Nivojske krivulje

Druga projektna naloga

Gašper Andrejc (63130002)
David Bašelj (63130009)
Marko Grešak (63130058)
Matic Repše (63130207)

11. junij 2015

1 Uvod

Namen projekte naloge je bil prikazati rekonstrukcijo neke funkcije ali naravnih podatkov, prikazanih s koordinatami *(zemljepisna širina, zemljepisna dolžina, nadmorska višina)*, z nivojskimi krivuljami. Izrisovali smo dve obliki, prva kot statični dvodimenzionalni graf njvojnic, na podoben način, kot smo to delali v prvem semestru pri predmetu OMA za funkcije dveh spremenjivk. Drug graf pa je aproksimirana rekonstrukcija v obliki interaktivnega 3D grafa.

2 Podatki

Za podatke smo si za umetne podatke izbrali nekaj že znanih funkcij: Ackleym, Beales, CrossInTray Sin, Matyas in Mishra. Pomagali pa smo si tudi z vizualizacijami grafov na platformi WolframAlpha, da smo lažje preverjali, kakšne razultate naj sploh pričakujemo.

2.1 Podatki iz narave

2.1.1 SRTM meritve (Nasa)

Za podatke iz narave pa smo najprej poizkusili s podatki Shuttle Radar Topography Mission (SRTM), ki jih ponuja Nasa in so bili navedeni kot priporočen vir podatkov v poročilu. Vendar pa smo sprva imeli problem z iskanjem vira podatkov, porabili smo namreč nekaj ur, da smo prišli do navedene povezave ter ugotovili, kako brati podatke.

Podatki so zakodirani v posebni obliki, zato je za iskano višino po posebnem algoritmu podatke odkodirati za dani zamljepisno širino ter dolžino. Za primer geolokacije Triglava *(46.379347, 13.832887)*, oziroma če pretvorimo decimalni del v minute in sekunde
($46^\circ\ 22'\ 45.6492"$*,* $13^\circ\ 49'\ 58.3926"$*)*. Potem moramo najprej poiskati ustrezno datoteko, v tem primeru se le-ta imenuje `N46E013.hgt`. Parametra `n` in `e` funkcije `get_height` pa izračunamo po formuli `min * 60 + sec`. Za točko iz primera je najdena višina *2490m*, kar po podatkih

iz Google Maps odstopa za nekaj metrov, na katero pa seveda vpliva tudi natančnost zbranih podatkov. Vendar pa večja natančnost pomeni večjo velikost datotek, za primer najbolj natančnih meritev (1/9 kotne sekunde) podatki nanesejo več kot 600GB, kar pa seveda v tako omejenem času za izdelavo naloge verjetno ne bi mogli prenesti, sploh pa ne obdelati na osebnih računalnikih.
Python koda za pridobivanje nadmorske višine dane datoteke ter odmika v minutah in sekundah:

```
def get_height(filename, n, e):
  i = 1201 - int(round(n / 3, 0))
  j = int(round(e / 3, 0))
  with open(filename, "rb") as f:
    f.seek(((i - 1) * 1201 + (j - 1)) * 2)
    buf = f.read(2)
    val = struct.unpack('>h', buf)
    if not val == -32768:
      return (round(n), round(e), val[0])
    else:
      return None
```

2.1.2 Google Elevetion API

Ker je zgoraj opisani del vzel kar veliko časa, smo začeli dvomiti, da nam bo uspelo najti podatke pred rokom oddaje naloge, smo se odločili za drugi plan: Google Maps Elevation API. Ta omogoča *2500* zahtev na dan, v vsaki zahtevi pa lahko zahtevamo podatke za do *512* točk. Torej lahko na vsakih 24 ur dobimo višine za $2500 \cdot 512 = 1,280,000$ geolokacijskih točk in če to še pomnožimo z vsemi člani, lahko dobimo do *5,120,000* točk na dan, kar je bilo dovolj, da smo lahko izračunali in vizualizirali nivojske krivulje za gori Uluru v Avstraliji ter Mt. Fuji, najvišje gore na Japonskem.

3 Triangulacija

Triangulacija je postopek, kjer določamo oddaljenost točke tako, da izmerimo kota α in β iz dveh drugih točk A in B, z medsebojno oddaljenostjo l, kateri skupaj s ciljno točko tvorijo trikotnik. Oddaljenost d izračunamo po formuli $d = l \cdot \frac{\sin\alpha \sin\beta}{\sin(\alpha+\beta)}$.

