



Semestrální práce z KIV/PC

Přebarvování souvislých oblastí ve snímku

Dominik Zappe
(A20B0279P)

Listopad 2021

Obsah

1	Zadání	1
1.1	Zadání	1
1.2	Algoritmus	2
1.3	Vstupní a výstupní formát	4
2	Analýza úlohy	4
2.1	Vstup	4
2.2	První průchod	4
2.3	Druhý průchod	6
2.4	Výstup	6
3	Popis implementace	6
3.1	pgm.h	6
3.2	disj_set.h	7
3.3	ccl.h	8
3.4	mem.h	9
3.5	main.c	9
4	Uživatelská příručka	9
4.1	Překlad	9
4.2	Spuštění	10
5	Závěr	13

1 Zadání

1.1 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.). Program se bude spouštět příkazem

```
ccl.exe <input-file[.pgm]> <output-file>
```

Symbol <input-file> zastupuje jméno vstupního souboru s binárním obrázkem ve formátu *Portable Gray Map*, přípona souboru nemusí být uvedena; pokud uvedena není, předpokládejte, že má souboru příponu .pgm. Symbol <output-file> zastupuje jméno výstupního souboru s obarveným obrázkem, který vytvoří vaše aplikace. Program tedy může být během testování spuštěn například takto:

```
...\>ccl.exe e:\images\img-inp-01.pgm e:\images\img-res.01.pgm
```

Úkolem vašeho programu tedy je vytvořit výsledný soubor s obarveným obrázkem v uvedeném umístění a s uvedeným jménem. Vstupní i výstupní obrázek bude uložen v souboru ve formátu PGM, který je popsán níže. Obarvení proveďte podle níže uvedeného algoritmu.

Testuje, zda je vstupní obraz skutečně černobílý. Musí obsahovat pouze pixely s hodnotou 0x00 a 0xFF. Pokud tomu tak není, vypíše krátké chybové hlášení (anglicky) a oznámte chybu operačnímu prostředí pomocí nenulového návratového kódu.

Hotovou práci odevzdejte v jediném archivu ZIP prostřednictvím automatického odevzdávacího a validačního systému. Postupujte podle instrukcí uvedených na webu předmětu. Archiv nechť obsahuje všechny zdrojové soubory potřebné k přeložení programu, **makefile** pro Windows i Linux (pro překlad v Linuxu připravte soubor pojmenovaný makefile a pro Windows makefile.win) a dokumentaci ve formátu PDF vytvořenou v typografickém systému $\text{T}_{\text{E}}\text{X}$, resp. $\text{L}_{\text{A}}\text{T}_{\text{E}}\text{X}$. Bude-li některá z částí chybět, kontrolní skript Vaší práci odmítne.

1.2 Algoritmus

Přebarvování souvislých oblastí (*Connected-Component Labeling, CCL*) je dvouprůchodový algoritmus z oblasti počítačového vidění. Jeho vstupem je binární obrázek, tj. takový, který obsahuje jen bílé a černé pixely. Bílé pixely představují jednotlivé objekty, které se tento algoritmus pokouší izolovat a označit různými barvami, resp. hodnotami intenzity šedé barvy (tedy tzv. labels, česky štítky, značkami). Černé pixely pak představují pozadí.

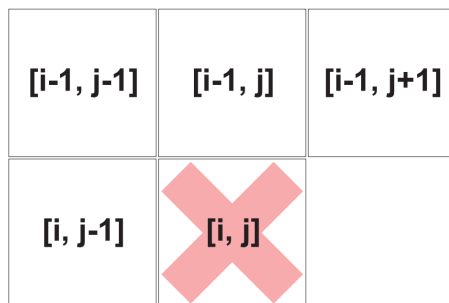
Algoritmus CCL pracuje tak, jak je uvedeno v následujícím popisu:

1. průchod: Procházejte obrázek po řádcích. Každému nenulovému pixelu (tj. pixelu představujícímu objekt) na souřadnicích $[i, j]$ přiřadte hodnotu podle hodnoty jeho sousedních pixelů, pokud nenulové sousední pixely existují (poloha sousedů je dána maskou na obrázku č. 1). Všechny sousední pixely dané maskou již byly obarveny v předchozích krocích (to je zajištěno tvarem masky).

