Problem gnilih pomaranč, Janez Vsezna

OSNUTEK

# **Povzetek \*\*\* Končno – priporočam, da je ta razdelek VEDNO že končen\*\*\***

*V prispevku so predstavljeni trije problemi in sicer poiskati največjo vsoto podseznama, največji zmnožek podseznama in koliko časa bi potrebovali za izračun najmanjše poti med dvema ogljiščema v zazankanem drevesu. Najprej si bomo ogledali način predstavitve vhodnih podatkov problema ter s pomočjo primera predstavili, zakaj pri posameznem problemu točno gre. Posamezen algoritem prikažemo še na primeru. Prispevek prvih dveh problemov zaključimo s prikazom kode rešitve.*

# **Problem največje vsote in največjega produkta podseznama**

V knjigi (Erickson, 2019) na strani 124, naloga 3., je predstavljen problem največje vsote podseznama in največjega produkta podseznama. Poglejmo, zakaj gre.

*Recimo, da imamo dan seznam A[1 ... n] števil, ki so lahko pozitivna, negativna ali nič. Števila niso nujno cela števila.*

1. *Opišite in analizirajte algoritem, ki najde največjo vsoto elementov v podseznamu seznama A[1 ... n]*
2. *Opišite in analizirajte algoritem, ki najde največji zmnožek elementov v podseznamu seznama A[1 ... n]*

*Primer: če imamo dan seznam števil [−6,12,−7,0,14,−7,5], bi moral prvi algoritem vrniti 19, drugi pa 504.*

*A black rectangular object with numbers and a square in the middle

Description automatically generated*

*Če pa imamo dan seznam [-306], bi moral prvi algoritem vrniti 0, drugi pa 1. (Prazen interval je še vedno interval!) Zaradi analize predpostavite, da primerjanje, seštevanje, ali množenje katerega koli para števil traja O(1) časa.*

# **Največja vsota podseznama**

## **Naivni način**

Tega problema bi se lahko lotili tako, da bi poiskali vse možne podsezname danega seznama in zanje izračunali vsoto. Na koncu bi vrnili največjo izmed poračunanih.

Recimo, da smo na mestu i. Potem pogledamo vse podsezname od i naprej, kot kaže slika. Na vsakem podseznamu poračunamo vsoto, ki jo primerjamo s prej dobljeno vsoto.

A diagram of a mathematical system

Description automatically generated with medium confidence

Na sliki je prav tako zapisano od kod do kod gre posamezna spremenljivka, potrebna za izvedbo tega algoritma.

Ta algoritem je časovno zelo potraten . Da se ga pa narediti precej hitrejšega. S tem se je ukvarjal Joseph Born Kadane, ki je ugotovil učinkovit algoritem. Ta se imenuje Kadane's Algorithm in sodi v poglavje dinamičnega programiranja.

## **Kadanov algoritem**

Ideja je imeti dve spremenljivki. Ena, ki hrani največjo vsoto podseznama do trenutnega indeksa (recimo ji trenutnaVsota) in drugo, ki hrani največjo vsoto podseznamov najdenih do sedaj (recimo ji maxVsota). Torej trenutnaVsota opredelimo kot maksimalno vsoto tistih podzaporedij, ki se končajo na indeksu, kjer smo. Ko smo na koraku i, vemo kakšna je trenutnaVsota podzaporedij, ki se končajo na (i-1) mestu. Ko pa gledamo i-ti element, novo tranutnaVsota naračunamo tako, da:

* Če je element pozitiven, bo trenutnaVsota stara povečana za element na i-tem mestu.
* Če je element negativen, pa imamo dve možnosti:
  + Element se nam morda splača vzeti, ker bo služil kot 'most' do nadaljnjih novih maksimalnih trenutnihVsot. To se splača vzeti, če stara trenutnaVsota s to prišteto negativno vrednostjo ne pade pod 0.
  + Če povečana vsota pade pod 0, pa se nam elementa ne splača vzeti. Bolje je vzeti podzaporedje na (i+1) mestu, zato trenutnaVsota nastavimo na 0.

Hkrati preverjamo tudi največjo vsoto do sedaj (maxVsoto), ki je lahko sedaj trenutnaVsota ali pa ostane maxVsota od prej.

Razumevanje zgoraj opisanega si lahko morda pomagamo s spodnjo skico.

A whiteboard with writing on it

Description automatically generated

## **Uporaba na primeru**

Recimo, da imamo dan seznam [-3,10,-9,80,-75,3,-10,20,41,3,-7,81,203,17,-306,-33].

Rešitev bomo pokazali na določenih indeksih, držali pa se bomo poimenovanj opisanih zgoraj.

Na začetku nastavimo trenutnaVsota in maxVsota na 0.