Triangulacija se danes uporablja za merjenja (npr. geodeti), navigacijo, izračun pozicije teles v vesolju, izračun leta naboja iz orožja ali leta rakete ipd. Ne smemo pa je zamešati s trilateracijo, katero smo že spoznali med predstavitvami za prvo projektno nalogo, saj je sam postopek precej drugačen.

4 Nivojske krivulje

Nivojska krivulja funkcije dveh spremenljivk je krivulja okoli katere ima funkcija konstantno vrednost. Je presek tridimenzonalnega grafa, ki predstavlja funkcijo, z neko poljubno ravnino $z = z1$. Uporablja se za najrazličnejše predstavitve v naravi; s skupnim imenom se v slovenščini imenujejo 'izočrte' - so črte, ki na zemljevidu povezujejo točke enakih vrednosti fizikalnih, meteoroloških ali jezikoslovnih količin (vir: Wikipedia: 'Izočrte').

V našem projektu smo se ukvarjali z izočrto pod imenom 'izohipsa, ki v naravi povezuje mesta z isto nadmorsko višino oziroma translirano za naš problem - je črta, ki povezuje točke, kjer ima funkcija enako vrednost. Ko pa smo se kasneje ukvarjali s podatki iz narave pa smo dejansko poskusili risati izohipse določenih geografskih področij.

5 Algoritem za triangulacijo domene in rekonstrukcijo ploskve

Algoritem za triangulacijo in rekonstrukcijo ploskve je napisan v jeziku Java, kjer nam class 'Network' (v nadaljevanju kot 'omrežje') opisuje celotno površino z vsemi vozlišči in povezavami, ki jih uporabljamo za risanje nivojnic.

Najprej si izberemo nekaj točk in iz njih naredimo kvadratno mrežo – kartezični produkt. Za princip naloge smo lahko predpostavljali, da so točke enakomerno razporejene na mreži (torej razmak med vsako točko je konstanten in enak za vse). Je pa glede na podano funkcijo pomembno s kakšnim razmakom med točkami gradimo kartezični produkt. Torej, če si mi izberemo vsa števila od 1 do 3 z razmakom med njimi 1, dobimo kartezični produkt sestavljen iz 9 točk, kjer je natančnost rekonstrukcije 1 enota. Primer zgoraj opisanega karteličnega produkta bi bile točke: ([1, 1], [1, 2], [1, 3] ....). Vsaka točka ima torej x in y koordinato. Naslednji korak je bil, da med vsemi temi točkami konstruiramo povezave in tako dobimo, v našem primeru 4, kvadrate. Za triangulacijo dodamo še v vsakem kvadratu eno (povsod isto) diagonalo. Rezultat je mreža sestavljena iz n trikotnikov (Slika 1).

Mreža nam na tej točki predstavlja le natančnost, s katero bomo rekonstruirali ploskev. Kot opisano zgoraj, ima vsaka povezava dve točki – začetno in končno točko, vsaka točka pa x,y koordinate. Predno nadaljujemo s triangulacijo moramo v algoritmu urediti še vse povezave tako, da imamo nekje seznam, katere povezave so sosednje in katera vozlišča so sosednja – to potrebujemo za kasnejšo delovanje algoritma.

Od tu naprej moramo x in y koordinato vsake točke vstaviti v našo funkcijo. Rezultat te funkcije nam predstavlja višino funkcije v podani točki. To vrednost dodelimo vsakemu vozlišču (Slika 1).

Sedaj si izberemo neko ravnino, ki jo definiramo z z višino (v nadaljevanju 'c'), s katero bomo presekali funkcijo. Postopek je tak, da pregledamo celotno omrežje in na vse daljice, kamor ta vrednost pade (odvisno od vrednosti funkcije, ki smo jo, kot je zgoraj opisano, dodelili vsakemu vozlišču), proporcionalno narišemo točko. Primer: če je začetek daljice definiran z vozliščem, ki ima vrednost 1.1, konec daljice 1.9, c pa 1.8, bomo nekje na konec daljice narisali točko. Točno kje se ta točka nariše je seveda pomembno, a je pomembno kasneje, ko se ta točka translira na

originalne koordinate kartezičnega produkta (Slika 3).

