Strategie di ricerca per problemi di vincoli

Chapter 3, Section 7 and Chapter 4, Section 4.4

Standard backtracking search

Stati definiti dai valori assegnati finora.

Stato iniziale: nessuna variabile assegnata

Operatori: assegnare un valore ad una variabile non assegnata

Test di goal: tutte la variabili assegnate, nessun vincolo violato

Lo stesso per tutti i CSP.

Complessita'

Massima profondita' dello spazio di ricerca $m =$?? n (number of variables)

Profondita' di uno stato di goal $d =$?? n (all vars assigned)

Algoritmo di ricerca da usare:?? depth-first

Fattore di branching $b =$?? $\sum_i |D_i|$ (at top of tree)

Puo' essere migliorato notando che:

1) L'ordine dell'assegnamento e' irrilevante, quindi molti cammini sono equivalenti
2) Aggiungere assegnamenti non puo' soddisfare un vincolo violato

Ricerca con backtracking

Usare la ricerca depth-first, ma

1) fissare l'ordine di assegnamento, $\Rightarrow b = |D_i|$

2) controllare le violazioni dei vincoli

Il test per la violazione dei vincoli puo' essere implementato in due modi:

1) modificare SUCCESSORS per assegnare solo valori che sono permessi, dati i valori gia' assegnati

o 2) controllare che i vincoli siano soddisfatti prima di espandere uno stato

La ricerca con backtracking e' l'algoritmo di ricerca non-informata di base per i CSP.

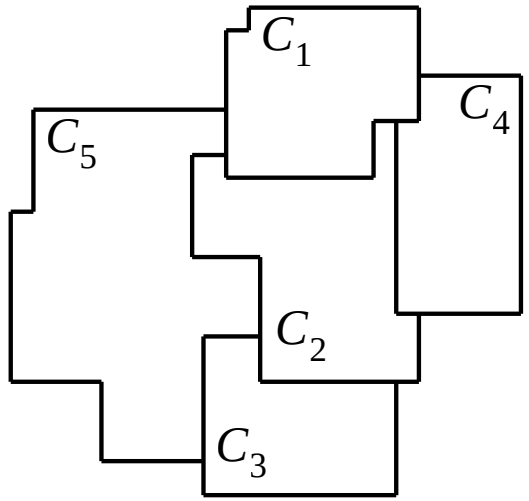
Puo' risolvere n-queens per $n \approx 15$

Forward checking

Idea: Ricordarsi dei valori legali rimasti per le variabili non ancora assegnate
Terminare la ricerca quando una variabile non ha piu' valori legali

Esempio: colorazione delle mappe:

	RED	BLUE	GREEN
C_1			
C_2			
C_3			
C_4			
C_5			



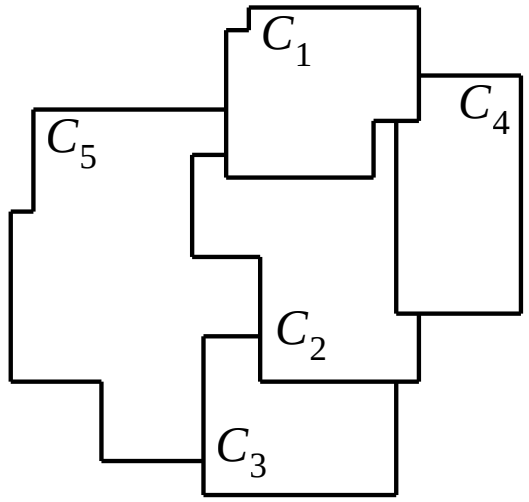
Puo' risolvere n-queens fino a $n \approx 30$

Forward checking

Idea: Keep track of remaining legal values for unassigned variables
Terminare la ricerca quando una variabile non ha piu' valori legali

Esempio: colorazione delle mappe:

	RED	BLUE	GREEN
C_1	$\surd$		
C_2	$\times$		
C_3			
C_4	$\times$		
C_5	$\times$		

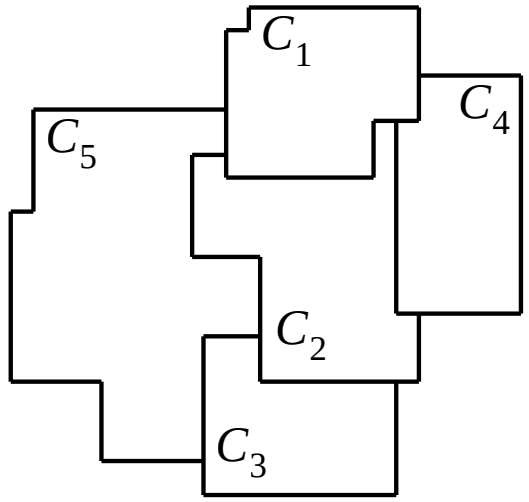


Forward checking

Idea: Keep track of remaining legal values for unassigned variables
Terminare la ricerca quando una variabile non ha piu' valori legali

Esempio: colorazione delle mappe:

