Univerza v Ljubljani
Fakulteta za računalništvo in informatiko
Fakulteta za matematiko in fiziko

Matjaž Cerar

Topološka obdelava slik

diplomsko delo

univerzitetni interdisciplinarni študij
računalništva in matematike

Mentor: prof. dr. Neža Mramor Kosta
Somentor: doc. dr. Žiga Virk

Ljubljana 2015

Fakulteta za računalništvo in informatiko podpira javno dostopnost znanstvenih, strokovnih in razvojnih rezultatov. Zato priporoča objavo dela pod katero od licenc, ki omogočajo prosto razširjanje diplomskega dela in/ali možnost nadaljne proste uporabe dela. Ena izmed možnosti je izdaja diplomskega dela pod katero od Creative Commons licenc http://creativecommons.si.

Morebitno pripadajočo programsko kodo praviloma objavite pod, denimo, licenco *GNU General Public License, različica 3.* Podrobnosti licence so dostopne na spletni strani http://www.gnu.org/licenses/.

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Cilj diplomskega dela je implementacija nekaj znanih algoritmov za topološko analizo slik in prikaz njihove uporabe. Digitalno sivinsko sliko predstavimo kot kubični kompleks s podanimi sivinskimi vrednostmi v ogliščih. Tega razširimo do diskretne Morsove funkcije na omenjenem kompleksu. Kritične celice dobljene Morsove funkcije zajemajo ključne podatke o značilnih oblikah na slikah.

V diplomskem delu opišite in implementirajte strukturo kubičnega kompleksa s podanimi funkcijskimi vrednostmi v ogliščih, algoritem za generiranje diskretne Morsove funkcije in pripadajočega diskretnega vektorskega polja ter algoritem za krajšanje odvečnih kritičnih celic in izračun prvih dveh Bettijevih števil kubičnega kompleksa. Prikažite delovanje implementiranih algoritmov pri analizi slik.

IZJAVA O AVTORSTVU DIPLOMSKEGA DELA

Spodaj podpisani Matjaž Cerar sem avtor diplomskega dela z naslovom:

Topološka obdelava slik

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom prof. dr. Neže Mramor Kosta in somentorstvom doc. dr. Žige Virka,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela na svetovnem spletu preko univerzitetnega spletnega arhiva.

V Ljubljani, dne 6. julij 2015 Podpis avtorja:

Zahvaljujem se svoji mentorici, prof. dr. Neži Mramor Kosta in somentorju, doc. dr. Žigi Virku, za pomoč, usmerjanje in koristne nasvete tekom izdelave diplomskega dela. Posebna zahvala gre mojim staršem in domačim za vso podporo med študijskim časom in nastajanjem diplomskega dela ter vsem prijateljem za vzpodbudne besede med študijem.

Kazalo

Povzetek

Abstract

1 Uvod **1**

1.1 Digitalne slike . . . 1

1.2 Orodja . . . 2

1.3 Vsebinski pregled . . . 3

2 Kubični kompleks **5**

2.1 Zgradba in lastnosti . . . 6

2.2 Podatkovna struktura . . . 7

2.3 Implementacija . . . 9

3 Diskretno vektorsko polje **13**

3.1 Kritične celice . . . 15

3.2 Algoritem “Process Lower Stars” . . . 17

3.3 Implementacija algoritma “Process Lower Stars” . . . 20

4 Morsov kompleks **25**

4.1 Algoritem “Extract Morse Complex” . . . 26

4.2 Implementacija algoritma “Extract Morse Complex” . . . 27

5 Računanje Bettijevih števil **33**

5.1 Robne matrike . . . 33

5.2 Algoritem krajšanja 35

6 Primeri uporabe **39**

6.1 Primer 1 40

6.2 Primer 2 41

6.3 Primer 3 42

7 Splošne ugotovitve **45**

Povzetek

V diplomskem delu si ogledamo implementacijo topološkega pristopa k analizi digitalnih 2-dimenzionalnih slik. Najprej predstavimo dva načina, kako sliko brez izgube informacij poenostavimo in pripravimo, da je primerna za nadaljnje algoritme. Obdelano sliko nato predstavimo kot topološko strukturo, ki jo imenujemo kubični kompleks. S pomočjo slednjega zgradimo vektorsko polje, ki odraža smeri, kamor funkcijske vrednosti padajo, ter pripadajoči seznam kritičnih celic. Iz obeh dobljenih struktur zgradimo Morsov kompleks, s katerim zajamemo bistvene informacije o posamezni sliki, in izračunamo Bettijeva števila, ki opisujejo ključne značilnosti slike. Za izračun Bettijevih števil prav tako predstavimo dva pristopa. Na koncu sledi še prikaz uporabe, kjer na izbranih primerih slik štejemo svetle objekte.

Ključne besede: diskretna Morsova teorija, topološka analiza podatkov, kubični kompleks, Morsov kompleks, Bettijeva števila.

Abstract

In this thesis we present an implementation of a topological approach to 2-dimensional digital images. First, we present two methods for simplifying and preparing the image, without loss of information, for further algorithms. We represent the image as a topological structure called a cubical complex. On the cubical complex, a discrete vector field encoding the directions of descent of grey scale values is constructed, together with the corresponding list of critical cells. From these, the Morse complex, which captures the vital information about the image, is built. Using Betti numbers, important features in the image are described. We present two approaches to computing Betti numbers. The thesis concludes with a presentation of how the implemented algorithms can be used for counting bright objects on specific examples of images.

Keywords: discrete Morse theory, topological data analysis, cubical complex, Morse complex, Betti numbers.

Poglavje 1

Uvod

Digitalne slike so danes že skoraj povsem nadomestile stare, t. i. analogne slike. Prav tako so vedno bolj kvalitetne in izostrene, pri čemer ima zasluge vedno boljša tehnologija, ki očitno ne pozna meja in omogoča prikazovanje slik z vedno večjim številom slikovnih elementov. Posledično so postali algoritmi za obdelavo slik počasnejši, saj število vhodnih podatkov hitro narašča.

Algoritmov in pristopov za obdelavo slik je ogromno. V tej nalogi se bomo osredotočili na topološko analizo slik. Tovrstna analiza omogoča kar nekaj algoritmov za analiziranje, kot so tanjšanje (ang. *thinning*), sledenje mejam in površini (ang. *border and surface tracing*), štetje komponent ali tunelov (ang. *counting of components or tunnels*), itd. [9] Pobližje si bomo ogledali štetje komponent. Kot problem si lahko predstavljamo slike večjih ločljivosti, za katere vemo, da je njihova obdelava časovno in prostorsko zahtevna. Naša motivacija je, kako iz slike razbrati najpomembnejše topološke informacije, jih ustrezno predstaviti v računalniku in s tem konkretno privarčevati tako na času kot prostoru.

1.1 Digitalne slike

Za začetek si najprej oglejmo nekatere pomembne lastnosti digitalnih slik. Vsaka digitalna slika je predstavljena kot končna množica slikovnih elementov

(v 2-dimenzionalnem prostoru imenujemo te elemente ang. *pixels*). Zaradi lažjega razumevanja algoritmov in podatkovnih struktur se bomo posvetili 2-dimenzionalnim slikam, čeprav vse obravnavano velja tudi za slike višjih dimenzij. Brez kakršne koli izgube informacij lahko posamezne slike izrazimo samo z eno barvo. S tem bi želeli poenostaviti vhodne podatke algoritmom, da bi lahko ti čim hitreje in optimalno opravili svoje delo. V tej nalogi bomo predstavili dva takšna načina, ki ju bomo podrobneje spoznali v naslednjih odstavkih.

Prvi način je, da slike predstavimo kot sivinske. V realnosti lahko takšne dobimo že v osnovi, toda z veliko verjetnostjo bo ravno nasprotno, kar pa ne predstavlja večjega problema, saj jih lahko kadar koli enostavno predstavimo oz. pretvorimo v sivinske vrednosti. To storimo tako, da vsak barvni slikovni element pretvorimo po naslednji formuli:

$$\textit{sivinska vrednost elementa} \;=\; 0.3 * R + 0.59 * G + 0.11 * B, \qquad (1.1)$$

kjer so R, G in B intenzitete rdeče, zelene in modre barve poljubnega slikovnega elementa. S to formulo zagotovimo, da se vsi barvni slikovni elementi spremenijo v sivinske vrednosti tako, da zajemajo vrednosti z intervala $[0, 255]$, kjer najmanjša vrednost opisuje najtemnejši odtenek (črna barva), najvišja pa najsvetlejšega (bela barva).

Drugi, preprostejši način, pa je, da namesto pretvarjanja intenzitet vseh treh barvnih odtenkov v sivinske vrednosti, vzamemo le eno izmed treh barvnih kanalov – rdečo, zeleno ali modro. S tem je naš vpliv na končen rezultat ničen [1, 7, 8].

1.2 Orodja

Vsi algoritmi, predstavljeni v diplomskem delu, so implementirani v programskem jeziku *C#*.

1.3 Vsebinski pregled

Najprej bomo pokazali, kako digitalno sliko predstavimo v topološkem prostoru, ki ga imenujemo kubični kompleks (ang. *cubical complex*), ga podrobneje opisali in predstavili njegovo implementacijo. Nato bomo pobližje spoznali, kaj so kritične točke, kaj je diskretno vektorsko polje in kako ga zgradimo (z opisom algoritma). Sledi opis, kako iz seznama kritičnih celic in diskretnega vektorskega polja zgradimo Morsov kompleks (ang. *Morse complex*), ki z vidika kompleksnosti predstavlja posplošen kubični kompleks z najpomembnejšimi topološkimi informacijami. S pomočjo dobljenih informacij bomo pokazali še, kako lahko preštejemo število iskanih objektov. Na koncu si bomo ogledali tudi nekaj primerov slik in jih analizirali za različne vhodne podatke.

Omenjen postopek, skozi katerega se bomo podali v nadaljevanju tega diplomskega dela, smo zasnovali na teoretični podlagi članka z naslovom *Theory and Algorithms for Constructing Discrete Morse Complexes from Grayscale Digital Images* [10].

