Università di Perugia
Dipartimento di Matematica e Informatica

Artificial Intelligent Systems -
Intelligent Application Development

Colorazione di un Grafo

(ricerca in Ampiezza)

Professore
Prof. Stefano Marcugini

Studente
Nicolò Vescera

Anno Accademico 2021-2022

Indice

1 Obiettivo

L'obiettivo di questo progetto è quello di realizzare un programma scritto in **OCaml** che risolva il *problema* della *colorazione di un grafo*: riuscire a colorare, se possibile, ogni nodo del grafo in modo da non avere mai nodi adiacenti con lo stesso colore. In via più formale, dato un grafo `g` ed un numero massimo di colori utilizzabili `N`, assegnare un colore (da 0 a $N-1$) ai nodi in modo tale che non esistano nodi adiacenti con lo stesso colore; qualora il numero di colori `N` non sia sufficiente per realizzare la colorazione, riportare un errore.

2 Struttura del Progetto

Di seguito sarà riportata la struttura delle cartelle e dei file del progetto con una breve descrizione per quelli più importanti. I file OCaml sono divisi in base al compito che svolgono le funzioni al loro interno (*e.g.:* ***`graphUtils`*** *conterrà le funzioni che operano sui grafi, ecc.*) e sono sempre formati da un file `.ml` che contiene il copro delle funzioni con tutte le varie espressioni ed un file `.mli` nel quale è presente solo il tipo delle funzioni ed espressioni che verranno utilizzate dagli altri file.

```
Makefile
src
  main.ml
  main.mli
  graphUtils.ml
  graphUtils.mli
  data.ml
  data.mli
  printer.ml
  printer.mli
  rappresentazione_grafo
    rappresentazione_grafo.py
```

Figura 1: Rappresentazione schematizzata dei file e cartelle del progetto.

- `Makefile`: utilizzato per compilare il progetto e generare l'eseguibile per avviarlo
- `main`: contiene la funzione principale che si occupa di avviare la colorazione e di gestire le scelte fatte dall'utente
- `graphUtils`: insieme di funzioni che operano sui grafi che svolgono l'effettivo compito di colorazione
- `data`: insieme di grafi per testare il corretto funzionamento del progetto. Questi possono essere aggiunti e rimossi in modo semplice ed efficiente

- `printer`: insieme di funzioni ed espressioni per stampare a video menu ed altri elementi con anche la presenza di colori ed emoji
- `rappresentazione_grafo`: script in python per rappresentare a video (in modo interattivo) un grafo

Una volta utilizzato il comando `make` per compilare il progetto, la struttura finale delle cartelle sarà la seguente:

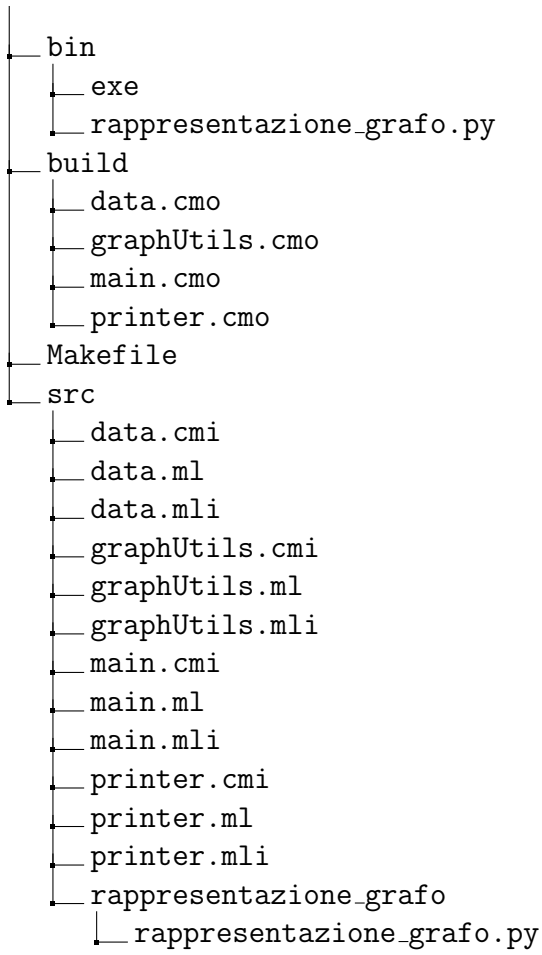


Figura 2: Rappresentazione dei file e cartelle del progetto dopo averlo compilato.

