

Lezione13

Table of contents

- [Standard Backtracking](#)
- [Search Heuristics](#)
- [Which Variable to Assign Next](#)
- [Which Value to Assign Next](#)
- [Constraint Propagation](#)
 1. [Forward Checking](#)
 1. [Arc Consistency](#)
- [Parametric CSPs](#)
 1. [\\$n\\$-Queens](#)
- [MiniZinc](#)
 1. [`Map1.mzn`](#)
 1. [Traduzione FlatZinc](#)
 2. [Output](#)
 2. [`Map2.mzn`](#)
 3. [`Money.mzn`](#)
 1. [Output](#)
 4. [`Banana.mzn`](#)
 1. [Output](#)

Standard Backtracking

Nella scorsa lezione abbiamo visto l'algoritmo di backtracking in uno pseudo codice quasi identico al linguaggio C. Vediamo ora una versione alternativa, che metta alla luce degli aspetti importanti dell'algoritmo, segnalando 2 punti in particolare che possono essere ottimizzati.

Il linguaggio usato è uno da supposti pseudo funzionale, non deterministico.

```
function Backtracking-Search(csp) returns a solution, or failure
    return Recursive-Backtracking({},csp)

function Recursive-Backtracking(assignment,csp) returns a solution, or failure
    if assignment is complete then return assignment
    var ← Select-Unassigned-Variable(Variable[csp],assignment,csp)
    for each value in Order-Domain-Values(var,assignment,csp) do
        if value is consistent with
            assignment according to Constraints[csp] then
                add {var = value} to assignment
                result ← Recursive-Backtracking(assignment,csp)
                if result ≠ failure then return result
                remove {var = value} from assignment
    return failure
```

`Recursive-Backtracking` sarebbe il nome dell'algoritmo visto a [Lezione12](#), ma riscritto per renderlo più semplice da capire: riceve come parametri un assegnamento parziale `assignment`, che alla prima chiamata di

funzione sarà vuoto, siccome le variabili che devono essere libere lo sono; riceve un `csp` per cui bisogna trovare una soluzione.

- Se l'assegnamento è completo, abbiamo trovato una soluzione che è anche consistente, siccome verificata parzialmente per ogni ciclo.
- Se non abbiamo un assegnamento completo, scegliamo una variabile `var` non ancora assegnata con `Select-Unassigned-Variable`. Questa funzione ha l'obiettivo di prendere una variabile che permette di arrivare prima al risultato; è il nostro primo punto che ammette ottimizzazioni: `Variable[csp]` è l'insieme delle variabili, `assignment` è l'assegnamento attuale e `csp` il problema.

Scelta la variabile, andiamo a enumerare tutti i suoi valori all'interno del dominio nell'ordine che vogliamo con `Order-Domain-Values`. Proprio perché l'ordine è arbitrario, questo sarà il secondo punto di possibile ottimizzazione.

Prendiamo poi un valore `var` e se questo è consistente con l'assegnamento, ovvero tutti i vincoli rispettano l'ipotesi che `value` sia `consistent` con `assignment`, allora aggiungiamo il valore all'assegnamento `add { var = value }` e procediamo ricorsivamente. Se il valore invece non è consistente perché qualche vincolo non viene soddisfatto, allora non facciamo l'assegnamento e passiamo al valore successivo.

Nel momento in cui la chiamata ricorsiva ritorna un risultato `if result ≠ failure then return result`, allora questa terminerà passandolo al chiamante. Se una soluzione non è stata trovata ancora (`result = failure`), allora la variabile con quell'assegnamento che si stava provando viene eliminata e si passa alla successiva, sempre ritornando un risultato `remove {var = value} from assignment`.

Arrivati in fondo, viene ritornato `failure` non perché non ci sia stata soluzione nel mentre, ma perché tutte le altre soluzioni sono state consumate. Stessa cosa succede se nessuno dei valori è consistente o se l'insieme dato è uno vuoto.

Search Heuristics

Definizione di "euristica" - Treccani

Aspetto del metodo scientifico che comprende un insieme di strategie, tecniche e procedimenti inventivi per ricercare un argomento, un concetto o una teoria adeguati a risolvere un problema dato.

