O więzach ciąg dalszy

Paweł Rychlikowski

Instytut Informatyki UWr

24 marca 2021

Problemy spełnialności więzów. Przypomnienie definicji

Definicja

Problem spełnialności więzów ma 3 komponenty:

1. Zbiór zmiennych $X_1, \ldots, X_n$
2. Zbiór dziedzin (przypisanych zmiennym)
3. Zbiór więzów, opisujących dozwolone kombinacje wartości jakie mogą przyjmować zmienne.

Przykład

Zmienne: X, Y, Z
Dziedziny: $X \in \{1, 2, 3, 4\}, Y \in \{1, 2\}, Z \in \{4, 5, 6, 7\}$
Więzy: $X + Y \geq Z, X \neq Y$

Więzy i maksymalizacja wartości

- Czasami do problemu więzowego dodajemy dodatkowo zadanie maksymalizacji wartości pewnej funkcji:
  - Przydział robotników do maszyn spełniający określone wymagania i maksymalizujący produktywność.
  - Poprawny plan lekcji, maksymalizujący liczbę spełnionych miękkich wymagań nauczycieli (np. wolałbym nie mieć zajęć w piątek po 12, ale ...)

Uwaga

W takich sytuacjach wybierając wartość bardzo często maksymalizujemy lokalne „zadowolenie” z rozwiązania.

Super słaby backtracking

- Zwróćmy uwagę, że zachłanny algorytm wybierający zmienną (trudną) i wartość (obiecującą) jest np. algorytmem układania planu (nawet bez backtrackingu).
- Z drugiej strony przestrzeń jest tak ogromna (kilkaset zajęć, każde w kilkudziesięciu terminach), że trudno spodziewać się, aby backtracking dał sobie z nią radę
(nawet backtracking na sterydach)

Super słaby backtracking

Limited Discrepancy Search

LDS (przeszukiwanie o ograniczonej rozbieżności) jest wariantem przeszukiwania, w którym jedynie d razy na całe przeszukanie, mamy prawo wziąć nie pierwszy najlepszy termin, lecz drugi! (d jest małe, rzędu 1,2,3)

Można myśleć o tym tak:

- Mamy ciąg decyzji (tyle ile zmiennych)
- Wybieramy d miejsc, w których podejmujemy „nieoptymalne" decyzje (z punktu widzenia heurystyki wyboru wartości)

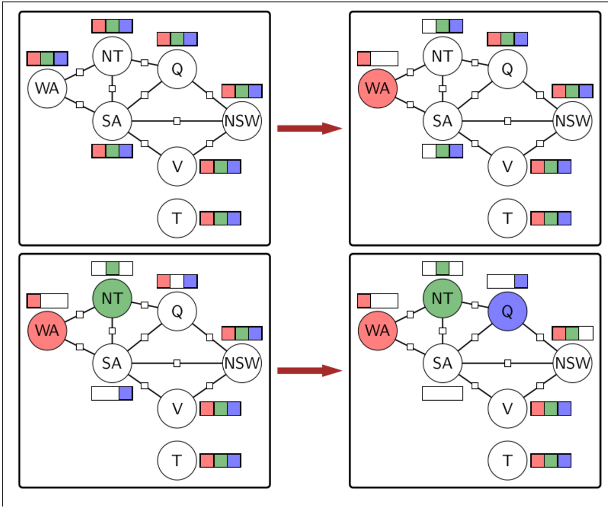
Przeplatanie poszukiwania i wnioskowania

- AC-3 może być kosztowne.
- Uproszczona forma: Forward Checking:
  - Zawsze, jak przypiszemy wartość, sprawdzamy, czy to przypisanie nie zmienia dziedzin innych zmiennych (które są w więzach z obsługiwaną zmienną)
  - I tu zatrzymujemy wnioskowanie.

Uwaga

Coś takiego można wykorzystać jako pełnoprawny algorytm. Wystarczy dodać jakąś losowość i restarty.

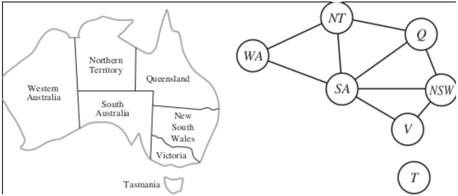
Forward Checking – przykład



Źródło: CS221: Artificial Intelligence: Principles and Techniques

First Fail w praktyce

- W kolorowaniu Australii wszystkie dziedziny na początku są równe...
- ale heurystyka First Fail w drugiej kolejności patrzy na liczbę więzów.



