





Contents

1	Algoritmo & Progetto		2
1.1	Obiettivo		2
1.2	Definizione di Tipi ed Eccezioni		2
1.3	Definizione dei Problemi		3
1.3.1	Problema 1		4
1.3.2	Problema 2		5
1.3.3	Problema 3		6
1.3.4	Problema 4		7
1.3.5	Problema 5		8
1.3.6	Problema 6		9
1.4	Inclusione Moduli		10
1.5	Librerie Python		10
1.6	Idea dell'Algoritmo		11
1.7	Funzioni Ausiliarie		11
1.8	Funzione Principale		15
1.9	Test dell'Algoritmo		16
2	I ♥ Jupyter		18
2.1	Makefile		18
2.2	Printer		20
2.3	Data		21
2.4	GraphUtils		22
2.5	Main		23
2.6	Python		24
2.7	Dimostrazione di Esecuzione		27
3	Ambiente di Sviluppo		30

1 Algoritmo & Progetto 📌

1.1 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.

1.2 Definizione di Tipi ed Eccezioni

Definisco il tipo di dato **grafo** e **problema** con i relativi costruttori di tipo **Grafo** e **Problema**. Il primo tipo rappresenta un *grafo*, quindi la funzione *successori* che dato un intero (un nodo) restituisce tutti i suoi nodi vicini (i successori). Il secondo va a rappresentare il *problema da risolvere*, questo è composto da un **grafo** (che andrà colorato), dal **nodo di partenza** e dal massimo numero di **colori utilizzabili** N .

```
[1]: type grafo = Grafo of (int -> int list);;  
type problema = Problema of grafo * int * int;;
```

```
[1]: type grafo = Grafo of (int -> int list)  
[1]: type problema = Problema of grafo * int * int
```

La seguente eccezione verrà lanciata per segnalare che il problema in questione non è risolvibile dato che il grafo non è colorabile con il numero di colori dato.

```
[2]: exception NumeroColoriInsufficiente;;
```

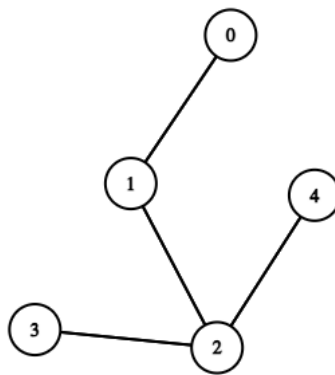
```
[2]: exception NumeroColoriInsufficiente
```

1.3 Definizione dei Problemi

Ho deciso di rappresentare ogni grafo da colorare come un **Problema** composto da 3 elementi distinti:

1. **succ**: la funzione successori che definisce tutti i nodi vicini raggiungibili da ogni altro nodo. Questo rappresenta il **Grafo** da colorare
2. **start**: nodo di partenza per la colorazione
3. **maxColori**: numero massimo (N) di colori da utilizzare durante la colorazione

Per esempio, il problema costituito dal seguente grafo



con partenza dal nodo 0 e con un numero di colori $N = 2$ è rappresentato dalla parte di codice sottostante:

```
let problema =
  let x = function
    0 -> [1]
    | 1 -> [0; 2]
    | 2 -> [1; 3; 4]
    | 3 -> [2]
    | 4 -> [2]
    | _ -> [] in
  let start = 0 in      (* Partenza *)
  let maxColori = 2 in  (* Massimo numero di colori *)
  let succ = Grafo x in (* Successori *)

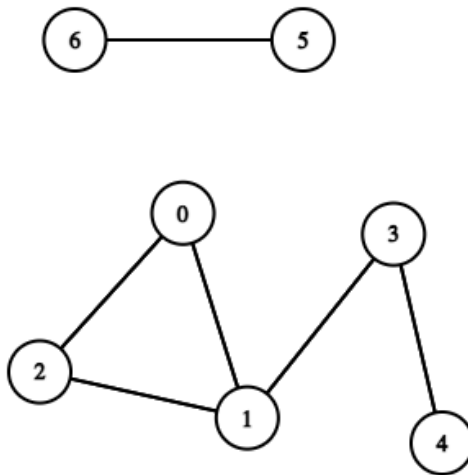
  (Problema (succ, start, maxColori))
;;
```

1.3.1 Problema 1

Il **problema 1** è formato da un grafo non orientato e sconnesso, composto da 7 nodi. Il punto di partenza è il *nodo 3* ed ha un numero massimo di colori $N = 3$. Questo problema è stato scelto per far vedere come l'algoritmo si comporta con grafi sconnessi: andrà a colorare solo i nodi che riesce a raggiungere.

```
[3]: let problema_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      (* Partenza *)  
    let maxColori = 3 in  (* Massimo numero di colori *)  
    let succ = Grafo x in (* Successori *)  
  
    (Problema (succ, start, maxColori))  
  ;;
```

```
[3]: val problema_1 : problema = Problema (Grafo <fun>, 3, 3)
```



1.3.2 Problema 2

Questo problema è composto da un grafo non orientato e connesso che presenta vari cicli. Ce ne sono 2 varianti:

1. la prima è quella che può essere colorata senza problemi con partenza dal *nodo 0* e numero massimo di colori $N = 4$
2. la seconda non è colorabile dato che il numero massimo di colori $N = 3$ non risultano sufficienti per la risoluzione del problema (ne servono minimo 4).

```
[4]: let problema_2_err =
      let x = function
        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]
      | _ -> [] in

      let start = 0 in      (* Partenza *)
      let maxColori = 3 in  (* Massimo numero di colori*)
      let succ = Grafo x in (* Successori *)

      (Problema (succ, start, maxColori))
;;
```

