INFORMATIKA

Minimální triangulace mnohoúhelníku (Úlohy z MO kategorie P, 50. část)

PAVEL TÖPFER

Matematicko-fyzikální fakulta UK, Praha

Dnešním jubilejním 50. dílem uzavíráme dlouhodobý seriál článků, ve kterém jsme vás postupně seznámili s vybranými soutěžními úlohami z Matematické olympiády kategorie P (programování). Zmíněných 50 článků vycházelo na stránkách našeho časopisu od roku 2000 až do současnosti, tedy po dobu plných 25 let.

Tentokrát si ukážeme jednu klasickou kombinatorickou úlohu, která byla zadána v celostátním kole 39. ročníku MO kategorie P ve školním roce 1989/90. Zabývá se tzv. triangulací mnohoúhelníku, což znamená rozdělení jeho plochy na samé trojúhelníky pomocí úhlopříček, které se vzájemně nikde nekříží. Jak uvidíme, efektivní řešení této úlohy bude založeno na myšlence dynamického programování, což je metoda velmi často využívaná nejen v různých soutěžních úlohách, ale i v programování obecně. V našem seriálu článků o úlohách z MO kategorie P jsme se s dynamickým programováním setkali již vícekrát, takže technika dynamického programování nyní celý seriál také symbolicky zakončí.

Jako obvykle se nejprve seznámíme s přesným zadáním úlohy. Původní soutěžní úlohu jen mírně formulačně upravíme, aniž bychom ovšem jakkoliv změnili její smysl.

* * * * * * * * * * * * *

Konvexní n-úhelník $A_1A_2 \ldots A_n$ je zadán kartézskými souřadnicemi jeho vrcholů v rovině. Navrhněte algoritmus, který určí minimální velikost jeho triangulace. Zdůvodněte správnost navrženého algoritmu.

Řekneme, že n-úhelník $A_1A_2\ldots A_n$, $n \geq 3$, je konvexní, jestliže všechny jeho vnitřní úhly jsou menší než 180°. *Úhlopříčkou* konvexního n-úhelníku budeme rozumět každou úsečku spojující dva různé vrcholy n-úhelníku, které spolu nesousedí na hranici (tj. nejsou spojeny hranou n-úhelníku). Z každého vrcholu n-úhelníku tedy vychází celkem $n-3$ úhlopříček. *Triangulací* konvexního n-úhelníku nazveme každý takový soubor navzájem se neprotínajících úhlopříček, které rozdělují plochu n-úhelníku na samé trojúhelníky. O součtu délek všech úhlopříček, které tvoří triangulaci konvexního n-úhelníku, budeme hovořit jako o *velikosti triangulace*.

* * * * * * * * * * * *

Pro vysvětlení úlohy můžeme doplnit, že každý konvexní n-úhelník s více než třemi vrcholy má více různých triangulací. Jejich počet nám v závislosti na hodnotě n určují Catalanova čísla, ale tím se zde zabývat nebudeme. Každá triangulace konvexního n-úhelníku je tvořena přesně $n-3$ úhlopříčkami, které rozdělují plochu n-úhelníku na $n-2$ trojúhelníků. Různé triangulace téhož n-úhelníku mohou mít různou velikost a my máme za úkol nalézt mezi nimi tu, jejíž velikost je nejmenší. Tuto hodnotu budeme nazývat *minimální triangulace*.

Naše rozbory úloh zpravidla začínáme primitivním řešením „hrubou silou“, tedy metodou zkoušení všech možností. Uděláme to také nyní. Minimální velikost triangulace daného mnohoúhelníku jistě můžeme určit tak, že postupně vygenerujeme všechny možné triangulace, spočítáme velikost každé z nich a z takto získaných hodnot vezmeme minimum. Vypadá to jednoduše, ale algoritmus na systematické procházení všech triangulací daného mnohoúhelníku není až tak úplně jednoduchý. Řešení nejsnáze zapíšeme ve tvaru rekurzivní funkce, která dostane ve svém vstupním parametru zkoumaný mnohoúhelník, v němž má zjistit minimální velikost triangulace. Pokud je to trojúhelník, funkce vrátí nulu, neboť triangulace trojúhelníku žádnou úhlopříčku neobsahuje. V opačném případě funkce umístí postupně všemi způsoby do mnohoúhelníku jednu úhlopříčku. Každou takovou volbou úhlopříčky se plocha mnohoúhelníku rozdělí na dva menší mnohoúhelníky, jejichž minimální triangulaci zjistíme rekurzivní aplikací téže funkce. Minimální velikost triangulace se zvolenou první úhlopříčkou je pak rovna součtu délky této úhlopříčky a minimálních triangulací obou vzniklých menších mnohoúhelníků. Výslednou funkční hodnotu získáme jako minimum z velikostí triangulací získaných pro všechny možné volby první úhlopříčky.

