V této sekci vám představím algoritmus, který řeší problém v n dimenzích. Součástí algoritmu není algoritmus na počítání determinantu matice, protože existují různé algoritmy na jeho výpočet. V sekci 4.3, kde jsem algoritmus naprogramoval, používám knihovnu NumPy, která je optimalizovaná na matematické výpočty.

Algoritmus dostane množinu bodů V. Dimenzi n algoritmus zjistí podle počtu souřadnic jednoho bodu.

Algoritmus 3: Algoritmus na hledání polytopu maximální dimenze s minimálním obvodem.

input : $V = \{x_1, \ldots, x_k\} \in \mathbb{R}$.
output: body tvořící polytop maximální dimenze s minimálním obvodem.

Function $p(\mathcal{P})$**:**
    $perimeter \leftarrow \sum_{i=1}^{n} \sum_{j=i+1}^{n+1} d(x_i, x_j)$;
    return $perimeter$;
$S \leftarrow \binom{V}{n+1}$;
$C \leftarrow \emptyset$;
for $\mathcal{P} \in S$ **do**
    $C \leftarrow C \cup \{\mathcal{P}\}$;
$min_{\mathcal{P}} \leftarrow \emptyset$;
while $C \neq \emptyset$ **do**
    $\mathcal{P} \leftarrow X \in C$, takový, že $p(X)$ je minimální;
    $x_1, \ldots, x_{n+1} \leftarrow \mathcal{P}$;
    $A \leftarrow \begin{pmatrix} {}_1x_2 - {}_1x_1 & {}_2x_2 - {}_2x_1 & \cdots & {}_nx_2 - {}_nx_1 \\ {}_1x_3 - {}_1x_1 & {}_2x_3 - {}_2x_1 & \cdots & {}_nx_3 - {}_nx_1 \\ \vdots & \vdots & \ddots & \vdots \\ {}_1x_{n+1} - {}_1x_1 & {}_2x_{n+1} - {}_2x_1 & \cdots & {}_nx_{n+1} - {}_nx_1 \end{pmatrix}$;
    if $det(A) \neq 0$ **then**
        $min_{\mathcal{P}} \leftarrow \mathcal{P}$;
        return $min_{\mathcal{P}}, p(\mathcal{P})$;
    else
        $C \leftarrow C \setminus \{\mathcal{P}\}$;
return $\emptyset$;

3.4 Důkaz algoritmu v nD

Dokázat korektnost algoritmu 3 nebude těžké, protože algoritmus zkouší všechny možnosti a pak vybere polytop s minimálním obvodem.

Tvrzení 7. *Algoritmus 3 je korektní.*

Důkaz. Algoritmus skončí, protože prochází konečnou množinu C. Algoritmus je správný, protože ze seřazených polytopů podle jejich obvodu vybere ten s minimálním obvodem a zkontroluje, jestli neleží v jedné nadrovině. Pokud leží, odebere tento polytop ze seznamu seřazených polytopů a vybere nový polytop s minimálním obvodem. □

Část II

Praktická část

Kapitola 4

Programování algoritmů

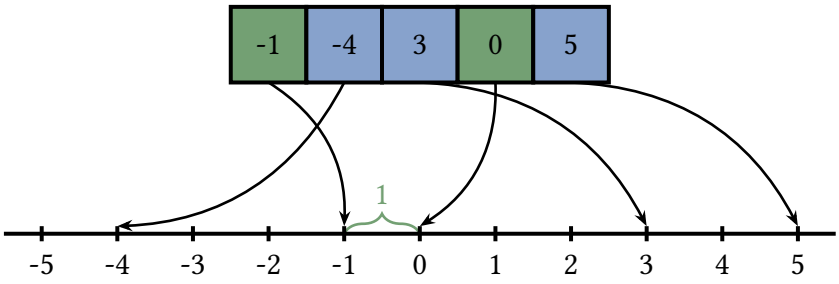
Moje praktická část bude programování problému, o kterém jsem psal v teoretické části. K programování použiji programovací jazyk Python, ve kterém už umím programovat. Python nabízí mnoho různých knihoven, které dokáží velmi ušetřit práci. Knihovna je sbírka funkcí a metod, které jsou předem napsané a můžeme je získat jako doplněk pro snadnější vývoj softwaru. Například ve své práci použiji knihovnu networkx, která je optimalizovaná pro práci s grafy.

Cílem programů je, aby fungovaly a aby nebyly moc těžké na pochopení. Programy například nebudou optimalizovány, aby měly co nejméně řádků. Komentáře jsou v kódu označeny # a tmavě zelenou barvou.