L'efficacia di un SBT può essere migliorata usando **euristiche di ricerca** a obiettivo multiplo per:

- selezionare quale variabile da usare nel prossimo assegnamento
 - con la funzione `Select-Unassigned-Variable`
- selezionare quale valore da usare nel prossimo assegnamento
 - con la funzione `Order-Domain-Values`
- identificare inconsistenze il prima possibile
- selezionare quale variabile rendere libera in caso di backtracking
 - con backtracking cronologico
 - con backtracking non cronologico (che non vedremo)

L'obiettivo di un SBT non è tanto arrivare a una soluzione il prima possibile, quanto piuttosto individuare quelle ottimizzazioni che rendono l'albero non necessariamente da visionare nel completo.

Which Variable to Assign Next

Usando l'esempio della mappa australiana sempre visto la lezione scorsa, visioniamo le possibilità nella scelta di quale variabile da assegnare.

- **min-domain** variable



L'euristica di questo tipo, prevede come variabile successiva d'*assegnare* quella che ha nel dominio il *numero inferiore di elementi*. Se abbiamo variabili con domini da 10 elementi, e altre variabili con domini da 2 elementi, sceglieremo una delle variabili che hanno per dominio l'insieme con 2 elementi. Lo facciamo perché se pensiamo a un albero di ricerca, siamo più ottimizzati eliminando metà dell'albero intero che si andrà a formare, piuttosto che un decimo.

Nei casi più comuni, il vantaggio c'è, nonostante non sia sicuro. È sicuro che non introduciamo approssimazioni.

(Osserviamo la mappa) All'inizio sarà tutto bianco, tutte le variabili hanno dominio di 3 valori (R, G, B). Prendiamo una variabile qualsiasi e assegniamole un valore (WA ← R) e andiamo a vedere quali sono i valori rimasti nei domini delle altre variabili, rimuovendone a causa del nuovo vincolo appena aggiunto.

- *max-degree* variable



Se abbiamo più di una variabile con il dominio di cardinalità minima, allora si applica questa tecnica per eliminare il pareggio. Viene prevista di prendere la *variabile* coinvolta nel *numero maggiore di vincoli*. Nella realtà, questa tecnica viene usata in combinazione con min-domain, siccome a volte possono sorgere questi problemi di chi scegliere se ci sono pareggi.

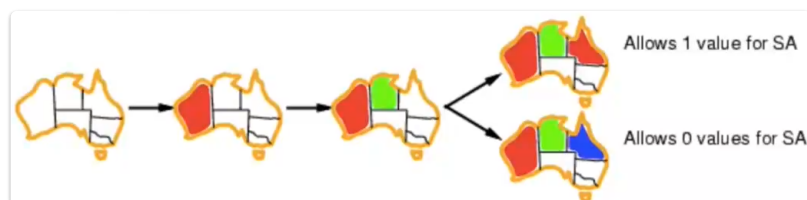
(Osserviamo la mappa) Lo stato che più è coinvolto è quello in basso al centro (SA ← B), quindi si parte da quello; i vincoli vengono aggiornati e si continua.

Sul perché sia meglio usare una tecnica piuttosto che un'altra, sta sempre nella questione dell'albero di ricerca:

- prendo variabili con numero minore di elementi nel dominio per aprire meno l'albero, un fallimento è più rilevante;
- considero variabili con numero massimo di vincoli per andare nella direzione di togliere un numero tanto grande di vincoli in proporzione.

Which Value to Assign Next

Di euristiche per la determinazione del valore d'assegnare ce ne sono diverse. Di queste ne vediamo una sola che è *min-conflict*.



Fissata una variabile per il prossimo assegnamento, si considera il *valore che viola il minor numero di vincoli*. Questo numero è abbastanza complicato da calcolare ma se ne vale la pena, viene fatto.

Per ogni valore possibile, viene controllato se il valore viola vincoli (e se così è può essere scartato) e se con alta probabilità questo sarà, allora suddetti vincoli verranno lasciati per ultimi: vengono scelti quello che violano meno vincoli.

(Osserviamo la mappa) Se scegliamo $Q \leftarrow R$, allora SA potrà assumere 1 solo valore (B). Se scegliamo $Q \leftarrow B$, allora SA non potrà assumere alcun valore, siccome circondato da tutte le possibilità (R,G,B), il che viola il presupposto originale che ogni stato deve essere accerchiato da un colore diverso.

