Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Přebarvování souvislých oblastí ve snímku

Programování v jazyce C

kiv/pc

Jiří Schödlbauer
jschodlb@students.zcu.cz
A19B0187P
14.01.2022

Obsah

Zadání **3**

1 Analýza úlohy **3**
1.1 PGM obrázek . . . 3
1.2 Uložení dat ze vstupu . . . 3
1.3 Label . . . 4
1.3.1 Množina . . . 4
1.3.2 Tabulka . . . 5
1.3.3 Spojový seznam . . . 6
1.3.4 Druhý průchod . . . 7
1.4 Obarvování . . . 7
1.4.1 Barvy . . . 7
1.4.2 Odstíny šedé . . . 7

2 Popis implementace **7**
2.1 Práce se souborem . . . 7
2.1.1 Načtení vstupu . . . 7
2.1.2 Zápis do souboru . . . 8
2.2 Spojový seznam . . . 8
2.2.1 Přidání uzlu . . . 9
2.2.2 Přidání ekvivalence . . . 9
2.2.3 Získání nejlepší ekvivalence . . . 9
2.2.4 Hledání uzlu . . . 9
2.2.5 Nastavení nejlepších hodnot ekvivalence . . . 9
2.2.6 Uvolnění paměti . . . 10
2.3 Spojový seznam s barvami . . . 10
2.3.1 Přidání uzlu . . . 10
2.3.2 Získání barvy podle labelu . . . 10
2.3.3 Nastavení hodnoty barev . . . 10
2.3.4 Uvolnění paměti . . . 10
2.4 Connected-Component Labeling . . . 11
2.4.1 První průchod . . . 11
2.4.2 Druhý průchod . . . 11
2.4.3 Obarvování . . . 11
2.5 Chyby . . . 12

3 Uživatelská příručka **12**
3.1 Překlad . . . 12
3.2 Spuštění . . . 12
3.3 Chyby . . . 14
3.3.1 Nebinární obrázek . . . 14
3.3.2 Nebyl zadán název souboru . . . 14

Závěr **15**

Zadání

Naprogramujte v ANSI C přenositelnou **konzolovou aplikaci**, která provede v binárním digitálním obrázku (tj. obsahuje jen černé a bílé body) **obarvení souvislých oblastí** pomocí níže uvedeného algoritmu *Connected-Component Labeling* z oboru počítačového vidění. Vaším úkolem je tedy implementace tohoto algoritmu a funkcí rozhraní (tj. načítání a ukládání obrázku, apod.). Odkaz na celé znění zadání naleznete ZDE.

1 Analýza úlohy

1.1 PGM obrázek

Vstupem programu je PGM obrázek. Tento obrázek je reprezentován dvěmi barvami, a to bílou a černou. Celkově má soubor čtyři řádky.

1. znaky 'P' a '5'
2. šířka a výška obrázku
3. nejvyšší hodnota barvy - 255 pro bílou
4. jednotlivé černé a bílé pixely obrázku

První tři řádky tedy informují o tvaru PGM souboru a čtvrtý řádek obsahuje pixely obrázku. Pixely jsou reprezentovány znaky. V programu se načítají jako `unsigned char`.

1.2 Uložení dat ze vstupu

K uložení dat do paměti může být použita struktura, která by vypadala například následovně.

```
struct PGM_image {
    char magic_number[3]; /* znaky 'P' a '5' */
    int width; /* šířka obrázku */
    int height; /* výška obrázku */
    unsigned char *pixels; /* jednotlivé pixely obrázku */
};
```

Zdrojový kód 1: Příklad struktury s daty PGM souboru.

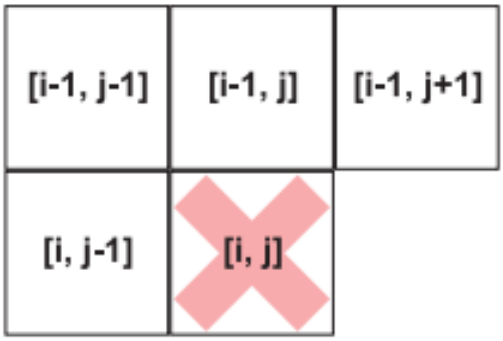
Ačkoliv je toto řešení ukládání dat ze vstupu nejlepší, napadlo mě to až ke konci implementace, proto jsem ho nevyužil. Místo ukládání s pomocí struktury jsem každou proměnnou uložil zvlášť.

K uložení jednotlivých pixelů je možno využít dvourozměrné nebo jednorozměrné pole. Dvourozměrné pole je ale oproti jednorozměrnému zbytečně složité. Musela by se přiřadit paměť k poli polí a poté ke každému poli zvlášť. Přístup k tomuto poli by byl například `pole[y][x]`.