Poglavje 2

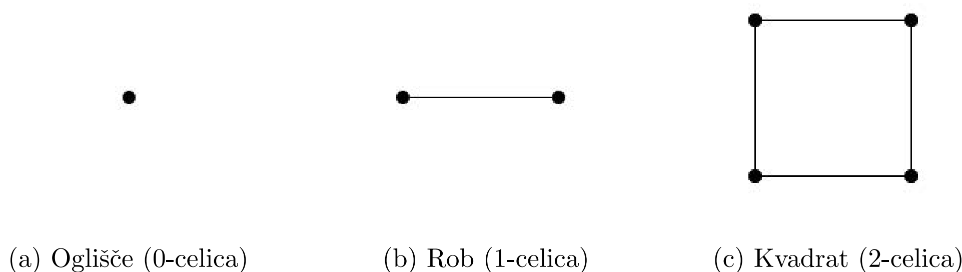
Kubični kompleks

Čeprav vsaka sivinska slika izgleda podobno kot zvezna funkcija na pravokotniku v ravnini, smo se v uvodu lahko prepričali, da ni tako. Ker gre v resnici za diskretne (slikovne) elemente, nam to dejstvo pravzaprav le olajša delo. Zvezni prostori v računalništvu namreč zaradi same predstavitve niso dobrodošli in jih zato poskušamo vedno predstaviti diskretno. V tem poglavju bomo spoznali enega izmed mnogih diskretnih prostorov, ki jih uporabljamo v topologiji.

Topološki prostor, ki predstavlja poljubno digitalno sivinsko več-dimenzionalno sliko kot skupek celic različnih dimenzij, imenujemo kubični kompleks (ang. *cubical complex*) [1, 9]. Kasneje bomo videli, da lahko z njegovo pomočjo natančneje prikažemo padanje in naraščanje sivinskih vrednosti. Osredotočili se bomo le na 2-dimenzionalne sivinske slike, čeprav bi lahko na podoben način obravnavali tudi 3- in več-dimenzionalne "slike". V nadaljevanju bomo predstavili zgradbo in lastnosti kubičnega kompleksa ter implementacijo podatkovne strukture, s katero lahko na hiter in učinkovit način dostopamo do posameznih elementov kompleksa.

2.1 Zgradba in lastnosti

Naj bo digitalna sivinska 2-dimenzionalna slika velikosti $M \times N$ definirana kot diskretna funkcija $g : D \to \mathbb{R}$, kjer je $D = \{(i, j) \mid 0 \leq i \leq M, 0 \leq j \leq N\}$ in $D \subset \mathbb{Z}^2$. Potem lahko v grobem 2-dimenzionalni kubični kompleks $K(D)$ (ali krajše K) opišemo kot zbirko p-celic (za $p = 0, 1, 2$), kjer so slikovni elementi *(ang. pixels)* $x \in D$ celice najmanjše dimenzije in predstavljajo oglišča (0-celice) kubičnega kompleksa. Vsaki dve sosednji oglišči (z vidika evklidske razdalje) sta med seboj povezani z višjo dimenzionalno 1-celico. Seznamu 1-celic pravimo robovi *(ang. edges)*, celicam, ki jih te posledično tvorijo, pa 2-celice, imenovane kvadrati *(ang. squares)* [10]. Gradniki kubičnega kompleksa so vidni na sliki 2.1.



(a) Oglišče (0-celica) (b) Rob (1-celica) (c) Kvadrat (2-celica)

Slika 2.1: p-celice 2-dimenzionalnega kubičnega kompleksa.

Celice različnih dimenzij so med seboj v relaciji tedaj, ko ima višje-dimenzionalna celica v svojem robu nižje-dimenzionalno celico. Za množico

$$\{\alpha^{(p)} < \beta^{(q)} \mid \alpha^{(p)}, \beta^{(q)} \in K, p < q, p,q \in \{0, 1, 2\}\},$$

kjer so oglišča celice $\alpha^{(p)}$ vsebovana v množici oglišč celice $\beta^{(q)}$, pravimo, da je $\alpha^{(p)}$ *lice* (ang. *face*) od $\beta^{(q)}$ ali $\beta^{(q)}$ *kolice* (ang. *coface*) od $\alpha^{(p)}$. Posledično tako vedno velja, da kompleks, ki vsebuje p-celico c, vedno vsebuje tudi vsa *lica* celice c [2, 10].

2.2 Podatkovna struktura

Z zgoraj naštetimi lastnostmi poznamo vse najpomembnejše lastnosti kubičnega kompleksa, ki jih teoretično in praktično potrebujemo za nadaljevanje. Želimo namreč zasnovati čim boljšo podatkovno strukturo, s katero bomo na hiter in učinkovit način dostopali do posameznih elementov kompleksa.

Naj bo I vhodna digitalna sivinska slika velikosti $M \times N$. V računalniku po navadi sliko predstavimo kot več-dimenzionalno tabelo, kjer so njene vrednosti bodisi barvni bodisi sivinski slikovni elementi in kjer je dimenzija tabele odvisna od dimenzije slike. Podatkovno strukturo, ki bo predstavljala omenjeno tabelo, bomo imenovali *Matrix*. Enako se imenuje tudi naš osnovni razred, iz katerega je izpeljan kubični kompleks.

Tako kot samo sliko tudi kubični kompleks predstavimo na podoben način, le da so njegovi elementi p-celice. Omenili smo, da za vsaki dve sosednji oglišči v mreži $\mathbb{Z}^2$ v evklidskem prostoru obstaja povezava, sosednja štiri oglišča v obliki kvadrata pa tvorijo 2-celico. V naši podatkovni strukturi to predstavimo z dodatnimi vrsticami in stolpci v matriki p-celic, zato so dimenzije kubičnega kompleksa za sliko I velikosti $(2 * M - 1) \times (2 * N - 1)$.

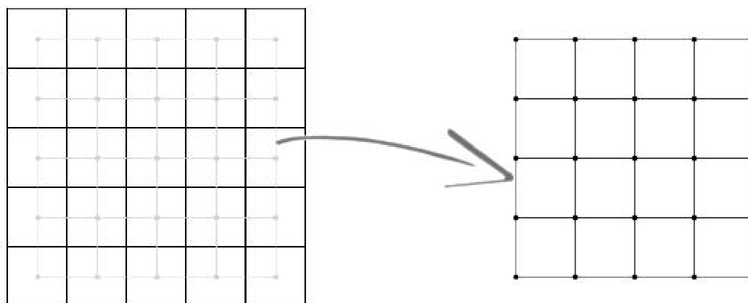
Slabost takšne podatkovne strukture je povečanje njene velikosti, saj se ta poveča za skoraj faktor 2 glede na velikost same slike, vendar pa teoretična prostorska zahtevnost še vedno ostaja enaka, tj. $O(M * N)$. Po drugi strani pa v prid taki predstavitvi govori dejstvo, da smo s takšnim načinom implementacije oblikovali strukturo, s katero lahko hitro dostopamo do posamezne celice. Tako lahko v vsakem trenutku poiščemo vsa *lica* in *kolica* poljubne celice, hkrati pa vemo, za katero dimenzijo celice gre ter kje se vse celice določene dimenzije nahajajo. Poleg tega nam takšna struktura omogoča dober vpogled v to, kako se sivinske vrednosti v sliki spreminjajo.

Naj bosta i in j indeksa, kjer je $i \in \{0, 1, ..., M\}$ in $j \in \{0, 1, ..., N\}$, poljubne p-celice iz kubičnega kompleksa slike I. Potem za vsako p-celico velja, da je:

- oglišče (0-celica), če sta i in j soda,
- rob (1-celica), če je i lih in j sod ali obratno,
- kvadrat (2-celica), če sta i in j liha.

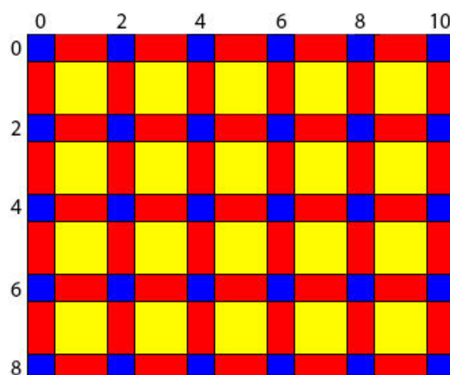
2.2.1 Splošni primer

Za lažje razumevanje si celoten postopek oglejmo na manjši simbolični sliki, velikosti 5×5, kot je prikazano na sliki 2.2. Na njej lahko vidimo prvotno sliko, iz katere preberemo sivinske slikovne elemente in jih v kubičnem kompleksu predstavimo kot njegova oglišča. Sosednja oglišča (tista, ki se med seboj razlikujejo le v eni koordinati) povežemo, tako da dobimo robove, s čimer pa že sami po sebi nastanejo kvadrati.



Slika 2.2: Primer slike velikosti 5×5 (levo) in njenega kubičnega kompleksa (desno).

Tako smo zgolj teoretično opisali in prikazali, kako nastane kubični kompleks. Sedaj pa se bomo podrobneje posvetili predstavitvi takšne strukture v računalniku, kar smo sicer v grobem že opisali in kar prikazuje tudi slika 2.3.



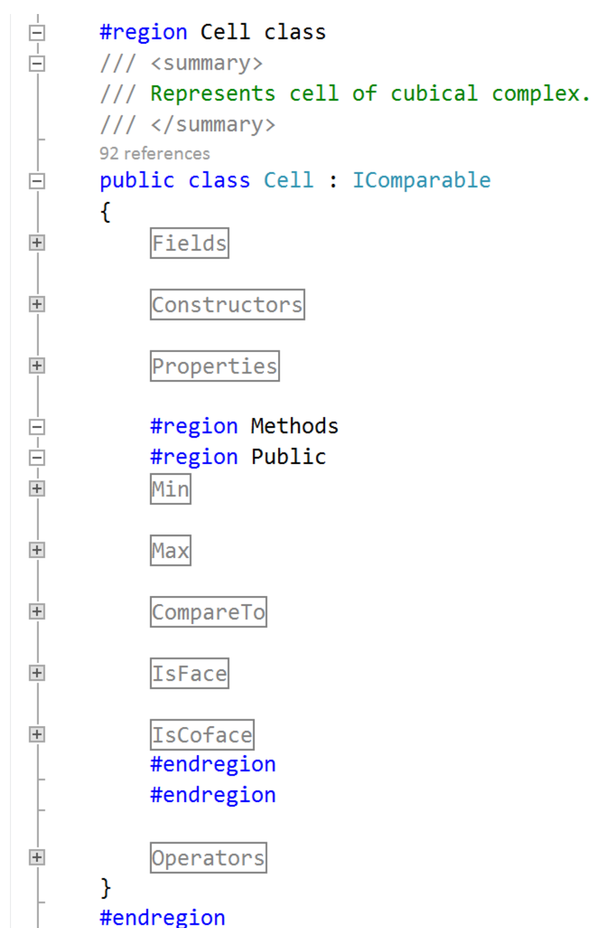
Slika 2.3: Kubični kompleks za sliko velikosti 5 $\times$ 5, kjer modri kvadratki predstavljajo oglišča (0-celice), rdeči robove (1-celice) in rumeni kvadrate (2-celice) le-tega. Prav tako lahko opazimo, da se določene p-celice res nahajajo na mestih (glede na parnost i in j), kot smo opisali malo prej.