Popsaný postup je jistě správný, neboť vybírá minimum ze všech možností, jak lze mnohoúhelník triangulovat. Časově je ovšem velmi neefektivní, protože pro mnoho menších mnohoúhelníků se mnohokrát opakovaně počítá jejich minimální triangulace. Tatáž práce se tak provádí zbytečně vícekrát. Zkusíme tedy popsaný algoritmus trochu urychlit. Všimněte si, že každá triangulace mnohoúhelníku musí obsahovat nějaký *okrajový trojúhelník*, tedy takový trojúhelník, jehož dvě strany jsou zároveň sousedními stranami mnohoúhelníku. Již jsme si ukázali, že triangulace n-úhelníku je vždy tvořena $n-3$ úhlopříčkami a ty dělí plochu daného n-úhelníku na $n-2$ trojúhelníků. Celý n-úhelník má ovšem n stran, takže alespoň dva z těchto trojúhelníků musí obsahovat dvě strany původního n-úhelníku.

Odtud plyne, že také hledaná minimální triangulace musí obsahovat nějaký okrajový trojúhelník. Budeme tedy postupně uvažovat taková umístění první úhlopříčky, která z plochy n-úhelníku oddělí nějaký okrajový trojúhelník. Takových možností je pouze n. Pro každou z nich nyní stačí provést vždy jen jedno rekurzivní volání funkce na vzniklý mnohoúhelník s $n-1$ vrcholy.

Než si ukážeme funkci realizující popsaný algoritmus, musíme ještě vyřešit několik technických implementačních záležitostí. V zadání úlohy se vrcholy n-úhelníku označují čísly od 1 do n. Pro snadnější výpočty se nám ale bude hodit očíslovat si je v programu od 0 do $n-1$. Nebude to ničemu vadit, je to jen interní značení, na výstupu programu s čísly vrcholů vůbec nepracujeme. Budeme totiž často potřebovat určit k vrcholu A_i číslo toho vrcholu, který je vzdálen o d vrcholů po hranici mnohoúhelníku. Takový vrchol má často číslo $i+d$, ale musíme počítat také s tím, že za posledním vrcholem následuje opět první vrchol atd., uspořádání vrcholů je cyklické. Při očíslování vrcholů n-úhelníku od nuly získáme číslo vrcholu, který následuje jako v pořadí d-tý za vrcholem i, jednoduchým výpočtem s operací modulo $(i+d)\%n$.

Kartézské souřadnice vrcholů zadaného n-úhelníku si uložíme jednoduše do seznamu a délky n, v němž na indexu i budou uloženy souřadnice vrcholu A_i ve tvaru dvojice reálných čísel (tedy dvě proměnné typu float). Dále potřebujeme počítat délky jednotlivých úhlopříček. K tomu použijeme pomocnou funkci *delka(a, i, j)*, která určí vzdálenost vrcholů A_i, A_j. Spočítá ji jednoduše z kartézských souřadnic obou vrcholů jako vzdálenost dvou bodů v rovině pomocí Pythagorovy věty.

```
import math
```

```
with open('triangul.txt', "r") as f:
    n = int(f.readline())
    a = []
    for _ in range(n):
        a.append([float(_) for _ in f.readline().split()])

def delka(a, i, j):
    return math.sqrt((a[i][0] - a[j][0]) ** 2 + \
                     (a[i][1] - a[j][1]) ** 2)

def triangul1(n, a):
    if n == 3:
        return 0
    minim = math.inf
    for i in range(n):
        x = delka(a, (i-1)%n, (i+1)%n) + \
            triangul1(n-1, a[:i]+a[i+1:])
        if x < minim:
            minim = x
    return minim

print(triangul1(n, a))
```

Takto upravené řešení je sice podstatně jednodušší z hlediska programové realizace i rychlosti výpočtu, než naše původní varianta, jeho časová složitost je ovšem stále ještě obrovská. V původním n-úhelníku zkoušíme n možností umístění první úhlopříčky, pro každou z nich získáme $(n-1)$-úhelník s $n-1$ možnostmi, pro každou z nich $(n-2)$-úhelník s $n-2$ možnostmi atd., takže celkově projdeme téměř $n!$ způsobů, jak rozmístit úhlopříčky (ne úplně, protože u trojúhelníků již končíme). Z nich ovšem mnohé získané triangulace budou navzájem shodné a budou se lišit jenom tím, v jakém pořadí jsme jejich úhlopříčky umisťovali. Stále tedy provádíme mnoho shodných výpočtů opakovaně.

