Progetto di Ricerca Operativa
Map Coloring (con variante) in AMPL

Vetere Francesco
Matricola: 313336
(francesco.vetere@studenti.unipr.it)

Indice

1 Descrizione del problema

Il problema affrontato in questo progetto consiste nell'implementazione in linguaggio `AMPL` del problema del `Map Coloring`, e di una sua interessante variante. Risolvere il problema del `Map Coloring` significa assegnare ad ogni regione di una mappa geografica un colore, in modo tale che due regioni confinanti abbiano sempre colori diversi. Il numero di colori cercato é ovviamente quello minimo (utilizzando sempre un numero di colori pari al numero di regioni infatti, il problema diventerebbe banale).
Il problema puó essere rappresentato tramite un grafo $G = (V, E)$ in cui:

- $V = \{v_1, ..., v_n\}$ rappresenta l'insieme delle n regioni della mappa geografica
- $E = \{(v_i, v_j)\}$ rappresenta l'insieme delle coppie di regioni i e j che sono confinanti sulla mappa geografica

Una variante del problema appena descritto consiste nel cercare di eliminare un numero di regioni minimo in modo tale che la mappa risulti colorabile con un certo numero k di colori. Anche in questa variante é subito evidente come prendendo $k = |V|$ il problema diventi banale.

Nel seguito di questo elaborato, si indicheranno con `map_coloring` e `map_coloring_2` rispettivamente il primo problema ed il secondo.

2 Modello matematico

Vengono di seguito mostrati i modelli matematici dei due problemi.

2.1 map_coloring

`map_coloring`

2.2 map_coloring_2

`map_coloring_2`

3 Modello AMPL

Vengono di seguito mostrati i file `.mod` dei due problemi.

3.1 map_coloring.mod

`map_coloring.mod`

```
# Insieme di nodi
set NODES;

# Insieme di archi (un arco e' individuato da una coppia di nodi)
set EDGES within (NODES cross NODES);

# Insieme di colori (cardinalita' sufficientemente elevata per trovare una
     soluzione ammissibile)
set COLORS;

# node_color[n, c] = 1 se il nodo n e' assegnato al colore c, 0 altrimenti
var node_color {NODES, COLORS} binary;

# color_used[c] = 1 se il colore c e' utilizzato, 0 altrimenti
var color_used {COLORS} binary;

# vincolo che impone l'assegnamento di un solo colore per ogni nodo
# (viene generato un vincolo per ogni elemento di NODES)
subject to one_color_per_node {n in NODES}:
    sum {c in COLORS} node_color[n, c] = 1;

# vincolo che impone l'assegnamento di colori diversi per due nodi adiacenti
# (viene generato un vincolo per ogni elemento dell'insieme formato dal prodotto
     cartesiano di EDGES e COLORS)
subject to different_color_adjacent {(i,j) in EDGES, c in COLORS}:
    node_color[i, c] + node_color[j, c] <= color_used[c];

# L'obiettivo e' minimizzare il numero di colori utilizzati
minimize num_colors:
    sum {c in COLORS} color_used[c];
```

Il file modella in linguaggio `AMPL` quanto visto nell'analogo modello matematico.
Per prima cosa vengono dichiarati gli insiemi che caratterizzano il problema: `NODES` e `EDGES` descrivono il grafo (`EDGES` é contenuto nel prodotto cartesiano `NODES` x `NODES`), mentre `COLORS` é l'insieme contenente i colori assegnabili.

Seguono poi le dichiarazioni di due variabili: `node_color` e `color_used`.
La prima modella un array bidimensionale avente indici sull'inieme dei nodi e dei colori, in cui ogni valore, vincolato ad essere di tipo binario, indica se un particolare nodo é assegnato ad un particolare colore o meno.
La seconda modella un array monodimensionale avente indici sull'insieme dei colori, con valori ancora binari che indicano se un particolare colore é stato

scelto per l'assegnamento o meno.