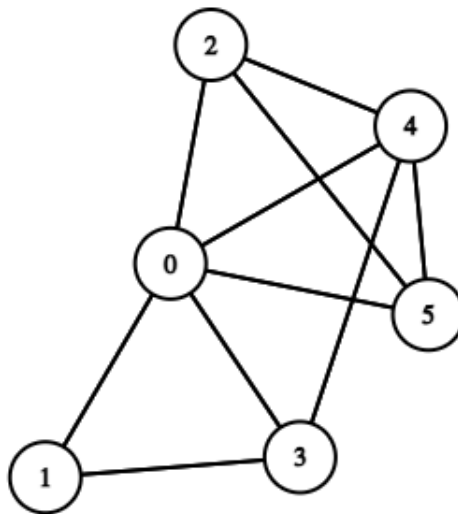
```
[4]: val problema_2_err : problema = Problema (Grafo <fun>, 0, 3)
```

```
[5]: let problema_2 =
      let x = function
        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]
      | _ -> [] in

      let start = 0 in      (* Partenza *)
      let maxColori = 4 in  (* Massimo numero di colori*)
      let succ = Grafo x in (* Successori *)

      (Problema (succ, start, maxColori))
;;
```

```
[5]: val problema_2 : problema = Problema (Grafo <fun>, 0, 4)
```



1.3.3 Problema 3

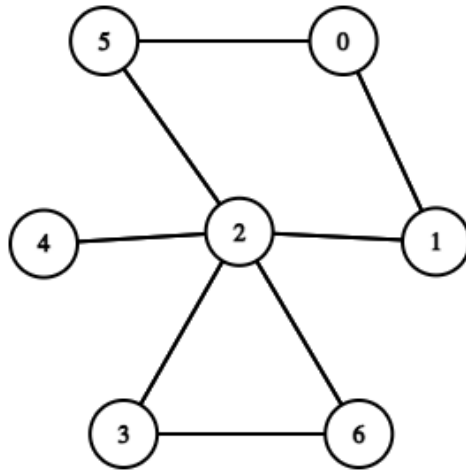
Il **problema 3** ha un altro grafo con 7 nodi, con una topologia abbastanza basilare, un numero massimo di colori $N = 3$ ed il *nodo 0* come punto di partenza.

```
[6]: let problema_3 =
      let x = function
        0 -> [1 ; 5]
      | 1 -> [0; 2]
      | 2 -> [5; 4; 3; 1; 6]
      | 3 -> [2; 6]
      | 4 -> [2]
      | 5 -> [0; 2]
      | 6 -> [2; 3]
      | _ -> [] in

      let start = 0 in      (* Partenza *)
      let maxColori = 3 in  (* Massimo numero di colori*)
      let succ = Grafo x in (* Successori *)

      (Problema (succ, start, maxColori))
;;
```

```
[6]: val problema_3 : problema = Problema (Grafo <fun>, 0, 3)
```



1.3.4 Problema 4

Il grafo di questo problema è *orientato* ed è stato scelto per verificare il comportamento dell'algoritmo. Il *nodo 0* è in collegamento con tutti gli altri, ma questi non possono raggiungerlo. Il punto di partenza è il *nodo 0* ed ha un numero di colori massimo $N = 3$.

```

[7]: let problema_4 =
      let x = function
        0 -> [1; 2; 3; 4]
        | _ -> [] in

      let start = 0 in      (* Partenza *)
      let maxColori = 3 in  (* Massimo numero di colori *)
      let succ = Grafo x in (* Successori *)

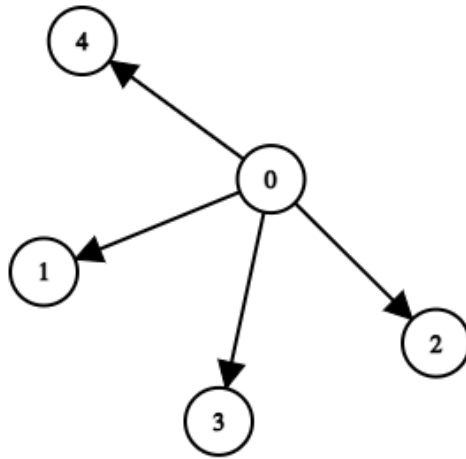
      (Problema (succ, start, maxColori))
;;

```

```

[7]: val problema_4 : problema = Problema (Grafo <fun>, 0, 3)

```



1.3.5 Problema 5

Il grafo del **problema 5** ha un numero maggiore di nodi, con la presenza di cicli e cammini ridondanti. È stato scelto per testare le performance dell'algoritmo e per accertare la sua correttezza. Ha un numero massimo di colori $N = 2$ e come punto di partenza il *nodo 0*.

```

[8]: let problema_5 =
      let x = function
        0 -> [1; 2; 6; 7]
      | 1 -> [0; 8]
      | 2 -> [0; 3]
      | 3 -> [2; 4; 5]
      | 4 -> [3; 5]
      | 5 -> [3; 4; 6; 10]
      | 6 -> [0; 5]
      | 7 -> [0; 8]
      | 8 -> [1; 7]
      | 10 -> [5]
      | _ -> [] in

      let start = 0 in      (* Partenza *)
      let maxColori = 2 in  (* Massimo numero di colori *)
      let succ = Grafo x in (* Successori *)

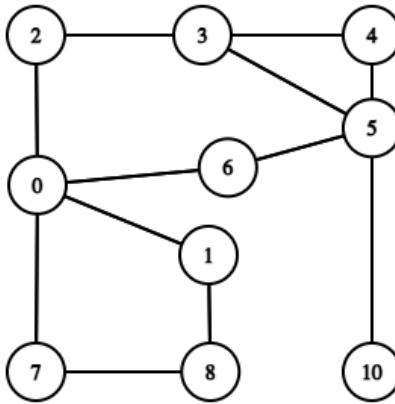
      (Problema (succ, start, maxColori))
;;

```

```

[8]: val problema_5 : problema = Problema (Grafo <fun>, 0, 2)

