Lorenzo Caselli 6170209

Progetto
Implementazione di TreeCSPSolver

Nel progetto qui descritto ho implementato un algoritmo di risoluzione dei vincoli: Tree-CSP-Solver. L'algoritmo è stato testato su uno scenario del problema di map coloring, le istanze delle quali sono state create in maniera randomica per rendere il più generico possibile il testing dello scenario. Per la generazione casuale delle mappe ho seguito la strategia di creazione proposta dall'esercizio numero 6.9 del libro *Artificial Intelligence a Modern Approach, S. Russell e P. Norvig, Pearson, Third Edition*. Una serie di test sono stati infine attuati sull'algoritmo, al variare dei vari dati necessari al problema. Una più accurata descrizione dei test e delle strategie implementate segue nel testo.

Algoritmo

L'algoritmo usato, Tree-CSP-Solver prende in ingresso una struttura dati in forma di albero, alla quale vengono accostati una serie di variabili, domini e vincoli del problema. L'algoritmo in seguito performa una serie di strategie per rendere i domini dei nodi(con arc consistency, o meglio directed arc consistency) e assegnazione dei valori (consistenti) con la serie di vincoli posti dal problema e se è possibile restituisce una serie di assegnazioni di dati alle variabili del problema (che sono i nodi dell'albero). Se ciò non è possibile restituisce errore. Questa serie di operazioni vengono fatte con una complessità lineare, grazie proprio alla struttura dei problemi che vengono trattati dall'algoritmo e grazie alla strategia di ordinamento dei nodi, restituita da un TopologicalSort, rendendo il meccanismo di controllo estremamente più semplice rispetto ad un arc consistency e una assegnazione delle variabili ad un grafo non sotto la forma di albero. Se la mappa non fosse sotto forma di albero sarebbe quindi impossibile applicare questo algoritmo direttamente, senza un qualche rilassamento del problema.

L'implementazione dell'algoritmo e della creazione delle mappe e delle conseguenti strutture ad albero viene eseguita con il linguaggio di programmazione Python 3.7.3 e facendo uso di alcune librerie in appoggio per alcune operazioni. Le librerie sono: *random*, *networkx* e *matplotlib* per la visualizzazione delle mappe, *timeit* e csv.
I risultati dei vari test sono scritti in

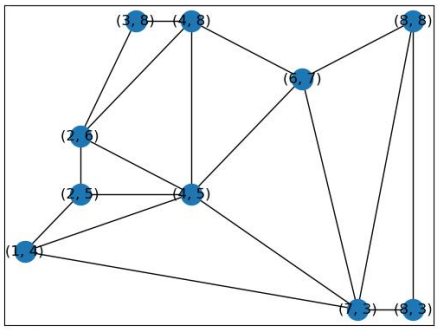


Figura 1

una serie di file csv, nei quali vengono salvati dati sul tempo di risoluzione, la grandezza del problema, relativo albero su cui è stato applicato il problema e la soluzione restituita.
L'implementazione dell'algoritmo è abbastanza generale da poter essere facilmente implementato in problemi di diversa natura, semplicemente modificando i domini e le funzioni che eseguono i controlli dei vincoli. Questi controlli e l'inserimento del dominio sono state codificate direttamente nel codice. Come ispirazione per la scrittura del codice è stato seguito lo pseudocodice nella figura 6.11 del libro *Artificial Intelligence a Modern Approach* al capitolo 6.5. Un esempio di grafo ottenuto è illustrato in figura 1, nella pagina precedente, mentre un albero di copertura con relativa soluzione è illustrato nella figura 2.

Descrizione problema map coloring

Data una mappa con collegamenti tra regioni, si vuole assegnare ad ogni regione uno dei k colori possibili in maniera tale da non ottenere colori uguali tra regioni tra loro connesse. Nel caso in questione le mappe devono avere una forma ad albero per poter essere applicato il TreeCSPSolver.