U jednorozměrného se přiřadí paměť jen k jednomu poli. K poli přiřadí paměť o velikosti `výška × šířka × sizeof(unsigned char)`. Jelikož je pole jednorozměrné a my se potřebujeme dostat k dvourozměrným datům, je třeba vytvořit si funkci pro výpočet indexu. Tato funkce bude vracet hodnotu `řádek × šířka + sloupec`.

1.3 Label

Algoritmus `CCL` prochází jednotlivé pixely popořadě od prvního k poslednímu. Jestliže je pixel černý, ignoruje se a pokračuje se na další pixel. Pokud okolní pixely jsou pouze černé, přiřadí se ke zkoumanému pixelu nový label. Pokud jsou okolo již jiné pixely s daným labelem, přiřadí se ke zkoumanému pixelu label jednoho z okolních pixelů, viz zadání.



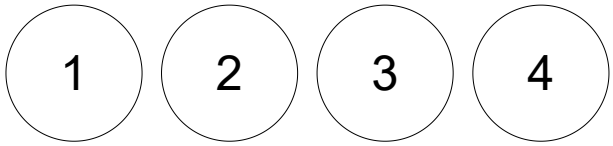
Obrázek 1: Zkoumané pixely.

Labely potřebujeme uložit do pole. Toto pole jsem pojmenoval `mask`, protože jsem si z počátku myslel, že se tak má jmenovat. Toto pole je opět jednorozměrné a je podobné poli `pixels`, s tím rozdílem, že bílé barvy jsou zaměněné za labely. V poli je pouze informace o jaký label se jedná a to jeho index. Pro samotné uložení labelů je třeba použít jinou datovou strukturu.

1.3.1 Množina

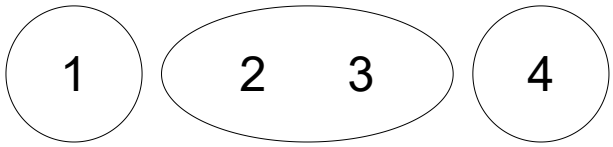
Při prvním průchodu `CCL` algoritmu se mimo jiné zkoumá, zda jsou body ekvivalentní. Pokud je zkoumaný pixel bílý a okolní pixely též, přiřadí se ke zkoumanému pixelu nejnižší label. Další body jsou poté k tomuto labelu ekvivalentní. Přidáme je tedy do ekvivalence. Label by se mohl do ekvivalence vkládat každý. Rychlejší a stále funkční je však přidávat jen label s nejvyšší hodnotou.

Ekvivalenci si jde lépe představit pomocí množin. Jedna množina bude obsahovat labely, které jsou k sobě ekvivalentní. Toto je následně předvedeno na příkladu. `CCL` postupně prochází pixely a najde čtyři různé labely. Každý label tedy dostane vlastní množinu.



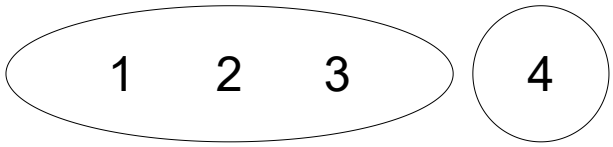
Obrázek 2: Body nejsou v ekvivalenci.

Program pokračuje ve zkoumání pixelů a zjistí, že pixel s labelem 2 je vedle pixelu s labelem 3. Množina, která obsahuje label 2 se tedy sloučí s množinou obsahující label 3.



Obrázek 3: Našla se ekvivalence mezi body 2 a 3.

Algoritmus později zjistí, že pixel s labelem 1 sousedí s pixelem s labelem 3. Znamená to tedy, že tyto dva labely jsou ekvivalentní. Množinu, která obsahuje label 1 tedy sloučíme s množinou obsahující label 3.

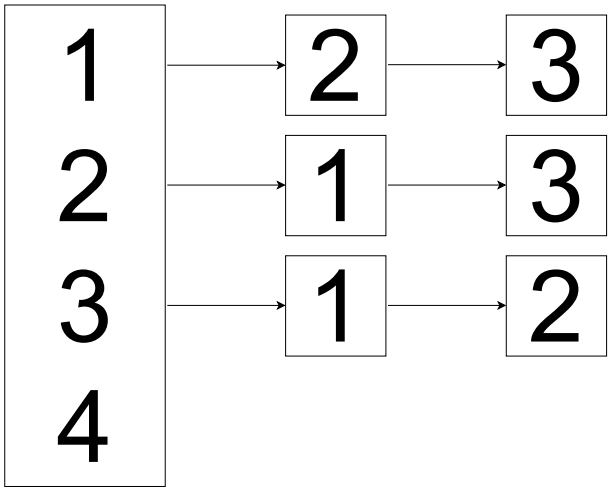


Obrázek 4: Bod 1, 2 a 3 jsou v ekvivalenci, bod 4 nikoliv.