I file `.cmi` vengono lasciati all'interno della cartella `src` per far funzionare correttamente l'estensione di **OCaml** per **Visual Studio Code** e per la corretta compilazione degli altri file.

3 Codice

3.1 Rappresentazione di un Grafo

In questo progetto i grafi (si parla sempre di grafi *orientati/diretti*) vengono rappresentati come una *tupla* di 3 elementi:

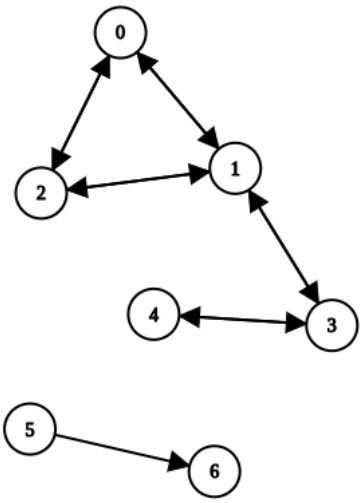
1. `succ` (funzione successori): definisce tutti i successori (nodi vicini) di tutti i nodi del grafo
2. `start`: nodo di partenza per la colorazione
3. `maxColors`: numero massimo di colori da utilizzare durante la colorazione

Di seguito un esempio di definizione di un grafo con la sua rappresentazione:

```
let grafo_1 =
  let x = function
      0 -> [1; 2]
    | 1 -> [0; 2; 3]
    | 2 -> [0; 1]
    | 3 -> [1; 4]
    | 4 -> [3]
    | 5 -> [6]
    | _ -> [] in
  let start = 3 in
  let maxColors = 3 in
  let succ = Successori x in

  (Grafo (succ, start, maxColors))
;;
```

(a) Esempio di definizione di un grafo



(b) Grafo prodotto dal codice (a)

Il tipo `grafo` e `succesori` sono definiti in `graphUtils` nel seguente modo:

```
  type successori = Successori of (int -> int list);;
  type grafo      = Grafo of successori * int * int;;
```

3.2 Data

In questo file ci sono tutte le definizioni dei grafi utilizzati come test per verificare il corretto funzionamento dell'algoritmo di colorazione; una funzione che permette di stampare un menù, con alcune informazioni come la descrizione dei grafi e gli ID unici, ed una funzione che dato un numero (ID), ritorna uno dei grafi detti prima. È anche presente una lista (`grafi`) che contiene tutti i grafi disponibili con una descrizione (una coppia `(grafo, string)`). È questa che permette di aggiungere e togliere elementi in maniera semplice e senza dover modificare alcuna funzione.

```
let grafi = [
  (grafo_1,   "Grafo 1");
  (grafo_2,   "Grafo 2 con numero sufficiente di colori");
  (grafo2err, "Grafo 2 con colori insufficienti");
  (grafo_3,   "Grafo 3");
  (grafo_4,   "Grafo 4")
];;
```

Listing 1: Lista di coppie (grafo,string) utilizzata per assiocare ad un grafo una descrizione.