Constraint Propagation

Tutte le volte che noi abbiamo un vincolo e lo utilizziamo per rimuovere valori dai domini delle variabili, stiamo facendo un'operazione che prende il nome di **propagazione del vincolo**. Se lasciate perdere l'euristiche, nello SBT questa operazione non è presente: al massimo ordiniamo i valori all'interno dei domini o rimuoviamo valori nel caso di fallimento.

Lo possiamo aggiungere:

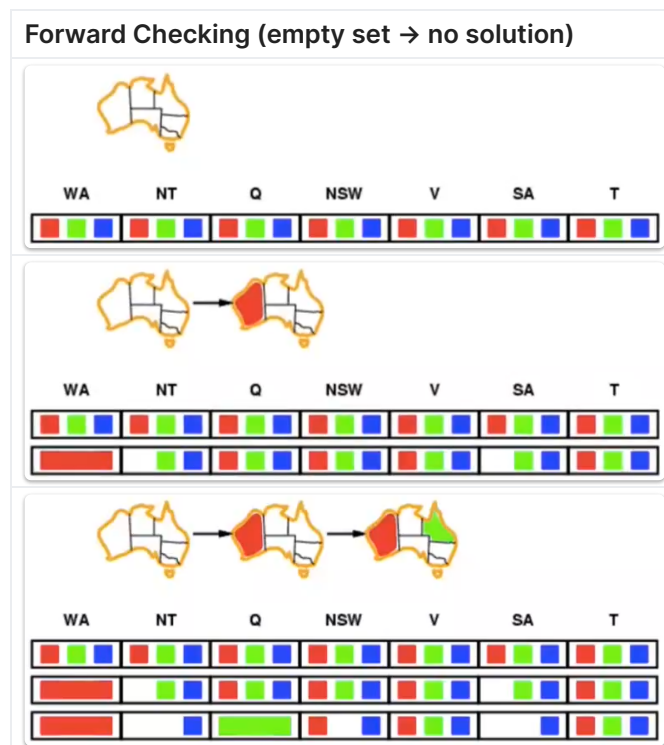
1. Vengono propagati i vincoli, potenzialmente eliminando valori dai domini.
2. Facciamo un assegnamento di una variabile soltanto, scegliendo quale variabile e quale valore in modo euristico.
3. Raggiungere `{ var = value }` vuole dire raggiungere un vincolo, propaghiamo i vincoli.

Questa tecnica serve a capire se un valore ha senso che rimanga nel dominio oppure no. Sfruttare la propagazione dei vincoli è abbastanza complicato per 2 motivi:

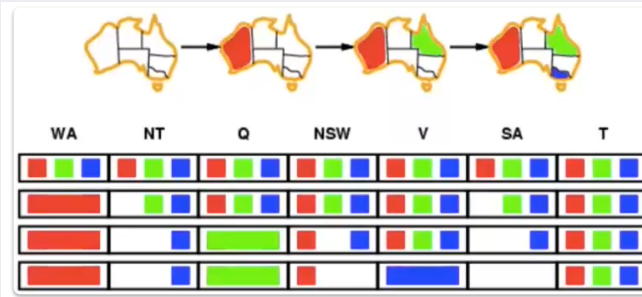
- dobbiamo fare modo di togliere valori dai domini; i valori possono essere talmente tanti che ci serve considerare l'efficacia del metodo;
- la rappresentazione dei dati deve essere fatta in modo da rimettere velocemente i vincoli se questi sono stati tolti perché se tolti all'inizio, non ci sono e siamo felici, ma se tolti nel mentre degli assegnamenti devono essere reinseriti.

Forward Checking

Ad assegnamento avvenuto, andiamo a togliere tutti i valori che sono inconsistenti con l'assegnamento appena fatto, con un passo soltanto.