Vengono dichiarati poi i due vincoli che caratterizzano il problema: `one_color_per_node` e `different_color_adjacent`.
La prima dichiarazione genera tanti vincoli quanti sono gli elementi dell'insieme `NODES`. Ogni vincolo generato impone che, per quel particolare nodo, la somma dei colori assegnati sia esattamente pari a 1 (questo per evitare assegnamenti di 0 oppure 2 colori per un singolo nodo).
La seconda dichiarazione genera un vincolo per ogni elemento dell'insieme formato dal prodotto cartesiano di `EDGES` e `COLORS`. Ogni vincolo generato impone che due nodi adiacenti non possano essere assegnati allo stesso colore (altrimenti si avrebbe una colorazione non valida).

Viene infine esplicitato l'obiettivo `num_colors`, che semplicemente minimizza la somma di tutti i colori utilizzati.

3.2 map_coloring_2.mod

map_coloring_2.mod

```
# Insieme di nodi
set NODES;

# Insieme di archi (un arco e' individuato da una coppia di nodi)
set EDGES within (NODES cross NODES);

# Insieme di colori (cardinalita' sufficientemente elevata per trovare una
     soluzione ammissibile)
set COLORS;

# node_color[n, c] = 1 se il nodo n e' assegnato al colore c, 0 altrimenti
var node_color {NODES, COLORS} binary;

# color_used[c] = 1 se il colore c e' utilizzato, 0 altrimenti
var color_used {COLORS} binary;

# node_deleted[n] = 1 se il nodo n e' stato eliminato, 0 altrimenti
var node_deleted {NODES} binary;

# vincolo che impone l'assegnamento di un solo colore per ogni nodo
# (viene generato un vincolo per ogni elemento di NODES)
subject to one_color_per_node {n in NODES}:
    sum {c in COLORS} node_color[n, c] = 1 - node_deleted[n];

# vincolo che impone l'assegnamento di colori diversi per due nodi adiacenti
# (viene generato un vincolo per ogni elemento dell'insieme formato dal prodotto
     cartesiano di EDGE e COLOR)
subject to different_color_adjacent {(i,j) in EDGES, c in COLORS}:
    node_color[i, c] + node_color[j, c] <= color_used[c];

# L'obiettivo e' minimizzare i nodi eliminati
minimize min_deleted:
    sum{n in NODES} node_deleted[n];
```

Questa variante é molto simile al problema originario.
Viene introdotta una nuova variabile, `node_deleted`, che modella un array monodimensionale avente indici sull'insieme dei nodi, con valori binari che indicano se un nodo ' stato eliminato o meno.

Cambia anche il vincolo `one_color_per_node`: per ogni nodo, é richiesto che la somma dei colori asegnati sia ora pari a `1 - node_deleted[n]`, per tenere conto del fatto che ora un nodo potrebbe anche non avere alcun colore assegnato, dal momento che si é scelto di eliminarlo.

Anche l'obiettivo é diverso: `min_deleted` richiede infatti che sia minimizzata la somma dei nodi eliminati.

4 Esempi di esecuzione

In questo capitolo vengono analizzati alcuni esempi di file `.dat` per entrambi i problemi.
Per testare l'esecuzione del modello corredato con i suoi dati, si utilizza per entrambi un file `.run`, molto simile per entrambi i problemi:

`map_coloring.run`

```
reset;
option solver '../ampl/gurobi';
model map_coloring.mod;
data map_coloring_dat/map_coloring_es1.dat;
solve;

display node_color;
display color_used;
```

`map_coloring_2.run`

```
reset;
option solver '../ampl/gurobi';
model map_coloring_2.mod;
data map_coloring_2_dat/map_coloring_2_es1.dat;
solve;

display node_color;
display color_used;
display node_deleted;
```

Come si nota, si é scelto di mostrare in output il contenuto delle variabili al termine della risoluzione da parte del solver, in modo da poter trarre qualche conclusione sui risultati ottenuti.

4.1 map_coloring

4.1.1 map_coloring_es1.dat

Per quanto riguarda `map_coloring`, analizziamo intanto il caso di un grafo completo, come descritto dalla seguente immagine:

$---Immaginegrafocompleto--$

Il file `.dat` che implementa questa situazione é il seguente:
`map_coloring_es1.dat`

```
set COLORS := red blue green yellow orange;

set NODES := A B C D E F;

set EDGES := (A, B) (A, C) (A, D) (A, E)
        (B, A) (B, C) (B, D) (B, E)
        (C, A) (C, B) (C, D) (C, E)
```

```
(D, A) (D, B) (D, C) (D, E)
(E, A) (E, B) (E, C) (E, D)
```

Si noti che é fondamentale definire un numero di colori sufficientemente alto per poter ottenere una soluzione ammissibile del problema: un upper bound sicuramente valido é dato dal numero di nodi presenti nel grafo.