* i = 0:
  + - * trenutnaVsota = max(-3, 0 + (-3)) = -3
      * maxVsota = max(0, -3) = 0
* i = 1:
  + - * trenutnaVsota = max(10, 10 + (-3)) = 10
      * maxVsota = max(0, 10) = 10
* i = 6:
  + - * trenutnaVsota = max(-10, -10 + 9) = -1
      * maxVsota = max(81, -1) = 81
* i = 11:
  + - * trenutnaVsota = max(81, 81 + 57) = 138
      * maxVsota = max(81, 138) = 138
* i = 12:
  + - * trenutnaVsota = max(203, 203 + 138) = 341
      * maxVsota = max(138, 341) = 341
* i = 13:
  + - * trenutnaVsota = max(17, 17 + 341) = 358
      * maxVsota = max(341, 358) = 358

Kar je z zeleno obrvano prikazuje enako število. Enako velja za modor, rumeno in vijolično obravna števila.

Ko postopek nadaljujemo do konca, dobimo, da je največja vsota enaka 358, njej pripadajoč podseznam pa je [20,41,3,-7,81,203,17].

## **Analiza časovne in prostorske zahtevnosti Kadanovega algoritma**

* Velikost problema: velikost seznama
* Karakteristična operacija: seštevanje in primerjanje velikosti
* Časovna zahtevnost: , ker iteriramo samo enkrat čez seznam
* Prostorska zahtevnost: , ker uporabljamo samo dve spremenljivki in nič dodatnega prostora npr. dodatnega seznama.

## **Programska rešitev Kadanovega algoritma**

A computer code with text

Description automatically generated with medium confidence

# **Največjegi produkt podseznama**

## **Naivni način**

Ideja problema je enaka kot za naivni način pri vsoti. Prav tako je tudi enaka časovna zahtevnost, zato se bomo raje osredotočili na uporabo Kadanovega algoritma na tem primeru.

## **Ideja algoritma**

Na mestu i imamo od prej poračunan največji in najmanjši produkt podseznama. Sedaj imamo dve možnosti za nov največji produkt. Lahko je to vrednost na mestu i ali pa je vrednost prejšnjega podseznama z največjim zmnožkom pomnoženega, z vrednostjo na mestu i. To označujemo z lokalni\_max. Enak razmislek je za najmanjšo vrednost produkta podseznama, ki ga označujemo z lokalni\_min. Pri seštevanju nimamo 'problema' s predznaki števila, tukaj pa lahko naletimo na tri možnosti, v primeru, da je število na mestu i negativno in sicer:

* Naj bosta lokalni\_max > 0 in lokalni\_min < 0. Ko ju pomnožimo z vrednostjo na mestu i, bo lokalni\_max \* i < 0 in lokalni\_min \* i > 0. Sedaj bo nov lokalni\_min večji od novega lokalnega\_max.
* Naj bosta lokalni\_max > 0 in lokalni\_min > 0. Ko ju pomnožimo z vrednostjo na mestu i, bo lokalni\_max \* i < 0 in lokalni\_min \* i < 0. Sedaj bo nov lokalni\_min večji od novega lokalnega\_max.
* Naj bosta lokalni\_max < 0 in lokalni\_min < 0. Ko ju pomnožimo z vrednostjo na mestu i, bo lokalni\_max \* i > 0 in lokalni\_min \* i > 0. Sedaj bo nov lokalni\_min večji od novega lokalnega\_max.

Od tod ideja, da ko naletimo na negativno vrednost, zamenjamo lokalni\_min in lokalni\_max.

Razumevanje zgoraj opisanega si lahko morda pomagamo s spodnjo skico.

A whiteboard with text and numbers

Description automatically generated

## **Uporaba na primeru**

Recimo, da imamo dan seznam [-3,10,-9,80,-75,3,-10,20,41,3,-7,81,203,17,-306,-33].

Na začetku nastavimo lokalni\_min, lokalni\_max in globalni\_min na 1, saj se bomo s tem držali pravila v navodilih.

* i = 1:
  + - * lokalni\_max = max(10, 10\* (-3)) = 10
      * lokalni\_min = min(10, 10\*(-3)) = -30
      * globalni\_max = max(1, 10) = 10
* i = 3:
  + - * lokalni\_max = max(80, 80\*270) = 21600
      * lokalni\_min = min(80, 80\*(-90)) = -7200
      * globalni\_max = max(270, 21600) = 21600
* i = 5:
  + - * lokalni\_max = max(3, 3\*540000) = 1620000
      * lokalni\_min = min(3, 3\*(-1620000)) = -4860000
      * globalni\_max = max(540000, 1620000) = 1620000
* i = 6: element na tem indeksu je negativen, zato zamenjamo prejšnji lokalni\_max in lokalni\_min (ta zamenjava je prikazana z zeleno in modro barvo)
  + - * lokalni\_max = max(-10, -10\*(-4860000)) = 4860000
      * lokalni\_min = min(-10, -10\*1620000) = -16200000
      * globalni\_max = max(1620000, 48600000) = 48600000

Na koncu dobimo globalni\_max = 78743280956632000000.