Wybór SA pozwala nam dalsze przeszukiwanie robić bez nawrotów.

Więzy globalne (1)

- Więzy globalne to takie, które opisują relacje dużej liczby zmiennych (np. klasa nie ma okienek)
- Dobrym przykładem jest więz alldifferent$(V_1, \ldots, V_n)$

Uwaga

Oczywiście da się wyrazić równoważny warunek za pomocą $O(n^2)$ więzów $V_i \neq V_j$.

Propagacja więzów globalnych

Przykład

Mamy taką sytuację: $X \in \{1, 2\}, Y \in \{1, 2\}, Z \in \{1, 2\}$,
Więzy: $X \neq Y, Y \neq Z, X \neq Z$

- Spójny łukowo (niemożliwa propagacja)
- Globalne spojrzenie umożliwia stwierdzenie, że wartości nie starczy

Daje to prosty algorytm wykrywania sprzeczności więzów (porównanie sumy mnogościowej dziedzin i liczby zmiennych).

Część II

Przeszukiwanie lokalne dla CSP

- Przeszukiwanie lokalne nie próbuje systematycznie przeglądać przestrzeni rozwiązań (ogólniej: przestrzeni stanów)
- Zamiast tego pamięta jeden stan (lub niewielką, stałą liczbę stanów)
- Dla CSP stanem będzie kompletne przypisanie (niekoniecznie spełniające więzy).

Problemy optymalizacyjne

- W tych problemach szukamy stanu, który maksymalizuje wartość pewnej funkcji (jakość planu).
- Często problemy z twardymi warunkami da się zamienić na problemy optymalizacyjne. Jak?

Można policzyć liczbę złych wierszy (kolumn) w obrazkach logicznych, albo liczbę szachów w hetmanach, albo....

MinConflicts

Uwaga

Możemy myśleć o spełnianiu CSP jako o zadaniu maksymalizacji liczby spełnionych więzów.

Możemy zatem stworzyć algorytm, w którym:

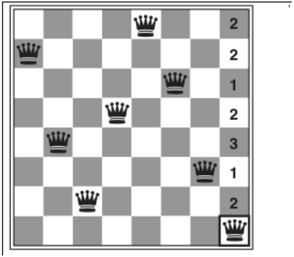
- Zmieniamy tę zmienną, która powoduje niespełnienie największej liczby więzów.
- Wybieramy dla niej wartość, która owocuje najmniejszą liczbą konfliktów.

Przykład: 8 hetmanów

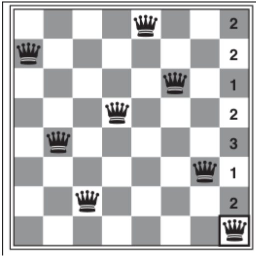
- Jak wybrać stan? (Wskazówka: powinniśmy umieć łatwo przejść ze stanu do stanu)
- Stan: w każdej kolumnie 1 hetman, Ruch: przesunięcie hetmana w górę lub w dół

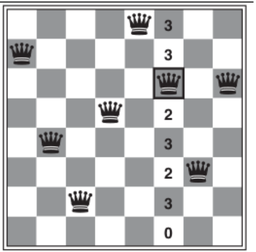
Popatrzmy, jak działa min-conflicts dla hetmanów.

Min-conflicts dla hetmanów

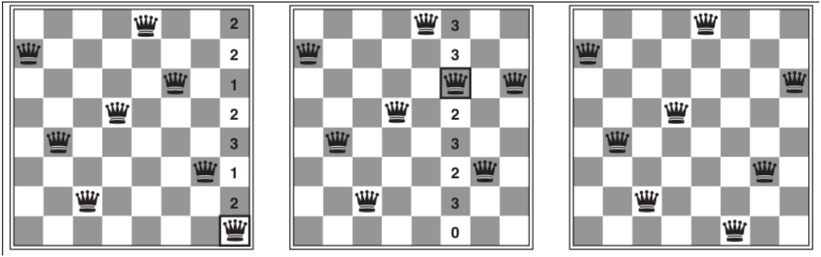


Min-conflicts dla hetmanów





Min-conflicts dla hetmanów



Hetmany. Wyniki

- Dla planszy 8×8 osiąga sukces w 14% przypadków.
- Niby niezbyt dużo, ale możemy go uruchomić na przykład 20 razy, wówczas p-stwo sukcesu to ponad 95%.
- Można dopuszczać pewną liczbę ruchów w bok (czyli, że nie musimy poprawić, ale wystarczy, że nie pogorszymy, jak na obrazkach).
- Jak dopuścimy ruchy w bok , to wówczas mamy sukces w 94%