`CCL` je dvouprůchodový algoritmus. V prvním průchodu se k jednotlivým pixelům přiřadí labely a mezi těmito labely se nastavuje ekvivalence, tedy postupně slučuje množiny. Druhý průchod jednotlivé labely zkoumá a nahrazuje je za nejnižší labely z množiny. Když se tedy při druhém průchodu dojde na pixel s labelem 3, nastaví se mu nejnižší label z jeho množiny a to je label 1.

1.3.2 Tabulka

Předchozí příklad by bylo možné implementovat pomocí datové struktury tabulka. Jednotlivé složky tabulky by ukazovaly na labely, se kterými jsou v ekvivalenci.



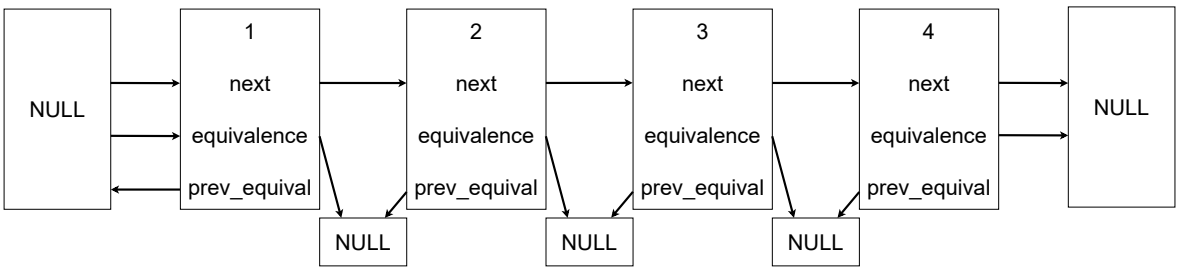
Obrázek 5: Příklad tabulky

V tabulce je znázorněn příklad z množiny. Label 1 je v ekvivalenci s labelem 2 a 3. Tyto labely jsou zase ekvivalentní s labelem 1. U tabulky bude neefektivní přidávání labelů do ekvivalence. Pokud bychom chtěli mezi labelem 1 a 2 nastavit ekvivalenci, museli bychom k labelu 1 přidat labely 2 a 3. A k labelům 2 a 3 přidat label 1.

1.3.3 Spojový seznam

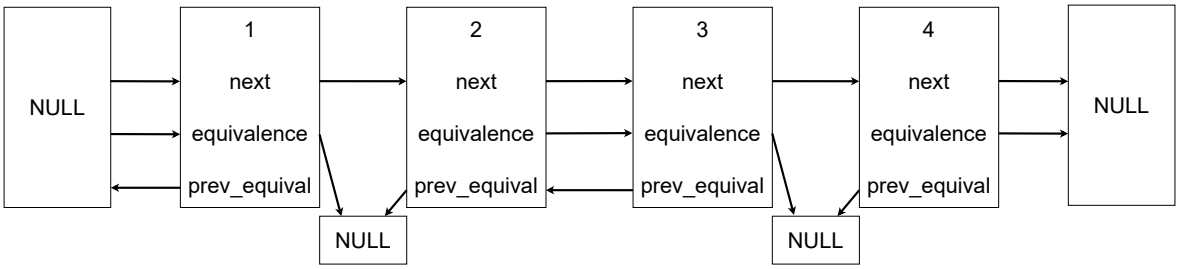
Spojový seznam je struktura, jenž je využita v programu. Spojový seznam se skládá z jednotlivých na sebe ukazujících bodů. Můžeme mít více druhů spojových seznamů. S přidáváním na konec, nebo s přidáváním na začátek. Spojový seznam může být dále například oboustranný, nebo jednostranný.

Pro řešení problému s ukládáním labelů může být použit dvojitý spojový seznam. Uzel tohoto seznamu bude obsahovat hodnotu labelu, ukazatel na další label a ukazatel na ekvivalenci. Tento ukazatel je oboustranný. Můžeme se s ním tedy pohybovat na obě strany a není třeba mít stále někde uložen jeho začátek. Použijeme tedy znovu minulý příklad pro lepší představu.



Obrázek 6: Spojový seznam s hodnotami 1, 2, 3 a 4, kde žádný bod není s žádným ekvivalentní.