2.3 Implementacija

Nekajkrat smo že omenili, da je podatkovna struktura matrika, zgrajena iz p-celic. Ker imajo v osnovi vse celice, ne glede na dimenzijo, večino enakih lastnosti, smo skonstruirali razred *Cell*, ki vsebuje skupne lastnosti vseh p-celic. Najpomembnejše takšne lastnosti so zagotovo X in Y ter *CellType*, ki poleg tipa hkrati povedo tudi dimenzijo celice, sledijo pa še takšne, ki so nam v pomoč pri izvedbi algoritmov.

Razred *Cell* poleg opisanih lastnosti vsebuje tudi pomožne in virtualne metode. Virtualne metode definirajo strukturo in so v dedovanih razredih ustrezno redefinirane v odvisnosti od dimenzije celice. Med slednje štejemo npr. *IsFace*, *IsCoface* ipd. Kakšen je njihov pomen in zakaj ta razred potrebuje vmesnik *IComparable*, bomo razložili pri predstavitvi algoritmov.

S tako definiranim razredom *Cell* smo na enostaven način izpeljali podrazrede, imenovane po p-celicah. To so *Vertex*, *Edge* in *Square*. Predvsem smo

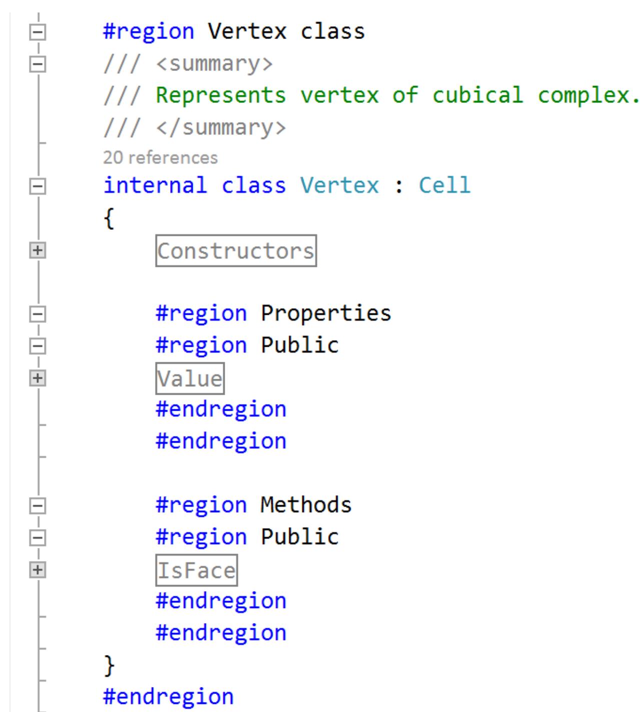


Slika 2.4: Okvirni prikaz implementacije nadrazreda *Cell*, iz katerega so izpeljane vse p-celice.

s takšnim načinom implementacije omogočili, da lahko zgradimo matriko, ki bo vsebovala vse omenjene izpeljane objekte.

Ena izmed glavnih posebnosti oz. razlik med izpeljanimi razredi je, da vsak od njih vsebuje tabelo različne velikosti, ki hrani unikatne sivinske vrednosti slikovnih elementov, urejene v padajočem vrstnem redu. Tako npr. podrazred *Vertex* vsebuje le eno vrednost, *Edge* dve in *Square* štiri vrednosti. Število vrednosti je namreč odvisno od števila oglišč, ki definirajo posamezno p-celico. Izpeljane razrede oz. njihove lastnosti in redefinirane metode

(v primerjavi z nadrazredom) lahko vidimo na slikah 2.5 in 2.6.


```
#region Vertex class
/// <summary>
/// Represents vertex of cubical complex.
/// </summary>
20 references
internal class Vertex : Cell
{
    Constructors

    #region Properties
    #region Public
    Value
    #endregion
    #endregion

    #region Methods
    #region Public
    IsFace
    #endregion
    #endregion
}
#endregion
```

Slika 2.5: Struktura izpeljanega razreda *Vertex*.

Z implementacijo omenjenih razredov smo pripravili vse potrebno za predstavitev kubičnega kompleksa v računalniku. Tega zgradimo na podlagi podatkovne strukture, izpeljane iz razreda *Matrix* in jo zapolnimo z objekti tipa *Cell*. Način implementacije je prikazan na sliki 2.7. Pomembno vlogo pri tem igra metoda *BuildCubicalComplex2D*, ki poskrbi za izgradnjo kubičnega kompleksa z vsemi pripadajočimi p-celicami. V resnici nam te metode ni potrebno posebej klicati, saj se pokliče že pri kreiranju objekta. Za to poskrbi konstruktor razreda, ki najprej določi velikost prostora, nato pa pokliče še omenjeno metodo.

```
#region Edge class
/// <summary>
/// Represents edge of cubical complex.
/// </summary>
9 references
internal class Edge : Cell
{
    Constructors

    #region Methods
    #region Public
    IsFace
    #endregion
    #endregion
}
```

```
#region Square class
/// <summary>
/// Represents square of cubical complex.
/// </summary>
7 references
internal class Square : Cell
{
    Constructors

    #region Methods
    #region Public
    IsFace
    #endregion
    #endregion
}
#endregion
```

Slika 2.6: Razreda *Edge* in *Square*. Oba dedujeta iste metode, iz slike pa je razvidno, da je le ena redefinirana glede na dimenzijo.

```
/// <summary>
/// Represents 2D cubical complex.
/// </summary>
8 references
public class CubicalComplex : Matrix<Cell>
{
    #region Constructors
    /// <summary>
    /// Initializes a new instance of the <see cref="CubicalComplex" /> class.
    /// </summary>
    /// <param name="image">The image.</param>
    1 reference
    public CubicalComplex(GrayscaleImage image)
        : base(2 * image.Width - 1, 2 * image.Height - 1)
    {
        BuildCubicalComplex2D(image);
    }
    #endregion
```

Slika 2.7: Kubični kompleks.

Poglavje 3

Diskretno vektorsko polje

Po gradnji kubičnega kompleksa poljubne digitalne sivinske slike bomo prešli na osrednji del diplomskega dela. S to podatkovno strukturo želimo namreč ustvariti diskretno vektorsko polje in hkrati dobiti tudi seznam vseh pripadajočih kritičnih celic. V tem poglavju bomo obravnavali, kaj sta omenjeno vektorsko polje in seznam kritičnih celic, pred tem pa bomo predstavili še nekaj uporabljenih lastnosti pri postopku in zahteve, ki jih moramo izpolniti za pravilnost in enoličnost vrnjenih rezultatov.

Naj bo t prag (ang. *treshold*), ki zavzema neko vrednost z intervala $[0, 255]$ in $g(x)$ funkcija, ki vrne sivinsko vrednost slikovnega elementa x. Potem za vsako p-celico α velja:

$$K_t = \{\alpha \in K \mid g(x) \leq t, \forall x \in \alpha\}. \tag{3.1}$$

V splošnem to pomeni, da K_t vsebuje tiste slikovne elemente, katerih vrednosti sivinskih odtenkov ne presegajo praga t. V primeru, ko iz kompleksa odstranimo oglišče oz. 0-celico, posledično odstranimo tudi vsa *kolica*.

Za sliko P in za vrednost t lahko dopolnimo definicijo moči množice K_t:

$$|K_t| = \begin{cases} 0 & \text{, če t manjši od min. sivinske vr. v } P \\ |K| & \text{, če t večji ali enak max. sivinski vr. v } P \end{cases} \tag{3.2}$$

Ko je $t_1 < t_2$, pravimo, da je K_{t_1} podkompleks (ang. *subcomplex*)

kompleksa K_{t_2}. Slednji namreč vsebuje vse celice prvega in še množico celic, ki je vsebovana v K in je ni v množici K_{t_1}.

Zaporedju vgnezdenih kubičnih kompleksov K_t pravimo filtracija (ang. *filtration*). Zanimajo nas torej topološke spremembe, ki se zgodijo ob naraščanju vrednosti t. Povečanju kompleksa K_{t-1} v stanje K_t pa pravimo ekspanzija (ang. *expansion*). V nadaljevanju bomo videli, da t v resnici nikoli ne naraste do maksimalne vrednosti, tj. 255, čeprav je teoretično tudi to možno. Videli bomo tudi, da praktično že na samem začetku vemo, do katere meje bo t naraščal. Nesmiselno bi bilo namreč graditi kubični kompleks do maksimalne sivinske vrednosti in nato tiste celice, ki presegajo na začetku definiran prag, zavreči. S tem bi lahko na slikah večjih razsežnosti po nepotrebnem povečali časovno in prostorsko zahtevnost.

Za potrebe algoritma, s katerim bomo dobili diskretno vektorsko polje in seznam kritičnih celic, moramo zagotoviti, da so vse vrednosti slikovnih elementov vedno različne. S tem se v prvi vrsti izognemo različnim rezultatom, ki bi jih omenjeni algoritem lahko vračal. To bi se predvsem odražalo pri padanju in naraščanju vrednosti med celicami, ki jih sestavljajo enaki slikovni elementi. Ker verjetnost velikega števila enakih elementov oz. odtenkov sivine raste z velikostjo slike (različnih odtenkov pa je, kot smo že omenili, 256), moramo zagotoviti, da se po perturbaciji vrednosti samih slikovnih elementov ne bodo spremenili za večjo konstanto, saj bi lahko s tem preveč spremenili prvotno sivinsko vrednost in bi to vplivalo na končni rezultat. Poleg tega ne smemo uporabiti takšne perturbacijske funkcije, ki bi ob vsaki ponovitvi vrnila drugačno vrednost.

