Elaborato Intelligenza Artificiale

Studente: Massimiliano Sirgiovanni
Matricola: 6357449
E-mail: massimiliano.sirgiovanni@stud.unifi.it

Obiettivi dell'elaborato
Nell'esecuzione di tale elaborato, l'obiettivo prefissato è stato quello di dimostrare l'assunto dell'esercizio, ovvero:
*"Spiega perché è una buona euristica scegliere la variabile * più * vincolata ma il valore * meno * vincolante in una ricerca CSP."*
Questo verrà dimostrato sia tramite quest'elaborato, ma anche tramite il confronto degli algoritmi richiesti, sviluppati in Java. Gli algoritmi sviluppati sono:

- ***> Backtracking (con euristiche semplici)***
- ***> Backtracking (con le euristiche richieste)***
- ***> Min Conflicts***

Come esempio per l'esecuzione è stato utilizzato il CSP riguardante la **colorazione dell'Australia**.
Tuttavia, l'implementazione è generica, il che consente di creare altri csp, sotto forma di grafo.

Perché è una buona euristica scegliere la variabile più vincolata ma il valore meno vincolante in una ricerca CSP?

Il motivo per cui un'euristica del genere porta a risultati migliori è intuibile. Se, durante l'esecuzione dell'algoritmo, si privilegiano le variabili che sono maggiormente vincolate, si restringono il maggior numero di domini nelle altre variabili. In questo modo, si raggiungerà più in fretta una situazione in cui i nodi rimanenti hanno possibilità di scelta minime, o nulle se vi è un unico valore nel dominio. Inoltre, in questo modo si eliminano, il prima possibile, i vincoli che potrebbero causare inferenze e potenziali errori. Di conseguenza questa euristica, nonostante comporti maggior tempo nella selezione dei nodi, recupera lo scarto di tempo nelle fasi finali dell'esecuzione ed inoltre **minimizza il numero di errori**.
Infatti, utilizzando altri approcci per la scelta del nodo, spesso si potrebbe incappare in situazioni che portano l'algoritmo a tornare sui suoi passi, per valutare altre possibilità.
Per esempio, se ci trovassimo in una situazione di questo tipo:
Vi sono tre stati, A, B e C.
C confina sia con A che con B, ma A e B non sono confinanti tra loro. Avremo un grafo di questo tipo:

Se utilizzassimo un'euristica semplice, potrebbe verificarsi la situazione in cui vengono visitati prima A e B e successivamente C. Se ad A ed a B venissero assegnati i medesimi valori, ci troveremo in una situazione di errore e sarebbe necessario "tornare indietro" e valutare nuove possibilità. Al contrario, se utilizzassimo l'euristica precedentemente vista, il primo nodo ad essere visitato, deve essere necessariamente C, il che non ci farà cadere in situazioni di errore.
D'altra parte, per la scelta dei valori è preferibile selezionare quello meno vincolante.
Ovvero, quel valore che è stato assegnato più spesso alle variabili già visitate.

Scegliendo un valore che sia già stato utilizzato per altri nodi, sarà meno probabile che questo possa risultare determinante per nodi successivi. Infatti, i nodi visitati in seguito, potrebbero essere vincolati al nodo appena visitato ed ai nodi a cui era stato assegnato quel valore precedentemente. In questo caso ci si potrebbe ritrovare in una situazione di errore.
Ritornando all'esempio sopracitato. Visitando i nodi utilizzando un'euristica semplice, potrebbe verificarsi la situazione in cui vengono visitati prima A e B e successivamente C.
Supponendo di avere due colori se non utilizzassimo tale euristica potrebbe verificarsi la situazione in cui si assegnano ad A ed a B colori distinti. Di conseguenza, C, essendo vincolato ad entrambi, non potrebbe assumere alcun valore e ci si ritroverebbe in una situazione di errore.
Al contrario, scegliendo il valore meno vincolante, si assegnerebbe lo stesso colore ad A e a B.
Vi sarà, dunque, un valore non assegnato da utilizzare per C.
In questo modo si è trovata una soluzione, senza trovare mai una situazione di errore.
Per confermare questi risultati, sono stati implementati gli algoritmi, presenti sul libro tramite pseudocodice, in Java

