Intelligenza Artificiale (2020/21)

In questo elaborato ho implementato il **cutset conditioning**, un algoritmo che preso in input un grafo approssimabile ad un albero trova tale albero. Successivamente si implementa anche il **map coloring,** il problema con vincoli. Non avendo mai programmato in Python, ho dovuto prendere diversi spunti da siti e documentazioni online. Inizialmente ho cercato un modo per poter raffigurare i grafi e ho utilizzato i dizionari del Python dove ogni chiave (**dict.keys**) rappresenta un vertice e i valori di esse (**dict.values**) rappresentano a quali vertici sono connessi.

Dopo aver implementato le varie funzioni che mi sarebbero tornate utili per creare grafi tramite dizionari, ho creato una funzione (**generate_graph**) che mi permettesse di generare casualmente grafi, dati in input il numero di vertici. Crea poi un numero di archi abbastanza alto da far sì che ci sia quasi sempre almeno un ciclo, ma tale che non superi un numero massimo di archi dato da: $\frac{(\text{n°vertici})(\text{n°vertici}-1)}{2}$ se il numero di vertici è minore o uguale a 6, $((\text{n°vertici}) * 2 - 2)$ se il numero di vertici è maggiore di 6. In questo modo riesco sempre a creare grafi planari.

Ho cercato, quindi, un modo per poter rappresentare il grafo in ordine topologico attraverso la funzione **topologic_graph**. L'ordine topologico viene fatto partendo da una radice (**root**) casuale tra una lista di possibili radici (**roots**) che si trovano attraverso la funzione **vertex_max_edges**, che ritorna la lista dei nodi ordinati per numero di archi.

L'algoritmo controlla se sono presenti cicli all'interno del grafo attraverso la funzione **yet_cycles**, che sfrutta a sua volta la funzione **yet_cycles_util**. Se il grafo generato contiene cicli, si applica prima **cutset conditioning**, altrimenti solamente **map coloring**.

In caso di presenza di cicli, ho utilizzato la funzione **cutset_conditioning** per effettuare un taglio che producesse in output un albero. Questa funzione cerca quale è il migliore nodo da scegliere come radice dell'ordine topologico tra la lista di nodi ordinati (**roots**). Se il primo nodo, che trova nella lista **roots,** crea un ordine topologico che mi permette di effettuare un taglio piccolo rispetto al numero di nodi del grafo e se tale taglio rimuove tutti i cicli, allora si rende in output i nodi del taglio. Altrimenti, la funzione si chiama ricorsivamente prendendo come **root** dell'ordine topologico il secondo nodo in **roots**.

Per poter rappresentare graficamente il grafo, ho cercato un modo per poterlo disegnare (**draw_graph**) in una nuova finestra. Questo è stato possibile grazie anche alla documentazione online sulla libreria *Python* ***NetworkX*** che permette di creare nuove finestre e rappresentare graficamente i vertici e gli archi del grafo.

Infine, ho implementato la funzione **tree_solver**, che prendendo in input l'ordine topologico **tp_sort** e il numero dei colori **num_colors,** rende consistenti i nodi e i loro figli. **tp_sort** è dato dalla funzione **topological_sort** nel caso di grafi aciclici o dalla funzione **cutset_conditioning** in caso di grafi con cicli.

La consistenza tra i nodi (**parent**) e i suoi nodi-figlio (**child**) è gestita da **make_arc_consistent** che assegna inizialmente a tutti i nodi lo stesso colore ("1") e poi aggiorna i colori di ogni nodo in modo tale che nodi connessi da un arco non abbiano lo stesso colore (indicato con i numeri 1-2-3-4). La funzione utilizza il numero minore di colori possibile ma, nel caso non sia possibile colorare i nodi con il numero di colori messi in input da in output l'errore.

Qui sono presenti alcuni dati sperimentali che permettono di trovare il taglio dato in input un grafo approssimabile ad un albero, utilizzando **Jupyter Notebook**:

Prima di tutto è necessario importare *exerciseIA,* dopodiché inizializzare il grafo che si desidera utilizzare *graph = {}*. Si ottiene il grafo in finestra scrivendo *%matplotlib qt* e chiamando poi la funzione *exerciseIA.draw_graph(graph)*. Per trovare i nodi da tagliare è necessario chiamare infine *exerciseIA.cutset_conditioning(graph)*

Risultati:

1. Esempio **Grafo 3 colori**:

```
>>> import exercise

>>> %matplotlib qt

>>>graph3colors = {'a': ['e', 'b'], 'b': ['c', 'a'], 'c': ['b', 'f'], 'd': ['g','f'], 'e': ['a', 'f', 'g'], 'f': ['e', 'c','d'], 'g': ['d', 'e']}

>>>exerciseIA.draw_graph(graph3colors)

>>>exerciseIA.cutset_conditioning(graph3colors)
```