La funzione `stampa_grafi_disponibili` mostra a video la descrizione dei vari grafi anteceduta da un ID. Questa scandisce la lista `grafi` ricorsivamente e va a stampare il secondo elemento di ogni coppia.

```
let stampa_grafi_disponibili () =
  let rec aux id = function (*lista di grafi*)
      []      -> print_string "\n"    (*caso base, stampa un \n*)
    | x::coda ->                      (*stampa descrizione ed id*)
      let stampa_elemento (_, descrizione) =
        print_string "  ";
        print_int id; print_string (") " ^ descrizione ^ "\n")

      (*id + 1 e continua con la coda*)
      in  stampa_elemento x;
          aux (id+1) coda

  in aux 1 grafi  (*avvia aux*)
;;
```

(a) Funzione per la stampa a video delle descrizione dei grafi antecedute da un ID

```
1) Grafo 1
2) Grafo 2 con numero sufficiente di colori
3) Grafo 2 con un numero di colori insufficiente per essere colorato
4) Grafo 3
5) Grafo 4
```

(b) Output del codice (a)

Infine, la funzione `scegli_grafo` che, dato un ID, ritorna il relativo grafo. È interessante notare come per selezionare un dato grafo non viene scorsa ricorsivamente tutta la lista, ma quest'utlima viene convertita in un `Array` per poter utilizzare il metodo `get` per accedere all'i-esima posizione. Vengono effettuati alcuni controlli sull'input: viene controllato se l'ID passato è valido (non superi per eccesso, o per difetto gli ID mostrati a video) e nel caso solleva un'eccezione (verrà utilizzata nella funzione `main` per segnalare il problema).

```
let scegli_grafo id_grafo =
  let array_tmp = Array.of_list grafi in    (*lista in array*)
    if id_grafo > Array.length array_tmp || (*controlla id*)
       id_grafo < 1
      then
        raise BadChoice    (* se non e valido lancia un'eccezione*)
    else
      fst (Array.get array_tmp (id_grafo-1))   (*ritorna grafo*)
;;
```

Listing 2: Funzione che ritorna il grafo selezionato

3.3 Printer

In questo file risiedono tutte le funzioni e dichiarazioni che gestiscono la stampa a video. Vi è una definizione di stringhe per stampare su terminale caratteri colorati (rosso, verde, blu, ecc.), elementi in grassetto ed emoji (a patto che quest'ultimo li supporti). La funzione più degna di nota è `stampa_grafo` che, dato un grafo, stampa a schermo tutti i nodi con i suoi vicini (in pratica rappresenta la funzione successori).

Di seguito alcuni dei codici utilizzati per rappresentare i colori nel terminale:

```
  type colore = Colore of string;;

  (* normali *)
  let rosso = Colore "\027[31m";;
  let verde = Colore "\027[32m";;
  let ciano =  Colore "\027[36m";;
  let bianco = Colore "\027[0m";;

  (* grassetto *)
  let rosso_b = Colore "\027[31;1m";;
  let verde_b = Colore "\027[32;1m";;
  let ciano_b = Colore "\027[36;1m";;
  let bianco_b = Colore "\027[0;1m";;

  (* valore di reset per stampare normalmente*)
  let reset = bianco;;
```

Listing 3: Codici per i colori ed elementi in grassetto

```
let stampa_grafo (Grafo((Successori succ), partenza, maxColori)) =
  print_colore rosso_b "Partenza: ";
  print_int partenza; print_colore rosso_b " Max Colori: ";
  print_int maxColori; print_string "\n\n";
  let rec search visitati = function  (* frontiera *)
      []         -> print_string "\n"    (*caso base, stampa \n*)
    | nodo::coda ->                      (*stampa nodo e vicini*)
      if List.mem nodo visitati     (* ignora nodi visti*)
        then
          search visitati coda
      else                          (* stampa il nodo e vicini.*)
        (print_colore verde_b (string_of_int nodo);
        print_colore bianco_b " -> "; stampa_lista (succ nodo);
        search (visitati@[nodo]) (coda@(succ nodo)))

  in search [] [partenza]   (*avvia la ricorsione*)
;;
```

(a) Funzione per la stampa di un grafo su terminale

```
Partenza: 3 Max Colori: 3

3 → 1 4
1 → 0 2 3
4 → 3
0 → 1 2
2 → 0 1
```

(b) Output della funzione (a)