	RED	BLUE	GREEN
C_1			
C_2		$\surd$	
C_3		$\times$	
C_4		$\times$	
C_5		$\times$	

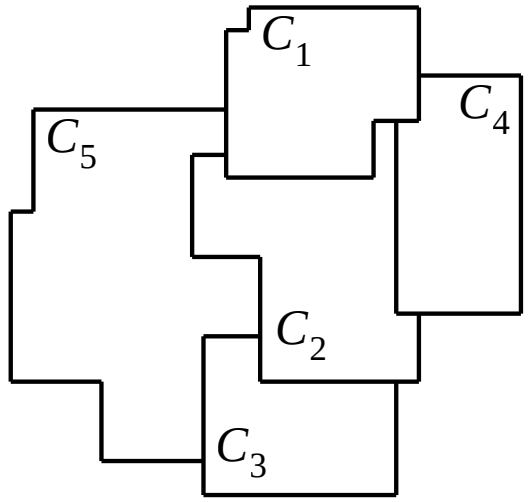


Forward checking

Idea: Keep track of remaining legal values for unassigned variables
Terminare la ricerca quando una variabile non ha piu' valori legali

Esempio: colorazione delle mappe:

	RED	BLUE	GREEN
C_1			
C_2			
C_3			$\surd$
C_4			
C_5			$\times$



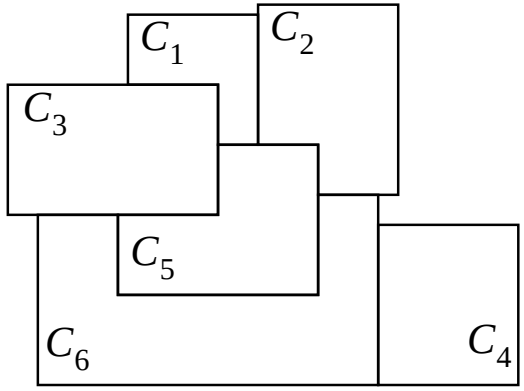
Forward checking: caso particolare di consistenza sugli archi

Euristiche per CSP

Decisioni piu' intelligenti su

quale valore scegliere per ogni variabile

quale variabile assegnare come prossima



Dato $C_1 = Red$, $C_2 = Green$, scegliere $C_3 =$??

$C_3 = Green$: *least-constraining-value*

Dato $C_1 = Red$, $C_2 = Green$, come procedere???

C_5: *most-constrained-variable*

Puo' risolvere n-queens per $n \approx 1000$

Algoritmi iterativi per CSP

Hill-climbing, simulated annealing funzionano con
stati "completi", cioe' stati dove tutte le variabili sono assegnate

Per applicarli ai CSPs:

permettere stati con vincoli violati

quindi l'operatore di successore $riassegna$ valori alle variabili

Selezione della variabile: a caso, sceglie una variabile coinvolta in un conflitto

min-conflicts heuristic:

sceglie il valore che viola il numero minimo di vincoli

i.e., hillclimb con $h(n) =$ numero totale di vincoli violati

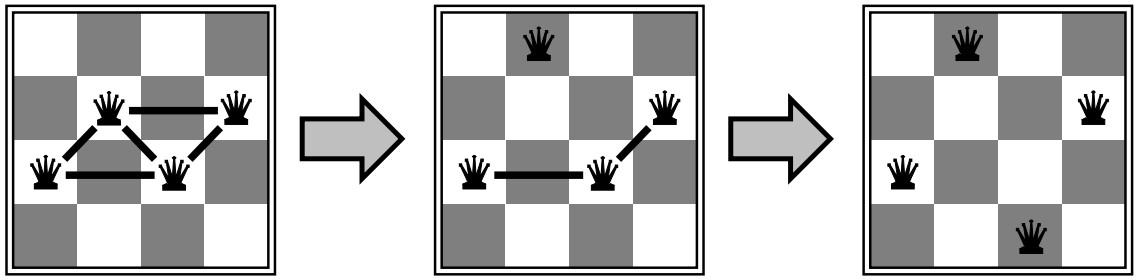
Esempio: 4-Queens

Stati: 4 regine in 4 colonne ($4^4 = 256$ stati)