Ważenie więzów

- Każdy więz ma wagę, początkowo wszystkie równe na przykład 1
- Waga więzów niespełnionych cały czas troszkę rośnie.
- Chcemy naprawiać nie zbiór więzów o liczności n, ale raczej zbiór więzów o największej sumarycznej wadze

Więzy trudne, rzadko spełniane będą miały coraz większy priorytet.

Więzy on-line

- Wyobraźmy sobie, że mamy problem, który się zmienia (ale w niewielkim stopniu)
- Przykład: obsługa linii lotniczych – bo zamykają się lotniska, pilot może złapać grypę, ...

On-line CSP

Min-conflicts umożliwia rozwiązywanie tego typu zadań: stan początkowy to ostatnie dobre przypisanie.

Część III

Constraint Logic Programming

- Spróbujemy powiedzieć o programowaniu logicznym z więzami mówiąc maksymalnie mało o samym programowaniu logicznym
- o którym z kolei trochę powiemy, jak będziemy zajmowali się logiką.

Uwaga

Możemy (na płytkim poziomie) potraktować CLP jako constraint solver, czyli system, w którym definiujemy zadanie więzowe i otrzymujemy rozwiązanie.

Constraint Solver

Deklaratywne podejście do programowania

- Wypisujemy więzy (w jakimś formalnym języku)
- (możemy się wspomóc programowaniem, więzów może być dużo)
- Rozwiązaniem zajmuje się Solver (nie musimy implementować propagacji więzów i backtrackingu)

Przykładowe systemy CLP

- SWI-Prolog (ma moduł clpfd)
- GNU-Prolog (trochę stary i nierozwijany)
- Eclipse (http://eclipseclp.org/)

- Zmienne FD (clpfd)
- Zmienne boolowskie (clpb)
- Zmienne rzeczywiste i wymierne (clpr)

Zajmiemy się tylko zmiennymi FD.

clp(X)

Rozważa się również inne X-y: napisy, zbiory, przedziały.

Składowe zadania w CLP

Przypominamy: musimy określić zmienne, ich dziedziny oraz więzy na nich.

Zmienne

Zmienne są zmiennymi prologowymi, piszemy je wielką literą.

Dziedziny

```
V in 1..10
[A,B,C,D] ins 1..10
```

Więzy

Języki CLP mają bardzo bogate możliwości wyrażania problemów za pomocą więzów.

Przyślijcie Więcej Pieniędzy


```
puzzle(Vars) :-
        Vars = [S,E,N,D,M,O,R,Y],
        Vars ins 0..9,
        all_different(Vars),
                     S*1000 + E*100 + N*10 + D +
                     M*1000 + O*100 + R*10 + E #=
        M*10000 + O*1000 + N*100 + E*10 + Y,
        M #\= 0, S #\= 0.
```

Wykorzystanie solwera więzowego

Postać programu CLP

```
name([V1, ..., Vn]) :-
      V1 in 1..K, ..., Vn in 1..K,
      V1 # >= V5, abs(V2+V6) # = abs(V7-V8)
      min(V3,V7) # > V3 + 2*V1
      labeling([options], [V1,...,Vn]).
```

Wykorzystanie solwera więzowego

Postać programu CLP

```
name([V1, ..., Vn]) :-
     V1 in 1..K, ..., Vn in 1..K,
     V1 # >= V5, abs(V2+V6) # = abs(V7-V8)
     min(V3,V7) # > V3 + 2*V1
     labeling([options], [V1,...,Vn]).
```

- Czyli mamy obsługę zmiennych i dziedzin, ustanowienie więzów oraz wywołanie przeszukiwania z nawrotami (labeling), a na końcu wywołanie głównego predykatu.