Naj bo $g(i,j)$ sivinska vrednost slikovnega elementa, ki se nahaja na (i,j)-tem mestu v sliki. Z upoštevanjem vseh omenjenih omejitev in lastnosti definirajmo funkcijo,

$$g'(i,\ j) \ = \ g(i,\ j) + \eta\ \frac{(i\ +\ Ij)}{2IJ}, \tag{3.3}$$

kjer je $\eta\ >\ 0$ konstanta. Kot lahko opazimo, funkcija upošteva koordinate $(i,\ j)$ posameznega slikovnega elementa ter širino in višino slike $(I,\ J)$.

Algoritem bo tako ob vsaki ponovitvi vedno vrnil enako diskretno vektorsko polje, temu pa tudi ustrezen seznam kritičnih celic, saj bo vrednost vsakega elementa, ne glede na ponovitev algoritma, vedno enaka. Kasneje bomo videli, da je za algoritem pomembno, da vedno prebere vrednosti slikovnih elementov v naraščajočem vrstnem redu. Z zgoraj opisano perturbacijsko funkcijo smo dosegli, da bo ta vrstni red ob vsaki ponovitvi enak. Vrednosti slikovnih elementov se po apliciranju te funkcije, za katero po navadi vzamemo za η vrednost iz intervala (0, 1), spremenijo v poljubno decimalno število. Kadar koli lahko to število pretvorimo v prvotno, tako da vzamemo samo celi del tega števila. Prišteta vrednost k vsakemu elementu, kjer je η iz omenjenega intervala, nikoli ne doseže 1 [10].

3.1 Kritične celice

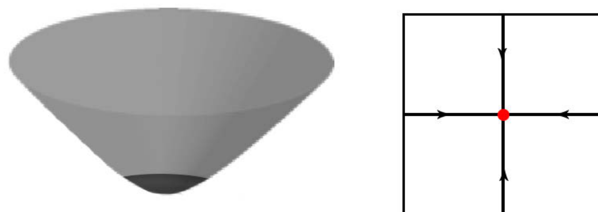
Vsaka celica, ki nastopa v kubičnem kompleksu s podanimi funkcijskimi vrednostmi v ogliščih, je potencialni kandidat, da postane *kritična celica*. V primeru 2-dimenzionalnih slik dobimo natanko tri vrste takšnih celic: *kritična oglišča*, *kritične robove* in *kritične kvadrate*. Na kakšen način pridemo do omenjenih celic, si bomo ogledali v naslednjem razdelku pri algoritmu, sedaj pa si podrobneje oglejmo vrste kritičnih celic in njihov pomen.

Kritično oglišče (*0-celica*)

Oglišče kubičnega kompleksa je kritično natanko tedaj, ko gre za celico, ki predstavlja *lokalni minimum* funkcijskih vrednosti. V splošnem to pomeni, da je oglišče α kritično, če so vse vrednosti sosednjih oglišč večje od vrednosti oglišča α. Za sosednja oglišča štejemo tista oglišča, ki skupaj z ogliščem α tvorijo več-dimenzionalne celice. Spomnimo še, da so vrednosti oglišč posamezne celice različne.

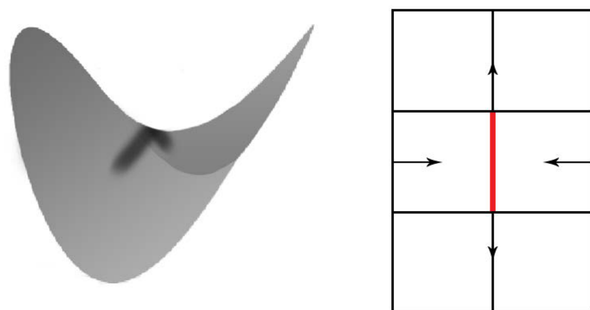
Kritičen rob (*1-celica*)

Kritičen rob v vektorskem polju je v obliki *sedla*, saj v smeri roba, tako v eno kot drugo stran, vrednosti padajo, medtem ko v levo in desno



Slika 3.1: Kritično oglišče – lokalni minimum (levo) in njegova predstavitev v vektorskem polju (desno).

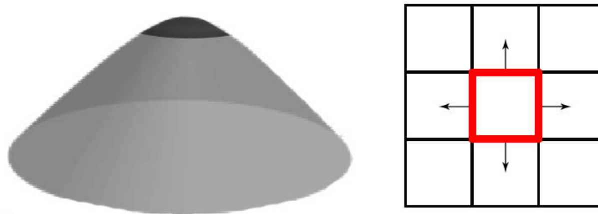
smer vrednosti naraščajo. Kot bomo videli na koncu, nam kombinacija kritičnega roba in oglišča lahko določa objekt, ki ga želimo prešteti.



Slika 3.2: Kritičen rob – sedlo (levo) in njegova predstavitev v vektorskem polju (desno).

Kritičen kvadrat (*2-celica*)

Element kubičnega kompleksa, s katerim lahko določimo, ali gre za iskani objekt na sliki ali ne, imenujemo kritičen kvadrat in ponazarja *lokalni maksimum*. V primeru, da je kritičen kvadrat prisoten v območju, ki ga omejujejo kritični robovi povezanimi s kritičnimi oglišči, to pomeni, da v tistem delu obstaja višje ležeče območje in ne gre za objekt, kakršne štejemo [5, 6].



Slika 3.3: Kritičen kvadrat – lokalni maksimum (levo) in njegova predstavitev v vektorskem polju (desno).

3.2 Algoritem "Process Lower Stars"

Kot lahko opazimo, nam kritične celice prikazujejo ekstremne spremembe na poljubnem lokalnem območju procesirane slike, saj opisujejo najnižja in najvišja področja ter sedla med njimi, kjer v eni dimenziji vrednosti vedno naraščajo, v drugi pa vedno padajo. Pripadajoče vektorsko polje povezuje kritična območja s pomočjo vektorjev tako, da ti kažejo smer padanja vrednosti, pri čemer vektor vedno povezuje nižjo-dimenzionalno celico s celico, ki je za največ eno dimenzijo večja.

Kandidate za kritične celice najdemo s pregledovanjem množice celic $L(x)$, kjer je x oglišče. Množico $L(x)$ imenujemo *spodnja zvezda* (ang. *lower star*) oglišča x in je definirana kot:

$$L(x) = \{ \alpha \in K \mid x \in \alpha \wedge g(x) = \max_{y \in \alpha} g(y) \}. \tag{3.4}$$

V splošnem to pomeni, da množica $L(x)$ vsebuje celico x in vse sosednje (višje-dimenzionalne) celice (v kubičnem kompleksu), katerih del je tudi x in kjer maksimalna vrednost oglišč teh posameznih celic ne presega vrednosti x. Na začetku smo omenili, da razred *Cell* potrebuje vmesnik *IComparable*. Ko imamo v množici $L(x)$ več kot en element, so le-ti urejeni v naraščajočem vrstnem redu. Ker so elementi oz. celice v množici lahko različnih dimenzij, je potrebno natančno definirati, v kakšni relaciji je vsak par celic. Tiste z

manjšo dimenzijo so v tej množici pred celicami z višjo dimenzijo, ko med njima obstaja relacija lice ali kolice. Npr.: za oglišče 8 in rob 85 bi veljal vrstni red $\{8, 85\}$, saj je oglišče 8 lice roba 85.

Pri implementaciji kubičnega kompleksa smo povedali, da so vrednosti oglišč posamezne celice urejene v padajočem vrstnem redu, zato na tem mestu definirajmo še funkcijo $G(\alpha)$, s katero bomo v nadaljevanju to urejenost označevali:

$$\begin{aligned} G(\alpha) &= (g'(x), g'(y_{i_1}), ..., g'(y_{i_k})), \\ &\qquad \text{kjer je } g'(x) > g'(y_{i_1}) > ... > g'(y_{i_k}) \end{aligned} \tag{3.5}$$

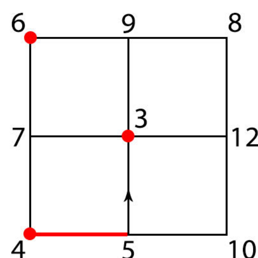
V nadaljevanju bomo potrebovali tudi dve podatkovni strukturi za shranjevanje celic, ki ju imenujemo *prioritetni vrsti*. Že ime nam pove, da bodo elementi (p-celice) razvrščeni po poljubni prioriteti. Ti dve vrsti bomo poimenovali *PQZero* in *PQOne*. Prva bo shramba za celice, ki nimajo prostih lic[1], druga pa za celice, ki imajo na voljo eno prosto lice.

Naš cilj je torej poiskati kritične celice in diskretno vektorsko polje, kar naredimo z algoritmom za procesiranje spodnjih zvezd (ang. *Process Lower Stars*).

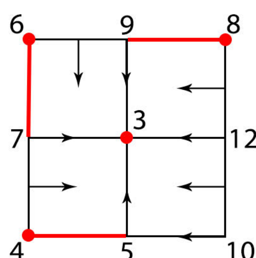
Postopek iskanja poteka tako, da na vsakem koraku poiščemo spodnjo zvezdo, tj. množico celic $L(x)$, za oglišče x kubičnega kompleksa iz naraščajoče urejenega seznama vseh oglišč po sivinski vrednosti slikovnega elementa. Če je $|L(x)| = 1$, potem je x lokalni minimum oz. kritično oglišče, sicer x sparimo s spodnjim robom iz $L(x)$ (tj. rob, ki je prvi v omenjeni množici po G), kar v resnici pomeni, da dodamo nov vektor v vektorsko polje. Preostale celice iz $L(x)$ dodamo v ustrezno prioritetno vrsto. Robove tako vstavimo v *PQZero*, saj nimajo lica, ki ga ne bi bilo na seznamu kritičnih celic ali v vektorskem prostoru, kvadrate pa v *PQOne*, ker se njihovo ustrezno lice zagotovo nahaja v *PQZero*. Za vsak kvadrat nato poiščemo rob, ki (še) ni

[1]Prosta lica so lica, ki že obstajajo na seznamu kritičnih celic ali v vektorskem polju.

kritičen in hkrati ni v paru s katerim od kvadratov v vektorskem polju. Če najdemo takšno kombinacijo, ju sparimo skupaj. Kritični kvadrati, ki ostanejo brez para, postanejo kritične celice. Podobno se zgodi vsem robovom iz $L(x)$, ko ne najdejo svojega para z nobenim kvadratom. Tudi ti postanejo kritične celice [10].