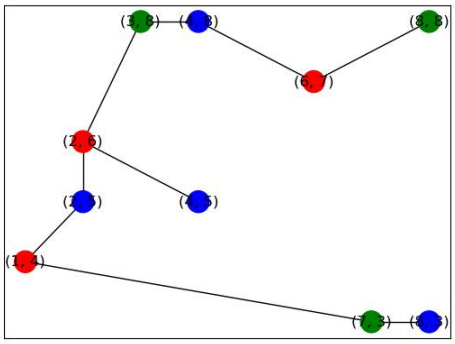


Figura 2

Codice: implementazione del problema

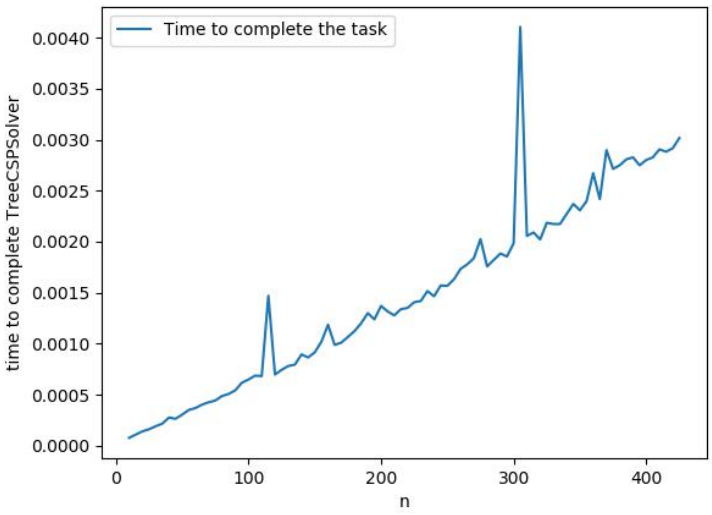
La maggior parte del codice che implementa il problema è racchiuso in due classi: *Grafo* e *ConstraintProblemTree*. In queste due classi divido in sottoproblemi il map coloring.
Nella classe *Grafo* implemento il codice per la creazione della mappa. L'effettiva creazione della mappa(ovvero del grafo) avviene grazie alla funzione *createMap* che prende in ingresso n, ovvero il numero di nodi che si vogliono creare e k che è la grandezza dello square nel quale creare tutti i nodi. In conseguenza della chiamata della funzione verranno creati gli n nodi e successivamente, grazie alla funzione *createGraph* chiamata dalla precedente verranno creati i vari archi in maniera da ottenere un grafo planare, ovvero un grafo nel quale non vi sono intersezioni tra i vari archi (la serie di controlli delle intersezioni sono eseguiti da *segmentIntersect*, che si appoggia ad altre funzioni come *direction*, *onSegmentCollinear*, *onSegment*), mentre per scegliere il nodo più vicino ad un altro nodo tra quelli non controllati ci si affida alla funzione *minimumDistance*.
Alla fine dell'esecuzione nell'attributo *graph* otterrò una lista di nodi, in *vertices* e il grafo. Il grafo è implementato come dict, dove le chiavi sono i nodi e i valori all'interno di un dict sono una lista di altri nodi. La coppia chiave-nodo nella lista mi

rappresenta la presenza di un arco che va da un nodo ad un altro(si tratta quindi di una lista di adiacenza).