Wykorzystanie solwera więzowego

Postać programu CLP

```
name([V1, ..., Vn]) :-
     V1 in 1..K, ..., Vn in 1..K,
     V1 # >= V5, abs(V2+V6) # = abs(V7-V8)
     min(V3,V7) # > V3 + 2*V1
     labeling([options], [V1,...,Vn]).
```

- Czyli mamy obsługę zmiennych i dziedzin, ustanowienie więzów oraz wywołanie przeszukiwania z nawrotami (labeling), a na końcu wywołanie głównego predykatu.
- Część czarna jest częścią *techniczną*, stanowiącą naszą *daninę* dla Prologa

Wykorzystanie solwera więzowego

Postać programu CLP

```
name([V1, ..., Vn]) :-
     V1 in 1..K, ..., Vn in 1..K,
     V1 # >= V5, abs(V2+V6) # = abs(V7-V8)
     min(V3,V7) # > V3 + 2*V1
     labeling([options], [V1,...,Vn]).
```

- Czyli mamy obsługę zmiennych i dziedzin, ustanowienie więzów oraz wywołanie przeszukiwania z nawrotami (labeling), a na końcu wywołanie głównego predykatu.
- Część czarna jest częścią *techniczną*, stanowiącą naszą *daninę* dla Prologa
- Ten program możemy napisać, używając Ulubionego Języka Programowania – wystarczy, że ma `print`, `printf`, `puts`, ...

Przykład

Popatrzmy, jak to działa dla zadania z N hetmanami.

- Warunki określające dziedziny:

```
def domains(Qs, N):
    return [ q + ' in 0..' + str(N-1) for q in Qs ]
```

- Brak szachów w poziomie (alldifferent)

```
def all_different(Qs):
    return ['all_distinct([' + ', '.join(Qs) + '])']
```

- Brak szachów po przekątnej

```
def diagonal(Qs):
    N = len(Qs)
    return [ "abs(%s - %s) #\\= abs(%d-%d)" % (Qs[i],Qs[j],i,j)
           for i in range(N) for j in range(N) if i != j ]
```

Sklejenie wszystkich części:

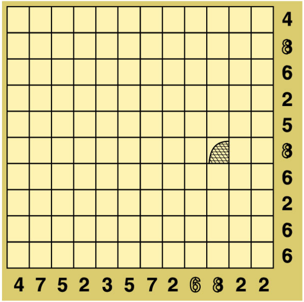
```
def queens(N):
    vs = ['Q' + str(i) for i in range(N)]
    print ':- use_module(library(clpfd)).'
    print 'solve([' + ', '.join(vs) + ']) :- '

    cs = domains(vs, N) + all_different(vs) + diagonal(vs)

    print_constraints(cs, 4, 70),
    print
    print '    labeling([ff], [' +  commas(vs) + ']).'
    print
    print ':- solve(X), write(X), nl.'
```

Testowanie hetmanów

- Zobaczmy, jak działa program queen_produce.py
- Jak wyglądają wynikowe programy
- Jak duże instancje jesteśmy w stanie rozwiązywać?

Przykład 2: burze

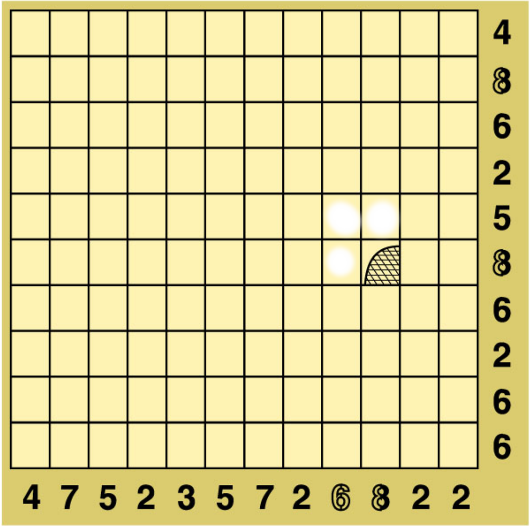
Może pojawią się na liście P3...