Do spojového seznamu se postupně přidávají labely. Pokud je seznam plný, přidá se nový uzel na počátek spojového seznamu a k ukazatelům se přiřadí hodnota `NULL`, jelikož zatím není s žádným dalším bodem v ekvivalenci a žádný další bod se nepřidal. Přidá-li se nový uzel do neprázdného seznamu, dojdeme pomocí ukazatele `next` na poslední uzel. Ukazatel `next` posledního uzlu ukazuje na hodnotu `NULL`. Do ukazatele `next` se tedy uloží přidávaný uzel. Přidaný uzel má opět své ukazatele `NULL`, protože je nejnověji přidaný.



Obrázek 7: Spojový seznam po přidání ekvivalence labelů 2 a 3.

Pro přidání ekvivalence mezi body 2 a 3 musíme nastavit jejich ukazatele ekvivalence, aby ukazovaly na sebe. Tedy aby `equivalence` uzlu 2 ukazovala na label 3 a `pre_equivalence` ukazoval na label 2. Jestliže bude mít label 2 delší seznam s ekvivalencí, musíme dojít na konec tohoto seznamu. Podobně u labelu 3, pokud má nějaké uzly na ukazovateli `prev_equivalence`, bude muset dojít na počátek svého seznamu ekvivalence. Tyto akce musíme provést, abychom neztratili žádné body ze seznamu ekvivalence. Připojíme takto počátek seznamu ekvivalence jednoho uzlu na konec ekvivalence druhého uzlu. Takto můžeme prohodit uzel 2 a 3 a dojít na počátek seznamu ekvivalence bodu 2 a konec seznamu ekvivalence bodu 3.

1.3.4 Druhý průchod

Při prvním průchodu se tedy pomocí spojového seznamu postupně ukládají labely a nastavují se mezi nimi ekvivalence. Druhý průchod pro každý pixel, který není černý, vyhodnotí nejnižší hodnotu v jeho seznamu ekvivalence a zapíše jí na své místo ve výstupním poli. Je tedy třeba do spojového seznamu přidat prvek nejlepší ekvivalence. Před spuštěním druhého průchodu se pro každý uzel spojového seznamu s labely spustí vyhodnocování nejlepší hodnoty ekvivalence a nejlepší hodnota se zapíše do prvku nejlepší ekvivalence. Poté se prochází celé výstupní pole a každému se přiřadí podle zapsaného labelu jeho nejlepší ekvivalence, která je uložená právě v této proměnné.

Původně bylo implementováno jiné pomalejší řešení bez tohoto prvku. Program při každém zkoumání pixelu hledal nejlepší ekvivalenci labelu. To, co se vyhodnocuje pro každý uzel, se vyhodnocovalo pro každý pixel.

1.4 Obarvování

Nyní by tedy mělo být připraveno pole a spojový seznam s labely. V poli jsou pixely reprezentovány pixely. Ve spojovém seznamu jsou labely mezi sebou propojené ekvivalencí. Jednotlivé uzly ve spojovém seznamu však neobsahují informaci o držené barvě. Tento fakt není zaznamenán ani ve zmiňovaném poli. V tomto poli jsou pouze labely a nuly (reprezentující černou barvu).

1.4.1 Barvy

Barvy ve vstupním obrázku jsou pouze bílé a černé. Ve výstupním obrázku černé pixely zůstanou černými. Bílé pixely se budou barvit. Těmto pixelům bude přiřazován různý odstín šedé, tedy jeho jas. Podle odstínů šedé poté můžeme rozeznat jednotlivé objekty. Minimální hodnota barvy je 1, protože 0 je černá, což je pozadí. Maximální barva je 255 (zaznamenáno v pgm souboru).

1.4.2 Odstíny šedé

Odstíny šedé, tedy hodnoty jednotlivých barev jsou v intervalu 1 až 255. Je třeba zjistit rozdíl jednotlivých hodnot. Nejjednoduším řešením by bylo nastavit rozdíl mezi hodnotami na 1. Program by fungoval jak měl a počítač by zvládl zjistit kolik různých objektů se v obrázku nachází.

Složitějším řešením bude určit rozdíl mezi barvami rovnoměrně tak, aby hodnoty byly vždy v intervalu 1 až 255. K tomu je třeba zjistit, kolik různých barev se ve výsledném obrázku bude nacházet. K této hodnotě se dostaneme spočítáním počtu různých labelů ve výstupním poli, mimo hodnoty 0, která znázorňuje černou barvu. Poté vydělíme nejvyšší možnou hodnotu barvy počtem barev a získáme tím rozdíl mezi jednotlivými barvami. Pokud tedy budeme mít labely 1, 2, 3 a 4, jejich počet bude 4. Hodnotu 255 vydělíme tímto počtem a získáme rozdíl 63. První odstín šedé je tedy 63 a další je o 63 vyšší a takto se hodnota zvyšuje až do hodnoty 252.