Operatore: muove una regina nella colonna

Test di goal: nessun attacco

Valutazione: $h(n) =$ numero di attacchi

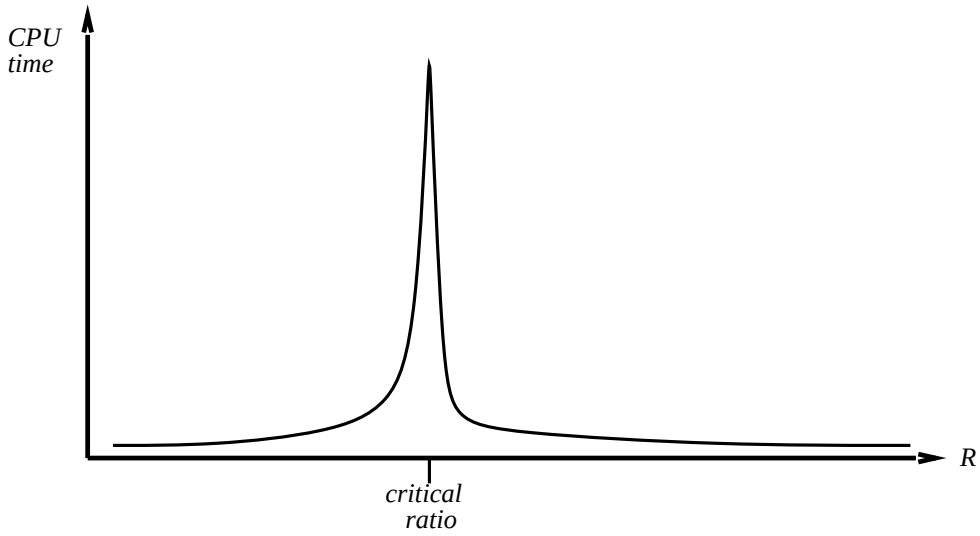


Performance di min-conflicts

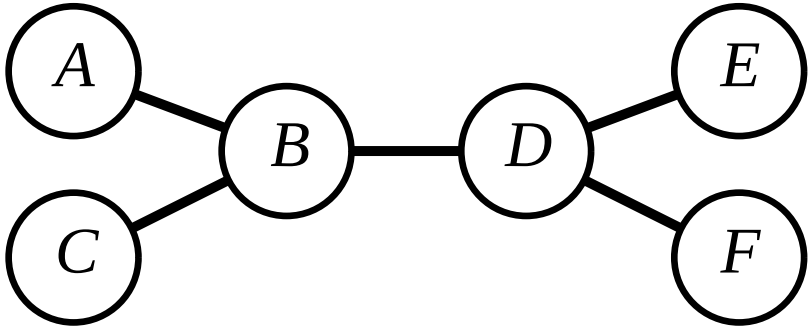
Dato uno stato iniziale random, puo' risolvere n-queens in tempo quasi costante per n arbitrario con alta probabilita' (es.: $n = 10{,}000{,}000$)

Lo stesso per altri CSP generati a caso
eccetto in una piccola parte della frazione

$$R = \frac{\text{numero di vincoli}}{\text{numero di variabili}}$$



Tree-structured CSPs



Theorem: if the constraint graph has no loops, the CSP can be solved in $O(n|D|^2)$ time

Compare to general CSPs, where worst-case time is $O(|D|^n)$

This property also applies to logical and probabilistic reasoning:
an important example of the relation between syntactic restrictions and complexity of reasoning.

Algorithm for tree-structured CSPs

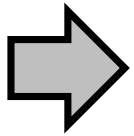
Basic step is called *filtering*:

FILTER(V_i, V_j)

removes values of V_i that are inconsistent with ALL values of V_j

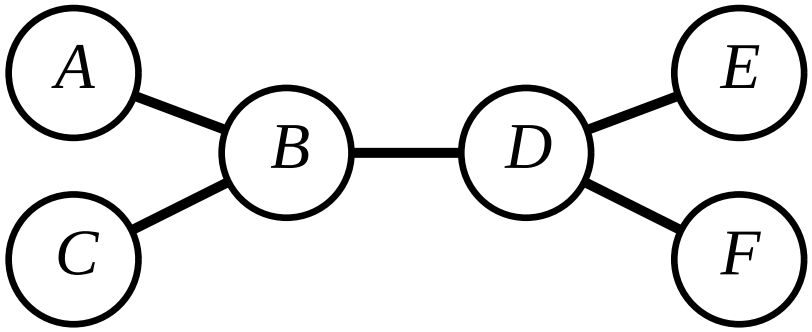
Filtering example:

allowed pairs:
< 1, 1 >
< 3, 2 >
< 3, 3 >

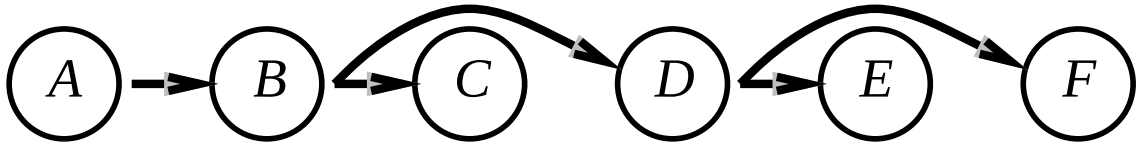


remove 2 from domain of V_i

Algorithm contd.



1) Order nodes breadth-first starting from any leaf:



2) For $j = n$ to 1, apply $\textsc{Filter}(V_i, V_j)$ where V_i is a parent of V_j

3) For $j = 1$ to n, pick legal value for V_j given parent value

Summary

CSPs are a special kind of problem:

states defined by values of a fixed set of variables

goal test defined by *constraints* on variable values

Backtracking = depth-first search with

1) fixed variable order

2) only legal successors

Forward checking prevents assignments that guarantee later failure

Variable ordering and value selection heuristics help significantly

Iterative min-conflicts is usually effective in practice

Tree-structured CSPs can always be solved very efficiently