Dopo aver lanciato `map_coloring.run` si ottiene il seguente risultato:

```
Gurobi 8.1.0: optimal solution; objective 5
6 simplex iterations
node_color [*,*]
: blue green orange red yellow    :=
A   0     0     0     1    0
B   0     0     0     0    1
C   0     0     1     0    0
D   0     1     0     0    0
E   1     0     0     0    0
F   0     0     0     1    0
;

color_used [*] :=
  blue  1
 green  1
orange  1
   red  1
yellow  1
;
```

Notiamo che il solver trova una soluzione ottima eseguendo 6 iterazioni dell'algoritmo del simplesso.
Il valore ottimo, ossia 5, era in questo caso facilmente prevedibile: trattandosi di un grafo completo, ogni nodo é adiacente ai restanti $|V| - 1$ nodi: questo implica il fatto che ad ogni nodo dovrá necessariamente essere assegnato un colore differente.
Il contenuto delle variabili ci conferma quanto ipotizzato: la variabile `node_color` mostra che ad ogni nodo é assegnato esattamente un colore (questo deve essere vero per qualsiasi tipo di grafo, in quanto é una diretta conseguenza del vincolo `one_color_per_node`) mentre la variabile `color_used` mostra che tutti i colori sono stati utilizzati.

Una possibile soluzione ottima trovata dal solver é la seguente:

$---Immaginegrafocompletocolorato---$

4.1.2 map_coloring_es2.dat

Analizziamo ora il caso di un grafo orientato non completo, come illustrato dalla seguente immagine:

$---Immaginegrafononcompleto---$

Fornendo sempre un numero di colori sufficientemente elevato, il file `.dat` che implementa questa situazione é il seguente:
`map_coloring_es2.dat`

```
set COLORS := red blue green yellow orange;

set NODES := A B C D E F G;

set EDGES := (A, B)
        (B, A) (B, C) (B, D) (B, E) (B, F)
        (C, B) (C, D)
        (D, B) (D, C) (D, E) (D, G)
        (E, F) (E, B) (E, D) (E, G)
        (F, B) (F, E)
        (G, D) (G, E)
```

Dopo aver lanciato `map_coloring.run` si ottiene il seguente risultato:

```
Gurobi 8.1.0: optimal solution; objective 3
30 simplex iterations
node_color [*,*]
: blue green orange red yellow    :=
A   0     0     0     0    1
B   0     1     0     0    0
C   1     0     0     0    0
D   0     0     0     0    1
E   1     0     0     0    0
F   0     0     0     0    1
G   0     1     0     0    0
;

color_used [*] :=
  blue  1
 green  1
orange  0
   red  0
yellow  1
;
```

Notiamo che il solver trova una soluzione ottima eseguendo 30 iterazioni dell'algoritmo del simplesso.
Come evidente dal valore ottimo, il numero minimo di colori per ottenere una

colorazione valida per il grafo é 3.
Utilizzando infatti i colori `blue`, `green` e `yellow`, notiamo che il solver riesce correttamente ad assegnare ad ogni nodo uno ed un solo colore, come richiesto dal problema.

4.2 map_coloring_2

4.2.1 map_coloring_2_es1.dat

Per quanto riguarda `map_coloring_2`, analizziamo intanto il caso del grafo completo visto in `map_coloring`.
Lasciando invariato il numero di colori (ossia 5), si ha che banalmente il solver non elimina alcun nodo, poiché 5 é giá il numero minimo di colori per cui il grafo é colorabile.
Il file `.dat` associato a questa situazione é lo stesso visto per `map_coloring`:

map_coloring_2_es1.dat

```
set COLORS := red blue green yellow orange;

set NODES := A B C D E F;

set EDGES := (A, B) (A, C) (A, D) (A, E)
       (B, A) (B, C) (B, D) (B, E)
       (C, A) (C, B) (C, D) (C, E)
       (D, A) (D, B) (D, C) (D, E)
       (E, A) (E, B) (E, C) (E, D)
```