Se stejným problémem neefektivity primitivně navržené rekurzivní funkce jsme se v našich článcích již dříve vícekrát setkali a už jsme si také ukázali vhodné postupy, jak lze tento problém řešit. Připomeňme si tedy, že se nabízejí dvě základní cesty:

1) buď rekurzivní funkci doplníme o vhodnou globální datovou strukturu, v níž si budeme ukládat minimální velikost triangulace všech již spočítaných menších mnohoúhelníků; tyto hodnoty budeme následně používat namísto zbytečného provádění dalších rekurzivních volání funkce,

2) nebo rekurzi nahradíme cyklem postupujícím zdola nahoru, tzn. ve vhodném pořadí od menších mnohoúhelníků k větším, až dojdeme k celému původně zadanému mnohoúhelníku.

Ukážeme si oba tyto postupy. Začneme první uvedenou cestou. Rekurzivní funkce zůstane v téměř nezměněné podobě, pouze ji doplníme o evidenci minimálních velikostí triangulace již spočítaných mnohoúhelníků. Po stránce technické může mít tato evidence různou podobu. Mohl by to být obyčejný seznam dvojic údajů (mnohoúhelník, velikost jeho minimální triangulace), v němž bychom ale pak museli každý zkoumaný mnohoúhelník sekvenčně vyhledávat. Mohli bychom použít binární vyhledávací strom, v němž by se kód mnohoúhelníku používal jako klíč. My využijeme možností programovacího jazyka Python a evidenci si uložíme ve tvaru slovníku, kde klíčem bude mnohoúhelník a uloženou hodnotou jeho minimální triangulace. Ve všech případech budeme ovšem potřebovat nějaké jednoznačné zakódování mnohoúhelníku. V naší programové ukázce jsme zvolili seznam čísel jeho vrcholů v rostoucím pořadí. Z toho důvodu funkce dostala třetí parametr *c*.

```
mintr = {} # slovník s hodnotami minimální triangulace

def triangul2(n, a, c):
    if n == 3:
        return 0
    if tuple(c) in mintr:
        return mintr[tuple(c)]
    minim = math.inf
    for i in range(n):
        x = delka(a, (i-1)%n, (i+1)%n) + \
                  triangul2(n-1, a[:i]+a[i+1:],
                  c[:i]+c[i+1:])
        if x < minim:
            minim = x
    mintr[tuple(c)] = minim
    return minim

print(triangul2(n, a, list(range(n))))
```

Tento program počítá naprosto shodně jako předchozí řešení a liší se pouze v jedné velmi důležité maličkosti: nepočítá opakovaně to, co už jednou spočítal. Kdykoliv jsme v původním řešení prováděli rekurzivní volání funkce, zde nejprve zkontrolujeme ve slovníku, zda požadovanou hodnotu

neznáme z dřívějšího výpočtu. Pokud jsme ji již někdy dříve spočítali a uložili do slovníku, nebudeme provádět znovu pracné rekurzivní volání funkce, ale potřebnou hodnotu si jednoduše vyzvedneme ze slovníku. Naopak, jestliže hodnotu ještě neznáme a spočítáme ji rekurzivním voláním funkce, kromě přímého použití ve výpočtu tuto hodnotu navíc uložíme do slovníku pro případ, že by v budoucnu byla ještě zapotřebí. Ukládáme si preventivně všechny spočítané hodnoty, některé možná i zbytečně, ale my dopředu nevíme, které z nich budeme ještě potřebovat.

Uvedená úprava značným způsobem sníží časovou náročnost výpočtu, rekurzivní funkci na určení velikosti minimální triangulace budeme volat pro každý mnohoúhelník nejvýše jednou. Můžete se o tom ostatně sami snadno přesvědčit. Porovnáte-li rychlost výpočtu obou uvedených funkcí na běžném osobním počítači pro data velikosti $n \leq 10$, rozdíl v dobách výpočtu bude celkem zanedbatelný. Pro jakákoliv větší vstupní data je již ovšem rozdíl velmi citelný.