3.4 GraphUtils

All'interno di questo file sono contenute tutte le funzioni e le espressioni che si occupano della gestione e della colorazione dei grafi. In particolare, la funzione `colora` è quella che effettua la colorazione vera e propria. Prende in input un grafo (una tupla formata da `(successori , partenza, maxColori)`) e come output restituisce una lista composta da coppie del tipo `(nodo, colore)`, associa quindi ad ogni nodo (`int`) un colore (`int`). Alla riga `2` viene calcolato il `refValue` (servirà successivamente per cercare di trovare un possibile colore da assegnare qualora i vicini del nodo siano già stati colorati); nella riga `3` viene definita la funzione ausiliaria `esplora` che, tramite un algoritmo di ricerca in ampiezza (*BFS*) andrà ad esplorare i nodi del grafo:

- Se, il nodo esaminato è già stato visitato (è presente nella lista `visitati`) lo ignora e continua richiamando se stessa ed estraendo un nuovo nodo dalla frontiera (righe `7 - 9`)
- Altrimenti (il nodo non è mai stato analizzato),
  - aggiunge il nodo alla frontiera (lista di nodi da visitare),

- se possibile colora il nodo (aggiunge la coppia `(nodo, colore)` alla lista `colorati`), in caso contrario verrà sollevata un'eccezione,
- infine aggiunge alla frontiera i nodi vicini al nodo esaminato.

Richiama quindi se stessa con le nuove liste continuando così la ricorsione (righe `11-18`)

Questa esplorazione avrà fine quando la frontiera non conterrà più nodi da esplorare. La riga `20` serve per avviare la ricorsione (e quindi l'esplorazione) impostando le liste `vicini` e `colorati` come vuote e la frontiera con il nodo di partenza al suo interno.

```
1  let colora (Grafo ((Successori succ), partenza, maxColori)) =
2    refValue := calcola_refValue maxColori;   (*per colore mancante*)
3    let rec esplora visitati colorati =
4      function (*frontiera*)
5          []            -> colorati  (*colorazione finita*)
6        | nodo::coda    ->           (*continua a colorare*)
7          if List.mem nodo visitati              (*nod visitato*)
8            then      (* ignora il nodo, successivo di frontiera*)
9              esplora visitati colorati coda
10
11         else  (*espande nodi vicini*)
12           esplora
13             (visitati@[nodo])       (*nodo visitato*)
14             (colorati@[
15               (nodo,
16               (scegli_colore (Successori succ) nodo colorati maxColori)
17               )]) (*colora il nodo*)
18             (coda@(succ nodo))      (*aggiunge vicini a frontiera*)
19
20   in esplora [] [] [partenza]   (*avvia esplora*)
21 ;;
```

Listing 4: Funzione che si occupa della colorazione di un grafo.

Un'altra funzione molto importante all'interno di questo file è `scegli_colore`, che dati in input la *funzione successori*, il *nodo* da colorare, la lista dei *nodi colorati* e il *numero massimo di colori*, restituisce un colore (`int`) se possibile, altrimenti lancia un'eccezione (il numero di colori scelto è insufficiente per colorare il grafo). Alla riga `2, 7` viene scelto un potenziale risultato che è dato dal massimo tra i colori vicini al nodo aumentato di 1, in modulo `maxColori`. Così facendo ottengo una scelta ciclica dei colori, potendo ritornare a scegliere il colore `0` quando tutti gli altri sono stati assegnati.

$$risultato\_mod = (\max(colori\_vicini) + 1) \bmod maxColori$$

Quando un nodo non è stato colorato gli viene assegnato il valore `-1`, questo non inficia nel calcolo descritto prima: supponiamo di star analizzando il nodo iniziale, quindi sia lui che tutti i suoi vicini non hanno colore (colore = `-1`), il massimo risulterà essere `-1`. Il colore scelto sarà quindi `0` e verrà assegnato al nodo iniziale, tutto funziona come previsto.