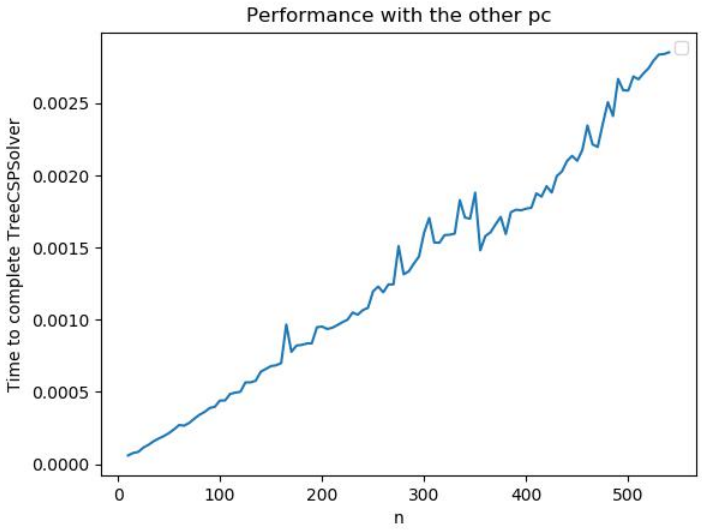
La classe *ConstraintProblemTree* invece conterrà tutte le informazioni per poter eseguire l'algoritmo *TreeCSPSolver*. Quando si instanzia, questa classe prende in ingresso una instanza della classe *Grafo*, salva i nodi che sono contenuti e ne crea uno spanning tree, cioè un albero di copertura del grafo di partenza. Inoltre viene preso in ingresso anche un parametro *domain*, che accoppiato alla funzione *setDomains* crea l'insieme dei domini dei nodi. A seconda di che tipo di problema si vuole affrontare è necessario modificare la funzione *setDomains* e i dati inseriti, oltre che le funzioni che hanno lo scopo di verificare i vincoli del problema. Nel nostro caso sono fatti ad hoc per risolvere istanze del problema di map coloring.
A quel punto viene implementata la funzione *treeCSPSolver*, la quale, in maniera piuttosto generale fa un ordinamento topologico dell'albero chiamando *topologicalSort*. Successivamente per ogni coppia di nodi viene chiamato *directedAC*, che verifica la arc consistency sulle coppie di nodi e esegue le varie assegnazioni in un successivo ciclo, usando la funzione *assignValue*, sui rispettivi domini e con gli adeguati vincoli, implementati nel codice con le funzioni *checkValue* e *reviseMapColour*.
A quel punto ci saranno due possibilità: o viene restituito false, se la assegnazione non è stata possibile perchè non tutti i vincoli potevano essere soddisfatti, oppure restituisce una assegnazione valida per il problema.
Una descrizione più dettagliata di come usare il codice per eseguire dei test sull'algoritmo è inclusa nel README.txt, insieme al resto dei file.

Test e Analisi dei risultati

Eseguendo vari test su taglie del problema incrementali, partendo da n uguale a 10 fino a taglie dell'ordine di 500, è stato facile verificare che qualsiasi istanza del problema TreeCSPSolver veniva risolta con un tempo linearmente proporzionale alla taglia del problema. I dati empirici riescono quindi a dimostrare l'ipotesi di partenza, ovvero che il TreeCSPSolver, quando



riesce a trovare una istanza di risoluzione di un problema, vi riesce con una complessità che equivale ad un O(n).
Mi è stato difficile eseguire test di taglia superiore a 500, poichè l'algoritmo di creazione dei grafi cominciava ad occupare un tempo molto grande. Sicuramente questo è anche causa di una poca ottimizzazione dei cicli nell'algoritmo e sicuramente anche dalla difficoltà conosciuta di Python di lavorare con una serie di cicli. Questo sicuramente rende la creazione delle istanze del problema più lenta di quanto sarebbe potuta essere con un altro linguaggio di programmazione.



Ciononostante questo non ha impedito di verificare che l'algoritmo da testare continui a comportarsi con la natura appropriata (ovvero lineare) a qualsiasi taglia del problema che viene proposta.
Successivamente alcuni test sono stati eseguiti al variare della grandezza del dominio, per 3,4,5 colori diversi e questo non impatta in alcuna maniera sulla complessità del problema.
Infine ho deciso di ripetere gli stessi test con un'altra macchina più performante sotto il punto di vista delle prestazioni. I risultati sono stati analoghi a quelli precedenti.