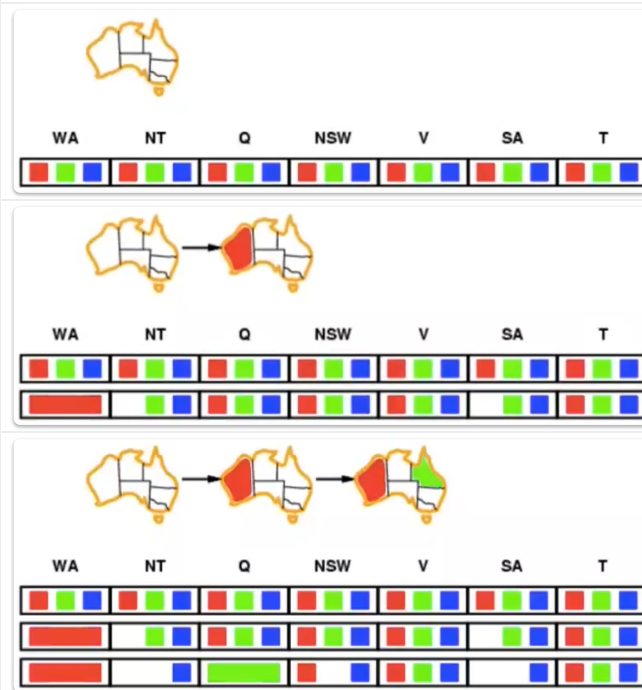
Forward Checking (empty set → no solution)



Nella tabella sopra, vediamo quali sono i passaggi dell'algoritmo di **forward checking** per la solita mappa australiana e notiamo come lo stato SA non abbia alcun valore assegnabile: in questo caso viene invocato SBT per tornare all'inizio e ricominciare, controllando usando nuovi valori compatibili.

Come tutti gli algoritmi di propagazione dei vincoli, attenzione va posta in quanto le inconsistenze non sono tutte identificabili nell'immediato: per come andiamo a controllare i vincoli, alcune sono subdole.

Forward Checking (inconsistent → no solution)



Usando la risoluzione degli stessi passi della mappa, vediamo come oltre al problema del set vuoto di possibilità, un'inconsistenza sia presente per NT e SA, che hanno come valori possibili solamente **B** (non va bene, siccome sono adiacenti $NT \neq SA$).

Arc Consistency

Siccome non possiamo accorgerci di inconsistenze, usiamo l'idea della **consistenza ad arco** (AC) come soluzione. Una volta tolto un valore dal dominio, perché fermarci solo ai prossimi vicini? Procediamo in avanti finché non arriviamo a punto fisso che può essere:

- un qualche dominio che si svuota (empty set), trovando quindi un'inconsistenza e invocando SBT;
- non ci sono più valori da togliere, arrivando a stato stabile con potenzialmente meno valori nei domini.

Questo algoritmo non è in grado di trovare tutte le inconsistenze, come già preannunciato, ma è un buon compromesso che spesso si applica. Il forward checking lo si descrive sul grafo dei vincoli: il grafo non è consistente se non esiste almeno una copia di valori tale per cui l'assegnamento su quell'arco non viola i vincoli.

Un CSP binario si dice consistente in base agli archi se i domini di variabili assicurano che tutti gli archi nel grafo dei vincoli siano consistenti:

- l'arco diretto $X \rightarrow Y$ è consistente se e soltanto se per tutti i valori $x \in \text{dom}(X)$ esiste un valore $y \in \text{dom}(Y)$ per cui il vincolo sia soddisfatto;
- l'arco non diretto $X \dashv Y$ è consistente se e soltanto se $X \rightarrow Y$ e $Y \rightarrow X$ sono entrambi consistenti.

Gli algoritmi AC sono una ventina, quasi tutti numerati.

Di questi vedremo AC-3, uno dei più facili da implementare e da seguire.

```
function AC-3(csp) returns the CSP, possibly with reduced domains
  inputs: csp, a binary CSP with variables {X1, X2, ..., Xn}
  local variables: queue, a queue of arcs, initially all the arcs in csp

  while queue is not empty do
    (Xi, Xj) ← Remove-First(queue)
    if RM-Inconsistent-Values(Xi, Xj) then
      for each Xk in Neighbours[Xi] do
        add (Xk, Xi) to queue
  ---
function RM-Inconsistent-Values(Xi, Xj) returns true iff remove a value
  removed ← false
  for each x in Domain[Xi] do
    if no value y in Domain[Xj] allows (x,y) to satisfy constraint(Xi, Xj)
      then delete x from Domain[Xi]; removed ← true
  return removed
```

AC-3 ha come parametro un `csp` e quello che produce è lo stesso CSP con potenzialmente meno elementi all'interno dei domini. L'obiettivo dell'algoritmo è di ridurre elementi all'interno dei domini delle variabili, magari a insieme vuoto.