Slika 3.4: Enostaven primer za prikaz postopka procesiranja spodnje zvezde. Oglišča so poimenovana po svojih sivinskih vrednostih $g(x)$, ostale celice pa po $G(x)$ (npr. zgornji levi kvadrat ima oznako 9763). Na sliki lahko opazimo, da smo zadnji korak izvedli za oglišče 6. Z rdečo barvo so označene kritične celice, puščice pa simbolizirajo vektorsko polje. Algoritem nadaljuje z naslednjim korakom za oglišče 7. Najprej poišče spodnjo zvezdo $L(7)$, tj. množica $\{7, 73, 74, 7543, 76\}$, ki je že urejena po $G(x)$. Ker množica vsebuje več kot en element, vemo, da nimamo kritičnega oglišča, pač pa po prej opisanem postopku v vektorsko polje dodamo vektor $7 \rightarrow 73$, vse preostale robove v *PQZero* in kvadrat v *PQOne*. Slednjega nato takoj vzamemo iz vrste in poiščemo ustrezno lice, ki ga najdemo v robu 74, saj ga ni med kritičnimi celicami in v vektorskem polju, zato v vektorsko polje dodamo nov vektor $74 \rightarrow 7543$. Iz množice $L(x)$ tako ostaja le še rob 76, ki nima ustreznega para, zato postane kritičen rob. Končni rezultat je na sliki 3.5.



Slika 3.5: Primer, obravnavan na sliki 3.4, po končanem zadnjem koraku.

3.3 Implementacija algoritma "Process Lower Stars"

Omenjeni algoritem vrne diskretno vektorsko polje in seznam kritičnih celic, implementirali pa smo ga kot svoj razred, imenovan *DiscreteVectorField*. Na sliki 3.6 je prikazano ogrodje celega razreda, v katerem sta tudi seznama, ki nas v resnici najbolj zanimata. To sta *CriticalCells* in *VectorField*. Oba hranita rezultat, ki ga dobimo tako, da zgradimo objekt tipa *DiscreteVectorField*, saj se že s konstruktorjem razreda kliče glavna metoda *ProcessLowerStars*. Da bi samo implementacijo čim bolje razumeli, si oglejmo vsako metodo posebej.

3.3.1 Metoda "LowerStar"

Metoda sprejme na vhodu kubični kompleks in oglišče, za katerega iščemo *spodnjo zvezdo* ter vrne seznam celic, ki vsebujejo vhodno oglišče. Pri tem mora maksimalna vrednost oglišča posamezne celice biti enaka vrednosti vhodnega oglišča. Na podlagi optimizirane implementacije strukture kubičnega kompleksa, ki smo jo predstavili v 2. poglavju, lahko te celice dobimo v trenutku, in sicer tako, da preverimo samo sosednje elemente podanega oglišča. Če se to oglišče nahaja na (i, j)-mestu, potem so kandidati za spodnjo zvezdo na mestih (i, j), $(i-1, j-1)$, $(i-1, j)$, $(i-1, j+1)$, $(i, j-1)$, $(i, j+1)$, $(i+1, j-1)$, $(i+1, j)$ in $(i+1, j+1)$. Seznam vrnjenih celic je urejen v

```
#region DiscreteVectorField class
/// <summary>
/// Represents topological features of 2D grayscale digital images.
/// </summary>
4 references
internal sealed class DiscreteVectorField
{
    Constructors

    #region Properties
    #region Public
    CriticalCells

    VectorField
    #endregion
    #endregion

    #region Methods
    #region Private
    ProcessLowerStars

    LowerStar

    Pair

    UnpairedFaces

    GetNumberOfUnpairedFaces
    #endregion
    #endregion
}
#endregion
```

Slika 3.6: Razred *DiscreteVectorField*.

naraščajočem vrstnem redu.

3.3.2 Metoda "UnpairedFaces"

Kot je mogoče razbrati iz imena metode, vrne metoda "UnpairedFaces" za dano celico vsa lica iz njene spodnje zvezde, ki niso niti na seznamu kritičnih celic niti v vektorskem polju. Velja poudariti, da so vsa lica, ki jih metoda vrne, še vedno urejena glede na G, saj se po sami množici elementov *spodnje zvezde* sprehodimo zaporedno in tako tudi vstavljamo elemente v seznam, ki

ga metoda vrne.

3.3.3 Metoda "Pair"

Pomožna metoda "Pair" vrne prvo celico s seznama, dobljenega od metode *UnpairedFaces*. Ta celica predstavlja edino dosegljivo celico, ki (še) ni v paru z nobeno drugo in hkrati tudi (še) ni kritična.

3.3.4 Metoda "GetNumberOfUnpairedFaces"

Pomožna metoda "GetNumberOfUnpairedFaces" vrne število lic, ki jih kot rezultat dobi od metode *UnpairedFaces*.

3.3.5 Metoda "ProcessLowerStars"

Metoda "ProcessLowerStars" kliče vse doslej omenjene metode. Postopek smo že natančno opisali v prejšnjem razdelku, zato na tem mestu omenimo le posebnosti. Da bi prišli do seznama kritičnih celic in vektorskega polja, potrebujemo kubični kompleks in seznam slikovnih elementov, urejenega po slikovnih vrednostih. Za prioritetni vrsti *PQZero* in *PQOne* smo dodatno implementirali razred *PriorityQueue* (slika 3.7) z vsemi potrebnimi metodami. Za bolj podroben opis postopka pa si oglejmo sledečo psevdokodo.

Algoritem 1: Process Lower Stars

Data: kubični kompleks K, urejeni slikovni elementi P

Result: kritične celice C, diskretno vektorsko polje $V[\alpha^{(p)}] = \beta^{(p+1)}$

```
for p ∈ P do
    x := K[p] (celica p v kubičnem kompleksu)
    if |L(x)| = 1 then
        dodaj x v C;
    else
        δ := prvi rob iz L(x);
        V[x] = δ;
        dodaj preostale robove iz L(x) v PQZero;
        dodaj α ∈ L(x) v PQOne, kjer je α > δ in št. neparnih lic(α)
        = 1;
        while PQOne ≠ ∅ or PQZero ≠ ∅ do
            while PQOne ≠ ∅ do
                α := iz PQOne vzemi celico;
                if št. neparnih lic(α) = 0 then
                    dodaj α v PQZero;
                else
                    V[par od α] := α;
                    odstrani par od α iz PQZero;
                    dodaj β ∈ L(x) v PQOne, kjer je (β > α ali
                    β > par(α)) in št. neparnih lic(β) = 1;
            if PQZero ≠ ∅ then
                γ := iz PQZero vzemi celico;
                dodaj γ v C;
                dodaj β ∈ L(x) v PQOne, kjer je α > γ in št. neparnih
                lic(α) = 1;
```

```
/// <summary>
/// Represents data structure Priority queue.
/// </summary>
/// <typeparam name="T"></typeparam>
6 references
public class PriorityQueue<T> where T : IComparable
{
    Fields

    Constructors

    #region Methods
    #region Public
    Enqueue

    Dequeue

    Peek

    Remove

    RemoveAt

    Count

    IsEmpty
    #endregion
    #endregion
}
```

Slika 3.7: Razred *PriorityQueue*.

V algoritmu se na vsakem koraku večkrat izvede preverjanje, ali določena celica že obstaja v seznamu kritičnih celic ali vektorskem polju, zato smo za njuni strukturi uporabili *HashSet* in *Dictionary*. Glavna prednost teh dveh struktur je predvsem hitro iskanje, ki deluje s pomočjo zgoščevalne funkcije in tako v najslabšem primeru ni potrebno preveriti celotnega seznama oz. polja, s čimer zelo pohitrimo celoten algoritem.

Poglavje 4

Morsov kompleks

Morsov kompleks zajema podatke o ključnih značilnostih funkcije sivinskih vrednosti v sliki, kar ima precej velik pozitiven učinek, saj prihrani predvsem pri prostorski in posledično tudi časovni zahtevnosti, da pridemo do želenih podatkov. Zato pravimo, da gre pri Morsovemu kompleksu pravzaprav za posplošeni kubični kompleks, ki pa je vendarle skonstruiran precej drugače, saj omenjene topološke spremembe hrani v drugačni obliki od kubičnega kompleksa.

Spremembe, o katerih govorimo, se skrivajo v Morsovem verižnem kompleksu (ang. *Morse chain complex*) in seznamu lic (ang. *Facelist*). Prva predstavlja seznam kritičnih celic, medtem ko druga *gradientne poti.* Slednje predstavljajo zaporedje celic dveh sosednjih dimenzij, vzdolž katerih sivinske vrednosti padajo. To pomeni, da za vsako kritično p-celico določijo smer padanja funkcijskih vrednosti proti kritičnim $(p-1)$-celicam. Algoritem, s katerim izluščimo vse te informacije, imenujemo *pridobivanje Morsovega kompleksa* (ang. *Extract Morse Complex*) [10].

V nadaljevanju tega poglavja si bomo ogledali, kako izluščimo Morsov kompleks iz kubičnega s pomočjo rezultatov prejšnjega algoritma. Nato pa sledi še opis implementacije, v katerem bomo predstavili algoritme, ki smo jih pri implementaciji uporabili.

4.1 Algoritem “Extract Morse Complex”

S tem algoritmom torej določimo gradientne poti od višje-dimenzionalnih kritičnih celic do kritičnih celic, ki so za eno dimenzijo manjše. V resnici tako pridemo do parov *kvadrat – rob* in *rob – oglišče*, ki predstavljajo večkrat omenjeni seznam lic.