Implementazione del codice

Per poter implementare gli algoritmi richiesti, bisogna rappresentare in Java il CSP.
Partendo dalle componenti più semplici, è stata necessaria la realizzazione di una classe *"**Node**"*.
Come si intuisce, questa classe permette la **rappresentazione di un nodo** del grafo, salvando in tale oggetto il suo nome, il suo dominio e tutti gli archi in cui compare. Il nome ed il dominio dovranno essere inseriti nel costruttore, durante la creazione del nodo, il dominio, tuttavia, potrà essere modificato in seguito. Per la rappresentazione degli archi si è usata una *LinkedList*.
Questa classe **prevede anche alcune operazioni sul dominio**, come la verifica che un valore sia presente nel dominio del nodo, la rimozione di un valore ed una funzione che restituisce il numero di elementi presenti nel dominio. Il valore "0" nel dominio rappresenta la cella vuota.
Dopo aver implementato una rappresentazione per i nodi, i quali rappresenteranno le nostre variabili, si può costruire il **grafo dei vincoli**.
Anche per il grafo è stata creata una classe, "***Graph***", la quale conterrà l'insieme di tutti i nodi facenti parte del grafo. È stata usata una *LinkedList*, dalla quale i nodi si potranno aggiungere, rimuovere e visualizzare, conoscendone il nome. Inoltre, è possibile aggiungere un arco tra due nodi e anche rimuovere un arco.
Infine, il grafo verrà inserito nella classe *"**CSP**"*, la quale terrà traccia, appunto, del grafo dei vincoli e del dominio delle variabili. Inoltre, la classe CSP, **contiene dei metodi che si occupano di verificare se inferenza è fattibile e di effettuarla.**
Prima di procedere con l'implementazione degli algoritmi, si osservi la classe *"**Assignment**"*.
Questa classe rappresenta gli assegnamenti di valori alle variabili. Per fare ciò, sono stati utilizzati due array, della stessa dimensione, contenenti uno i nomi delle variabili e l'altro i valori da assegnare. L'ultimo array contiene, inizialmente, il valore *"0"* in ogni casella. Oltre ai metodi di base, per restituire i due array e per modificare i valori delle variabili, **tale classe può verificare**, tramite opportune funzioni, **se l'assegnamento è completo, se ad una variabile è già stato assegnato un valore, se l'assegnamento è una soluzione e se un determinato valore risulti essere consistente**.
Ora che sono state introdotte le classi di base, si può passare all'implementazione degli algoritmi.