L'input è un CSP binario (vincoli di sole 2 variabili).

Le variabili locali sono una coda di archi `queue of arcs`, indicante la sequenza in cui andiamo a esaminare tutti gli archi all'interno del grafo. Questa coda contiene, nell'ordine dell'algoritmo, tutti gli archi del `csp`.

Uno alla volta, gli archi vengono tutti tolti `RM-Inconsistent-Values`, se si decide che qualcuno va rianalizzato, questi andranno rimessi nella coda e così continuiamo.

Ogni arco viene tolto e rimesso un numero finito di volte, il che significa che la coda prima o poi diverrà vuota, e non potremo più eliminare niente (il CSP è consistente con meno valori, oppure qualche dominio è vuoto).

Con il ciclo `while` quindi, tiriamo fuori dalla coda un arco alla volta, il che comporta lo svuotamento. Il primo arco (X_i, X_j) viene rimosso `Remove-First(queue)`, chiamiamo `RM-Inconsistent-Values(Xi, Xj)` per rimuovere valori inconsistenti nell'arco che va da X_i a X_j .

`RM-Inconsistent-Values` ritorna `true` se almeno un valore è stato rimosso, mentre ritorna `false` se $\text{dom}(X_i)$ e $\text{dom}(X_j)$ sono rimasti quelli di prima. Nel caso siano cambiati i domini, quindi, gli archi vengono riaggiunti per riconsiderare tutti i vincoli che potenzialmente sono da prendere. Per come è fatta la funzione, questa prende in considerazione soltanto i nodi vicini a X_i , siccome X_j non lo tocca: solo X_i ha dominio che cambia, considerando l'insieme dei nodi collegati a esso, aggiungiamo tutti gli archi da nodo $\rightarrow X_i$, all'interno della coda.

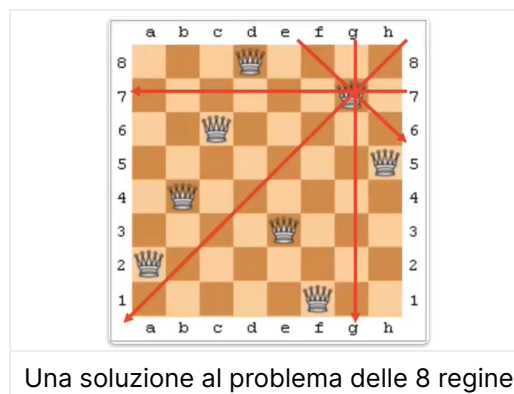
Se prendiamo l'algoritmo AC e lo inseriamo in un algoritmo SBT con euristiche a scelta, stiamo facendo un algoritmo detto **maintaining arc consistency** (MAC). Sarebbe un algoritmo che forza la consistenza per ciascuna delle variabili del dominio; anziché andare fare il test per fare SBT, verificando la consistenza degli assegnamenti, *togliamo valori dai domini*. Seguendo questa tecnica, siamo in grado di risolvere CSP in modo efficace.

Parametric CSPs

Le ottimizzazioni che abbiamo visto fino adesso sono mirate, ma hanno comunque un certo costo computazionale non discriminabile. Quello che bisognerebbe fare, sarebbe un'analisi del CSP che sia scalato di difficoltà, in modo da determinare se la nostra soluzione sia lo stesso applicabile al nuovo problema più complicato.

I **CSP parametrici** hanno almeno 1 parametro che permette variazione di dimensione, da utilizzare per studiare le caratteristiche di complessità della tecnica in considerazione.

n -Queens



Il problema delle n -regine degli scacchi, è un esempio tipico e popolarizzato nell'informatica, usato per mostrare come un concetto semplice all'apparenza, nasconda concetti intricati.

Pubblicato nel 1848 da Max Bezzel(1824-1871) e popolarizzato da matematici come Carl Friedrich Gauss(1777-1855), il puzzle delle 8 regine aveva come obiettivo quello di chiedere di sistemare 8 regine, su una scacchiera, in modo che nessuna di queste si minacciasse l'una con l'altra, o in parole più semplici: che nessuna coppia di regine si minacci. Nel '72, questo problema viene utilizzato da Edsger Dijkstra(1930-2002) per dimostrare la potenza della sua programmazione strutturata.