## **Analiza časovne in prostorske zahtevnosti algoritma**

* Velikost problema: velikost seznama
* Karakteristična operacija: množenje in primerjanje velikosti
* Časovna zahtevnost: , ker iteriramo samo enkrat čez seznam
* Prostorska zahtevnost: , ker uporabljamo samo tri spremenljivki in nič dodatnega prostora npr. dodatnega seznama.

## **Programska rešitev algoritma**

A screen shot of a computer code

Description automatically generated

# **Problem najkrajše poti med dvema vozliščema v zazankanem drevesu (angl. Looped tree)**

V knjigi (Erickson, 2019) na strani 297, 1. naloga, je predstavljen problem najkrajše poti med dvema vozliščema v zazankanem drevesu. Poglejmo, zakaj gre.

*Zazankano drevo je utežen, usmerjen graf, zgrajen iz binarnega drevesa z dodajanjem povezave od vsakega lista nazaj do korena. Vsaka povezava ima nenegativno težo.*

*A diagram of a tree

Description automatically generated*

1. *Koliko časa bi Dijkstrov algoritem potreboval za izračun najkrajše poti med dvema vozliščema u in v v zankanem drevesu z n vozlišči?*
2. *Opišite in analizirajte hitrejši algoritem.*

## **Dijkstrov algoritem**

Ideja Dijkstrovega algoritma je najti najkrajšo pot med začetno točko in vsemi drugimi točkami v grafu. Algoritem vzdržuje množico točk, kjer je za vsako točko shranjena trenutna najkrajša znana pot od vira do te točke. Postopek poteka iterativno, pri čemer se v vsakem koraku izbere točka, ki ni v množici, vendar ima najmanjšo trenutno znano razdaljo od vira. Nato se posodobi razdalja do vseh sosednjih točk izbrane točke. Postopek se ponavlja, dokler niso vse točke vključene v množico ali pa ni več možnih izboljšav poti. Torej naš končen rezultat bo drevo najcenejših povezav.

Vendar v naši nalogi imamo opravka s potjo med izbranima vozliščema. Originalni Dijkstrov algoritem potem dopolnimo tako, da vedno začnemo v izbranem vozlišču u in končamo postopek, ko najdemo vozlišče v.

Potrebovali bomo:

* Seznam razdalj, ki hrani najmanjšo razdaljo od izvornega do vsakega vozlišča v grafu. Na začetku je za izbrano vozlišče razdalja enaka 0, za vse ostale pa neskončno. Čim bomo našli krajšo razdaljo, se bo seznam ustrezno posodobil.
* Vrsta , ki je prednostna vrsta vseh vozlišč v grafu. Na koncu bo prazna.
* Niz , ki označuje, katera vozlišča je algoritem že obiskal. Na koncu bo vseboval vsa vozlišča.

## **Uporaba Dijsktrovega algoritma na primeru**

Želeli bi najkrajšo pot od vozlišča 4 do vozlišča 8.

1. Postopek začnemo v vozlišču 4.
2. Pogledamo povezave iz njega in izračunamo povezave za sosednja vozlišča.
3. Naslednje vozlišče izberemo tako, da ima najkrajšo povezavo. To je 1.

Postopek nadaljujemo.

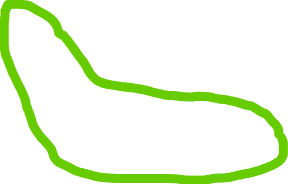
A diagram of a network

Description automatically generated

Naše drevo sedaj izgleda kot označeno na sliki. Izberemo tisto vozlišče, ki ima najkrajšo povezavo. Te povezave so zapisane z rdečo in obkrožene s temno zeleno. Vidimo, da je to ravno vozlišče 2.

A diagram of a network

Description automatically generated



Potem ga dodamo v naše drevo in ponovimo postopek, dokler ne pridemo do vozlišča 8.

A diagram of a network

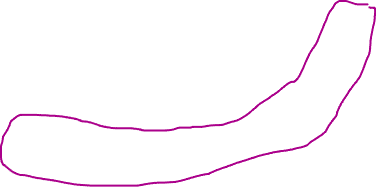
Description automatically generated



Sedaj smo prišli do vozlišča 8 in postopek končali. Dobili smo drevo najcenejših povezav. Ampak nas zanima samo od vozlišča 4 do vozlišča 8, kar je pa del naše cele poti, zato je rezultat obarvan vijolično.