Závěrečné řešení založené na metodě dynamického programování přistupuje k problému z druhé strany. Místo rekurzivního vkládání úhlopříček do zadaného mnohoúhelníku a postupného dělení jeho plochy na menší mnohoúhelníky budeme postupovat iteračně v cyklu od nejmenších mnohoúhelníků k větším, ještě větším atd., až se dostaneme k celému původně zadanému mnohoúhelníku. Pro každý zkoumaný mnohoúhelník budeme určovat velikost jeho minimální triangulace. Tyto údaje si budeme ukládat do tabulky a budeme je využívat pro řešení větších mnohoúhelníků. Jak uvidíme později, můžeme se omezit pouze na ty mnohoúhelníky, jejichž všechny strany až na jednu jsou tvořeny stranami původního zadaného n-úhelníku a pouze jedna jejich strana je úhlopříčkou původního n-úhelníku.

Naší nejdůležitější datovou strukturou bude dvourozměrná tabulka t velikosti $(n + 1) \times n$. Index řádku k bude určovat počet vrcholů uvažovaného mnohoúhelníku, index sloupce i udává číslo vrcholu, od kterého je tento k-úhelník umístěn v původním zadaném n-úhelníku. Hodnoty uložené v tabulce budou představovat minimální velikost triangulace. Číslo $t[k][i]$ tedy udává velikost minimální triangulace k-úhelníku $A_iA_{(i+1)\%n} \ldots A_{(i+k-1)\%n}$. Řádky s indexy 0, 1, 2 a 3 budou zjevně celé nulové, protože žádný jednoúhelník ani dvojúhelník neexistuje a všechny trojúhelníky mají triangulaci rovnou 0. Dále budeme tabulku t vyplňovat postupně po řádcích, počínaje řádkem číslo 4. Při výpočtu $t[k][i]$ již budeme znát všechny hodnoty na řádcích s číslem menším než k.

Každá triangulace uvažovaného k-úhelníku $A_iA_{(i+1)\%n} \ldots A_{(i+k-1)\%n}$ obsahuje jistě nějaký trojúhelník se stranou $A_iA_{(i+k-1)\%n}$. Třetím vrcho-

lem tohoto trojúhelníku je některý z vrcholů $A_{(i+1)\%n} \dots A_{(i+k-2)\%n}$. Můžeme tedy vyzkoušet všechny takové volby vrcholu j pro j od $(i+1)\%n$ do $(i+k-2)\%n$ včetně, pro každou z nich spočítat velikost minimální triangulace, z těchto hodnot vybrat tu nejmenší a vložit ji do $t[k][i]$. Minimální velikost triangulace pro zvolené j je rovna součtu délek úhlopříček A_iA_j, $A_{(i+k-1)\%n}A_j$ a hodnot minimální triangulace obou menších mnohoúhelníků, které vzniknou vložením těchto dvou úhlopříček. Oba tyto mnohoúhelníky mají menší počet vrcholů než k a jejich minimální triangulaci proto už známe, je uložena v tabulce t na některém z předchozích řádků.

Poznamenejme ještě, že pro volbu $j = (i+1)\%n$ nebo $j = (i+k-2)\%n$ vzniká trochu odlišná situace a musíme ji správně ošetřit (máme jen jednu úhlopříčku, která oddělí jeden menší mnohoúhelník – viz program). Po zaplnění tabulky t až do řádku číslo n budou všechny hodnoty v řádku n shodné a budou udávat hledaný výsledek.

```
def triangul3(n, a):
    t = [[0]*n for _ in range(n+1)]
    for k in range(4, n+1):
        for i in range(n):
            t[k][i] = min(delka(a, i, (i+k-2)%n) + t[k-1][i],
                          delka(a, (i+1)%n, (i+k-1)%n) + \
                          t[k-1][(i+1)%n])
            j = (i+2)%n
            while j != (i+k-2)%n:
                x = delka(a, i, j) + delka(a, (i+k-1)%n, j) + \
                    t[(j-i+1)%n][i] + t[(i+k-j)%n][j]
                if x < t[k][i]:
                    t[k][i] = x
                j = (j+1)%n
    return t[n][0]

print(triangul3(n, a))
```

Asymptotická časová složitost závěrečného řešení úlohy dynamickým programováním je pouze $O(n^3)$. Postupně počítáme $O(n^2)$ hodnot tabulky t a každou z nich zjistíme v čase $O(n)$. Opět se můžete sami snadno přesvědčit, že výpočet této funkce je dostatečně rychlý i pro velmi rozsáhlá vstupní data.