```

1.3.6 Problema 6

L'ultimo problema è formato da un grafo abbastanza grande, con un elevato numero di nodi, presenza di cicli e cammini ridondanti. Anche questo è stato utilizzato per vedere come si comporta l'algoritmo su grafi di modeste dimensioni.

```

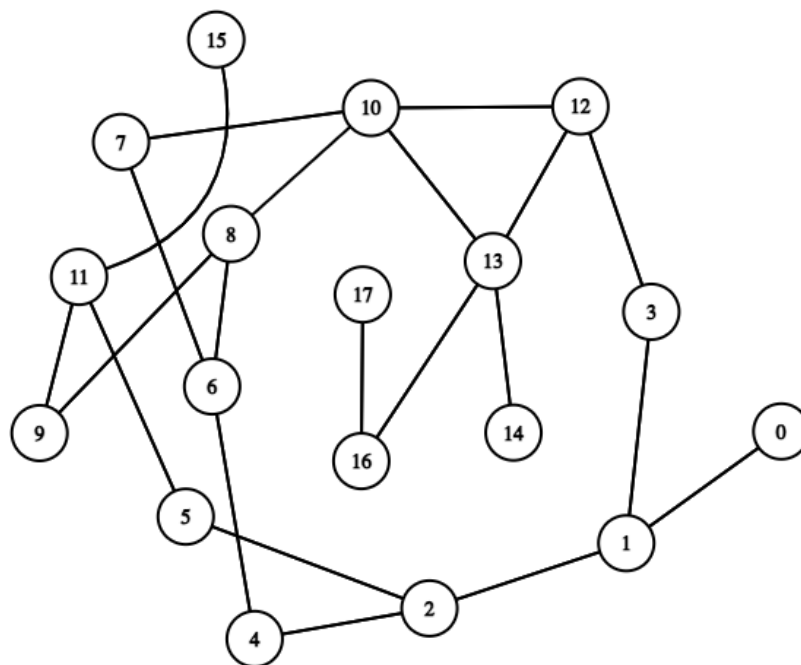
[9]: let problema_6 =
      let x = function
        0 -> [1]
      | 1 -> [0; 2; 3]
      | 2 -> [1; 4; 5]
      | 3 -> [1; 12]
      | 4 -> [2; 6]
      | 5 -> [2; 11]
      | 6 -> [4; 7; 8]
      | 7 -> [6; 10]
      | 8 -> [6; 9; 10]
      | 9 -> [8; 11]
      | 10 -> [7; 12; 13]
      | 11 -> [5; 9; 15]
      | 12 -> [3; 10; 13]
      | 13 -> [10; 12; 14; 16]
      | 14 -> [13]
      | 15 -> [11]
      | 16 -> [13; 17]
      | 17 -> [16]
      | _ -> [] in
      let start = 0 in (* Partenza *)
      let maxColori = 3 in (* Massimo numero di colori *)
      let succ = Grafo x in (* Successori *)
      (Problema (succ, start, maxColori))
    ;;

```

```

[9]: val problema_6 : problema = Problema (Grafo <fun>, 0, 3)

```



1.4 Inclusione Moduli

Includo il modulo `jupyter` per poter gestire stampe ed immagini nel notebook direttamente da OCaml, dato che i comandi speciali *Magic* (`%` e `%%`) sono solo disponibili con il kernel python 🙄.

```
[10]: #require "jupyter.notebook" ;;
      open Jupyter_notebook ;;
```

1.5 Librerie Python

Procedo all'installazione (tramite `pip`) delle librerie necessarie al funzionamento dello script python.

```
[11]: Process.sh "pip3 install matplotlib";;
      Process.sh "pip3 install pyvis";;
```

1.6 Idea dell'Algoritmo

L'idea su cui si basa questo algoritmo è quella di andare ad aggiornare i colori dei vicini di un nodo se questi hanno lo stesso colore.

Prima di iniziare la fase di colorazione viene inizializzata una lista con elementi formati da coppie (nodo, colore). Questa rappresenterà il risultato finale ed inizialmente tutti i nodi avranno colore 0. Viene creata esplorando il grafo (tramite una BFS) iniziando dal nodo di partenza e, se siamo di fronte ad un grafo sconnesso verrà presa in considerazione solo la parte di nodi raggiungibile dal quello iniziale ignorando così gli altri.

Ora avviene la fase di colorazione dove, si parte dal nodo iniziale e, sempre con una BFS, si va a scandire il grafo effettuando queste operazioni:

- Controllo che il nodo che sto esaminando non sia già stato visitato, nel caso lo ignoro e proseguo con il prossimo
- Per ogni suo vicino, controllo che il colore del vicino ed il suo siano uguali, se lo sono incremento di 1 il colore del vicino, altrimenti lascio tutto inalterato.
- Prima di effettuare l'aggiornamento del colore di un nodo controllo di non aver superato il massimo numero di colori N :
 - se ciò accade, arresto l'esecuzione e avviso l'utente che il problema non è risolvibile,
 - altrimenti procedo con l'assegnamento e l'esplorazione
- Continuo fin quando non ho esplorato tutto il grafo.

1.7 Funzioni Ausiliarie

Ritorna il colore del nodo dato. La funzione va a cercare il nodo all'interno della lista di nodi colorati (risultato) e ritorna il suo colore. La lista `colorati` è composta da coppie del tipo (nodo, colore).

```
[11]: let get_colore nodo colorati =  
  let rec aux nodo = function (*lista di nodi colorati*)  
    (x,y)::coda -> (*prende il primo elemento della forma (x,y)*)  
      if x = nodo    (* se l'elemento in questione è il nodo che cerco*)  
      then y        (*  ritorna il colore del nodo*)  
      else  
        aux nodo coda (* continua con la ricorsione*)  
  
    | _ -> (-1) (*se la lista finisce vuol dire che il nodo non esiste.*)  
  
  in aux nodo colorati  
  (*avvia la ricorsione con nodo=nodo lista_di_nodi_colorati=colorati*)  
;;
```

```
[11]: val get_colore : 'a -> ('a * int) list -> int = <fun>
```

Aggiorna i colori dei nodi adiacenti a quello in esame. Modifica la lista `risultato` incrementando (`colore + 1`) il colore dei nodi vicini a quello preso in esame solo se hanno lo stesso colore. Quando si cerca di assegnare un nuovo colore si controlla prima che non venga superato il limite massimo di colori utilizzabili, se succede, viene lanciata un'eccezione e termina il programma.

```
[12]: let incrementa_colore_nodi risultato nodi colore maxColori =
      List.map (
        fun(x, y) ->
          (*prende solo i          controlla che il colore          *)
          (*nodi vicini          del vicino sia uguale a quello del *)
          (*al nodo dato          nodo in esame                     *)
          if ((List.mem x nodi) && (colore=(get_colore x risultato)))
          then
            let nuovo_colore = y + 1 in
            if ((nuovo_colore) >= maxColori)
              (*controlla che il numero massimo di colori*)
              (*non venga superato                          *)
              then
                raise NumeroColoriInsufficiente
              else
                (x, nuovo_colore)
            else
              (x, y)
          ) risultato
      ;;
```

```
[12]: val incrementa_colore_nodi :
      ('a * int) list -> 'a list -> int -> int -> ('a * int) list = <fun>
```