A diagram of a network

Description automatically generated



## **Psevdokoda Dijsktrovega algoritma**

**def** dijakstra(graf = (, ), izbrani):

dist[izbrani] ← 0

postavi dist[] = 0 za vse in ≠ izbrani

dodaj v

←

**whil**e ≠ :

← vozlišče v z min(dist[])

odstrani

dodaj v

za vse sosede od dodanega vozlišča , ki še niso v drevesu najkrajših

poti popravimo dist[]

**return** dist[]

## **Analiza časovne in prostorske zahtevnosti Dijsktrovega algoritma**

Naj bo velikost vozlišča in velikost povezave/cene.

Časovna zahtevnost je odvisna od izbire podatkovne strukture za . Če je:

* je običajna vrsta:
* je prednostna vrsta v obliki kopice:
* je prednostna vrsta v obliki Fibonaccijeve kopice: .

Najhitrejši algoritem je z uporabo Fibonaccijeve kopice.

## **Analiza časovne zahtevnosti na zazankanem drevesu z uporabo Dijkstrovega algoritma**

Opazimo lahko, da ima vsako vozlišče največ dve povezavi, ki gresta iz njega. Torej v najslabšem primeru, če gremo skozi vse povezave, gremo dvakrat skozi vsa vozlišča. Časovna zahtevnost za povezave je , saj je . Skupna časovna zahtevnost za Dijakstrov algoritem je .

## **Analiza časovne zahtevnosti in opis boljšega algoritma na zazankanem drevesu**

Če pogledamo sliko lahko ugotovimo, da obstaja samo eno vozlišče, ki ima več kot eno povezavo, ki kaže vanj. To je koren. Najkrajša pot med vozliščema in bo lahko šla čez koren ali pa ne. Poglejmo si oba primera.

1. Vozlišči in **ne gresta čez koren**. Rešitev je tako samo ena in sicer seštevek cen povezav od do . To lahko opravimo v z uporabo DFS (Depth-first search) algoritma oz. algoritma iskanja v globino.

A diagram of a tree

Description automatically generated

1. Vozlišči in **gresta čez koren**. Od korena do vozlišča bo pot enolična in jo lahko naračunamo enako kot v primeru a). Potrebujemo izračunati le najkrajšo pot od do korena. Od do korena pa imamo usmerjeni acikličen graf, torej graf, ki je usmerjen in brez ciklov. Zanj prav tako obstaja algoritem za računanje najkrajših poti (angl. Directed Acyclic Graph), ki ima časovno zahtevnost Skupen čas za algoritem je torej

A diagram of a tree

Description automatically generated

## **Prikaz boljšega algoritma na nekaj primerih**

**A drawing of a diagram

Description automatically generated**A drawing of a diagram

Description automatically generated

A drawing of a diagram

Description automatically generated**A drawing of a diagram

Description automatically generated**

**A diagram of a tree

Description automatically generated with medium confidenceA diagram of a tree

Description automatically generated**

# Viri

Erickson, J. (2019). *Algorithms.*

*Maximum subarray problem*. (2023). Pridobljeno iz Wikipedia: https://en.wikipedia.org/wiki/Maximum\_subarray\_problem

*Kadanes algorithm explained*. (2023). Pridobljeno iz Algodaily: https://algodaily.com/lessons/kadanes-algorithm-explained

*Largest sum contiguous subarray*. (2023). Pridobljeno iz Geeksforgeeks: https://www.geeksforgeeks.org/largest-sum-contiguous-subarray/

Abhishek, S. w. (8. June 2021). *Max Product Contiguous Subarray*. Pridobljeno iz You Tube: https://www.youtube.com/watch?v=bLHrFVx-OJQ

*Maximum product subarray in an array*. (2023). Pridobljeno iz TakeUForward: https://takeuforward.org/data-structure/maximum-product-subarray-in-an-array/

*Dijkstra time complexity*. (2023). Pridobljeno iz Baeldung: https://www.baeldung.com/cs/dijkstra-time-complexity

*Priority queue*. (2023). Pridobljeno iz Baeldung: https://www.baeldung.com/cs/priority-queue

*Shortest path for directed acyclic graphs*. (2023). Pridobljeno iz Geeksforgeeks: https://www.geeksforgeeks.org/shortest-path-for-directed-acyclic-graphs/

*H21 solution*. (2023). Pridobljeno iz Bowdoin: https://tildesites.bowdoin.edu/~ltoma/teaching/cs231/duke\_cps130/Homeworks/Solutions/H21-solution.pdf

Person, C. t. (15. january 2022). *Dijkstra's Algorithm Visualized and Explained*. Pridobljeno iz You Tube: https://www.youtube.com/watch?app=desktop&v=71Z-Jpnm3D4