Sedaj si podrobneje oglejmo algoritem, s katerim dobimo Morsovo verigo in za vsako kritično p-celico seznam njenih kritičnih $(p-1)$-lic. V osnovi algoritem deluje v treh korakih, saj se postopoma premikamo od nižje-dimenzionalnih kritičnih celic do višjih. Na prvem koraku, ko obravnavamo kritična oglišča, vsako oglišče samo dodamo v Morsovo verigo. Ko se sprehodimo čez kritične robove in kvadrate, pa je stvar bolj zapletena. Kritičen rob bo namreč preko svojih dveh oglišč vedno povezan z enim ali dvema kritičnima ogliščema, ki ju moramo poiskati. Zato za vsak kritičen rob najprej poiščemo oglišča, ki predstavljajo kandidate, s katerimi bi rob lahko bil povezan. V primeru, da izbrano oglišče obstaja v vektorskem polju, ga dodamo v prioritetno vrsto *Qbfs*, saj to ni kritično. Takšno poimenovanje vrste izvira iz dejstva, da v resnici za konstruiranje seznama lic uporabimo princip iskanja v širino (ang. *Breadth-first search algorithm).* Če oglišča ne najdemo v vektorskem polju, to pomeni, da je to oglišče tudi kritično in kombinacijo dodamo v seznam lic. Vsa preostala oglišča, ki se nahajajo v *Qbfs*, obravnavamo posebej in za vsakega od njih poiščemo rob, po katerem funkcijska vrednost pada. Če nobeno od oglišč najdenega roba še vedno ni kritično, ponovimo postopek. To počnemo toliko časa, dokler ne najdemo kritičnega oglišča in takrat ponovno dodamo kombinacijo kritičnega roba in najdenega kritičnega oglišča v seznam lic.

Pri kritičnih kvadratih je postopek povsem enak, le da celoten algoritem izvedemo za eno dimenzijo višje in iščemo ujemanja med kritičnimi robovi in kvadrati.

Če pri kritičnih robovih točno vemo, koliko oglišč mu pripada, pa to ne

velja za kritične kvadrate. Za slednje ne moremo natančno napovedati, s koliko kritičnimi robovi so povezani [10].

V nadaljevanju si oglejmo, kako smo implementirali algoritem in katere strukture smo pri tem uporabili.

4.2 Implementacija algoritma "Extract Morse Complex"

Podobno kot za kubični kompleks smo tudi za Morsovega ustvarili svoj razred, imenovan *MorseComplex*. Osnovna ideja je, da s kreiranjem objekta slednjega že v ozadju zgradimo kubični kompleks ter dobimo seznam kritičnih celic in diskretno vektorsko polje, s čimer se izognemo odvečnemu pisanju kode in tako avtomatiziramo večji del postopka. Kako smo to naredili, bomo pokazali v zaključku tega poglavja, pred tem pa si oglejmo, kako smo dejansko implementirali algoritem. Sama implementacija je precej podobna psevdokodi, zato si na njej oglejmo podrobnosti.

Algoritem 2: Process Lower Stars

Data: kubični kompleks K, kritične celice C, vektorsko polje V

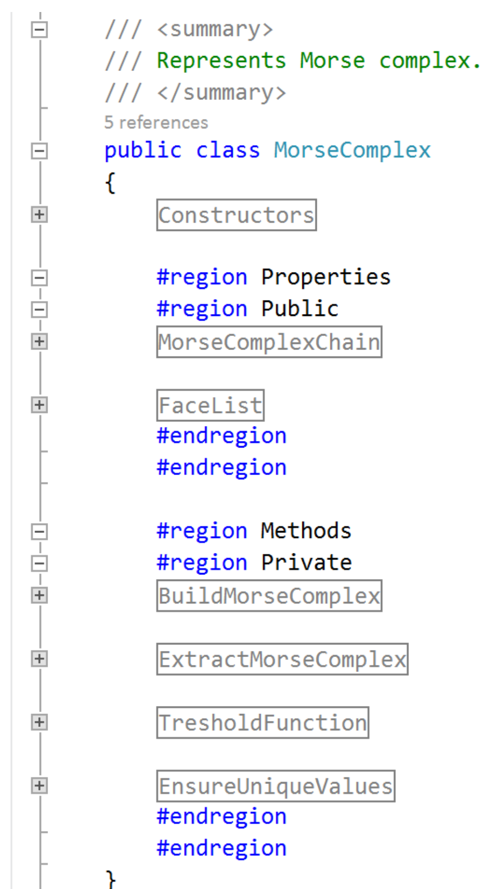
Result: Morsova veriga M, seznam lic $FaceList$

for $p \in \{0, 1, 2\}$ **do**
- **for** $\gamma^{(p)} \in C$ **do**
  - dodaj γ v M;
  - **if** $p > 0$ **then**
    - **for** $\alpha^{(p-1)} < \gamma^{(p)}$ **do**
      - **if** $V[\alpha] \neq \emptyset$ **then**
        - dodaj α v *Qbfs*;
      - **else**
        - $FaceList[\gamma] = \alpha$
    - **while** *Qbfs* $\neq \emptyset$ **do**
      - $\alpha :=$ iz *Qbfs* vzemi celico;
      - $\beta := V[\alpha]$;
      - **for** $\delta^{(p-1)} < \beta \wedge \delta \neq \alpha$ **do**
        - **if** $\delta \in C$ **then**
          - dodaj δ v $FaceList(\gamma)$;
        - **else**
          - dodaj δ v *Qbfs*;

Razred vsebuje dva seznama, ki ju dobimo kot rezultat pravkar opisanega algoritma. To sta *MorseComplexChain* in *FaceList*, njuna implementacija pa predstavlja glavni del te diplomskega dela, v kateri želimo pokazati, kako z najpomembnejšimi informacijami opišemo poljubno sliko in na njih naredimo določeno poizvedbo. Ravno te informacije so namreč shranjene v Morsovi verigi in seznamu lic.

Na sliki 4.1 lahko vidimo, kakšno je ogrodje razreda *MorseComplex*, v

katerem se nahajo tudi 4 metode, ki so bile že opisane v poglavju 3.3, z izjemo metode *BuildMorseComplex*.



Slika 4.1: Razred *MorseComplex*.

Metoda *BuildMorseComplex* je glavna metoda, ki se pokliče pri samem kreiranju objekta *MorseComplex*. Vse, kar je potrebno narediti, da zgradimo Morsov kompleks poljubne digitalne 2-dimenzionalne slike, je, da na vhodu podamo sivinsko sliko in prag, o katerem smo pisali v poglavju 3. Nekaj besed bomo o pragu namenili še v poglavju s primeri, ki sledi.

Vsako vhodno sliko, ki jo preberemo z diska, najprej shranimo kot objekt tipa *RGBImage*, katerega razred smo prav tako implementirali. Omenjeni

```
#region BuildMorseComplex
/// <summary>
/// Builds the Morse complex.
/// </summary>
/// <param name="image">The image.</param>
/// <param name="treshold">The treshold.</param>
1 reference
private void BuildMorseComplex(GrayscaleImage image, int treshold)
{
    // Thresholding
    TresholdFunction(image, treshold);

    // Ensure unique pixel values
    EnsureUniqueValues(image);

    // Build cubical complex
    CubicalComplex complex = new CubicalComplex(image);

    // Remove nulls in array and sort pixels ascending
    // (g(x0) < g(x1) < ... < g(xN))
    IEnumerable<Pixel> pixels = (image.GetArray()).
        Where<Pixel>(x => x != null).OrderBy(x => x.Value);

    // Construct discrete vector field (C, V)
    DiscreteVectorField dvf = new DiscreteVectorField(complex, pixels);

    // Extract Morse complex
    ExtractMorseComplex(complex, dvf.CriticalCells, dvf.VectorField);
}
#endregion
```

Slika 4.2: *BuildMorseComplex* metoda.

razred vsebuje tudi metodo *ConvertToGrayscale*, ki sliko pretvori v sivinsko po formuli, omenjeni na začetku tega diplomskega dela in vrne zahtevan objekt tipa *GrayscaleImage* za metodo *BuildMorseComplex*.

Na sliki 4.2 si lahko pobližje pogledamo, kako izgleda metoda *BuildMorseComplex*, ki zajema celoten postopek za gradnjo Morsovega kompleksa. Najprej (z metodo *TresholdFunction*) apliciramo prag, ki ga na vhodu podamo ob kreiranju objekta *MorseComplex* in z njim posredno v kubičnem kompleksu kreiramo celice, ki z vrednostmi oglišč ne presegajo te pragovne

vrednosti. Nato pred samo gradnjo kubičnega kompleksa z metodo *EnsureUniqueValues* poskrbimo, da so vse vrednosti oglišč, ki so pod pragom, različne. Za potrebe pravilnega delovanja samega algoritma *Process Lower Stars* pripravimo seznam slikovnih elementov, urejenih po naraščajočem vrstnem redu in za tem zgradimo seznam kritičnih celic in diskretno vektorsko polje. V samem zaključku iz seznama kritičnih celic in vektorskega polja, zgradimo še Morsov kompleks.

Poglavje 5

Računanje Bettijevih števil

Z Morsovim kompleksom in Bettijevimi števili, ki so ime dobila po italijanskem matematiku *Enricu Bettiju*, bomo prikazali enega izmed načinov, kako prešteti iskane objekte na sliki. Na koncu bomo pokazali še drug način, ki smo ga prav tako implementirali in poudarili ključno prednost slednjega v tej nalogi. O podrobnostih Bettijevih števil se v tem diplomskem delu ne bomo ukvarjali, pač pa se bomo posvetili le najpomembnejšim dejstvom.

5.1 Robne matrike

Bettijeva števila računamo z *robnimi matrikami* (ang. *Boundary matrices*), ki opisujejo, kako so kritične p-celice v Morsovem verižnem kompleksu povezane s kritičnimi $(p-1)$-celicami. Formalno predstavljajo robne matrike homomorfizme iz vektorskega prostora na $\mathbb{Z}_2$, ki ima za bazo kritične p-celice in ga označimo s C_p, v vektorski prostor C_{p-1}, ki ima za bazo kritične $(p-1)$-celice. Vektorski podprostor v C_p, ki ga robna matrika preslika v 0, imenujemo podprostor *ciklov* in označujemo z Z_p, vektorski podprostor, ki je slika robne matrike iz C_{p+1} v C_p, pa imenujemo podprostor *robov* in označujemo z B_p. Cikli in robovi določajo Bettijeva števila, kot bo razloženo v nadaljevanju poglavja.