Come previsto, il solver restituisce un output di questo tipo:

```
Gurobi 8.1.0: optimal solution; objective 0
9 simplex iterations
node_color [*,*]
: blue green orange red yellow    :=
A   0     0     0     0    1
B   0     1     0     0    0
C   0     0     1     0    0
D   1     0     0     0    0
E   0     0     0     1    0
F   0     0     0     1    0
;

color_used [*] :=
  blue  1
 green  1
orange  1
   red  1
```

```
yellow  1
;

node_deleted [*] :=
A  0
B  0
C  0
D  0
E  0
F  0
;
```

4.2.2 map_coloring_2_es2.dat

Analizziamo ora il caso dello stesso grafo non completo visto in `map_coloring`:

$---Immaginegrafononcompleto---$

Dalla soluzione ottenuta precedentemente, sappiamo che il grafo richiede un minimo di 3 colori per poter essere colorato in maniera ammissibile. Proviamo dunque a fissare un numero di colori $k = 2$: il solver, se possibile, dovrá scegliere di eliminare alcuni nodi al fine ottenere una colorazione valida.

Il file `.dat` che implementa questa situazione é il seguente:
`map_coloring_2_es2.dat`

```
set COLORS := red blue;

set NODES := A B C D E F G;

set EDGES := (A, B)
        (B, A) (B, C) (B, D) (B, E) (B, F)
        (C, B) (C, D)
        (D, B) (D, C) (D, E) (D, G)
        (E, F) (E, B) (E, D) (E, G)
        (F, B) (F, E)
        (G, D) (G, E)
```

Dopo aver lanciato `map_coloring_2.run` si ottiene il seguente risultato:

```
Gurobi 8.1.0: optimal solution; objective 2
13 simplex iterations
1 branch-and-cut nodes
node_color :=
A blue   1
A red    0
B blue   0
```

```
B red    0
C blue   1
C red    0
D blue   0
D red    0
E blue   0
E red    1
F blue   1
F red    0
G blue   1
G red    0
;

color_used [*] :=
blue  1
 red  1
;

node_deleted [*] :=
A  0
B  1
C  0
D  1
E  0
F  0
G  0
;
```

Notiamo che il solver riesce ad eliminare un certo numero di nodi per ottenere una colorazione valida con 2 colori: una soluzione ottima é infatti trovata eseguendo 13 iterazioni dell'algoritmo del simplesso e 1 iterazione dell'algoritmo branch and bound.
Il valore ottimo é in questo caso 2: affinché il grafo sia 2-colorabile, é necessario eliminare 2 nodi. Come evidenziato dal contenuto delle variabili, una possibile soluzione ottima consiste nell'eliminare i nodi `B` e `D`: cosí facendo, il grafo risultante é 2-colorabile:

$$---Grafononcompletodopoeliminazionedi\,B\,e\,D\,colorato---$$

4.2.3 map_coloring_2_es3.dat

Analizziamo ora il caso di un albero binario:

$---Immaginealberobinario---$

In generale, un albero binario necessita di almeno 2 colori per poter essere colorabile. Analizziamo il comportamento del solver nel caso venga richiesto su questo albero una 1-colorazione, scrivendo un file .dat come il seguente:

map_coloring_2_es3.dat

```
set COLORS := red;
set NODES := A B C D E F G;

set EDGES := (A, B) (A, C)
        (C, D) (C, E)
        (D, F) (D, G)
```

Dopo aver lanciato `map_coloring_2.run` si ottiene il seguente risultato:

```
Gurobi 8.1.0: optimal solution; objective 3
node_color :=
A red   0
B red   1
C red   0
D red   0
E red   1
F red   1
G red   1
;

color_used [*] :=
red  1
;

node_deleted [*] :=
A  1
B  0
C  1
D  1
E  0
F  0
G  0
;
```

Notiamo che il solver é costretto ad eliminare qualsiasi collegamento esistente tra i nodi: una soluzione ottima é quella di eliminare i nodi `A`, `C` e `D`.

Cosí facendo, il grafo risultante essere 1-colorabile (il solver sceglie il colore `red` per la colorazione):

$---Alberodopoeliminazionedi A, C, Dcolorato---$