- Jsou-li všechny sousední pixely součástí pozadí (mají hodnotu 0x00), přiřadte pixelu $[i, j]$ dosud nepřidělenou hodnotu – novou barvu.

- Má-li právě jeden ze sousedních pixelů nenulovou hodnotu, přiřadte obarvovanému pixelu hodnotu nenulového sousedního pixelu.

- Je-li více sousedních pixelů nenulových, přiřadte obarvovanému pixelu hodnotu kteréhokoli nenulového pixelu ze zkoumaného okolí. Pokud byly hodnoty pixelů v masce různé (došlo k tzv. *kolizi barev*), zaznamenejte ekvivalenci dvojic hodnot do zvláštní datové struktury – tabulky ekvivalence barev.



Obr. 1: Maska pro přebarvování

2. průchod: Po průchodu celého obrázku jsou všechny pixely náležející oblastem (objektům) obarveny, některé oblasti jsou však obarveny více barvami (díky kolizím). Proto projděte znovu celý obraz po řádcích a podle informací o ekvivalenci barev (z tabulky ekvivalence barev získané v průběhu 1. průchodu) přebarvěte pixely těchto oblastí. Z množiny kolizních barev je možné nějak vybrat jednu (první, poslední, náhodně) nebo opět postu-

povat při přiřazování barev „od začátku“ a jako novou barvu použít index množiny (nesmí být samozřejmě nulový, protože barva 0x00 představuje pozadí).

Po tomto kroku odpovídá každé oblasti označení (obarvení) jedinou, v jiné oblasti se nevyskytující hodnotou (barvou).

Obrázek č. 2 představuje binární (černobílý) obrázek na vstupu algoritmu. Černé oblasti představují pozadí, bílé oblasti (získané tzv. *prahováním*) představují objekty. Na obrázku č. 3 vidíte výsledek činnosti algoritmu – každý objekt je obarven unikátní barvou. Lze tedy např. zjistit počet objektů ve scéně, zkoumat jejich tvar, apod.



Obr. 2: Vstupní binární obrázek



Obr. 3: Výstupní obarvený obrázek

Obarvením se zde – v případě šedotónových obrázků – pochopitelně nemyslí změna barvy pixelů v reprezentaci např. RGB (jak již bylo naznačeno), ale označení každé souvislé blasti jinou úrovní šedi, takže je pak každý „obarvený“ objekt v obrázku snadno identifikovatelný touto svojí „barvou“.

1.3 Vstupní a výstupní formát

Na vstupu i výstupu Vašeho programu budiž obrázek v **souboru ve formátu PGM**. Formát tohoto souboru je následující:

P5	znaky 'P' a '5' a bílý znak*
1024 768	počet sloupců, bílý znak*, počet řádek, bílý znak*
255	index nejvyšší hodnoty šedé, resp. úrovně jasu, bílý znak*
ÈÈÈuu\$AAAA...	byty (tj. v jazyce C datový typ unsigned char) představující hodnoty jednotlivých pixelů

*) bílým znakem se ve formátu PGM rozumí jakýkoliv znak, pro který vrací funkce ze standardní knihovny překladače jazyka C `isspace(int c)` nenulovou hodnotu.

Každý pixel je v souboru uložen jako jeden byte (hodnota pixelu je tedy vlastně úroveň šedé). Byte s hodnotou 0x00 znamená úplně černý pixel, zatímco byte s hodnotou 0xFF znamená úplně bílý pixel.