Il problema delle 8 regine può essere esteso per n -regine per mostrare come più soluzioni sono possibili, introducendo più variabili. Data una scacchiera $n \times n$, il problema chiede di posizionare n regine su di essa in modo che:

- nessuna sia sulla stessa riga;
- nessuna sia sulla stessa colonna;
- nessuna sia sulla stessa diagonale.


A oggi, set di soluzioni per questo tipo di problema sono noti e il più grande è 234,907,967,154,122,528 con $n = 27$. Stiamo parlando di soluzioni: le configurazioni possibili sono estremamente maggiori; nel 2021, Michael Simkin provò che il numero di soluzioni per il puzzle delle n -regine per n grande, è approssimativamente pari a $(0.143n)^n$, dal costo asintotico $\mathcal{Q}(n)$ con α costante tra 1.939 e 1.945 di

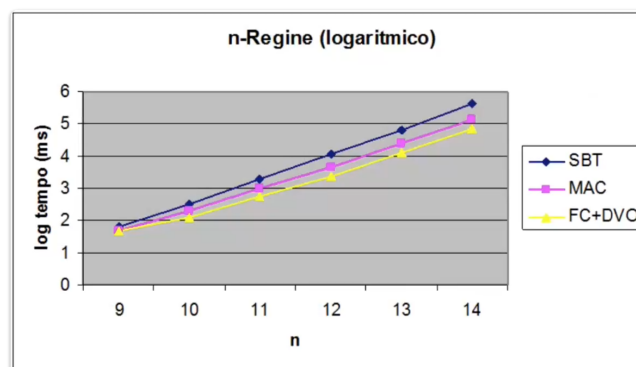
$$\mathcal{Q}(n) = (1 + o(1))ne^{-\alpha}n$$

Un problema con $n = 8$, può avere 16,777,216 possibili configurazioni, delle quali tuttavia soltanto 92 sono soluzioni.

| n | # possible solutions | G&T (ms) | SBT (ms) | MAC (ms) | FC + DV0 (ms) |
|---|----------------------|----------|----------|----------|---------------|
| 4 | 2 | 0 | 0 | 0 | 0 |
| 5 | 10 | 0 | 0 | 0 | 16 |
| 6 | 4 | 32 | 0 | 0 | 31 |
| 7 | 40 | 625 | 0 | 0 | 32 |

| n | # possible solutions | G&T (ms) | SBT (ms) | MAC (ms) | FC + DVO (ms) |
|----|----------------------|----------|----------|----------|---------------|
| 8 | 92 | 15,188 | 0 | 15 | 31 |
| 9 | 352 | 321,828 | 63 | 47 | 47 |
| 10 | 724 | > 1h | 328 | 204 | 125 |
| 11 | 2680 | N/A | 1,953 | 984 | 547 |
| 12 | 14200 | N/A | 12,016 | 4,391 | 2,328 |

Il problema a ricerca esaustiva quindi non porta molto lontano, siccome G&T dovrebbe essere in grado di vedere tutte le possibilità per estrarre le soluzioni. Sopra vediamo una tabella che mostra quanto tempo ciascun algoritmo implementato in JAVA , impieghi a trovare una soluzione: notare come il semplice algoritmo SBT in pseudo codice C, senza euristiche e quant'altro, c'impieghi nettamente meno rispetto a un G&T, ma anche come MAC o FC+DVO (Forward-Checking + Dynamic Variable Ordering) si comportino molto bene.



Logaritmicamente, vediamo come scalano i nostri algoritmi (escludendo G&T).

SBT comunque rimane sempre più alto, MAC al centro e FC+DVO, sono esattamente quello che c'interessava sapere: se il nostro problema è qualche cosa come il problema delle n -regine, conviene fare l'ultimo degli algoritmi, con meno propagazione dei vincoli ed euristica di ordinamento dinamico delle variabili.