2 Popis implementace

2.1 Práce se souborem

Práce se soubory se nachází v souboru `main.c`.

2.1.1 Načtení vstupu

O otevření vstupního souboru se stará funkce `load_file()`, které jsou předávány argumenty zadané v konzoli (proměnná `FILE` je mezi globálními proměnnými). Tato funkce nejdříve kontroluje, zda jsou zadány potřebné argumenty. Pomocí pomocné funkce `check_pgm_suffix()` zjistí, jestli se na konci názvu souboru nachází přípona `.pgm`. Pokud se přípona na konci nenachází, přidá jí pomocí funkce `add_pgm_suffix()`.

Čtení ze souboru zajišťuje funkce `read_from_file()`. Funkci jsou předáváné reference na hodnoty souboru, které pak jsou dále využívány. Načte základní informace o `.pgm` souboru a zapíše je do argumentů funkce. Jednotlivé pixely poté načte a zároveň zkontroluje, jestli souboru obsahuje pouze černé a bílé pixely. Jednotlivé pixely ukládá do globální proměnné `unsigned char pixels`.

2.1.2 Zápis do souboru

Funkce `write_into_file()` zajišťuje zápis do souboru. Obsahuje následující parametry:

- **char *argv[]**: obsahuje název výstupního souboru
- **int *mask**: pixely výstupního obrázku
- **char *magic_number**: znaky 'P' a '5'
- **int width**: šířka obrázku
- **int height**: výška obrázku
- **int max_value**: hodnota bílé barvy

Nejdříve se načte název souboru do proměnné `char *file_name`. Zkontroluje se pomocí `check_pgm_suffix`, zda má příponu `.pgm`. Pokud jí nemá, zavolá se funkce `add_pgm_suffix()`, která příponu přidá. Pokusí se otevřít souboru pro čtení. Pokud se soubor úspěšně otevřel, můžeme do něj vpisovat. Na první tři řádky zapíšeme základní informace o souboru. To jsou `*magic_number`, `width`, `height` a `max_value`. Poté se jednotlivé pixely z pole `mask` zapíšou na další řádku. Jakmile je hotový zápis, soubor se zavře.

2.2 Spojový seznam

Spojový seznam pro labely se nachází v souboru `linked_list.c`. Obsahuje všechny potřebné funkce pro řešení tohoto problému. Do spojového seznamu se přidává na začátek.

```
typedef struct thenode {
    int value;
    struct thenode *next;
    struct thenode *equivalence;
    struct thenode *prev_equivalence;
    int best_equivalence;
} node;
```

Zdrojový kód 2: Struktura uzlu spojového seznamu.

- **value**: hodnota, resp. index labelu
- **next**: ukazatel na další uzel ve spojovém seznamu
- **equivalence**: ukazatel na další uzel ve spojovém seznamu s ekvivalencí
- **prev_equivalence**: ukazatel na předešlý uzel ve spojovém seznamu s ekvivalencí
- **best_equivalence**: nejnižší hodnota ve spojovém seznamu s ekvivalencí

2.2.1 Přidání uzlu

Přidání uzlu do spojového seznamu zařizuje funkce `add_node(node **head, int *value)`. Jejími parametry jsou ukazatele na první prvek a na hodnotu labelu. Ukazatel na hodnotu se předává, abychom ho mohli zvýšit přímo ve funkci. Funkce vytvoří nový uzel, uloží do něj předávanou hodnotu a ukazatele nastaví na `NULL`. Prvek `best_equivalence` nastaví na předávanou hodnotu, protože zatím není s žádným prvkem ekvivalentní. Pokud ukazatel na první prvek ukazuje na `NULL`, spojový seznam je prázdný a nový uzel přiřadíme k ukazateli na první prvek. Pokud je seznam již nějak zaplněný, musí se nejdříve nastavit ukazatel `next` nového uzlu, aby ukazoval na první prvek, resp. teď už druhý, a ukazatel na první prvek nastavíme aby na tento uzel ukazoval.

2.2.2 Přidání ekvivalence

Funkce `add_equivalence(node *node1, node *node2)` přidá přidá ekvivalenci mezi uzlem `node1` a uzlem `node2`. Nejdříve se s pomocnou proměnnou `node *last_node1` dostane na poslední uzel ve spojovém seznamu ekvivalence uzlu `node1`. Poté se pomocí `node *first_node2` dostane na začátek spojového seznamu ekvivalence uzlu `node2`.