Predno se točke translirajo v originalne koordinate, je potrebno narisati še dejansko povezave, ki nam bodo predstavljale nivojske krivulje. To naredimo tako, da povežemo vse točke, ki so sosednje – tu nam pride prav seznam vseh sosednjih vozlišč in povezav, za kar smo poskrbeli na začetku algoritma. Po tem delu je čas, da, kot je opisano zgoraj, točke transliramo nazaj v prvotne koordinate.

```
Edge nedge = neighbourEdges.get(j);
/*
 * if point is on edge, create a new Node with the calculated x and y
 * properly proportioned depending on the original values
 * we do that by calculating like this:
 *       dist........100%
 *       c - min.....x %
 *
 * this same proportion we later on apply to our own scale that we're
 * going to use for graphic
 */
if (pointIsOnEdge(nedge, c)) {
  distance = Math.abs(nedge.startValue - nedge.endValue);
  min = Math.min(nedge.startValue, nedge.endValue);
  // proportion in %
  proportion = ((Math.abs(c - min)) * 100) / distance;
  // now we have to calculate x and y of the node
  // if edge is vertical, we already have x and need to calculate y
  // if edge is horizontal, the other way
  newNodeX = 0;

  newNodeY = 0;
  // for the above, first calculate 'k'
  Node startNode2 = net.nodes.get(nedge.startID);
  Node endNode2 = net.nodes.get(nedge.endID);
  boolean reverse2 = false;
  if (startNode2.nodeValue > endNode2.nodeValue) {
    reverse2 = true;
  }
```

```
    k = (startNode2.y - endNode2.y) / (startNode2.x - endNode2.x);
    if (k == 0) {
      newNodeY = startNode2.y;
    } else {
      // need to calculate y
      newNodeY = Math.min(startNode2.y,endNode2.y) + ((proportion * minDist)/100);
      if (reverse2) {
        newNodeY = Math.max(startNode2.y,endNode2.y) - ((proportion * minDist)/100);
      }
    }
    if (k == Double.POSITIVE_INFINITY || k == Double.NEGATIVE_INFINITY) {
      newNodeX = startNode2.x;
    } else {
      // need to calculate x
      newNodeX = Math.min(startNode2.x,endNode2.x) + ((proportion*minDist)/100);
      if (reverse2) {
        newNodeX = Math.max(startNode2.x,endNode2.x) - ((proportion*minDist)/100);
      }
    }
  }
```

Primer translacije nazaj v prvotne koordinate; 'reverse' zgoraj opisuje primer, ko se riše v drugo smer; predvsem če je funkcija definirana tudi na negativnih oseh (2. ali 4. kvadrant)

6 Grafični prikaz ter težave algoritma

Za prikaz delovanja algoritma smo se odločili, da vse nivojnice prikažemo tudi grafično. Uporabljali smo javanske swing in awt knjižnice. Izhod zgoraj opisanega algoritma je množica vseh daljic (ena daljica je en del nivojske krivulje na podani višini), ki jih prejmejo razredi za risanje. Grafični prikaz kot tak je relativno preprost, ugotovili pa smo, da je potrebno za lepo sliko izredno dobro poznati obnašanje funkcije v različnih delih definicijskega območja. Poleg tega je za vsako funkcijo bilo potrebno ugotovili ravno pravšnjo velikost kartezičnega produkta ter korak povečevanje le tega. Za ugotavavljanje približnih vrednosti vseh teh parametrov smo si pomagali s prikazom funkcij na strani WolframAlpha (http://www.wolframalpha.com/) ter na strani Wikipedie (http://en.wikipedia.org/wiki/Test_functions_for_optimization). Torej kot primer: če je zaloga vrednosti neke funkcije od 0 do 1, smo naš izbor 'c'-ja seveda omejili na ta razpon. Podobno smo morali ugotavljati parametre za začetni kartezični produkt – če je definicijsko območje funkcije od -10 do 10, je največja vrednost kartezičnega produkta padla v ta razpon. Poleg tega, da smo ugotovili te mejne vrednosti, pa smo morali za lep prikaz funkcije upoštevati tudi korak, s katerim smo povečevali tako vrednosti za kartezični produkt kot vrednost 'c'ja. Bolj na gosto kot smo postavili te vrednosti, bolj lepo bi se funkcija (oziroma nivojnice) izrisale, problem pa nastane, ker je naš algoritem relativno časovno zahteven.

Kot je opisano v opisu algoritma je bilo potrebno za vsako točko kartezičnega produkta izračunati vrednost, vzpostaviti povezave med točkami, iskati sosednja vozlišča, sosednje povezave itd. Torej že pri izbiri velikosti kartezičnega produkta (in s tem korak povečevanje razdalj me točkami), je to lahko

ogromno vplivalo na čas izvajanja algoritma. Podoben problem je bil pri izbiri 'c'ja oziroma bolj pri izbiri koraka, s katerim se povečuje (kako na gosto bomo risali nivojnice).

Vse to je bilo potrebno vzeti v račun ko smo risali nivojnice in za kakovostno sliko je bilo potrebno kar dosti iskanja pravih parametrov sorazmerno s tem, da nam je v nekem relativno kratkem času algoritem zaključil.