Output:

```
The graph is {'a': ['e', 'b'], 'b': ['c', 'a'], 'c': ['b', 'f'], 'd': ['g', 'f'], 'e': ['a', 'f', 'g'], 'f': ['e', 'c', 'd'], 'g': ['d', 'e']}
---------------------------------------------------------------------------------------------
The cutset part is: {'e': []}
---------------------------------------------------------------------------------------------
The topological sort is:
{'e': ['a', 'f', 'g'],
'a': ['b'],
'f': ['c', 'd'],
'g': ['d'],
'b': ['c'],
'c': [],
'd': []}
```

Inizializzando con **tp_sort** il topological sort trovato sopra:

```
>>>tp_sort = {'e': ['a', 'f', 'g'], 'a': ['b'], 'f': ['c', 'd'], 'g': ['d'], 'b': ['c'], 'c': [], 'd': []}

>>>exerciseIA.tree_solver(tp_sort, 3)
```

Output:

```
The color of each node is: {'e': '1', 'a': '2', 'f': '2', 'g': '2', 'b': '1', 'c': '3', 'd': '1'}
```

.

2. Esempio Grafo **2 colori** (Con la stessa sintassi descritta sopra):

```
>>> import exercise
>>> %matplotlib qt
>>> graph2colors = {'a': ['e', 'b', 'c'], 'b': ['a','d'], 'c': ['a', 'd'], 'd': ['e', 'c','b'], 'e': ['d', 'a']}
>>> exerciseIA.draw_graph(graph2colors)
>>> exerciseIA.cutset_conditioning(graph2colors)
```

Output:

```
The graph is {'a': ['e', 'b', 'c'], 'b': ['a', 'd'], 'c': ['a', 'd'], 'd': ['e', 'c', 'b'], 'e': ['d', 'a']}
-------------------------------------------------------------------------------------------
The cutset part is: {'a': []}
-------------------------------------------------------------------------------------------
The topological sort is:
{'a': ['e', 'b', 'c'],
'e': ['d'],
'b': ['d'],
'c': ['d'],
'd': []}
```

Come osservato precedentemente, inserendo **tp_sort** e chiamando **tree_solver**:

```
>>>tp_sort = {'a': ['e', 'b', 'c'],
'e': ['d'],
'b': ['d'],
'c': ['d'],
'd': []}
>>>exerciseIA.tree_solver(tp_sort, 3)
```

Output:

```
The color of each node is: {'a': '1', 'e': '2', 'b': '2', 'c': '2', 'd': '1'}
```

Anche mettendo in input 3 colori, l'algoritmo utilizza il minor numero di colori possibile

3. Esempio **Grafo 4 colori**:

```
>>> import exerciseIA

>>> %matplotlib qt

>>> graph4colors = {'a': ['g', 'c'], 'b': ['g', 'f', 'e'], 'c': ['d', 'g', 'a'], 'd': ['e', 'c', 'g'], 'e': ['f', 'd', 'g', 'b'],
                    'f': ['e', 'b', 'g'], 'g': ['b', 'e', 'a', 'f', 'd', 'c']}
>>> exerciseIA.draw_graph(graph4colors)

>>> exerciseIA.cutset_conditioning(graph4colors)
```

Questo renderà come output il grafo in una finestra e inoltre:

```
The graph is {'a': ['g', 'c'], 'b': ['g', 'f', 'e'], 'c': ['d', 'g', 'a'], 'd': ['e', 'c', 'g'], 'e': ['f', 'd', 'g', 'b'], 'f': [' e', 'b', 'g'],
              'g': ['b', 'e', 'a', 'f', 'd', 'c']}
-------------------------------------------------------------------------------------------------
The cutset part is: {'g': [], 'b': []}
-------------------------------------------------------------------------------------------------
The topological sort is:
{'g': ['b', 'e', 'a', 'f', 'd', 'c'], 'b': ['f', 'e'], 'e': ['f', 'd'], 'a': ['c'], 'f': [], 'd': ['c'], 'c': []}
```

Quindi, sarà necessario inizializzare **tp_sort** con il grafo trovato qui sopra e proseguire con:

```
>>> tp_sort = {'g': ['b', 'e', 'a', 'f', 'd', 'c'], 'b': ['f', 'e'], 'e': ['f', 'd'], 'a': ['c'], 'f': [], 'd': ['c'], 'c': []}
>>> exerciseIA.tree_solver(tp_sort, 4)
```

Questo renderà come output:

```
The color of each node is: {'g': '1', 'b': '2', 'e': '3', 'a': '2', 'f': '4', 'd': '2', 'c': '3'}
```

Osservazioni finali

Il programma funziona in un tempo limitato solo se con un numero di archi minori di 20. Per poter generare grafi casuali, decidendo il numero di vertici che il nostro grafo deve avere, possiamo utilizzare la funzione **exerciseIA.generate_graph (***numero vertici***)**. In tal caso genera mappe casuali e le colora con un numero di colori che viene chiesto di immettere in input. Per esempio:

>>> exerciseIA.generate_graph (7)

<<< How many colors?

Output: Applica cutset conditioning a un grafo casuale di 7 vertici e lo colora con il numero di colori messi in input