2 Analýza úlohy

2.1 Vstup

V zadání již byla struktura PGM dobře popsána. Je důležité správně načítat data ze vstupního souboru v tomto formátu. Načítač se nesmí nechat oklamat byty s hodnotou 0x00, nejedná se o žádný konec řetězce ani nic podobného, je to prostě šedotónová hodnota bytu (tedy kompletně černá). Prvním úkolem tedy je vůbec vstupní soubor načíst a správně si ho uložit. Jako nejlepší řešení se jeví, si vstupní byty uložit do pole. Kvůli výpočetní náročnosti indexace dvou (či více) dimenzionálních polí skvěle postačí pole jednorozměrné. Mapovací funkce jednorozměrného pole při uvažování dvou rozměrů (což tedy každý obrázek má – šířka a výška) je $x = y \cdot \text{šířka}$.

2.2 První průchod

Po úspěšném načtení vstupních dat je nyní třeba vyhledávat v obrázku spojitě komponenty v popředí. Popředí a pozadí obrázku se jednoduše rozliší, jak již bylo v zadání popsáno – hodnota pozadí je 0x00 a hodnota popředí je 0xFF (tedy pozadí je černou barvou, popředí je bílou barvou). Spojitost komponenty zajišťuje maska daná zadáním, viz

obrázek č. 1. Tvar této masky zajišťuje, že sousedé prohledávaného pixelu už byli zpracováni.

Tvar ale také znemožňuje prohledání prvního řádku a sloupce a posledního sloupce pixelů v obrázku, neboť nic není před prvním řádkem a sloupcem a nic není ani za posledním sloupcem. Je tedy nutné si předpřipravit prvních pár pixelů jiným způsobem, než-li procházením touto maskou, jako tomu tak bude u zbytku algoritmu. Možností je projít separátně prvních pár pixelů s jinou maskou, např. první řádek tak, že každý pixel, se dívá na souseda pouze doleva (tedy kromě prvního pixelu). Jiný možný přístup k tomuto problému může být duplikace „imaginárních“ pixelů. Tím jest myšleno představit si pixely před prvním a posledním sloupcem a prvním řádkem a přiřadit jim hodnotu jejich nejbližších sousedů.

Díky datové struktuře disjunktních množin lze snadno detekovat spojitě komponenty v obrázku, tedy jednotlivé objekty na popředí. Neboli v řeči disjunktních množin, každý separátní objekt bude mít své pixely v jedné množině, která je disjunktní se všemi ostatními množinami, neboli nebudou mít žádný průnik – kdyby dva objekty měli společný průnik pixelů, nejsou to dva separátní objekty, ale měla by to být jedna komponenta. Tedy v algoritmu při průchodu pixelů danou maskou probíhá pouze kontrola sousedů, zda-li jsou nenulové, tedy jsou součástí popředí. Když jsou všechny sousední pixely nulové, tedy všechny jsou součástí pozadí, nyní navštěvovaný pixel, označovaný za současný, si založí svou vlastní novou množinu. Když je jen jeden ze sousedů nenulový, tedy už musí být součástí nějaké množiny (to je zajištěno tvarem masky), nyní vyšetřovaný pixel se přidá do jeho množiny. Pokud by nastala situace, že je více nenulových sousedů, může se současně vyšetřovaný pixel přidat do jakékoli z množin a všechny tyto množiny všech nenulových sousedů je třeba sjednotit, využila by se funkce datové struktury disjunktních množin *union*. Tato funkce mimo jiné zajistí pro druhý průběh výše zmíněnou tabulku ekvivalencí, neboť při volání funkce *find* s indexem jakéhokoliv pixelu ze stejné množiny bude vždy vracet stejnou hodnotu – hodnota reprezentativního předka. Jinou možností by bylo vést si tabulku ekvivalencí v pomocné datové struktuře. Poštačilo by klidně jednoduché pole, kde např. vždy na lichý index by se uložila barva a na sudý index vedle by se uložila ekvivalentní barva. Tento přístup by vyžadoval sofistikovanější ukládání do pole. Proto jako další skvělou alternativou se může zdát spojový seznam. Element tohoto spojového seznamu by byl pouze číslo dané barvy a ukazoval by na následující ekvivalentní barvu.