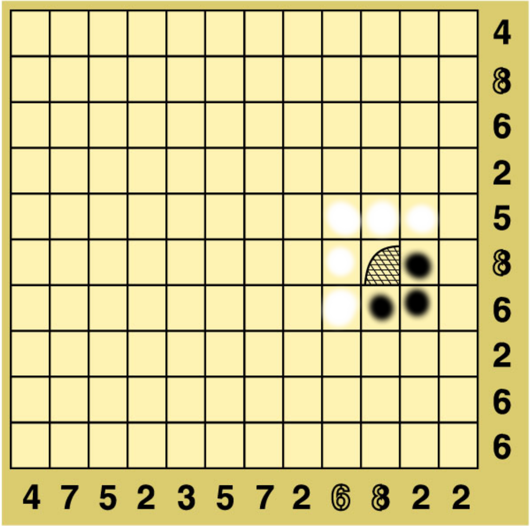
Zasady

1. Radary mówią, ile jest pól burzowych w wierszach i kolumnach.
2. Burze są prostokątne.
3. Burze nie stykają się rogami.
4. Burze mają wymiar co najmniej 2×2.

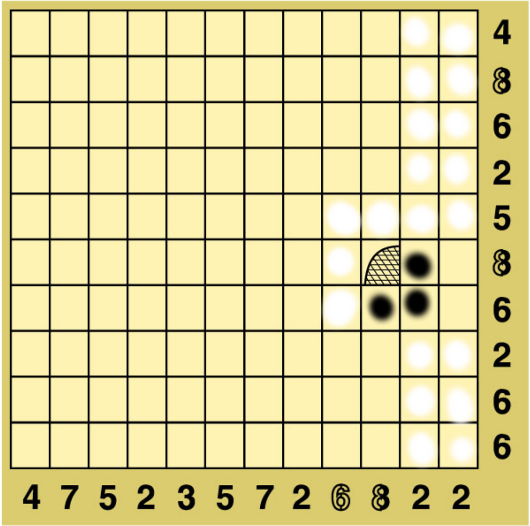
Burze: wnioskowanie



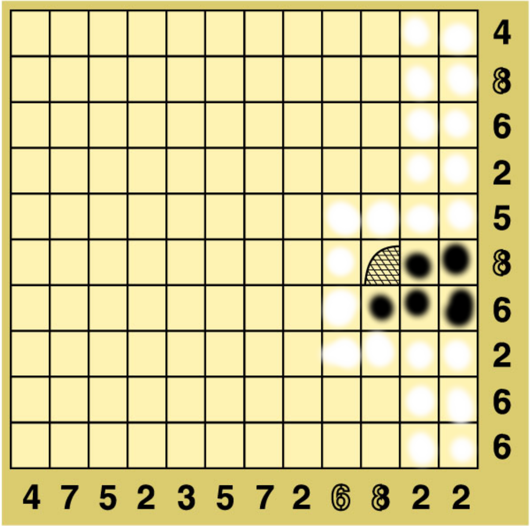
Burze: wnioskowanie



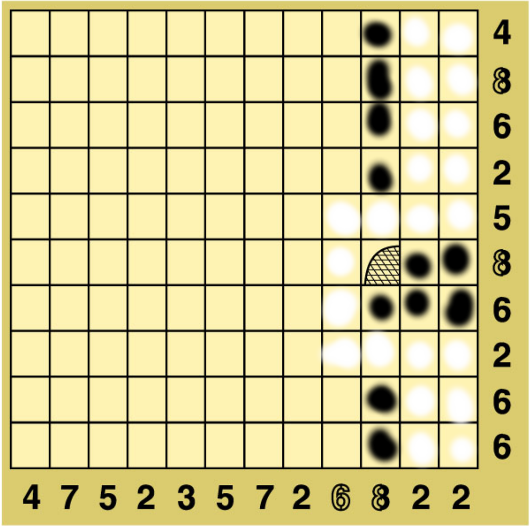
Burze: wnioskowanie



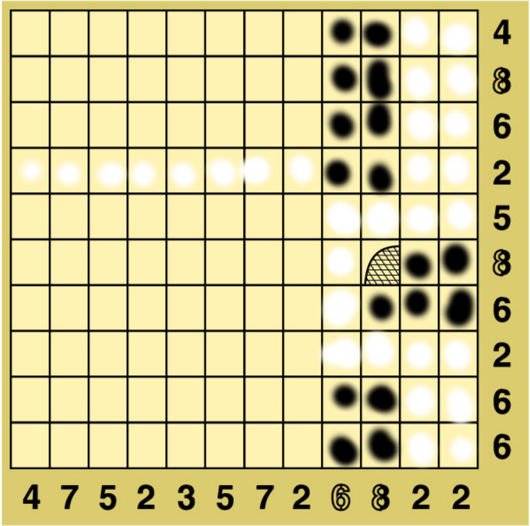
Burze: wnioskowanie



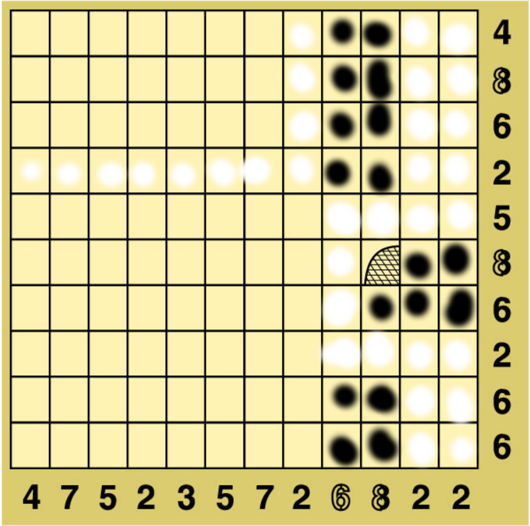
Burze: wnioskowanie



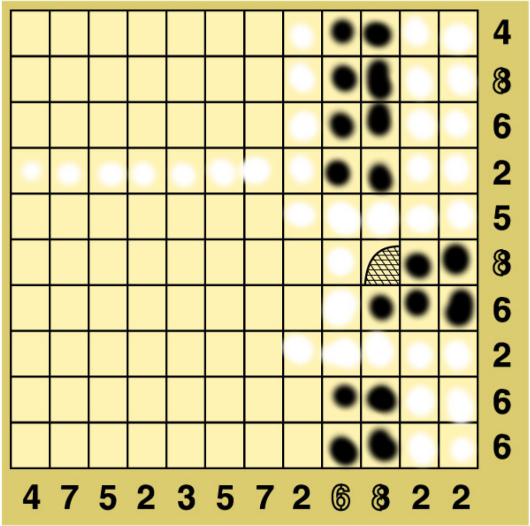
Burze: wnioskowanie



Burze: wnioskowanie



Burze: wnioskowanie



Rozwiązanie

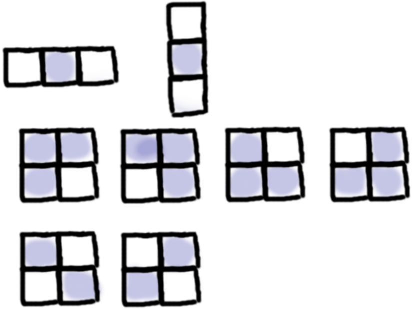
- Strategia 1: jak obrazki logiczne, + wnioskowanie
- Strategia 2: wykorzystujemy SWI-Prolog

Kodowanie burz

- Zmienne, dziedziny: piksele, 0..1
- Radary: $b_1 + b_2 + \cdots + b_n = K$
- Prostokąty: ?
- Co najmniej 2×2?