Nyní se ve dvojitém cyklu zkoumá, zda už tato ekvivalence není zjištěna. Tento cyklus používá pomocné proměnné `node *walk1`, pro průchod uzlu `node1`, a `node *walk2`, pro průchod uzlu `node2`. Vnější cyklus prochází spojový seznam ekvivalence uzlu `node2` a vnitřní napodobně pro uzel `node1`. Zde si je třeba uvědomit, že do proměnné `walk1` se přiřadí uzel `last_node1`. Takže se při průchodu jde pozadu na začátek spojového seznamu. Naopak u `walk2`, kde se prochází naopak na konec spojového seznamu. Ve vnitřním cyklu je pak podmínka, zkoumající, zda se dvě hodnoty nerovnají a jestli už tedy tato ekvivalence není zjištěná.

Pokud ekvivalence není zjištěná, musí se přidat. K tomuto se právě hodí proměnné `first_node1` a `last_node2`. Zde nastavíme příslušné ukazatele, aby tyto uzly ukazovaly na sebe.

```
last_node1->equivalence = first_node2;
first_node2->prev_equivalence = last_node1;
```

Zdrojový kód 3: Přidání ekvivalence.

2.2.3 Získání nejlepší ekvivalence

Funkce `int get_equivalence(node *examined_node)` prochází celý spojový seznam s ekvivalencí uzlu `examined_node`. Zde si je třeba uvědomit, že spojový seznam může mít hodnoty po obou stranách. Proto pomocí pomocné proměnné `node* walk` prochází spojový seznam nejdříve na konec spojového seznamu a poté podobně na počátek spojového seznamu. Při procházení si zaznamenává nejnižší hodnotu, kterou nakonec vrátí.

2.2.4 Hledání uzlu

Pro algoritmus `CCL` je třeba implementace funkce pro nalezení uzlu podle jeho hodnoty. K tomuto účelu slouží funkce `node *get_node(node *head, int value)`. Pomocí proměnné `node *walk` projde celý spojový seznam a vrátí hledaný uzel.

2.2.5 Nastavení nejlepších hodnot ekvivalence

Funkce `set_best_equivalence(node *head)` zajišťuje nastavení prvku `best_equivalence` u všech uzlů spojového seznamu. Pomocí `node *walk` prochází celý spojový seznam a u každého uzlu zavolá funkci `get_node()`. Hodnotu, kterou tato funkce vrátí se uloží do prvku `best_equivalence` daného uzlu.

2.2.6 Uvolnění paměti

Uvolnění paměti zaplněné spojovým seznamem zařizuje funkce `free_list(node *head)`. Tato funkce pomocí `node *walk` projde každý uzel spojového seznamu a použije na něj funkci `free()`. Takto se uvolní celý spojový seznam z paměti.

2.3 Spojový seznam s barvami

Tento spojový seznam se nachází v souboru `colour_linked_list.c`. V tomto seznamu se nachází hodnota barvy a label, ke které je přiřazena.

```
typedef struct the_colour_node {
    int label;
    int value; /* hodnota odstínu šedé */
    struct the_colour_node *next;
} colour_node;
```

Zdrojový kód 4: Struktura uzlu spojového seznamu s barvami.

2.3.1 Přidání uzlu

Přidání uzlu obstarává funkce `add_colour_node(colour_node **head, int label)`. Funkce funguje v podstatě stejně jako u `linked_list.c`. Vytvoří se nový uzel `colour_node *new_colour_node`, do kterého se načtou potřebné informace a to hodnotu labelu. Hodnota barvy se prozatím nastaví na hodnotu labelu. Pokud je tento spojový seznam prázdný, přidáme `new_colour_node` do ukazatele `*head`. Pokud ale spojový seznam již nějaké prvky obsahuje, musíme `new_colour_node->next` nastavit aby ukazoval na první prvek ve spojovém seznamu. Poté ukazatel na první prvek `*head` nastavíme, aby ukazoval na prvek `new_colour_node`.

2.3.2 Získání barvy podle labelu

Funkce `int get_colour(colour_node *head, int label)` zařídí vrácení hodnoty barvy podle zadáného labelu. Pomocí pomocné proměnné `colour_node* walk` procházíme v cyklu while celý spojový seznam, pokud najdeme label rovnající se labelu zadaném jako parametr funkce, vrátíme hodnotu barvy.

2.3.3 Nastavení hodnoty barev

Nyní má každý uzel hodnotu barvy rovnající se hodnotě labelu. Potřebujeme tedy každému uzlu přiřadit správnou barvu. Funkce `set_colours()` barvy přiřadí. Parametry funkce jsou reference na první uzel `colour_node *head`, počet unikátních barev `int unique_colours` a hodnota bílé barvy `int max_value`.