4.1 Program v 1D

Naprogramovat algoritmus 1, který řeší problém v dimenzi 1, je velmi jednoduchý. Využijeme funkci sort() z knihovny NumPy, která dokáže seřadit prvky v seznamu. Vstupem bude množina bodů a výstupem budou dva body, které tvoří nejkratší úsečku.

```
1 def algorithm(points:np.array):
2 points = np.sort(points, kind='quicksort') # Seřazení seznamu čísel
3   shortest = float('inf') # Proměnná pro úsečku s minimální délkou.
4   for i in range(len(points)-1): # Cyklus procházení bodů.
5     if abs(points[i] - points[i+1]) < shortest: # Pokud je úsečka menší.
6       shortest = points[i] - points[i+1] # Aktualizujeme délku nejkratší úsečky.
7       closest_points = points[i], points[i+1] # Aktualizujeme nejkratší úsečku.
8   return closest_points
```



Obrázek 4.1: Příklad, kdy algoritmus seřadil čísla a vybral nejkratší úsečku.

4.2 Program ve 2D

Ve 2D je problém trochu těžší, protože hledáme tři body, které tvoří trojúhelník s minimálním obvodem a musíme pouze kontrolovat, jestli nejsou na přímce. Tento program dostane jako vstup množinu bodů v $\mathbb{R}^2$ a výstupem budou tři body tvořící trojúhelník s minimálním obvodem.

```
from math import dist  # Importuje funkci dist z knihovny math pro výpočty.
from networkx import Graph, neighbors  # Importuje třídu Graph a funkci neighbors z knihovny networkx pro práci s grafy.
from itertools import combinations
import numpy as np  # Import dalších užitečných knihoven.

def find_path(G:Graph, start, end):
  min_path = None, None, None, float("inf")  # Proměnná pro nejkratší cestu s nekonečnou vzdáleností.
  for x in neighbors(G, start):  # Prochází sousedy počátečního bodu v grafu G.
    path = G.get_edge_data(start, x)["weight"] + G.get_edge_data(end, x)["weight"]  # Spočítá vzdálenost cesty přes aktuálního souseda.
    if float(min_path[3]) > path:  # Pokud je nová cesta kratší než dosavadní.
      min_path = start, x, end, path  # Aktualizuje nejkratší cestu.
  return min_path  # Vrátí nejkratší cestu.

def algorithm(V:np.array):
  G = Graph()  # Prázdný graf.
  subsets = combinations(V, 2) # Všechny dvouprvkové seznamy z V.
  for edge in subsets: # Pro hranu v subsets.
    G.add_weighted_edges_from([((edge[0][0], edge[0][1]), (edge[1][0], edge[1][1]), dist(edge[0], edge[1]))])  # Přidá hranu edge do grafu G s její váhou.

  min_triangle = None  # Proměnná pro trojúhelník s minimální váhou.
  min_triangle_weight = float("inf")  # Proměnná pro minimální váhu trojúhelníku.
  for edge in G.edges():
    edge_weight = G.get_edge_data(edge[0], edge[1])["weight"]  # Váha hrany edge.
    G.remove_edge(edge[0], edge[1])  # Odebere hranu z grafu.
    u, j, v, weight= find_path(G, edge[0], edge[1])  # Najde nejkratší cestu mezi body edge[0] a edge[1].
    if (v[1]-u[1])*(j[0]-u[0]) == (j[1]-u[1])*(v[0]-u[0]):  # Pokud body u, j, v leží na jedné přímce.
      if dist(u, j) > dist(u, v):  # Pokud je vzdálenost mezi body u a j větší než vzdálenost mezi body u a v.
        G.remove_edge(u, j)  # Odebere hranu mezi body u a j.
      elif dist(j, v) > dist(u, v):  # Pokud je vzdálenost mezi body j a v větší než vzdálenost mezi body u a v.
        G.remove_edge(v, j)  # Odebere hranu mezi body v a j.
      else:
        continue  # Pokračuje na další iteraci cyklu.
    elif weight + edge_weight < min_triangle_weight:  # Pokud je váha cesty plus váha hrany menší než váha nejmenšího trojúhelníku.
      min_triangle = (u, j, v)  # Aktualizuje nejmenší trojúhelník.
      min_triangle_weight = weight + edge_weight  # Aktualizuje váhu nejmenšího trojúhelníku.
    G.add_weighted_edges_from([(u, v, dist(u, v))])  # Přidá hranu u, v zpátky do grafu s její váhou.

  return min_triangle, min_triangle_weight  # Vrátí trojúhelník a jeho obvod.
```