Questa funzione crea una lista formata da coppie del tipo `(nodo, colore)`. Questa lista rappresenta il risultato finale della colorazione ed inizialmente viene assegnato ad ogni nodo il colore 0, che poi verrà modificato in seguito durante la risoluzione del problema. Questa lista viene creata esplorando il grafo del problema tramite una BFS.

```
[13]: let inizializza_risultato (Grafo succ) partenza =
      let rec bfs visitati risultato = function (* frontiera *)
        [] -> risultato (*esplorazione finita, ritorno il risultato*)
      | nodo::coda ->
        if List.mem nodo visitati (*se il nodo è già stato visitato*)
        then
          bfs visitati risultato coda (*lo ignoro e continuo l'esplorazione*)
        else
          bfs (*procedo con l'esplorazione*)
            (visitati@[nodo])
            (*aggiungo il nodo attuale alla frontiera*)
            (risultato@[nodo, 0])
            (*assegno il colore iniziale al nodo attuale*)
```

```

        (coda@(succ nodo))
        (*aggiungo alla coda di nodi da esaminare i vicini del nodo attuale*)

    in bfs [] [] [partenza]
;;

```

[13]: val inizializza_risultato : grafo -> int -> (int * int) list = <fun>

Genera un file contenente il risultato della colorazione che verrà passato a python per poter visualizzare il grafo in un modo più comprensibile.

```

[14]: let grafodati_file = "grafo.data";; (*file su cui viene salvato il grafo*)
let salva_grafo_colorato (Grafo succ) colorati =
  let oc = open_out grafodati_file in (*Apertura del file in scrittura*)
  let rec salva_lista =
    (*Funzione ausiliaria per salvare su file una lista data*)
    function (* lista *)
      [x] -> Printf.fprintf oc "%d" x
      (* caso base, se la lista ha un solo elemento lo stampa (senza " ")*)
    | x::coda ->
      (* caso ricorsivo, la lista ha più elementi*)
      Printf.fprintf oc "%d " x; (* stampa l'elemento*)
      salva_lista coda (* continua la ricorsione*)
    | _ -> ()
  in let rec salva =
    (*Funzione ausiliaria per salvare nodo - vicini - colore su file*)
    function (* lista nodi_colorati *)
      [] -> Printf.fprintf oc "\n"; close_out oc
      (* caso base, la lista è finita. Stampo un \n e chiudo il file*)
    | (nodo, colore)::coda ->
      (* caso ricorsivo, stampo nodo, lista vicini, colore nodo*)
      Printf.fprintf oc "%d," nodo; salva_lista (succ nodo); Printf.fprintf oc ", ";
      Printf.fprintf oc "%d\n" colore;
      salva coda
  in salva colorati
  (*avvia la funzione ausiliaria per salvare il grafo su file*)
;;

```

[14]: val salva_grafo_colorato : grafo -> (int * int) list -> unit = <fun>

Il successivo set di funzioni serve per mostrare immagini all'interno del notebook ed avviare lo script in python responsabile della rappresentazione grafica del risultato della colorazione. È anche presente una funzione per la rimozione dei file temporanei, come il risultato della risoluzione del problema che OCaml passerà a python o l'immagine generata dallo script. La funzione **rappresenta** è quella che combina tutte le altre per ottenere il risultato desiderato e questa verrà invocata solo

quando la colorazione del grafo sarà completa ed avvenuta con successo.

Nelle varie funzioni ausiliare c'è una parte di codice comune a tutte, `forza_unit`, definita come segue:

```
let forza_unit _ = ();;
```

Questo permette di ignorare il valore di ritorno di qualunque espressione passata a questa funzione ed ottenere così `unit` come valore finale di ogni espressione.

```
[15]: (*forza unit come valore di ritorno ignorando quello dell'espressione data*)
let forza_unit _ = ();;
let avvia_python = fun () ->
  forza_unit (Process.sh "python3 progetto/src/rappresentazione_grafo/
↳rappresentazione_grafo.py grafo.data injupyter")
;;
let mostra_immagine = fun () ->
  forza_unit (Jupyter_notebook.display_file ~base64:true "image/png"
↳"risultato.png")
;;
let pulisci = fun () ->
  forza_unit (Process.sh "rm grafo.data risultato.png")
;;

let stampa_risultato risultato =
  let rec aux = function (*lista da stampare*)
    [] -> () ;
    (*caso base, la lista è finita.*)
  | (nodo, colore)::coda ->
    (*caso ricorsivo, stampa l'elemento e continua*)
    forza_unit (Jupyter_notebook.display "text/html" ((string_of_int
↳nodo) ^ "-" ^ (string_of_int colore))); ()
    aux coda
  in aux risultato;
;;

let rappresenta = fun () ->
  avvia_python ();
  mostra_immagine ();
  pulisci ()
;;
```

```
[15]: val forza_unit : 'a -> unit = <fun>
```

```
[15]: val avvia_python : unit -> unit = <fun>
```

```
[15]: val mostra_immagine : unit -> unit = <fun>
```

```
[15]: val pulisci : unit -> unit = <fun>
[15]: val stampa_risultato : (int * int) list -> unit = <fun>
[15]: val rappresenta : unit -> unit = <fun>
```