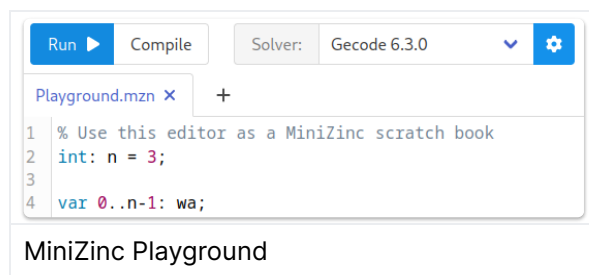
MiniZinc

Per realizzare questi algoritmi, non si scrivono programmi in C o JAVA che siano, quando piuttosto si usano linguaggi dedicati. Uno di questi linguaggi non Turing completo prende il nome di **MiniZinc** (<https://github.com/MiniZinc/libminizinc>).

MiniZinc permette l'espressione semplice di CSP che può poi essere interfacciato con risolutore, sfruttando le caratteristiche di questo per risolvere nel modo migliore possibile il problema. È un linguaggio di descrizione: diciamo quale è il nostro CSP specificando variabili, domini, vincoli e diciamo quale interfaccia usare di MiniZinc per risolverlo.

MiniZinc, vale la pena specificare, è versione ridotta del linguaggio di basso livello Zinc, con un modello che risiede tra ASM, C e C++, con una sintassi a somiglianza di Ruby. Facciamo distinzione perché alcune funzioni quali aggiungere variabile durante l'esecuzione, sono originarie di Zinc e non vengono implementate in MiniZinc.

Per realizzare CSP, possiamo usare un risolutore online al link <https://play.minizinc.dev>. Il sito si presenta con una finestra interattiva in cui possiamo digitare il nostro codice, dei bottoni per eseguire e compilare e un menù a tendina, con lo scopo di lasciarci selezionare quale risolutore usare.



Gecode v6.3.0 è il "solver" che usiamo, perché il più famoso ed efficace.

Map1.mzn

Il programma qui sotto, è la realizzazione del CSP della mappa australiana usando il linguaggio MiniZinc. L'estensione del file è `mzn`. Una volta scritto un programma, per vederne l'output è sufficiente avviarlo cliccando il bottone "Run".

```
% una costante
int: n = 3;

% le variabili
var 0..n-1: wa;
var 0..n-1: nt;
var 0..n-1: sa;
var 0..n-1: q;
var 0..n-1: nsw;
var 0..n-1: v;

% i vincoli
constraint wa ≠ nt;
constraint wa ≠ sa;
constraint nt ≠ sa;
constraint nt ≠ q;
constraint sa ≠ q;
constraint sa ≠ nsw;
constraint sa ≠ v;
constraint q ≠ nsw;
constraint nsw ≠ v;

% richiesta di risoluzione a Gecode
solve satisfy;

% richiesta di stampa a minizinc
output [
    "WA = ", show(wa), "\n",
    "NT = ", show(nt), "\n",
    "SA = ", show(sa), "\n",
    "Q = ", show(q), "\n",
    "NSW = ", show(nsw), "\n",
    "V = ", show(v), "\n"
];
```

Traduzione FlatZinc

Una versione ancora più "cruda" di MiniZinc è FlatZinc.

Si tratta dello stesso tipo di codice, ma con un linguaggio ancora di più basso livello: una traduzione a questo tipo può essere fatta, cliccando il bottone "Compile" dell'UI del playground di MiniZinc.

Può tornare utile programmare usando questo linguaggio, nel caso si volesse creare solver di un CSP molto mirato, in quanto parlare con il solver in questo caso, diventa più semplice.

```

array [1..2] of int: X INTRODUCED_0_ = [1,-1];
var 0..2: wa:: output_var;
var 0..2: nt:: output_var;
var 0..2: sa:: output_var;
var 0..2: q:: output_var;
var 0..2: nsw:: output_var;
var 0..2: v:: output_var;
constraint int_lin_ne(X INTRODUCED_0_, [wa,nt],0);
constraint int_lin_ne(X INTRODUCED_0_, [wa,sa],0);
constraint int_lin_ne(X INTRODUCED_0_, [nt,sa],0);
constraint int_lin_ne(X INTRODUCED_0_, [nt,q],0);
constraint int_lin_ne(X INTRODUCED_0_, [sa,q],0);
constraint int_lin_ne(X INTRODUCED_0_, [sa,nsw],0);
constraint int_lin_ne(X INTRODUCED_0_, [sa,v],0);
constraint int_lin_ne(X INTRODUCED_0_, [q,nsw],0);
constraint int_lin_ne(X INTRODUCED_0_, [nsw,v],0);
solve satisfy;

```

Output

L'uscita del programma è il testo che abbiamo chiesto a MiniZinc di stampare, ovvero le variabili. Il tempo che ha impiegato a trovare la soluzione, è scritto in fondo.

```

WA = 2
NT = 1
SA = 0
Q = 2
NSW = 1
V = 2
-----
Finished in 179msec.