- Nie stykają się rogami.

- Jak wygląda każdy kwadrat 2×2?
- Jak wygląda każdy prostokąt 1×3 albo 3×1?

Zabronione układy



Pytanie

Jak wyrazić to językiem relacji arytmetycznych?

Warunek dobrych 3 pól

Mamy zmienne A, B, C

- $A + 2B + 3C \neq 2$
- $B \times (A + C) \neq 2$

Reifikacja. Warunki w więzach

Naturalne sformułowanie

Jeżeli środkowy piksel jest ustawiony, to wówczas przynajmniej 1 z otaczających go jest jedynką.

$$B \Rightarrow (A + C > 0)$$

Reifikacja (cd)

- Inny przykład: `A #<=> B #> C`
- Naturalna propagacja:
  - Ustalenie A dorzuca więz
  - Jak wiemy, czy prawdziwy jest `B #> C`, to znamy wartość `A`

Inne więzy globalne

tuples_in

Wymieniamy explicite krotki wartości, jakie może przyjmować krotka zmiennych

Uwaga

Zauważmy, że ten więz pasuje do lokalnych warunków dla burz, na przykład dla prostokątów 3×1::

```
tuple_in( [A,B,C], [ [0,0,0], [1,1,0], [1,0,0],
[0,1,1], [0,0,1], [1,1,1], [1,0,1]]
```