6.1 3D Risanje

Razred Draw3D.java prejme nivojnice, ki vsebuje robove (edges), na dolocenih visinah. Paziti je treba na scale, saj se mora približno ujemati s prejetimi podatki. Sam program za risanje je dokaj preprost, vendar pa je zelo pomembno, kakšne podamo začetne parametre za izračun. Kartezični produkt se mora ujemati s definicijskim območjem funkcije. Manjši kot je korak kartezičnega produkta, večja je natančnost. Prav tako je potrebno paziti na razpon 'Z' osi, saj se mora ta ujemati z zalogo vrednosti funkcije. Manjši kot je korak razpona, več je izohips in s tem tudi večja natančnost. V razredu Draw3D imamo tudi parameter z, ki ga za vsako izohipso povečamo. Ta "z" parameter je v večini enak razponu po Z osi pri računanju izohips. Konstantno in ločeno se povečuje za to, ker pride do težave pri funkcijah, ki imajo zelo veliko zalogo vrednosti, v tem primeru slika pride razpotegnjena po Z osi.

Rezultati algoritma (več slik in videov izrisovanja: gresak.io/mm)

6.2 Realni podatki

Algoritem smo stestirali tudi na realnih podatkih in sicer tako, da smo namesto kartezičnega produkta vzeli geografsko širino in geografsko dolžino. Obe te vrednosti smo najprej translirali tako, da smo vse skupaj postavili v sredino koordinatnega sistema. Nato smo za vrednosti teh vozlišč vzeli geografsko višino, ki smo jo pridobili s pomočjo google-apija 'elevation'. Seveda smo morali paziti na omejitve tega apija (število requestov ipd.).

Kot posledica teh omejitev ter zgoraj opisane časovne zahtevnosti algoritma nismo bili najbolj zadovoljni z rezultatom. Problem je nastal, ker če smo hoteli natančno izrisati neko področje, smo morali vzeti kar velik razpon na drobno postavljenih točk (tudi do 100.000 točk), kar pa je pomenilo ogromno časovno zahtevo. Bolj kot smo to manjšali, manj 'zanimive' rezultate smo dobivali. Poleg tega pa smo morali ustrezno povečevati še 'c'. 3D izris je bil še dokaj lep, za 2D pa moramo imeti kar dobro domišljijo, da vidimo prave nivojnice.

7 Zaključek

S projektom smo spoznali izredno zanimiv način risanja nivojnic. Sprva, predno smo ga dodobra razumeli, smo bili malo skeptično o samem delovanju. Vedeli smo, s teoretične plati, da deloval bo, a nikakor si nismo predstavljali, da bodo tako natančni rezultati. Še posebno s 3D risanjem se vidi, da s pravimi parametri (velikost kartezičnega, korak kartezičnega, minC, maxC ter korakC) dobimo skoraj identične funkcije, kot pa jih nariše, na primer, WolframAlpha.

Rezultati, ki smo jih dobivali, so nam vedno bolj vzbujali zanimanje in na koncu nikakor nismo bli zadovoljni z rezultati – dokler nismo res dobili popoln rezultat. Izbirali smo si vedno bolj zanimive funkcije in za katerokoli funkcijo smo si izbrali, naj je bila še tako kompleksna, je algoritem deloval popolno (če tukaj ne upoštevamo časovnega parametra), ko pa smo se začeli ubadati z realnimi podatki, pa se je stvar malce obrnila. Namreč kot opisano zgoraj nam nikakor ni uspelo dobro parametrizirati realnih podatkov, da bi nam v nekem zglednem času algoritem vrnil spodobne rezultate. Bodisi smo

vzeli preveliko področje in preveč na drobno, kar je rezultiralo v (pre)dolgo delovanje algoritma, bodisi pa so bili rezultati nenatančni / nezadovoljivi.

Algoritem bi seveda bilo možno tudi pohitriti do neke meje, a v vsakem primeru je potrebno n-krat izračunati določene stvari za delovanje algoritma (n – število vozlišč oz točk). Risanje nivojnic s pomočjo triangulacije je torej mogoče in zredno zanimivo, a za velika področja in natančnost lahko časovno zahtevno. Mi smo morali določati ali želimo veliko področje in malo natančnost ali pa veliko natančnost in manjše področje.

Z rezultati in projektom smo izredno zadovoljni – res zanimiv projekt ki daje konkretne in zanimive, "oprijemljive" rezultate. Če bi imeli še malo več časa, bi lahko izpopolnili še risanje za realne podatke (najti bi bilo potrebno ravno prave parametre).