Ker se nahajamo v 2-dimenzionalnem prostoru oz. obravnavamo slike iz tega prostora, moramo zgraditi dve takšni matriki. Za potrebe računanja bomo uporabili binarna števila in posledično vse operacije izvajali na binarnih robnih matrikah. Brez kakršnih koli težav bi sicer lahko uporabili tudi števila iz kakšnega drugega obsega $\mathbb{Z}_p$, kjer je p praštevilo.

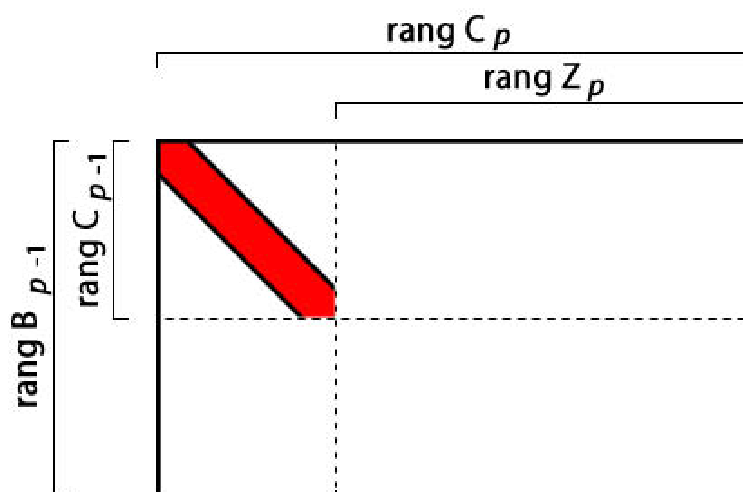
Robni matriki sta pri incializaciji vedno ničelni. Prva vsebuje le 0-celice in 1-celice, torej oglišča in robove, ki jih lahko enostavno dobimo iz Morsovega verižnega kompleksa. Celice z manjšo dimenzijo v matriki vedno predstavljajo vrstice, medtem ko celice z višjo dimenzijo stolpce. Iz seznama lic preberemo vse pare med prej omenjenimi celicami, ki jih obravnavamo v tej prvi matriki, in prištejemo 1 na mestih, kjer obstaja povezava med določenim ogliščem in robom. Ker operacije izvajamo na binarnih matrikah, se lahko zgodi, da bi na poljubnem mestu dvakrat prišteli 1, s čimer bi vrednost nastavili nazaj na 0. Do takšnega primera pridemo, ko imamo npr. dve gradientni poti od kritičnega roba do istega kritičnega oglišča. Na podoben način zgradimo tudi drugo matriko, s to razliko, da ta vsebuje 1-celice (robove) in 2-celice (kvadrate).

Na obeh matrikah nato z Gaussovimi eliminacijami naredimo zgornjo trikotno obliko, tako da dobimo *Smithovo normalno obliko*, kjer imamo samo po diagonali lahko 1, povsod drugod pa 0 (kot kaže slika 5.1).

S sledečo Bettijevo formulo iz ustrezno formiranih matrik enostavno dobimo informacijo o številu objektov na sliki.

$$\beta_p = rang(Z_p) - rang(B_p) \tag{5.1}$$

Število iskanih objektov na sliki, kjer so ti izraženi s svetlejšimi odtenki, poiščemo z izračunom števila β_1, ki vrne število vseh t. i. lukenj v kubičnem kompleksu. V primeru, da bi bili iskani objekti poudarjeni s temnejšimi odtenki na svetlejši podlagi, nam jih ne bi bilo potrebno spreminjati v obratno situacijo. Objekte lahko takrat preštejemo z β_0, saj sta si β_0 in β_1 v resnici inverzni [4].



Slika 5.1: V robni matriki predstavlja obarvano območje vrednost 1, povsod drugod pa so vrednosti enake 0.

Slabost pravkar predstavljenega načina je, da matriki za veliko sliko oz. sliko, kjer je veliko kritičnih celic, zelo hitro naraščata, zato se prej ali slej soočimo z nekaterimi problemi, kot je npr. maksimalna velikost strukture, s katero lahko predstavimo matriko v določenem programskem jeziku ali računalniku nasploh.

5.2 Algoritem krajšanja

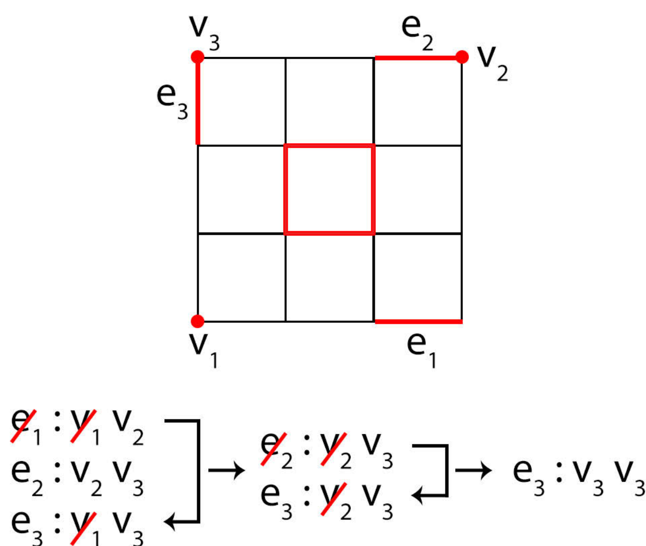
Pri drugem načinu, ki ga bomo predstavili, se prej omenjenim slabostim izognemo. Pri opisu kritičnih celic smo omenili, da kritičen rob in oglišče skupaj zajemata luknjo v kubičnem kompleksu, če se v tem območju ne nahaja kakšen lokalni maksimum, tj. kritičen kvadrat. Na podlagi tega dejstva bomo skonstruirali naslednji algoritem, ki mu pravimo *krajšanje* (ang. *Cancelling*).

Ideja tega algoritma je, da iščemo kritične robove, kjer sta obe oglišči kritičnega roba povezani z enim kritičnim ogliščem. Stanje v seznamu lic je do tega trenutka takšno, da zaradi morebitnega šuma ali kakšnega drugega

dejavnika obstaja več lokalnih minimumov okoli *luknje* v kubičnem kompleksu in posledično tudi sedel. Z omenjenim algoritmom bomo take anomalije odstranili iz seznama lic oz. ga spremenili tako, da bomo dobili želeno obliko, tj. seznam kritičnih robov, kjer obe oglišči posameznega roba kažeta na isto kritično oglišče. V splošnem to pomeni, da dve kritični celici in vse pare celic na gradientni poti med njima nadomestimo z novimi pari, ki zajemajo obe kritični celici.

Splošni korak algoritma deluje tako, da se sprehodimo le po robovih v seznamu lic, kjer za vsak rob velja, da je povezan z natanko dvema ogliščema. To lahko simbolično prikažemo kot $e \rightarrow v_1\ v_2$, kjer je e rob, v_1, v_2 pa oglišči. Pri tem odstranimo rob e in enega od oglišč, npr. v_1, zaradi česar moramo posodobiti celoten seznam, saj odstranjeni celici od tega trenutka ne obstajata več. Povsod drugod, kjer se pojavi odstranjeno oglišče v_1, tega nadomestimo s preostalim, tj. oglišče v_2 [5].

Za lažje razumevanje opisanega postopka si oglejmo sliko 5.2.



Slika 5.2: Primer krajšanja kritičnih celic, ki se pojavijo na zgornji simbolični sliki.

Po krajšanju kritičnih celic imamo rezultat praktično že na dlani, saj preštejemo le št. robov, ki so po krajšanju ostali v seznamu lic in od njih odštejemo št. kritičnih kvadratov. Število slednjih nam pove, koliko lokalnih maksimumov obstaja znotraj posameznega območja, ki ga sestavlja kritičen rob in kritično oglišče. Takšne primere želimo odšteti od končnega rezultata, ker so posledica šuma ali kakšne druge napake. Za lažje razumevanje spomnimo, da po krajšanju za vsak rob velja, da sta obe njegovi oglišči povezani z istim kritičnim ogliščem.

Poglavje 6

Primeri uporabe

Kot smo že v uvodu omenili, bomo v diplomskem delu kot primer uporabe implementiranih algoritmov prikazali štetje svetlih objektov na temni podlagi. Predmete, ki jih želimo prešteti, poskusimo s predprocesiranjem slike (če je to potrebno), čim bolj izpostaviti tako, da ti postanejo karseda svetlejši od ostalih objektov. Drugače povedano, iskani objekti morajo imeti čim višjo, ostali pa čim nižjo sivinsko vrednost. Že tekom naloge smo omenili, da na samem začetku določimo pragovno vrednost z intervala [0, 255]. Namen praga je torej, da z njim naredimo t. i. *luknje* v pripadajočem kubičnem kompleksu in sicer tako, da iz njega izrežemo dele z velikimi sivinskimi vrednostmi. Kakšna naj bo vrednost praga, je seveda precej odvisno od same slike oz. objektov na sliki, od tega pa je na koncu odvisen tudi sam rezultat. V primeru, da bi bil prag prenizek, bi lahko zajeli predmete, ki so neustrezni. Prav tako ne bi bilo nič boljše, če bi bil prag previsok, saj bi s tem lahko izgubili množico relevantih rešitev. Zato je toliko bolj pomembno samo predprocesiranje slik, da z njim objekte, ki jih želimo prešteti, čim bolj poudarimo oz. ločimo od ostalih.

V nadaljevanju si bomo ogledali nekaj primerov s komentarji in statističnimi podatki ter rezultate, ki jih dobimo z implementiranimi algoritmi. Slike, ki smo jih izbrali, predstavljajo nočno nebo, rezultat pa število zvezd z dovolj veliko svetlostjo na njem.

6.1 Primer 1

Za začetek vzemimo najbolj enostavno sliko (slika 6.1), na kateri bomo preverili, ali opisani algoritmi vračajo pravilne rezultate.

Slika 6.1: Demonstracija nočnega neba z umetnimi zvezdami [11].