4.3 Program v nD

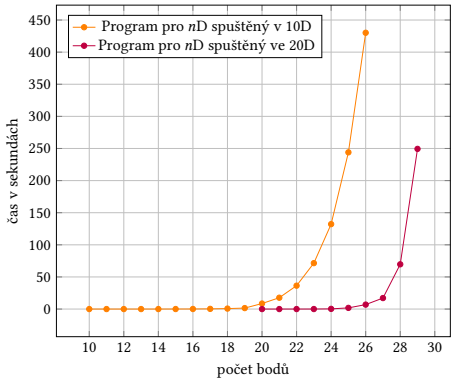
V nD program vytvoří všechny $(n+1)$-prvkové podmnožiny z množiny bodů V. Poté program spočítá obvod polytopu. Je-li menší, než ten uložený, ověří, že má maximální dimenzi. Pokud ano, uloží ho jako minimální.

```
num_points = len(points)   # Počet bodů.
n = len(points[0])     # Dimenze.
subsets = generate_subsets(points, n+1) # Vytvoří všechny (n+1)-prvkové
    podmnožiny.
polytopes = [] # Seznam polytopů.
for subset in subsets   # Pro podmnožinu (polytop).
  perimeter = 0
  edges = generate_subsets(subset, 2) # Množina všech 2-prvkových podmnožin.
  for edge in edges:
    perimeter += euclidean(edge[0], edge[1]) # Délka hrany.
  polytopes.append((subset, perimeter)) # Přidá hranu do seznamu polytopů.
sorted_polytopes = sorted(polytopes, key=lambda x: x[-1]) # Seřadí seznam
    polytopů podle jejich obvodu.
while len(sorted_polytopes) > 0:
  polytope, distance = sorted_polytopes[0] # Polytop a jeho obvod.
  x1 = polytope[0] # První bod polytopu.
  differences = [] # Seznam rozdílů.
  for x in polytope[1:]:
    difference = x - x1 # Rozdíl bodů.
    differences.append(difference) # Přidá rozdíl do seznamu rozdílů.
  matrix = np.vstack(differences) # Vytvoří matici z rozdílů.
  determinant = np.linalg.det(matrix) # Spočítá determinant.
  if determinant != 0: # Pokud je determinant nenulový, řešením je tento polytop.
    return polytope, distance
```

Kapitola 5

Porovnání programů

V kapitole 4 jsem programoval algoritmy, které jsem uvedl v teoretické části. Programy jsem zkoušel na náhodně generovaných bodech a také jsem je zkontroloval jiným algoritmem, který zkouší úplně všechny možnosti. Podle testů, které jsem provedl, by měly fungovat správně. Může se ale stát, že Python nebo nějaké knihovny, které používám, se změní a poté se bude muset program upravit.



Obrázek 5.1: Three simple graphs

Závěr

V této práci jsme hledali polytopy maximálních dimenzí s minimálním obvodem v různých dimenzích. Jako první čtenář mohl vidět základní definice a použitou notaci.

Problém jsme si rozdělili na 1D, kde jsme hledali nejkratší úsečku, pak jsme ve 2D hledali trojúhelník s minimálním obvodem a nakonec jsme problém zobecnili na n dimenzí. Problém ve 2D by nejspíš šel urychlit tím, že bychom zakazovali cesty, které jsou větší, než obvod nejmenšího trojúhelníku. Všechny algoritmy jsme dokázali, na druhou stranu, nebylo těžké je dokázat, protože zkoušely všechny možnosti. Aspoň algoritmus v nD je v průměru rychlejší, protože ověřujeme, jestli polytop tvoří maximální dimenzi až na konec počínajíce polytopem s nejmenším obvodem. V nejlepším případě se determinant matice, jehož výpočet je časově náročný, spočítá pouze jednou. V nejhorším případě se může stát, že pouze polytop, který má nejdelší obvod, má maximální dimenzi.

V praktické části jsem naprogramoval algoritmy v Pythonu. Kód je okomentovaný, ale už jsem ho znovu nevysvětloval, protože princip algoritmu jsem vysvětlil v teoretické části.

Literatura

[Hoa62] C. A. R. Hoare. „Quicksort“. In: *The Computer Journal* 5.1 (led. 1962), s. 10–16. ISSN: 0010-4620. DOI: 10.1093/comjnl/5.1.10. eprint: https://academic.oup.com/comjnl/article-pdf/5/1/10/1111445/050010.pdf. URL: https://doi.org/10.1093/comjnl/5.1.10 (cit. 09. 03. 2024).

[Cox73] H.S.M. Coxeter. *Regular Polytopes*. third. Dover Books on Advanced Mathematics. Dover Publications, 1973. ISBN: 978-0-486-61480-9. URL: https://books.google.cz/books?id=iWvXsVInpgMC (cit. 22. 04. 2024).