Infine viene analizzato questo possibile risultato (righe `8 - 16`):

- Se il colore scelto è presente tra i colori dei nodi vicini (un nodo vicino ha già lo stesso colore), tenta di trovare un colore valido: somma tutti i colori vicini al nodo e sottrae questo valore al `refValue`. Se questo nuovo colore è ancora uguale ad un colore vicino al nodo, viene lanciata un'eccezione dato che non è possibile risolvere il problema, altrimenti viene ritornato come nuovo colore (righe `8 - 15`).
- Se non lo è allora vuol dire che è stato scelto il giusto colore e viene ritornato.

In questo modo, mi assicuro che ad ogni passaggio viene scelto il colore giusto e non necessito di dover ricontrollare tutto il grafo nuovamente.

```
1  let scegli_colore (Successori succ) nodo colorati maxColori =
2    let risultato =   (*trova il massimo tra i colori vicin al nodo +1*)
3      ((max_colore_vicini (Successori succ) nodo colorati)+1) in
4
5    (*trova tutti i colri vicini*)
6    let colori_vicini = tutti_colori_vicini (succ nodo) colorati in
7    let risultato_mod = risultato mod maxColori in  (*calcola il modulo*)
8      if List.mem risultato_mod colori_vicini
9        then   (*colore scelto = colore vicino*)
10         let mancante =  (* prova scegliere un altro colore*)
11           trova_mancante colori_vicini maxColori in
12             if List.mem mancante colori_vicini
13               (* numero di colori insufficiente *)
14               then raise InsufficentColorNumber
15             else mancante (*ritorna colore mancante*)
16     else risultato_mod  (*trovato un colore valido*)
17 ;;
```

Listing 5: Funzione ausiliaria a Listing 4 che sceglie un colore per il nodo (se possibile).

Infine un breve sguardo alla funzione che si occupa di salvare i dati del grafo colorato su file per poterli poi processare da python e mostrare una versione interattiva del grafo a video. Viene scandita tutta la lista prodotta dalla funzione Listing 4 (lista di coppie `(nodo, colore )`) e per ogni elemento viene scritto su file:

`nodo,vicini,colore`

Un esempio: `0,1 2 9 3,1`. Questo vuol dire che il nodo `0` ha come vicini `1, 2, 9, 3` e come colore `1`. Nelle righe `4 - 10` viene definita una funzione ausiliaria che permette di salvare la lista dei nodi adiacenti e la parte di codice che effettua le operazioni descritte prima è nelle righe `12 - 21`.

```
let grafodati_file = "grafo.data";; (*file dove salvare il grafo*)
let salva_grafo_colorato (Successori succ) colorati =
  let oc = open_out grafodati_file in (*apertura file in scrittura*)
    let rec salva_lista =   (*salva su file una lista*)
      function (* lista *)
         [x]      -> Printf.fprintf oc "%d" x   (* caso base*)
       | _        -> ()
       | x::coda ->                             (* caso ricorsivo*)
         Printf.fprintf oc "%d " x;          (* stampa l'elemento*)
         salva_lista coda                    (* continua*)

    in let rec salva =      (*salva nodo - vicini - colore su file*)
      function (* lista nodi_colorati *)
         []  ->
         Printf.fprintf oc "\n";
         close_out oc  (*chiude file*)
       |(nodo, colore)::coda -> (* salva nodo,vicini,colore*)
         Printf.fprintf oc "%d," nodo;
         salva_lista (succ nodo); Printf.fprintf oc ",";
         Printf.fprintf oc "%d\n" colore;
         salva coda  (*continua*)

  in salva colorati (*avvia salva*)
;;
```

Listing 6: Funzione che salva un grafo su file.