Nejdříve se spočte proměnná `int interval`, která reprezentuje rozdíl mezi jednotlivými barvami. Spočteme jí vydělením hodnoty bílé barvy `max_value` počtem různých barev `unique_colours`. V cyklu poté procházíme pomocí pomocné proměnné `colour_node *walk` všechny uzly. Použita je proměnná `int index`, která násobí `interval` podle toho, u kolikátého uzlu se nacházíme, a tento násobek poté uloží jako hodnotu barvy v daném uzlu.

2.3.4 Uvolnění paměti

Funkce `free_colour_list(colour_node *head)` uvolní paměť zaplněnou spojovým seznamem. Proměnná `colour_node *walk` projde celý seznam a pro každý uzel zavolá funkci `free()` pro uvolnění z paměti.

2.4 Connected-Component Labeling

Algoritmus se nachází v souboru `CCL.c` a spouští se funkcí `run()`. Tato funkce přiřadí paměť výstupnímu poli `int *mask` a provede spuštění funkcí `first_walk_through()` pro první průchod, `second_walk_through()` pro druhý průchod a `paint()` pro obarvení pixelů ve výstupním poli `mask`.

2.4.1 První průchod

Jak už je zmíněno, první průchod zajišťuje funkce `first_walk_through()`. V prvním průchodu je třeba postupně zkoumat každý pixel a jeho sousedy a přidávat mezi nimi ekvivalenci, pokud je to třeba. Okolní body se kontrolují podle šablony na obrázku 1.

Nejdříve se zavolá funkce `first_line()` pro kontrolu prvního řádku obrázku. Protože pro první pixel nejsou žádné okolní pixely, které se musí zkontrolovat, přiřadíme tomuto pixelu ve výstupním poli buď nový label, nebo necháme hodnotu 0, pokud se jedná o černý pixel. U dalších pixelů z prvního řádku zkoumáme pouze pixel, který je nalevo od zkoumaného pixelu. Pokud je pixel černý, necháme mu hodnotu 0. Je-li pixel bílý a jeho soused černý, přiřadíme pixelu nový label. Pokud však sousední pixel má přiřazený label, přiřadí se zkoumanému pixelu tento label. Při zkoumání první řádky není kde nastavovat ekvivalenci mezi body.

Při zkoumání ostatních řádek dochází k tomu podobnému, co bylo u prvního řádku. Kontrola je rozdělena do tří funkcí (`first_pixel_value()`, `middle_pixel_value()` a `last_pixel_value()`. Každá funkce dělá to samé, jen zkoumá jiné sousedy podle polohy zkoumaného pixelu. Tyto funkce se ještě starají o ekvivalenci. Nejdříve se vytvoří pole `integer`ů, do kterého se načtou hodnoty okolních pixelů. Funkce `is_all_black()` poté zjistí zda toto pole neobsahuje pouze hodnoty 0. Pokud jsou všechny hodnoty nulové, přiřadí se pixelu nová hodnota a tato hodnota se přidá do spojového seznamu. Jsou-li sousedé nenuloví, přiřadí se hodnota souseda s nejnižší hodnotou labelu. Dále se vyhodnotí soused s nejvyšší hodnotou a přidá se do ekvivalence pomocí `add_equivalence()` a `get_node()`.

2.4.2 Druhý průchod

Nejdříve se ke každému uzlu ze spojového seznamu labelů přiřadí nejlepší hodnota ekvivalence pomocí funkce `set_best_equivalence()`. Poté se v cyklu každému bodu ve výstupním poli nastaví nejlepší hodnota ekvivalence.

2.4.3 Obarvování

K naplnění pole hodnotami barev slouží funkce `paint(node **head, int max_value, int *mask)`, kde `head` je reference na první prvek ve spojovém seznamu, `max_value` je hodnotou bílé barvy a `mask` je obarvované pole.

Pomocí pomocné proměnné `node *walk` zjistí v cyklu počet unikátních labelů (barev) a postupně tyto labely ukládá do spojového seznamu s barvami. Jaké labely vybrat zjistí podle podmínky, pokud se hodnota labelu rovná nejlepší ekvivalenci daného uzlu, poté se label přidá.

Poté se zavolá funkce `set_colours()`, která jednotlivým uzlům ve spojovém seznamu s barvami nastaví správné hodnoty odstínu šedé tak, aby měli mezi sebou stejný rozdíl.

Nyní je správně nastavený spojový seznam s barvami. Stačí už tedy hodnoty barev přiřadit jednotlivým bodům v poli `mask`. Funkce `paint_mask(colour_node *head, int *mask)` prochází všechny body pole `mask` a místo labelu jim přiřadí barvu, která k danému labelu patří využitím funkce `int get_colour(colour_node *head, int label)` ze souboru `colour_linked_list.c`.