1.8 Funzione Principale

```
[16]: let risolvi (Problema ((Grafo succ), partenza, maxColori)) =
  let risultato = inizializza_risultato (Grafo succ) partenza in
  let rec esplora visitati colorati =
    function (*frontiera*)
      [] -> (*fine delle ricorsione*)
        ( stampa_risultato colorati;
          salva_grafo_colorato (Grafo succ) colorati;
          rappresenta ()
        )

    | nodo::coda -> (*caso ricorsivo, continua a colorare*)
      if List.mem nodo visitati (*se il nodo è già stato visitato*)
      then esplora visitati colorati coda
        (* ignora il nodo ed estrae il successivo dalla frontiera*)

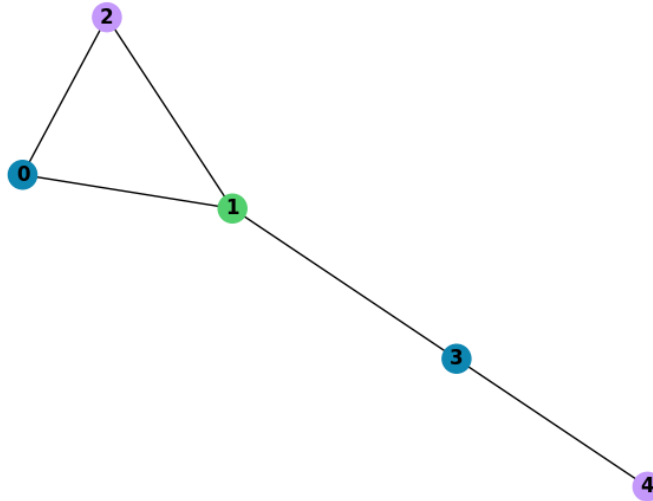
      else
        (*continua con la ricorsione espandendo i nodi vicini al nodo*)
        esplora
          (visitati@[nodo])
          (*aggiunge il nodo attuale ai visitati*)
          (incrementa_colore_nodi
            colorati (succ nodo)
            (get_colore nodo colorati)
            maxColori
          )
          (*colora i nodi vicini a quello attuale *)
          (coda@(succ nodo))
          (*espande i vicini del nodo*)

  in esplora [] risultato [partenza]
;;
```

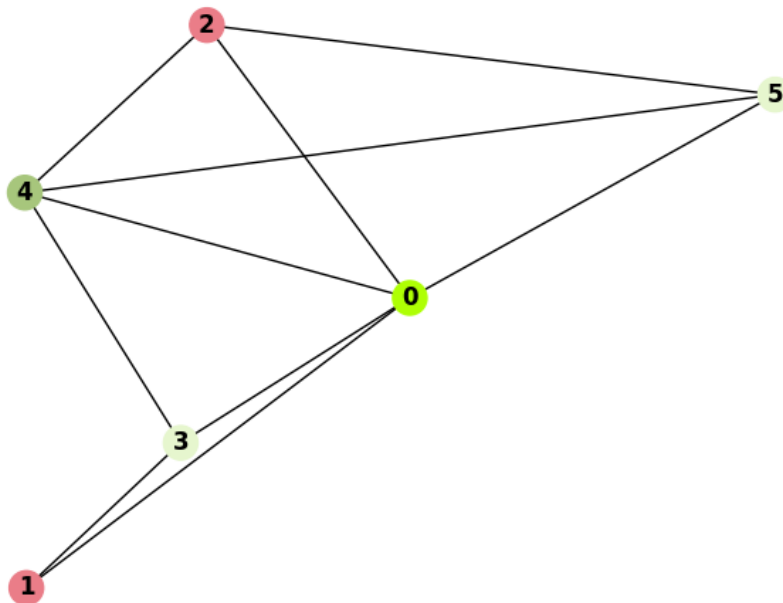
```
[16]: val risolvi : problema -> unit = <fun>
```

1.9 Test dell'Algoritmo

```
[17]: risolvi problema_1;;
```



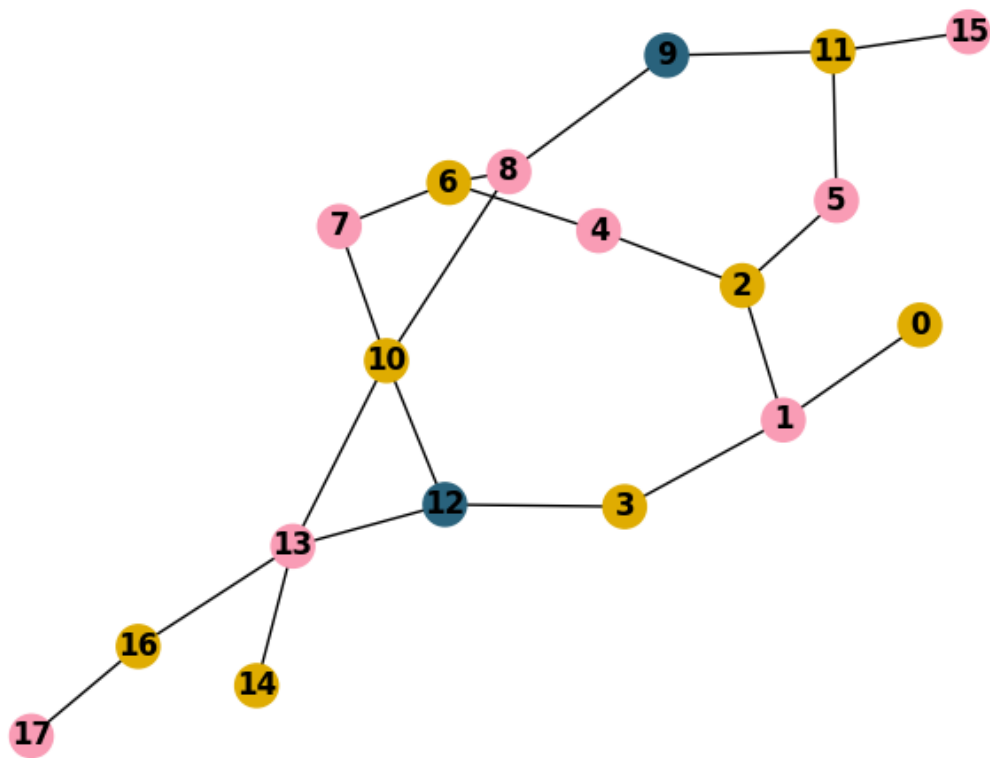
```
[18]: risolvi problema_2;;
```



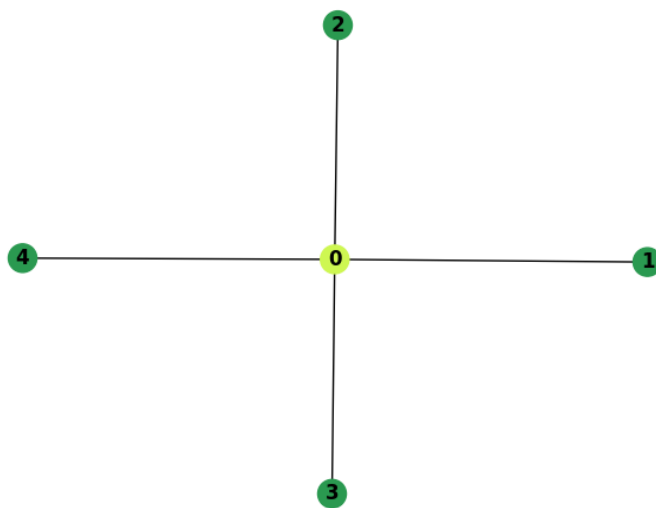
```
[19]: risolvi problema_2_err;;
```

Exception: NumeroColoriInsufficiente.


```
[20]: risolvi problema_6;;
```



```
[21]: risolvi problema_4;;
```