3.5 Main

In questo file è presente la funzione principale `main` che si occupa di gestire tutto il flusso del programma: stampa il menu iniziale, fa scegliere all'utente su quale grafo testare l'algoritmo di colorazione, avvia la colorazione del grafo selezionato, stampa il risultato ed avvia lo script python.

```
let main () =
  stampa_logo ();                    (*stampa il logo*)
  stampa_grafi_disponibili ();       (*stampa i grafi disponibili*)

  let dati =
    let rec aux () =                  (*utente sceglie il grafo*)
      try                             (*controllo input valido*)
        scegli_grafo (scelta ())
      with BadChoice ->
        aux ()
    in aux()
      in let avvia_colorazione (Grafo (g, partenza, maxColori)) =
        print_string "\nIl Grafo selezionato: \n\n";
        stampa_grafo (Grafo (g, partenza, maxColori));
        print_string "Coloro ...\n\n";

        (*colora il grfo*)
        let colorati = colora (Grafo (g, partenza, maxColori)) in
          stampa_nodi_colorati colorati;   (*stampa il grafo colorato*)
          salva_grafo_colorato g colorati; (*salva grafo colorato*)
          avvia_python ()                  (* avvia python*)

          in      (*con un grafo scelto, lo colora*)
            try
              avvia_colorazione dati
            with InsufficentColorNumber ->
              stampa_errore ()
;;
```

Listing 7: Parte del codice contenuto in main.ml

Alla riga `2 - 3` viene stampato sul terminale il logo iniziale e tutti i grafi disponibili per testare l'algoritmo. Nelle successive righe (`5 - 27`) vengono definite alcune funzioni ausiliarie per mettere in pratica i comportamenti descritti prima:

- `dati`: rimane fermo sulla scelta del grafo fin quando l'utente non seleziona un ID esistente. Una volta selezionato un ID valido conterrà il grafo scelto.
- `avvia_colorazione`: stampa alcune informazioni utili all'utente per poi colorare il grafo, salvarlo su file e avviare lo script python. Qui vengono anche gestite le possibili eccezioni.

Nelle prime righe del file vengono inclusi gli altri codici descritti nelle precedenti sezioni con:

```
open Printer;;
open GraphUtils;;
open Data;;
```

Listing 8: Inclusione degli altri file .ml

3.6 Python

All'interno del progetto è anche presente un breve script in python che verrà avviato una volta riuscito a colorare il grafo. Questo script legge il file `.data` generato da OCaml (contiene il grafo ed i colori di ogni nodo) e lo rappresenta all'interno di una finestra in modo interattivo e soprattutto più comprensibile rispetto alla visualizzazione su terminale.

```
from pyvis.network import Network


def main(fname):
  collegamenti = []
  nodi = []
  colori = []

  # lettura dei dati da file
  with open(fname, 'r') as f:
    for text in f.readlines():
    text = text.strip("\n")                # elimina i vari \n nel file
    if text != '':                         # controlla validita' riga
      nodo, vicini, colore  = text.split(",")

      vicini = vicini.split(" ")          # prende i vicini
      colori.append(int(colore))          # aggiunge il colore alla lista

      if vicini[0] != '':                 # controlla se il nodo ha vicini
        for vicino in vicini:             # crea gli archi del grafo
          tmp = (int(nodo), int(vicino))
          collegamenti.append(tmp)

  nodi = trova_nodi(collegamenti)
  colori = aggiusta_colori(colori)  # converte i colori per pyvis

  # rappresenta il grafo con pyvis
  rappresenta_grafo(nodi, collegamenti, colori)


 def rappresenta_grafo (nodi, collegamenti, colori):
  net = Network(width='100%', height='600px', directed=False)

  net.add_nodes(nodi, color=colori)
  net.add_edges(collegamenti)
  net.toggle_physics(True)
  net.inherit_edge_colors(False)

  net.show('rappresentazione_grafo.html')

```

Listing 9: Alcune funzioni dello script python

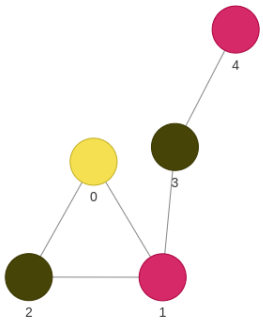


Figura 6: Output dello script python.