```

Map2.mzn

Una versione alternativa a `Map1.mzn`, può essere scritta includendo la libreria `alldifferent.mzn`, che ci permette di usare vincoli molto comuni dei problemi come **alldifferent**. In verità, anche se i vincoli sono sempre gli stessi per l'esempio, quello che stiamo facendo è ottimizzare il codice in quanto meno sono i vincoli da scrivere: algoritmi di propagazione saranno applicati per espandere i vincoli a tutte le variabili, rendendo il tempo d'esecuzione, minore.

```

include "alldifferent.mzn";

%...

% i vincoli
constraint alldifferent([wa,nt,sa]);
constraint alldifferent([nt,q,sa]);
constraint alldifferent([sa,q,nsw]);
constraint alldifferent([sa,nsw,v]);

%...

```

Money.mzn

Questo codice invece è una soluzione a un problema di cripto-aritmetica.

Il problema ha dai due termini dell'operazione somma: SEND e MORE. La soluzione sarà MONEY. Ad ogni lettera è associata una cifra.

```
include "alldifferent.mzn";

var 1..9: S;
var 0..9: E;
var 0..9: N;
var 0..9: D;
var 1..9: M;
var 0..9: O;
var 0..9: R;
var 0..9: Y;

constraint 1000 * S + 100 * E + 10 * N + D
           + 1000 * M + 100 * O + 10 * R + E
           = 10000 * M + 1000 * O + 100 * N + 10 * E + Y;

constraint alldifferent([S,E,N,D,M,O,R,Y]);

solve satisfy;

output [
    " ", show(S), show(E), show(N), show(D), "\n",
    "+ ", show(M), show(O), show(R), show(E), "\n",
    "= ", show(M), show(O), show(N), show(E), show(Y), "\n"
];
```

Output

```
  9567
+ 1085
= 10652
-----
Finished in 188msec.
```

Banana.mzn

Possiamo anche trovare un assegnamento completo e consistente che per esempio massimizzi una funzione aritmetica o un insieme di funzioni viste con una certa priorità. Aniché fare un CSP, facciamo un **constraint optimization problem** (COP).

```
var 0..100: b; % # of banana cakes
var 1..100: c; % # of chocolate cakes

% flour
constraint 250*b + 200 * c ≤ 4000;
% banana
constraint 2*b ≤ 6;
% sugar
constraint 75*b + 150*c ≤ 500;
% cocoa
```

```

constraint 75*c ≤ 500;

% maximize profit
solve maximize 400*b + 450*c;

output [
    "# of banana cakes = ", show(b), "\n",
    "# of chocolate cakes = ", show(c), "\n"
];

```

`b` e `c` sono variabili da 0..100.

I vincoli sono operazioni matematiche.

Un obbiettivo viene definito con `solve maximize`: stiamo chiedendo a MiniZinc di trovare l'assegnamento completo e consistente, potenzialmente tutti, tale da massimizzare il valore espresso in funzione di variabili.

Output

Tutte le esplorazioni vengono elencate: per ogni assegnamento completo e consistente viene calcolato il target tenendo memorizzato il massimo calcolato fino ad un certo istante; procedendo con l'esecuzione, se viene trovato un nuovo assegnamento con valore minore di quello memorizzato questo verrà scartato, altrimenti aggiornato.

```

# of banana cakes = 0
# of chocolate cakes = 1
-----
# of banana cakes = 1
# of chocolate cakes = 1
-----
# of banana cakes = 2
# of chocolate cakes = 1
-----
# of banana cakes = 3
# of chocolate cakes = 1
-----
# of banana cakes = 2
# of chocolate cakes = 2
-----
=====
Finished in 192msec.

```

02/05/2023