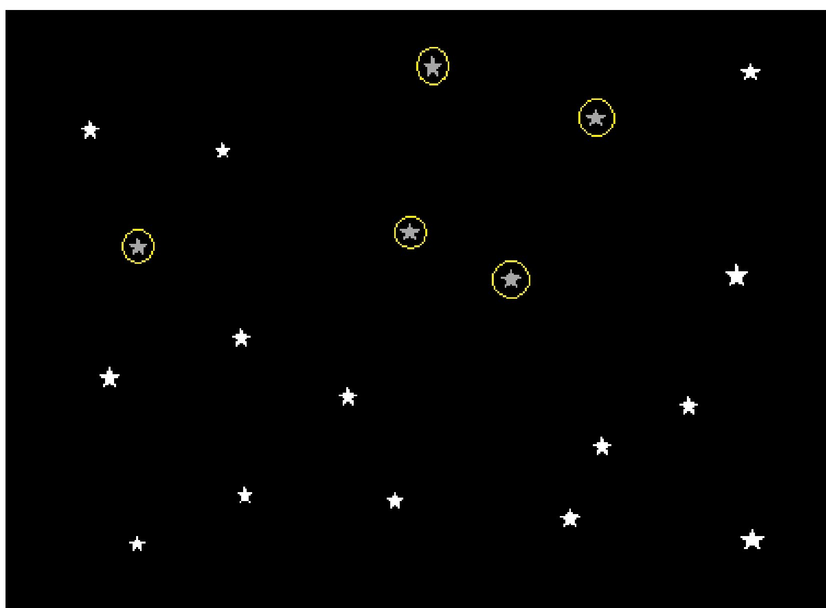
Tabela 6.1: Analiza slike 6.1.

Podatki o sliki	
Velikost slike	259 x 194
Prag	250
Velikost kubičnega kompleksa (po št. celic)	200.014
Velikost k. k. z upoštevanim pragom (po št. celic)	199.796
Velikost Morsovega kompleksa (po št. celic)	80
Št. najdenih zvezd	**19**
Čas poizvedbe	manj kot 1 s

Opazimo, da so po večini vrednosti slikovnih elementov zvezd, morda celo vse, večje od 250, saj smo s to pragovno vrednostjo zajeli vse vidne zvezde na sliki. Posebej zanimiva je razlika v številu celic med kubičnim in Morsovim kompleksom. Najpomembnejše značilnosti, v našem primeru luknje v sliki, ki predstavljajo zvezde, smo opisali z zgolj $0,0004$ % celotnega kubičnega kompleksa in v manj kot 1 sekundi dobili pravilen rezultat. Razlika je v tem primeru še posebej velika, saj je primer res preprost.

6.2 Primer 2

Na drugem primeru smo nekatere zvezde, ki so označene z rumenim krogom, potemnili, s čimer bi radi pokazali, da za dovolj visok prag algoritem teh zvezd ne bo našel, saj jih ne štejemo med iskane objekte na sliki.



Slika 6.2: Demonstracija nočnega neba z nekaj potemnjenimi umetnimi zvezdami.

Tabela 6.2: Analiza slike 6.2.

Podatki o sliki		
Velikost slike	452 x 328	
Prag	150	180
Velikost kubičnega kompleksa (po št. celic)	591.465	
Velikost k. k. z upoštevanim pragom (po št. celic)	587.214	588.349
Velikost Morsovega kompleksa (po št. celic)	170	199
Št. najdenih zvezd	**19**	**14**
Čas poizvedbe	$\sim$ 2 s	

Zopet lahko opazimo občutno razliko med številom celic v kubičnem in Morsovem kompleksu, kar še enkrat dokazuje, da lahko res občutno zmanjšamo prostor oz. lahko *luknje* opišemo s precej manj podatki, preden se lotimo štetja.

6.3 Primer 3

Prva dva primera sta bila namenjena temu, da preverimo, ali algoritmi delujejo pravilno. Zadnji primer (slika 6.3) predstavlja realno sliko neba, pa tudi dimenzija in kompleksnost sta bolj realistični.

Tabela 6.3: Analiza slike 6.3 za dva različna praga.

Podatki o sliki		
Velikost slike	1600 x 1003	
Prag	220	250
Velikost kubičnega kompleksa (po št. celic)	6.413.995	
Velikost k. k. z upoštevanim pragom (po št. celic)	6.392.855	6.409.728
Velikost Morsovega kompleksa (po št. celic)	489.143	489.696
Št. najdenih zvezd	**394**	**186**
Čas poizvedbe	$\sim$ 50 s	

Slika 6.3: Slika nočnega neba [12].

Čeprav se prag razlikuje “le” za 30 odtenkov, pa je razlika v številu najdenih zvezd precej očitna. Ta namreč znaša za več kot faktor 2. Tudi sama velikost slike je bistveno večja kot v prejšnjih primerih, s čimer se je povečal tudi čas, ki smo ga porabili za štetje. Odstotkovno gledano, v tem primeru razlika v številu celic ni tako očitna, vendar pa je še vedno zelo velika. S pragovno vrednostjo 220 je Morsov kompleks “težak” približno $0,08$ % kubičnega kompleksa.

Poglavje 7

Splošne ugotovitve

V diplomskem delu smo spoznali podatkovno strukturo, imenovano kubični kompleks in jo uporabili za opis digitalnih sivinskih slik. Predstavili in implementirali smo tudi algoritma, ki iz danega kubičnega kompleksa zgenerirata novo strukturo. Dobljena struktura se imenuje Morsov kompleks in ima običajno veliko manjšo prostorsko zahtevnost ter omogoča vpogled v to, kako se sivinske vrednosti na sliki spreminjajo. Morsov kompleks je zelo uporabno orodje v topološki analizi. V literaturi [3] obstaja vrsta uspešnih primerov uporabe Morsovih kompleksov pri analizi različnih, lahko tudi zelo velikih, podatkovnih množic. Za konec smo prikazali uporabo Morsovega kompleksa, zgrajenega na digitalni sliki, na treh konkretnih primerih, kjer smo šteli svetle objekte na temni podlagi.

Opisani algoritmi delujejo na 2-dimenzionalnih slikah, čeprav bi jih brez večjih sprememb lahko uporabili tudi za slike višjih dimenzij, z izjemo algoritma za krajšanje, ki bi ga morali prilagoditi višjim dimenzijam. Večje spremembe pa bi morali narediti na podatkovnih strukturah, saj te omogočajo shranjevanje samo za podatke, predstavljene v dveh dimenzijah.

Glavni prispevek diplomskega dela je učinkovitost implementiranih podatkovnih struktur in algoritmov. Predvsem način implementacije prvih omogoča, da je porabljen čas za izvedbo algoritmov izdatno manjši, vendar na račun prostora. Teoretična prostorska zahtevnost kljub temu ostane

enaka tisti iz prvotne slike. Gre le za razliko v konstanti, ki jo pri tovrstnem računanju zanemarimo. Veliko časa smo pridobili tudi pri štetju objektov, saj smo pokazali, kako se lahko izognemo ogromnim matrikam in izvajanju operacij na njih.

Uspešnost implementiranih algoritmov na prikazanih primerih štetja objektov na sliki je odvisna od več dejavnikov. Najpomembnejša sta zagotovo kompleksnost vhodne slike in določitev pragovne vrednosti. V resnici niti ni nujno, da je slika kompleksna v smislu same prepletenosti sivinskih odtenkov ali pa količine vseh (tudi neiskanih) objektov na sliki. Dovolj je, da se dva objekta delno prikrivata, saj tega algoritmi ne prepoznajo. V veliki meri na rezultat vpliva tudi izbrana pragovna vrednost. Eden od izzivov za nadaljnje delo je poiskati dobre avtomatične metode za določanje smiselnega praga.

Problem delnega prekrivanja nam lahko koristi kot dobra motivacija za nadgradnjo diplomskega dela. Torej, kako prepoznati, da sta na sliki npr. dva objekta, ki se prekrivata, in pravilno prešteti število objektov tudi v tem primeru. Vemo pa, da idealnega algoritma ni, zato pa lahko vedno stremimo k čim boljšemu približku.

Literatura

[1] (2011) J. Sumner, ImageWorsener: Conversion to grayscale. Dostopno na: http://entropymine.com/imageworsener/grayscale

[2] D. Gunther, J. Reininghaus, H. Wagner, I. Hotz, "Efficient Computation of 3D Morse-Smale Complexes and Persistent Homology using Discrete Morse Theory". *The Visual Computer* 28 (10): 959-969 (2012)

[3] A. Gyulassy, P.T. Bremer, B. Hamann, V. Pascucci, "A Practical Approach to Morse-Smale Complex Computation: Scalability and Generality," *IEEE Transactions on Visualization and Computer Graphics*, 14 (6): 1619–1626 (2008).

[4] H. Edelsbrunner, J. Harer, *Computational topology: an introduction*, Providence, RI: American Mathematical Society, 2010.

[5] R. Forman, "A user's guide to discrete Morse theory". *Sém. Lothar. Combin.*, 48 (2002).

[6] R. Forman, "Morse Theory for Cell Complexes". *Adv. Math.*, 134 (1): 90–145 (1998).

[7] (2012) Tanner Helland: Seven grayscale conversion algorithms. Dostopno na: http://www.tannerhelland.com/3643/grayscale-image-algorithm-vb6

[8] Three algorithms for converting color to grayscale. Dostopno na: http://www.johndcook.com/blog/2009/08/24/algorithms-convert-color-grayscale

[9] T. Y. Kong, A. Rosenfeld, *Topological algorithms for digital image processing*, Amsterdam, The Nertherlands: Elsevier Science B.V., 1996.

[10] V. Robins, P.J. Wood, A.P. Sheppard, "Theory and Algorithms for Constructing Discrete Morse Complexes from Grayscale Digital Images". *IEEE Transaction on Pattern Analysis and Machune Intelligence* 33 (8): 1646–1658 (2011).

[11] http://i.imgur.com/PPyCqFp.gif

[12] http://barbarashdwallpapers.com/wp-content/uploads/blogger/ItvV6aL0OjI/T0ONcLZ1LsI/AAAAAAAAZj8/ZuPkM9T1Gck/s1600/Pictures-star-wallpapers-star-wallpaper-stars-backgrounds-06.jpg