Uložení datové struktury disjunktních množin se nabízí velice jednoduše – udělat stejně velké pole elementů této datové struktury, jako je velikost pole pixelů. Alternativou by opět mohl být spojový seznam, ale ztrácí se tím výhoda pole – konstantní přístup a indexování. Neboť pole těchto elementů je stejně velké jako pole pixelů, indexace je také stejná. Tedy index pro současně vyšetřovaný pixel v poli pixelů je stejným indexem pro přístup do pole disjunktních množin na současně vyšetřovanou pozici.

2.3 Druhý průchod

Při druhém průchodu (jak již bylo zmíněno, algoritmus je dvouprůchodový) je již vše připraveno ke zhotovení finálního výsledku. Nyní nebrání nic tomu zavolat funkci datové struktury disjunktních množin *find* s indexem každého pixelu jako parametr a ke každé unikátně vrácené hodnotě z této funkce přiřadit jednu unikátní barvu. Tímto by měli všechny spojitě pixely tvořící jednotlivé objekty mít svou vlastní unikátní barvu.

Nutno dodat, že v okruhu šedotónových obrázků je dostupných pouze 256 barev. Jedna barva je už vyhrazená pro pozadí, tedy zbývá jen 255 barev. Tedy tímto algoritmem je možné obarvit objekty v obrázku pouze tehdy, je-li v obrázku maximálně 255 samostatných objektů.

2.4 Výstup

Výsledná přepsaná data už je jen zapotřebí uložit do výsledného souboru formátu PGM. Prakticky dojde jen k opsání hlavičky z původního vstupního souboru a jen zapsání změněných dat jednotlivých pixelů.

3 Popis implementace

Celá aplikace je napsána v ANSI C a měla by být díky své implementaci i přenositelná (minimálně byly testovány operační systémy Windows a Linux). Obsahuje čtyři hlavičkové soubory, k nim samozřejmě i soubory s příponou .c. Navíc ještě obsahuje jeden soubor přípony .c, který zastřešuje celou aplikaci a obsahuje funkci *main*.

3.1 pgm.h

V zadání byla popsána struktura PGM souboru, je ale zapotřebí si tento soubor nějak adekvátně vnitřně reprezentovat. Byla k tomu užita struktura *struct pgm*. Tato struktura si uchovává tzv. *magické číslo* PGM (např. „P5“), dále *šířku* a *výšku* daného obrázku, maximální jasovou hodnotu (u binárního obrázku to skoro vždy, ne-li opravdu vždy, je 255) a v neposlední řadě samozřejmě jednotlivé byty, určující barvu pixelu v šedotónu.

Dále jsou implementovány funkce pro načtení PGM souboru do této vnitřní struktury a následné zapsání do souboru formátu PGM za použití vnitřní reprezentace strukturou.

Načítání probíhá za užití funkce *fopen*, dále se vytvoří výše popsaná struktura a naplní se daty ze souboru, který byl otevřen. Zápis probíhá velmi podobně, jen se do souboru zapisuje a data se berou ze struktury. Tato knihovna dále zavádí funkci pro ověření binarity obrázku, tedy zkontrolují se jednotlivé načtené pixely, zda jsou jejich hodnoty buď 0x00 nebo 0xFF. Nesmí samozřejmě chybět funkce na uvolnění paměti, ta nejdříve uvolní vnitřní pole struktury, pole pixelů, pak se uvolní struktura jako taková.

3.2 disj_set.h

V analýze byla zmíněna *datová struktura disjunktních množin* (také známá jako *Union and Find*). Tato knihovna zavádí strukturu *struct disj_set_element* reprezentující jeden daný element množiny. Jeden takový element si musí pamatovat do jaké množiny patří, takže musí mít referenci na svého předka, měl by také znát svou hodnotu, která následně reprezentuje danou množinu. Z optimalizačních, níže popsaných, důvodů si ještě každý element vede informaci o své hloubce – tedy v jaké úrovni stromu se nachází. S tímto elementem následně pracují všechny funkce definované touto knihovnou.