Partendo dal **Backtracking Search**, è stato realizzato tramite una classe con due metodi. Il primo, inizializza un assegnamento, vuoto, sul csp dato in input e successivamente richiama il metodo ricorsivo, per la ricerca, dandogli come argomento il csp, l'assegnamento e due operatori.
Questi due operatori sono stati realizzati tramite l'utilizzo di uno **Strategy**, design pattern, e sono *SelectVariables* ed *OrderDomain*. Il primo fornisce i metodi per la selezione dei nodi da espandere, il secondo ci restituisce l'ordine dei valori, presenti nel dominio, da seguire per le assegnazioni.
L'utilizzo di uno Strategy ci permette di evitare una modifica dell'algoritmo e la ripetizione di codice. Basta passare al metodo un oggetto concreto di una classe che implementa l'interfaccia dello Strategy, direttamente nella classe main. Di conseguenza, tramite l'utilizzo dello stesso algoritmo, si possono ottenere versioni del Backtracking con euristiche diverse.
Tutte le euristiche sono contenute nel pacchetto *"csp.heuristics"*.
Proseguendo, la funzione ricorsiva si comporta esattamente come lo pseudocodice di partenza, visto a lezione e presente sul libro. Dunque, per prima cosa si verifica che l'assegnamento sia completo, se così fosse si sarebbe trovata la soluzione.
Nel caso contrario, si procede a selezionare un nodo, non ancora assegnato, da espandere. Il metodo di selezione, come accennato prima, dipenderà dall'euristica scelta.
In modo similare, si ottiene un dominio ordinato. Infine, si procede a ricercare, per la variabile scelta, un valore che sia consistente e che ci permetta di effettuare l'inferenza. Trovato questo valore si procede con la ricorsione e si effettuano questi passaggi su ogni altra variabile del csp.
Anche per l'algoritmo ***MinConflicts***, si è seguito lo pseudocodice visto a lezione. Tale algoritmo, prende come argomenti un csp, un'euristica per la selezione delle variabili ed infine un numero massimo, ***maxSteps***, di passi da eseguire. Tale algoritmo non è completo, infatti può non trovare una soluzione anche qualora ve ne sia una.
Per cominciare si è creata una funzione che restituisse un **assegnamento di valori casuali**.
Una volta fatto ciò, si entra in un ciclo for, che avrà al massimo *maxSteps* iterazioni, e si verifica se l'assegnamento appena creato sia una soluzione. In caso contrario si procederà con la scelta delle variabili. Anche in questo caso si è utilizzato lo Strategy, ma con un metodo diverso. Infatti, tale funzione deve restituire una variabile che sia già stata assegnata e non una ancora da assegnare. Questa caratteristica, rende molte delle euristiche, utilizzate per il Backtracking, inutilizzabili nel *MinConflicts*, dato che restituirebbero sempre lo stesso valore. L'euristica usata sarà quella per la selezione randomica delle variabili.
Una volta selezionata la variabile, si dovrà scegliere il valore che minimizza i conflitti all'interno dell'intero grafo (in questo caso non è stata usata un'euristica, poiché tale metodo di scelta dei valori è parte fondante dell'algoritmo). Infine, sostituisce il valore scelto e si procede con una successiva iterazione del ciclo for.
Se non è stata trovata una soluzione, il metodo restituisce un valore null.

Confronto degli algoritmi

Per il confronto degli algoritmi è stato utilizzato, come esempio, il problema della colorazione dell'Australia. Gli algoritmi confrontati quattro, di cui tre versioni, con euristiche diverse, del Backtracking Search ed il MinConflicts. Le euristiche usate per i tre algoritmi di Backtracking sono:

- ***> FIFO per le variabili e per i valori***
- ***> Random per le variabili e FIFO per i valori***
- ***> Variabile più vincolata e valore meno vincolante***

Nella classe main si è quindi costruito prima il csp e poi sono stati inizializzati gli algoritmi e le euristiche. Inoltre, si è usata la funzione *"currentTimeMillis()"* per misurare il tempo impiegato dai vari algoritmi per l'esecuzione. Per riprodurre questi risultati, è sufficiente eseguire la classe Main presente nel progetto Java allegato. Per importare il codice in eclipse basta seguire le istruzioni presenti nel file ***README.txt***. Risultati del test:

```
------------BACKTRACKING ALGORITHM----------------

[WA] --> [1]
[NT] --> [2]
[Q] --> [1]
[SA] --> [3]
[NSW] --> [2]
[V] --> [1]
[T] --> [1]

Execution time: 2 milliseconds

--------------------------------------------------

------------BACKTRACKING ALGORITHM (RANDOM)----------

[WA] --> [1]
[NT] --> [2]
[Q] --> [1]
[SA] --> [3]
[NSW] --> [2]
[V] --> [1]
[T] --> [1]

Execution time: 1 milliseconds

--------------------------------------------------

------------BACKTRACKING ALGORITHM (Better Heuristics)----------

[WA] --> [1]
[NT] --> [2]
[Q] --> [1]
[SA] --> [3]
[NSW] --> [2]
[V] --> [1]
[T] --> [1]

Execution time: 0 milliseconds

--------------------------------------------------

-------------MIN CONFLICTS-----------------

[WA] --> [3]
[NT] --> [1]
[Q] --> [3]
[SA] --> [2]
[NSW] --> [1]
[V] --> [3]
[T] --> [2]

Execution time: 1 milliseconds
```

```
------------BACKTRACKING ALGORITHM----------------

[WA] --> [1]
[NT] --> [2]
[Q] --> [1]
[SA] --> [3]
[NSW] --> [2]
[V] --> [1]
[T] --> [1]

Execution time: 4 milliseconds

--------------------------------------------------

------------BACKTRACKING ALGORITHM (RANDOM)----------

[WA] --> [1]
[NT] --> [2]
[Q] --> [1]
[SA] --> [3]
[NSW] --> [2]
[V] --> [1]
[T] --> [1]

Execution time: 11 milliseconds

--------------------------------------------------

------------BACKTRACKING ALGORITHM (Better Heuristics)----------

[WA] --> [1]
[NT] --> [2]
[Q] --> [1]
[SA] --> [3]
[NSW] --> [2]
[V] --> [1]
[T] --> [1]

Execution time: 2 milliseconds

--------------------------------------------------

-------------MIN CONFLICTS-----------------

[WA] --> [1]
[NT] --> [3]
[Q] --> [1]
[SA] --> [2]
[NSW] --> [3]
[V] --> [1]
[T] --> [3]

Execution time: 20 milliseconds
```