3.7 Makefile

File utilizzato per la compilazione del progetto che avviene tramite il comando `make`. Qualora un file venga modificato, questo permette di non dover ricompilare tutto il progetto ma solo quello che è stato cambiato. In una prima fase compila tutti i file `.ml` con i relativi `.mli`, generando 2 altri file:

- `.cmi`: interfacce compilate utili per il corretto funzionamento dell'estensione *OCaml* in *Visual Studio Code* e per la corretta compilazione degli altri file.
- `.cmo`: file oggetto che verranno poi spostati nella cartella `build` per essere utilizzati nella fase di *linking*

Quindi, ogni file `.cmo` dipende dai relativi file `.ml` e `.mli`. Successivamente, avvia la fase di *linking* unendo insieme tutti i file `.cmo` e generando l'eseguibile `exe` nella cartella `bin`. All'interno di quest'ultima viene copiato anche lo script python per permettere il corretto funzionamento del progetto.

```
cc = ocamlc
cflags = -c
cinclude = -I src
srcdir = src
buildir = build
bindir = bin
target = $(bindir)/exe

.PHONY: clear
all: cartelle $(target)

$(target):
  $(buildir)/graphUtils.cmo $(buildir)/printer.cmo
  $(buildir)/data.cmo $(buildir)/main.cmo
    $(cc) -o $@ $^
    cp $(srcdir)/rappresentazione_grafo/rappresentazione_grafo.py $(bindir)
```

Listing 10: Breve estratto del file Makefile

4 Dimostrazione

Di seguito andremo a vedere una breve guida dimostrativa del funzionamento del progetto, dalla compilazione iniziale fino al grafo colorato.

4.1 Compilazione

Per prima cosa è necessario compilare il progetto. Bisogna spostarsi all'interno della cartella `progetto`, dove è contenuto il file `Makefile` e la cartella `src`. Ora avviare la compilazione con il comando:

```
make
```

Verranno create due nuove cartelle:

- `bin`: al suo interno ci sarà l'eseguibile `exe` ed altri file necessari al funzionamento del progetto
- `build`: conterrà tutti i file generati durante la fase di compilazione (`.cmo`)

```
➜  progetto git:(main) ✗ make
mkdir -p build
mkdir -p bin
ocamlc -I src -c src/graphUtils.mli src/graphUtils.ml
mv src/graphUtils.cmo build
ocamlc -I src -c src/printer.mli src/printer.ml
mv src/printer.cmo build
ocamlc -I src -c src/data.mli src/data.ml
mv src/data.cmo build
ocamlc -I src -c src/main.mli src/main.ml
mv src/main.cmo build
ocamlc -o bin/exe build/graphUtils.cmo build/printer.cmo build/data.cmo build/main.cmo
cp src/rappresentazione_grafo/rappresentazione_grafo.py bin
➜  progetto git:(main) ✗
```

Figura 7: Output del comando make

Qualora si voglia ripulire tutti i file generati da questa operazione è possibile tramite il comando:

```
make clear
```

```
➜  progetto git:(main) ✗ make clear
rm -f build/*.cmo
rm -f bin/*
rm -f src/*.cmi
➜  progetto git:(main) ✗
```

Figura 8: Output del comando make clear

4.2 Esecuzione

Per avviare l'eseguibile del progetto generato alla fase precedente è prima necessario spostarsi all'interno della cartella `bin`:

```
cd bin
./exe
```

Verrà mostrato il menù iniziale e l'utente dovrà scegliere quale grafo utilizzare per testare l'algoritmo di colorazione. Se verrà inserito un ID invalido si rimane bloccati in questa scelta fin quando un ID corretto non verrà inserito. Una volta selezionato un ID valido verrà stampato a video il *grafo*, con il *nodo di partenza*, il *massimo numero di colori* ed il *risultato* della fase di colorazione. Infine verrà avviato lo script python che aprirà una finestra con al suo interno il grafo disegnato. Qualora l'utente selezionerà un grafo che non è colorabile per via del numero insufficiente di colori, verrà mostrato a video un messaggio di errore e non verrà avviato lo script python.