Zavedeny byly standardní – ve smyslu pro tuto datovou strukturu – funkce, tedy *make_set*, *find* a *union*.

Make_set vytvoří výše zmíněnou strukturu a nastaví nově vzniklému elementu jeho hodnotu na hodnotu z parametru. Dále nastaví předka na NULL a hloubku logicky na 1.

Find přebírá jako parametr ukazatel na element, dokud daný element má předka, nastaví se nynější ukazatel na něj. Tímto cyklem dojdeme až na hlavního předchůdce, tedy reprezentanta dané množiny. Funkce vrací ukazatel na tohoto reprezentanta

Union přebírá dva ukazatele na elementy jako své parametry. Zavolá výše popsanou funkci *find* pro oba elementy. Pokud oba elementy mají stejného reprezentanta, znamená to, že už se nacházejí ve stejné množině a není důvod je slučovat, neboť už sloučení vlastně jsou. Pokud tomu tak není, tak na základě hloubky stromů se jeden přiřadí pod druhý a spojí se tak i naše abstraktní množiny.

V neposlední řadě nesmí ani chybět funkce na uvolňování paměti, tedy uvolnění struktury elementu

3.3 ccl.h

Tato knihovna nezavádí žádnou strukturu, narozdíl od předchozích. Zavádí pouze jednu jedinou funkci, kterou je hlavní algoritmus, popsáný jak v zadání, tak v analýze, tedy dvou průchodové přebarvování souvislých komponent v binárním obrázku.

Úplně nejdříve se provede inicializace pole elementů disjunktních množin. Toto pole je stejně velké jako pole pixelů vstupního obrázku.

V prvním průchodu se nejdříve musí projít první řádek, první sloupec a poslední sloupec, kvůli již zmíněnému tvaru masky. Pak se mohou projít i zbylé pixely. Při takovém průchodu nespeciálního pixelu – tedy pixel, který není ani v prvním řádku či sloupci, ani v posledním sloupci – se děje následující. Algoritmus zkontroluje hodnoty pixelů jeho čtyř sousedů. Je-li nějaký z nich nenulový – tedy není pozadí – přistoupí se do pole elementů množin, které je vlastně indexované stejně jako pole pixelů. Jelikož byl nějaký ze sousedů nenulový, musí se na stejném indexu v poli elementů vyskytovat existující element (to je zajištěno tvarem masky). Na indexu nynějšího pixelu se vytvoří nový element v poli elementů a za jeho hodnotu se nastaví hodnota nenulového souseda. Dále se sloučí všichni nenulový sousedé v tomto kroku průchodu. Sloučením se zajišťuje tabulka ekvivalencí barev, neboť sloučené množiny mají jednoho jediného reprezentanta množiny. Pokud byly všechny sousední pixely nulové, nemůže existovat ani žádný sousední element. Je tedy zapotřebí založit nový element a dát mu unikátní novou hodnotu. To lze jednoduše zajistit inkrementujícím se počítadlem.

Nyní před druhým průchodem se napočítá počet unikátních množin, tedy jednotlivých spojitých komponent v obrázku. Z poznámky v sekci analýza, druhý průchod, je známo, že je k dispozici pouze 255 barev. Bylo-li by tedy v obrázku více než 255 komponent, je nutné o tom dát vědět uživateli a chod programu ukončit. Je-li komponent méně než 255, je vhodné nějak inteligentně barvy rozdělit. Tedy stačí podělit 255 možných barev počtem unikátních komponent, a pak každé komponentě přiřadit barvu jako její pořadí \times výsledek podílu.

Druhý průchod je velice jednoduchý. Stačí projít pole elementů a pro každý element zavolat funkci *find*. Dle hodnoty výsledného reprezentativního elementu se jednoduše přiřadí dříve spočítaná barva. Tuto barvu je jen zapotřebí přepsat v poli dat ve vnitřní struktuře reprezentující obrázek ve formátu PGM.