2.5 Chyby

Chybové hlášení jsou v souboru `error.c`. Zde se nacházejí dvě funkce pro chyby nejčastěji se objevované v programu. Jedná se o funkce `sanity_check(char *function_name)`, která napíše, že nastala chyba při `sanity check`, a funkci `malloc_fail(char *function_name)`, která napíše, že nastala chyba při použití funkce `malloc()`. Chybové hlášky se vypíší společně s `function_name`, aby bylo znát, kde k chybě nastalo.

Další chybové hlášení jsou ve funkcích `get_node` v souboru `linked_list.c` a `get_colour` v souboru `colour_linked_list.c`, které hlásí, že nebyl nalezen hledaný label. Dále se v programu nachází chybové hlášky v případě neúspěchu při otevírání souboru.

3 Uživatelská příručka

3.1 Překlad

Program se překládá pomocí přiložených `MakeFilů`. Příkazem `make` se vytvoří spustitelný soubor.

```
C:\Users\jirka\CLionProjects\pc>make
gcc -c -Wall -pedantic -ansi error.c -o error.o
gcc -c -Wall -pedantic -ansi linked_list.c -o linked_list.o
gcc -c -Wall -pedantic -ansi colour_linked_list.c -o colour_linked_list.o
gcc -c -Wall -pedantic -ansi main.c -o main.o
gcc -c -Wall -pedantic -ansi ccl.c -o ccl.o
gcc error.o linked_list.o colour_linked_list.o main.o ccl.o -o ccl.exe
```

Obrázek 8: Překlad programu příkazem make.

3.2 Spuštění

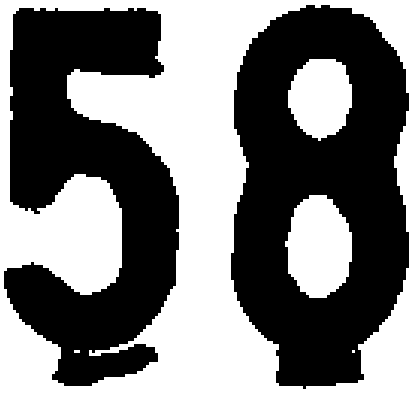
Soubor je možné spustit pomocí příkazu: `ccl.exe <input file> <output file>`
Po zadání tohoto příkazu se program spustí a vypíše informace o běhu.

```
C:\Users\jirka\CLionProjects\pc>ccl.exe D:\w2test.pgm D:\output.pgm
Reading from file...
Running CCL algorithm...
First walk through...
Second walk through...
Painting...
Writing into file...
Everything is done.
Time elapsed: 0.624 seconds
```

Obrázek 9: Spuštění a běh programu.

Obrázek 10: Příklad vstupu.

Obrázek 11: Příklad výstupu.



Obrázek 12: Další příklad vstupu.

Obrázek 13: Další příklad výstupu.

3.3 Chyby

3.3.1 Nebinární obrázek

Pokud zadáme na vstupu je obrázek, který neobsahuje pouze černou a bílou barvu, vypíše se chyba.

```
C:\Users\jirka\CLionProjects\pc>ccl.exe D:\wrong_test.pgm D:\output.pgm
Reading from file...
Incorrect PGM format.
```

Obrázek 14: Spuštění s nebinárním obrázkem.

3.3.2 Nebyl zadán název souboru

```
C:\Users\jirka\CLionProjects\pc>ccl.exe
Reading from file...
File name not entered.
```

Obrázek 15: Chybí názvy výstupního a vstupního souboru.

```
C:\Users\jirka\CLionProjects\pc>ccl.exe D:\w2test.pgm
Reading from file...
File name not entered.
```

Obrázek 16: Chybí název výstupního souboru.

Závěr

Program splňuje zadání. Program na obrázku detekuje objekty a správně je obarví. Program běží rychle, což bude svědčit o tom, že program není implementován špatně. Programovací jazyk C je sám o sobě velmi rychlý. Čas trvání programu byl u obrázku 10 v průměru z deseti měření `0,638 sekundy`. U obrázku 12 byl tento průměr `0,015 sekundy`. Specifikace počítače, na kterém byly časy testovány:

- Windows 10 64-bit
- Intel core i5-7300HQ @ 2,5GHz, jádra: 4
- 8GB DDR4 RAM

Našel by se prostor pro další vylepšení programu. Některé části by se daly naprogramovat efektivněji. Jen si nejsem jistý kde, jaké a hlavně jak. K programu by se mohlo implementovat grafické rozhraní. V programu by se mohla zkusit použít jiná datová struktura než je spojový seznam.

Díky tomuto projektu jsem se naučil programovat v jazyce C. Nyní už bych měl být schopen naprogramovat nějaký složitejší program.