Figura 9: Menu iniziale

Figura 10: Esempio di multipli ID errati

```
> 1

Il Grafo selezionato:

Partenza: 3 Max Colori: 3

3 → 1 4
1 → 0 2 3
4 → 3
0 → 1 2
2 → 0 1

Coloro ...

Nodo: 3 - Colore: 0
Nodo: 1 - Colore: 1
Nodo: 4 - Colore: 1
Nodo: 0 - Colore: 2
Nodo: 2 - Colore: 0
```

Figura 11: Esempio di colorazione di un grafo

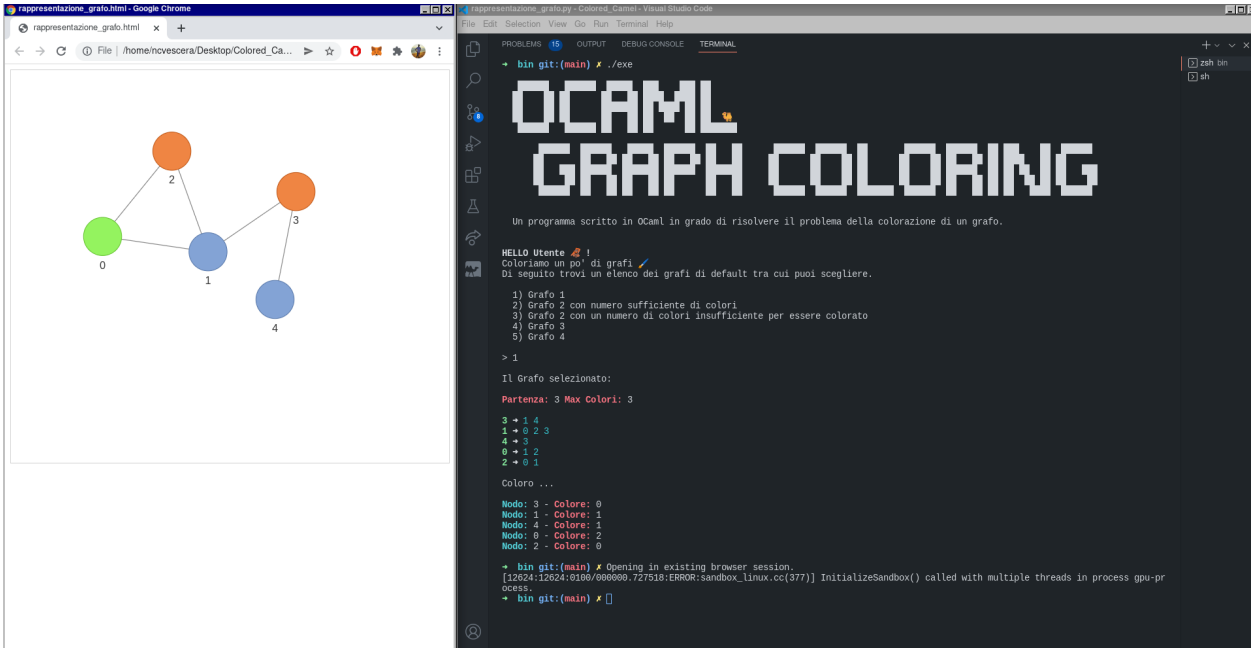


Figura 12: Finestra aperta da python con terminale affiancato

```
> 3

Il Grafo selezionato:

Partenza: 0 Max Colori: 3

0 → 1 2 3 4 5
1 → 0 3
2 → 0 5 4
3 → 0 1 4
4 → 0 3 2 5
5 → 0 2 4

Coloro ...

ERRORE: Numero di colori insufficienti per colorare questo grafo !!
➜  bin git:(main)
```

Figura 13: Messaggio di errore per numero di colori insufficienti