Na závěr je důležité nezapomenout vše správně uvolnit.

3.4 mem.h

Tato knihovna reprezentuje jednoduchou implementaci kontroly úniků paměti. Není vůbec dokonalá, neboť pouze počítá bloky alokované paměti, nikoliv jednotlivé byty. Obsahuje funkce *mymalloc*, *mycalloc* a *free*, které všechny volají jejich „stejnomené“ ekvivalenty knihovnických funkcí ze standardní knihovny *stdlib.h*. Samozřejmostí je ještě veřejné globální počítadlo, které se při alokování paměti inkrementuje a při uvolňování paměti se dekrementuje.

3.5 main.c












Soubor obsahující hlavní spouštěcí funkci *main*. Má ještě další funkcionalitu mimo pouhé spouštění aplikace, tou jest kontrola vstupu od uživatele. Nejdříve se kontroluje počet argumentů z příkazové řádky. Očekávají se přesně dva – vstupní a výstupní soubory. Vstupní i výstupní soubory se v případě nutnosti doplňují o příponu formátu PGM souboru – .pgm. Dále se ještě kontroluje, zda vstupní soubor je opravdu binární pomocí výše zmíněné funkce z knihovny *pgm.h*. Následně už se jen využívá hlavního algoritmu z knihovny *ccl.h*.


4 Uživatelská příručka

4.1 Překlad

Aplikace obsahuje soubory *Makefile* a *Makefile.win*, které se starají o správný překlad. Stačí jen otevřít příkazovou řádku v daném adresáři se zdrojovými kódy a napsat příkaz **make**, popř. **make -f Makefile**, pokud je překlad žádán překladačem **GCC**. Pro překlad překladač od Windows je to stejné, jen se musí uvést soubor s příponou .win, tedy **make -f Makefile.win**. Vykonání příkazu pro překlad demonstruje obrázek č. 4. Všechny demonstrace budou předváděny na operačním systému Windows s užitím překladače GCC.

Následně by v adresáři měly vzniknout objektové soubory. Tedy soubory s příponou .o nebo .obj (záleží na užitém překladači). Samozřejmě by také měla vzniknout spustitelná aplikace s příponou .exe (na Windows). Vše je znázorněno na obrázku č. 5.

Název	Datum změny	Typ	Velikost
 ccl.c	28.11.2021 14:04	JetBrains CLion	12 kB
 ccl.h	28.11.2021 14:01	JetBrains CLion	1 kB
 disj_set.c	28.11.2021 14:04	JetBrains CLion	4 kB
 disj_set.h	28.11.2021 13:52	JetBrains CLion	1 kB
 main.c	28.11.2021 14:03	JetBrains CLion	4 kB
 Makefile	28.11.2021 14:07	Soubor	1 kB
 Makefile.win	28.11.2021 14:19	Soubor WIN	1 kB
 mem.c	28.11.2021 12:28	JetBrains CLion	2 kB
 mem.h	28.11.2021 14:01	JetBrains CLion	1 kB
 pgm.c	28.11.2021 14:00	JetBrains CLion	5 kB
 pgm.h	28.11.2021 14:01	JetBrains CLion	1 kB


















 C:\Windows\System32\cmd.exe


C:\Users\SpeekeR\Documents\ccl>make_

Obr. 4: Obsah adresáře před vykonáním příkazu make

4.2 Spuštění

Nyní do příkazové řádky lze napsat příkaz **ccl.exe** **<input-file[.pgm]>** **<output-file>**, kde input-file značí cestu k vstupnímu souboru ve formátu PGM a output-file značí cestu k výstupnímu souboru, tedy obarvenému produktu. Následuje série obrázku demonstrující ukázkové vstupy.