[BS99] Holger Benl a Helmut Schwichtenberg. „Formal Correctness Proofs of Functional Programs: Dijkstra's Algorithm, a Case Study“. In: *Computational Logic*. Ed. Ulrich Berger a Helmut Schwichtenberg. Berlin, Heidelberg: Springer, 1999, s. 113–126. ISBN: 978-3-642-58622-4. DOI: 10.1007/978-3-642-58622-4_4.

[Mat03] Jiří Matoušek. „Convexity“. In: *Introduction to Discrete Geometry*. Department of Applied Mathematics, Charles University Malostranské nám. 25, 118 00 Praha 1, Czech Republic: Springer, zář. 2003, s. 1–16. ISBN: 978-1-4613-0039-7. URL: https://kam.mff.cuni.cz/~matousek/kvg1-tb.pdf (cit. 21. 04. 2024).

[Jin05] Jindřich Bečvář. *Lineární algebra*. third. Sokolovská 83, 186 75 Praha 8: Matfyzpress, 2005. 436 s. ISBN: 80-86732-57-6. URL: https://www.karlin.mff.cuni.cz/~halas/becvar_-_linearni_algebra.pdf (cit. 20. 04. 2024).

[Bae19] Sammie Bae. „Big-O Notation“. In: *JavaScript Data Structures and Algorithms*. Berkeley, CA: Apress, 2019, s. 362. ISBN: 978-1-4842-3988-9. DOI: 10.1007/978-1-4842-3988-9_1. URL: https://doi.org/10.1007/978-1-4842-3988-9_1 (cit. 20. 04. 2024).

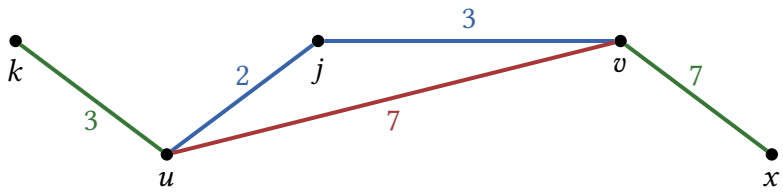
[DJ19] Dan Margalit a Joseph Rabinoff. *Interactive Linear Algebra*. Ve spol. s Larry Rolen. master version. Georgia Institute of Technology, 3. čvn. 2019. 455 s. URL: https://textbooks.math.gatech.edu/ila/ila.pdf (cit. 20. 04. 2024).

[Ada24] Adam Klepáč. *Definice polytopu*. 9. led. 2024.

Příloha A

Dijkstrův algoritmus

Dijkstrův Algoritmus, dokáže v ohodnoceném grafu najít cestu s minimální váhou mezi dvěma body v čase $O(\#V^2)$. [BS99]



Obrázek A.1: V obecném grafu nemusí platit trojúhelníková nerovnost.

A.1 Popis Dijkstrova algoritmu

Dijkstrův algoritmus počítá také nejkratší vzdálenosti z bodu a do všech ostatních. Záleží kdy je algoritmus ukončen. Jestli je ukončen, když aktuální bod je cílový, algoritmus najde cestu s minimální váhou z bodu a do bodu b. Pokud ale ukončen není, najde cesty s minimální váhou do všech bodů. Následuje popis Dijkstrova algoritmu.

1. Vytvoříme množinu všech nenavštívených bodů a vybereme startovní a cílový bod. Označíme všechny body nenavštívenými.

2. Každému bodu přiřadíme vzdálenost od počátečního bodu; prozatím na ∞. Vzdálenost počátečního bodu od sebe samého nastavíme na 0.

3. Přesuneme se na nenavštívený bod s minimální vzdáleností od počátečního bodu (poprvé to bude počáteční bod). Tento bod označíme jako aktuální a začneme kontrolovat jeho sousedy. Je-li součet vzdálenosti od počátečního bodu do aktuálního s váhou hrany vedoucí k sousedu menší než vzdálenost, kterou má u sebe soused uloženou, změníme ji. Je třeba myslet na to, že když přepíšeme vzdálenost souseda od startovního bodu, souseda neoznačujeme za navštíveného. Počáteční bod odebereme z množiny nenavštívených bodů, až zkontrolujeme všechny jeho sousedy.

4. Čtvrtý bod opakujeme, dokud nevybereme za aktuální bod ten cílový. V tomto okamžiku jsme našli nejkratší cestu. Pokud bychom chtěli získat nejkratší vzdálenosti z počátečního bodu do všech ostatních, čtvrtý bod budeme opakovat dokud nebude množina nenavštívených bodů prázdná.