2 I ❤️ Jupyter

Oltre alla stesura di questo notebook (che presenta una versione certamente funzionante, seppur minimale, dell'algoritmo) ho riscritto tutto il progetto in modo tale che sia compilabile ed eseguibile anche da terminale approfondendo anche altri argomenti e funzionalità del linguaggio.

È disponibili all'interno della cartella **progetto** e presenta la seguente struttura:

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

- **Makefile**: utilizzato per compilare il progetto e generare l'eseguibile
- **main**: contiene la funzione principale che si occupa di avviare la risoluzione dei problemi e di gestire le scelte dell'utente.
- **graphUtils**: insieme di funzioni per risolvere il problema della colorazione.
- **data**: insieme di problemi 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 menù 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.

2.1 Makefile

È il 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. Questi vengono lasciati all'interno della cartella **src** e non spostati in quella designata alla fase di building per permettere il corretto funzionamento delle estensioni dell'editor.
- **.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
cinclue = -I src

srcdir = src
builddir = build
bindir = bin

target = $(bindir)/exe

.PHONY: clear

all: cartelle $(target)

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

$(builddir)/graphUtils.cmo: $(srcdir)/graphUtils.mli $(srcdir)/graphUtils.ml
    $(cc) $(cinclue) $(cflags) $^
    mv $(srcdir)/graphUtils.cmo $(builddir)

$(builddir)/printer.cmo: $(srcdir)/printer.mli $(srcdir)/printer.ml
    $(cc) $(cinclue) $(cflags) $^
    mv $(srcdir)/printer.cmo $(builddir)

$(builddir)/data.cmo: $(srcdir)/data.mli $(srcdir)/data.ml
    $(cc) $(cinclue) $(cflags) $^
    mv $(srcdir)/data.cmo $(builddir)

$(builddir)/main.cmo: $(srcdir)/main.mli $(srcdir)/main.ml
    $(cc) $(cinclue) $(cflags) $^
    mv $(srcdir)/main.cmo $(builddir)

cartelle: $(builddir) $(bindir)

$(builddir):
    mkdir -p $@

$(bindir):
```

```

mkdir -p $@

clear:
    rm -f $(builddir)/*.cmo
    rm -f $(bindir)/*
    rm -f $(srcdir)/*.cmi

```

Contenuto di Makefile

Un esempio pratico di come eseguire la fase di compilazione:

```

[29]: (* compilazione del progetto *)
      Process.sh "cd progetto; make; ls -LR";;

      (* pulizia e rimozione dei file compilati *)
      Process.sh "cd progetto; make clear; ls -LR";;

```

```

[29]: 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

```

```

[29]: rm -f build/*.cmo
      rm -f bin/*
      rm -f src/*.cmi

```

2.2 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 questo li supporti). La funzione più degna di nota è `stampa_problema` che, dato un problema, stampa a schermo tutti i suoi dati:

- il grafo
- il nodo di partenza
- il massimo numero di colori N

```
type colore = Colore of string ;;
```

```

(* normali *)
let rosso = Colore "\027[31 m";;

```

```

let verde = Colore " \027[32 m";;
let ciano = Colore " \027[36 m";;
let bianco = Colore " \027[0 m";;

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

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

Alcune definizioni di colori

let stampa_problema (Problema((Grafo 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 *)

```

Codice della funzione *stampa_problema*

```

Partenza: 3 Max Colori: 3
3 → 1 4
1 → 0 2 3
4 → 3
0 → 1 2
2 → 0 1

```

Output della funzione

2.3 Data

In questo file ci sono tutte le definizioni dei problemi 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, gli ID unici, ecc; ed una funzione che dato un numero (ID), ritorna uno dei precedenti problemi. È anche presente una lista (`problemi`) che contiene tutti i problemi disponibili con una descrizione (una coppia (`problema`, `string`)). È

questa che permette di aggiungere e togliere elementi in maniera semplice senza dover modificare alcuna funzione.

```
let problemi = [
  ( problema_1 , " Grafo 1" ) ;
  ( problema_2 , " Grafo 2 con numero sufficiente di colori " ) ;
  ( problema_2_err , " Grafo 2 con colori insufficienti " ) ;
  ( problema_3 , " Grafo 3" ) ;
  ( problema_4 , " Grafo 4" )
];;
```

Parte della lista `problemi`

```
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
```

Output per la selezione del problema da risolvere

Infine, la funzione `scegli_problema` che, dato un ID, ritorna il relativo problema. È interessante notare come per selezionare un dato problema non viene scorsa ricorsivamente tutta la lista, ma quest'ultima 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 superiori 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_problema id_problema =
  let array_tmp = Array.of_list problemi in (* lista in array *)
    if id_problema > Array.length array_tmp || id_problema < 1
    then
      raise SceltaErrata (* lancia un ' eccezione *)
    else
      fst ( Array.get array_tmp ( id_problema -1 ) )
;;
```

Codice della funzione `scegli_problema`

2.4 GraphUtils

Nel file `graphUtils.ml` possiamo trovare tutte le funzioni implementate ed analizzate fin ora che riguardano l'algoritmo di colorazione. La parte delle *funzioni ausiliarie* è esattamente identica, l'unica differenza possiamo trovarla nella funzione `risolvi`, dove sono state apportate alcune piccole modifiche per permettere l'esecuzione da riga di comando:

- Ora ritorna il risultato (la lista `colorati`) e non stampa o mostra a schermo più nulla, sarà un problema del `main`,
- Lo script python non viene più fatto partire qui dentro ma sarà compito del `main` avviarlo.

```

let risolvi (Problema ((Grafo succ), partenza, maxColori)) =
  let risultato = inizializza_risultato (Grafo succ) partenza in
  let rec esplora visitati colorati =
    function (*frontiera*)
      [] -> (*fine delle ricorsione*)
        (salva_grafo_colorato (Grafo succ) colorati; colorati)

    | nodo::coda -> (*caso ricorsivo, continua a colorare*)
      if List.mem nodo visitati (*se il nodo è già stato visitato*)
      then esplora visitati colorati coda (* ignora il nodo e continua*)

      else
        (*continua con la ricorsione espandendo i nodi vicini al nodo*)
        esplora
          (visitati@[nodo]) (*aggiunge il nodo attuale ai visitati*)
          (incrementa_colore_nodi
            colorati
            (succ nodo)
            (get_colore nodo colorati) maxColori)
          (coda@(succ nodo))
          (*espande i vicini del nodo attuale e li aggiunge alla frontiera*)

  in esplora [] risultato [partenza]
;;

```

Codice della funzione *risolvi* in *graphUtils.ml*

Qui vengono anche definiti i tipi **grafo** e **problema**, con i relativi costruttori di tipo **Grafo** e **Problema**. Viene anche creata l'eccezione **NumeroColoriInsufficiente** che servirà per segnalare che il dato problema non è risolvibile per il numero insufficiente di colori selezionato.

2.5 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 problema testare l'algoritmo di colorazione, avvia la risoluzione del problema selezionato, stampa il risultato ed avvia lo script python.

```

let main () =
  stampa_logo ();
  stampa_problemi_disponibili ();

  let dati =
    let rec aux () = (* utente sceglie il problema *)
      try (* controllo input valido *)
        scegli_problema ( scelta () )
      with SceltaErrata ->
        aux ()
    in aux ()

```

```

in let avvia_colorazione (Problema(g, partenza, maxColori)) =
  print_string "\nIl Problema selezionato : \n\n";
  stampa_problema ( Problema (g , partenza , maxColori ) ) ;
  print_string " Coloro ...\ n\n";

  (* colora il grafo *)
  let colorati = risolvi (Problema(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 NumeroColoriInsufficiente ->
    stampa_errore ()
;;

```

Parte del codice del file main.ml

Alla riga 2 – 3 viene stampato sul terminale il logo iniziale e tutti i problemi 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 problema fin quando l'utente non seleziona un ID esistente. Una volta selezionato un ID valido conterrà il problema 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 ;;

```

Inclusione degli altri file .ml

2.6 Python

All'interno del progetto è anche presente un breve script in python che verrà avviato una volta risolto il problema della colorazione. 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.

Presenta 2 modalità di utilizzo:

- **notebook**: in questa modalità produce in output un'immagine compatibile con i notebook di jupyter. Sarà poi compito di OCaml andare ad importarla e disegnarla.
- **eseguibile**: questa modalità viene utilizzata quando lo script è invocato dal progetto compilato, genera una pagina HTML, che verrà aperta in automatico, con la visualizzazione interattiva del grafo colorato.


```

import networkx as nx
from pyvis.network import Network
import matplotlib.pyplot as plt

def main(args):
    # controlla che il nome del file sia stato passato
    if len(args) <= 0: return

    fname = args[0]
    in_jupyter = True if len(args) >= 2 else False

    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 che la riga sia valida
                nodo, vicini, colore = text.split(",")

                vicini = vicini.split(" ")
                colori.append(int(colore))

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

    # controlla che abbia trovato tutti i nodi
    nodi = trova_nodi(collegamenti)
    # converte i colori da int a stringa hex per pyvis
    colori = aggiusta_colori(colori)

    rappresenta_grafo(nodi, collegamenti, colori, in_jupyter)

...

def rappresenta_grafo (nodi, collegamenti, colori, in_jupyter):
    if in_jupyter:
        G = nx.Graph()

        G.add_nodes_from(nodi)

```

```

G.add_edges_from(collegamenti)

nx.draw(
    G,
    node_color=colori,
    with_labels=True,
    font_weight='bold'
)
plt.savefig("risultato.png")

else:
    net = Network(
        width='100%',
        height='600px',
        directed=False
    )

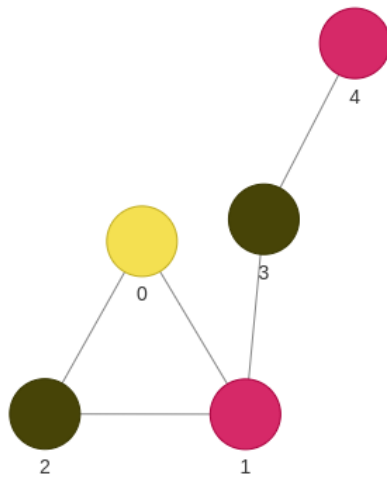
    net.add_nodes(
        nodi,
        label=[f"{x}" for x in nodi],
        color=colori
    )

    net.add_edges(collegamenti)

    net.toggle_physics(True)
    net.inherit_edge_colors(False)
    net.write_html('rappresentazione_grafo.html')

```

Parte delle funzioni dello script python



Output dello script python

2.7 Dimostrazione di Esecuzione

Una volta compilato il progetto tramite il comando **make**, è prima necessario spostarsi all'interno della cartella **bin** per poi avviare l'eseguibile:

```
cd bin
./exe
```

Verrà mostrato il menù iniziale e l'utente dovrà scegliere quale problema 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 problema 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.

```
→ bin git:(main) ✖ ./exe

OCAML
GRAPH COLORING

Un programma scritto in OCaml in grado di risolvere il problema della colorazione di un grafo.

HELLO Utente 🐼 !
Coloriamo un po' di grafi 🖋️
Di seguito trovi un elenco dei grafi di default tra cui puoi scegliere.

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

> █
```

Menù iniziale

```
→ bin git:(main) ✖ ./exe

OCAML
GRAPH COLORING

Un programma scritto in OCaml in grado di risolvere il problema della colorazione di un grafo.