Název	Datum změny	Typ	Velikost
 ccl.c	28.11.2021 14:04	JetBrains CLion	12 kB
 ccl.exe	04.12.2021 12:50	Aplikace	96 kB
 ccl.h	28.11.2021 14:01	JetBrains CLion	1 kB
 ccl.o	04.12.2021 12:50	Soubor O	5 kB
 disj_set.c	28.11.2021 14:04	JetBrains CLion	4 kB
 disj_set.h	28.11.2021 13:52	JetBrains CLion	1 kB
 disj_set.o	04.12.2021 12:50	Soubor O	2 kB
 main.c	28.11.2021 14:03	JetBrains CLion	4 kB
 main.o	04.12.2021 12:50	Soubor O	3 kB
 Makefile	28.11.2021 14:07	Soubor	1 kB
 Makefile.win	28.11.2021 14:19	Soubor WIN	1 kB
 mem.c	28.11.2021 12:28	JetBrains CLion	2 kB
 mem.h	28.11.2021 14:01	JetBrains CLion	1 kB
 mem.o	04.12.2021 12:50	Soubor O	2 kB
 pgm.c	28.11.2021 14:00	JetBrains CLion	5 kB
 pgm.h	28.11.2021 14:01	JetBrains CLion	1 kB
 pgm.o	04.12.2021 12:50	Soubor O	3 kB

 C:\Windows\System32\cmd.exe

```

C:\Users\SpeekeR\Documents\ccl>make
gcc -c -Wall -pedantic -ansi mem.c -o mem.o
gcc -c -Wall -pedantic -ansi pgm.c -o pgm.o
gcc -c -Wall -pedantic -ansi disj_set.c -o disj_set.o
gcc -c -Wall -pedantic -ansi ccl.c -o ccl.o
gcc -c -Wall -pedantic -ansi main.c -o main.o
gcc mem.o pgm.o disj_set.o ccl.o main.o -o ccl.exe

C:\Users\SpeekeR\Documents\ccl>_
    
```

Obr. 5: Obsah adresáře po vykonání příkazu make

```
C:\Users\SpeekeR\Documents\ccl>ccl.exe
ERROR: Expected execution of program as shown below:
ccl.exe <input-file[.pgm]> <output-file>

C:\Users\SpeekeR\Documents\ccl>ccl.exe vice parametru nez jen dva
ERROR: Expected execution of program as shown below:
ccl.exe <input-file[.pgm]> <output-file>
```

Obr. 6: Reakce programu na špatně zadané parametry (konkrétně žádné, nebo více než 2)

```
C:\Users\SpeekeR\Documents\ccl>ccl.exe cesta/k/vstupnimu/souboru cesta/k/vystupnimu/souboru
ERROR: Filepath is invalid, input file might not exist!
ERROR: Out of memory!
```

Obr. 7: Reakce programu na neexistující soubory

```
C:\Users\SpeekeR\Documents\ccl>ccl.exe w2test vysledek
C:\Users\SpeekeR\Documents\ccl>ccl.exe w2test.pgm vysledek
C:\Users\SpeekeR\Documents\ccl>ccl.exe w2test vysledek.pgm
C:\Users\SpeekeR\Documents\ccl>ccl.exe w2test.pgm vysledek.pgm
```

Obr. 8: Správné spuštění programu; všechny 4 varianty zajišťují stejný výsledek

5 Závěr

Obrázek č. 9 je ukázkou správnosti obarvovacího algoritmu, jedná se o obarvení obrázku již známého ze zadání

Program funguje rychle, neboť se všude pracuje s poli, přístup k poli je v konstantním čase. Paměťově je program náročnější než časově, protože se vnitřně ukládají dvě stejně velké matice, které závisí na počtu pixelů v obrázku. Ale opět se nejedná o žádnou velkou náročnost, neboť se dá předpokládat normální velikost obrázků.

Možné vylepšení do budoucna by mohl být spojový seznam uchovávající elementy datové struktury disjunktních množin. Časová náročnost by se zvýšila z konstantního času na lineární, což je stále dostatečně rychlé, a hlavně by se ubrala paměťová náročnost, neboť by se neuchovávaly matice dvě.



Obr. 9: Programem obarvený ukázkový obrázek ze zadání