In generale, come ci si poteva aspettare, si può osservare come l'algoritmo di backtracking con le euristiche migliori risulti essere il più veloce nella sua esecuzione. Gli algoritmi randomici sono invece quelli più imprevedibili. Infatti, se, casualmente, escono valori che facilitino la risoluzione del problema, questi possono risultare estremamente convenienti e veloci, al contrario, se i valori selezionati dovessero essere poco convenienti, potrebbero impiegare un tempo molto maggiore.
Nel caso del *MinConflicts*, dato che vi è un massimo numero di iterazioni fattibili prima che l'algoritmo termini con un fallimento, potrebbe anche non trovare soluzione.

Quando la soluzione non viene trovata dal *MinConflicts*, ovviamente, il tempo di esecuzione risulta essere molto più alto rispetto a quello degli altri algoritmi.
In generale, questi risultati, anche se effettuati su un problema giocattolo, confermano la tesi per cui le euristiche proposte dall'esercizio siano preferibili per la risoluzione di un CSP. Naturalmente, più si affronta un problema grande e complesso, più queste differenze risulteranno si noteranno.

```
------------BACKTRACKING ALGORITHM----------------

[WA] --> [1]
[NT] --> [2]
[Q] --> [1]
[SA] --> [3]
[NSW] --> [2]
[V] --> [1]
[T] --> [1]

Execution time: 2 milliseconds

--------------------------------------------------

------------BACKTRACKING ALGORITHM (RANDOM)----------

[WA] --> [1]
[NT] --> [2]
[Q] --> [1]
[SA] --> [3]
[NSW] --> [2]
[V] --> [1]
[T] --> [1]

Execution time: 1 milliseconds

--------------------------------------------------

------------BACKTRACKING ALGORITHM (Better Heuristics)----------

[WA] --> [1]
[NT] --> [2]
[Q] --> [1]
[SA] --> [3]
[NSW] --> [2]
[V] --> [1]
[T] --> [1]

Execution time: 0 milliseconds

--------------------------------------------------

-------------MIN CONFLICTS-----------------

NO SOLUTION WAS FOUND!

Execution time: 10 milliseconds
```

```
------------BACKTRACKING ALGORITHM----------------

[WA] --> [1]
[NT] --> [2]
[Q] --> [1]
[SA] --> [3]
[NSW] --> [2]
[V] --> [1]
[T] --> [1]

Execution time: 3 milliseconds

--------------------------------------------------

------------BACKTRACKING ALGORITHM (RANDOM)----------

[WA] --> [1]
[NT] --> [2]
[Q] --> [1]
[SA] --> [3]
[NSW] --> [2]
[V] --> [1]
[T] --> [1]

Execution time: 2 milliseconds

--------------------------------------------------

------------BACKTRACKING ALGORITHM (Better Heuristics)----------

[WA] --> [1]
[NT] --> [2]
[Q] --> [1]
[SA] --> [3]
[NSW] --> [2]
[V] --> [1]
[T] --> [1]

Execution time: 0 milliseconds

--------------------------------------------------

-------------MIN CONFLICTS-----------------

NO SOLUTION WAS FOUND!

Execution time: 12 milliseconds
```