HELLO Utente 🐼 !
Coloriamo un po' di grafi 🖋️
Di seguito trovi un elenco dei grafi di default tra cui puoi scegliere.

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

> 9
> 10
> 111
> 987
> █
```

Esempio di multiple scelte errate di ID

```
> 1

Il Grafo selezionato:

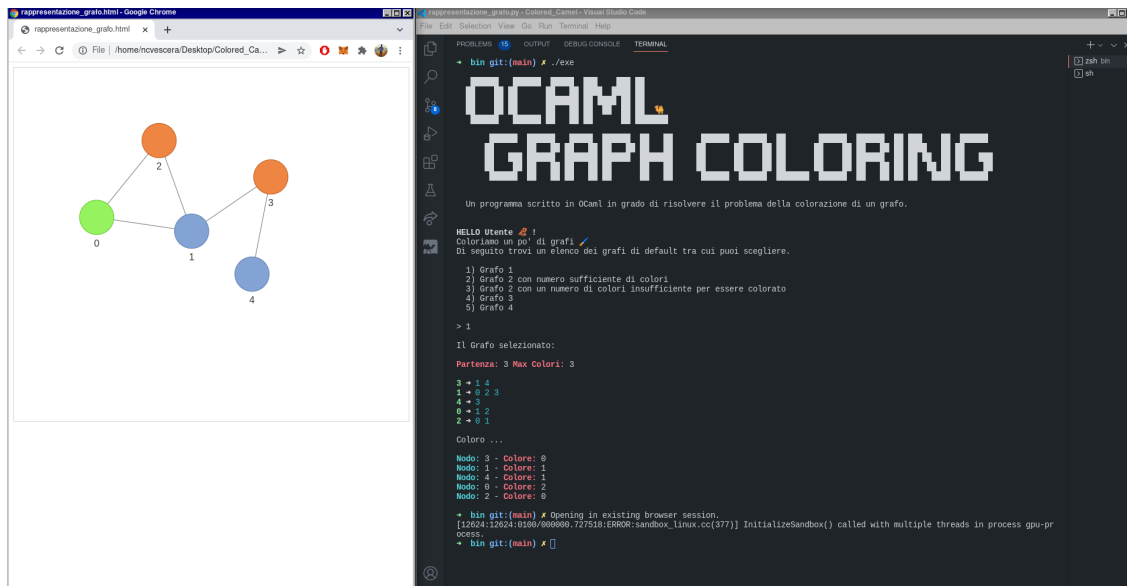
Partenza: 3 Max Colori: 3

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
```

Esempio di risoluzione di un problema



Comportamento dello script python

```
> 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) █
```

Messaggio di errore per numero di colori insufficienti

3 Ambiente di Sviluppo 🐳

L'intero progetto è stato realizzato all'interno di un **container docker** utilizzato come ambiente di sviluppo, questo mi ha permesso di risparmiare notevole tempo nella configurazione e installazione di tutto il software necessario al funzionamento di OCaml, ma soprattutto mi dà la possibilità di avere un ambiente di sviluppo portatile che può essere condiviso e avviato su qualunque macchina in pochissimo tempo.

Il tutto si basa sull'estensione [Remote Containers](#) per l'editor Visual Studio Code che permette di utilizzare un container docker come un ambiente di sviluppo *full-featured*, di poter aprire qualunque cartella all'interno del container e gestirla tramite l'editor avendo a disposizione tutte le funzionalità che ci offre. È anche possibile scegliere quali estensioni di VSCode andare ad installare e utilizzare all'interno del nostro nuovo ambiente di sviluppo.

Il file `devcontainer.json` (presente all'interno della cartella `.devcontainer`) è quello che si occuperà di dire a VSCode come accedere, o creare, il container e quali elementi e strumenti andare a utilizzare. Di seguito un esempio:

```
{
  "name": "ocamldev",
  "build": {
    "dockerfile": "Dockerfile",
    "context": ".."
  },

  "features": {
    "git": "latest",
    "github-cli": "latest"
  },
  "customizations": {
    "vscode": {
      "extensions": [
        "ocaml-labs.ocaml-platform",
        "ms-python.python",
        "ms-toolsai.jupyter"
      ]
    }
  },

  "remoteUser": "opam"
}
```

Possiamo notare alcuni campi importanti come:

- **build**: che indica il Dockerfile da utilizzare per la generazione del container
- **customizations**: che permette di configurare le impostazioni dell'editor e scegliere anche quali estensioni andare ad installare
- **remoteUser**: specifica l'utente con cui verrà effettuato l'accesso al container, di default è `root`

Il file `Dockerfile` è responsabile della creazione del nuovo container. Il seguente codice è quello utilizzato per il mio ambiente di sviluppo:

```
FROM ocaml/opam:latest

RUN sudo apt-get update && sudo apt upgrade -y
RUN sudo apt-get install -y zlib1g-dev \
    libffi-dev libcairo2-dev libgmp-dev libzmq5-dev pkg-config
RUN sudo apt install -y make python3 python3-pip
RUN pip3 install jupyter
ENV PATH $PATH:/home/opam/.local/bin
RUN opam update
RUN opam user-setup install
RUN opam install -y ocaml-lsp-server
RUN opam install -y merlin
RUN opam install jupyter
RUN opam upgrade jupyter
RUN grep topfind ~/.ocamlinit || echo '#use "topfind";;' >> ~/.ocamlinit
RUN grep Topfind.log ~/.ocamlinit || echo 'Topfind.log:=ignore;;' >> ~/.ocamlinit
RUN opam exec -- ocaml-jupyter-opam-genspec
RUN jupyter kernelspec install \
    --user --name ocaml-jupyter \
    "$(opam config var share)/jupyter"
```

Sono partito da un'immagine già configurata per il corretto funzionamento di OCaml (solo con questa si ha un ambiente pronto per scrivere ed eseguire codice) per poi modificarla andando ad aggiungere alcuni pacchetti necessari al mio progetto (`make`, `python` e le varie librerie per python). Ho poi deciso d'installare un server jupyter (che può essere avviato all'evenienza) con un `kernel` in grado di eseguire OCaml all'interno dei notebook. Non è potente come quello di python ma risulta comunque utilizzabile e più che sufficiente per le mie necessità.