Języki definiowania więzów i zachłanne rozwiązywanie więzów

Paweł Rychlikowski

Instytut Informatyki UWr

29 marca 2019

Przyślijcie Więcej Pieniędzy. Przypomnienie


```
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.
```

CLP bez Prologa. Przypomnienie

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) #> V4 + 2*V1,  % etc

    labeling([options], [V1,...,Vn]).

:- name(Solution), write(Solution), nl.
```

- Czyli mamy obsługę zmiennych, ustanowienie więzów oraz wywołanie przeszukiwania, a na końcu wywołanie głównego predykatu.
- Ten program możemy napisać, używając Ulubionego Języka Programowania – wystarczy, że ma `print`, `printf`, `puts`, …

Pajtono-prolog

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 ]
```

Pajtono-prolog (2)

Sklejenie wszystkich części:

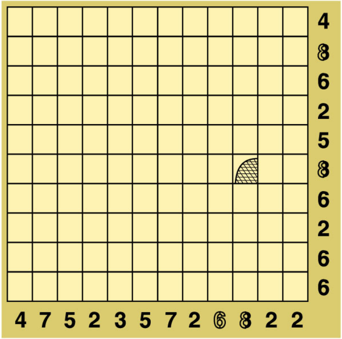
```
def queens(N):
    vs = ['Q' + str(i) for i in range(N)]
    print ':- use_module(library(clpfd)).'
    print 'solve([' + ', '.join(vs) + ']) :- '

    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

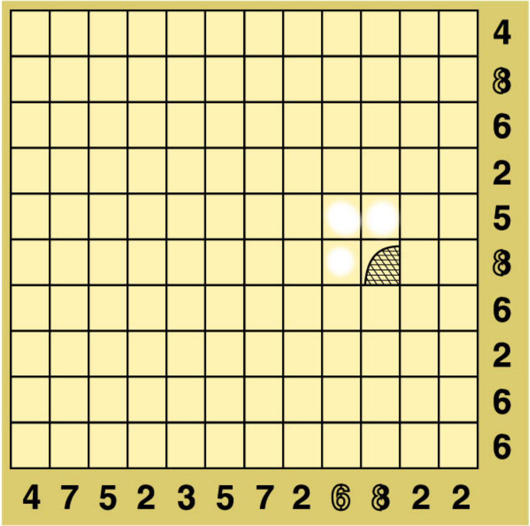
Może pojawią się na liście P3...



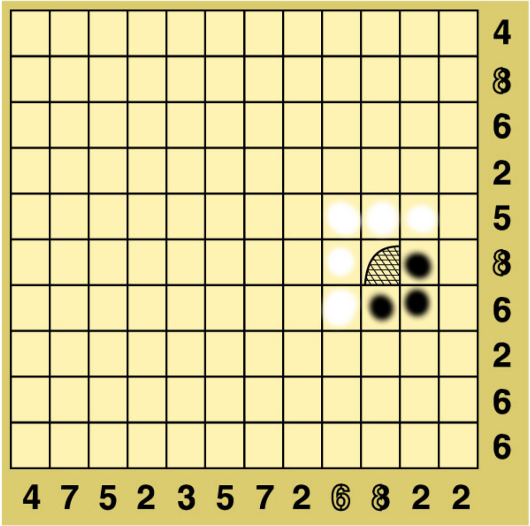
Zasady

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.

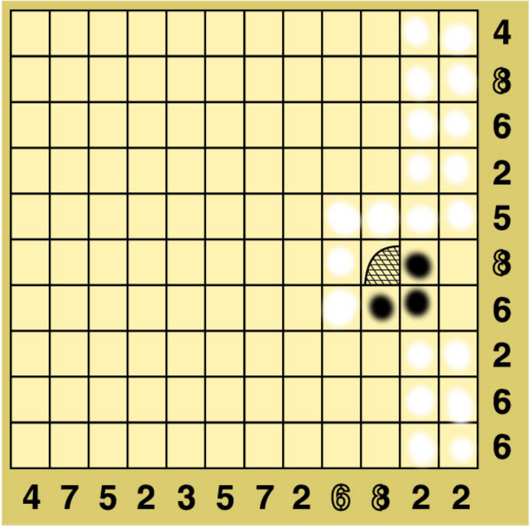
Burze: wnioskowanie



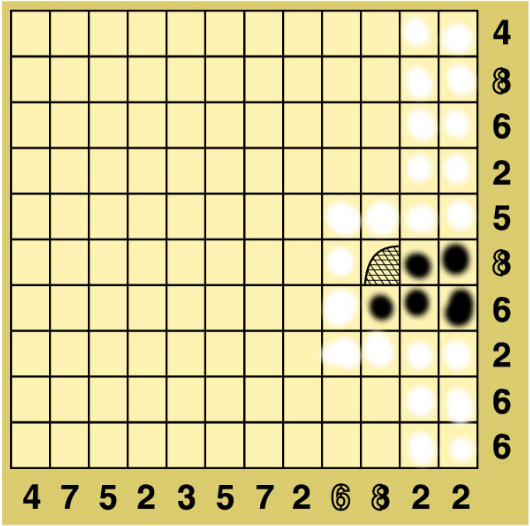
Burze: wnioskowanie



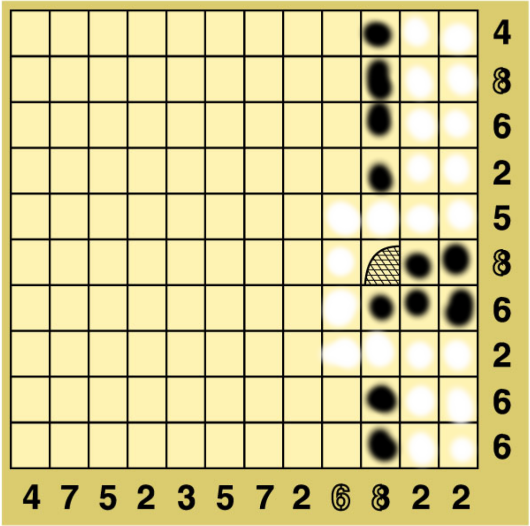
Burze: wnioskowanie



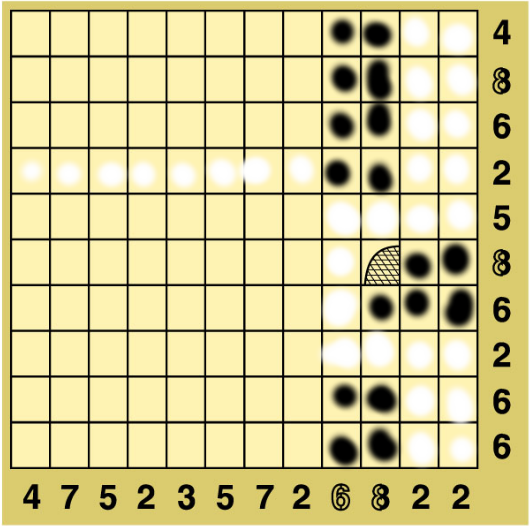
Burze: wnioskowanie



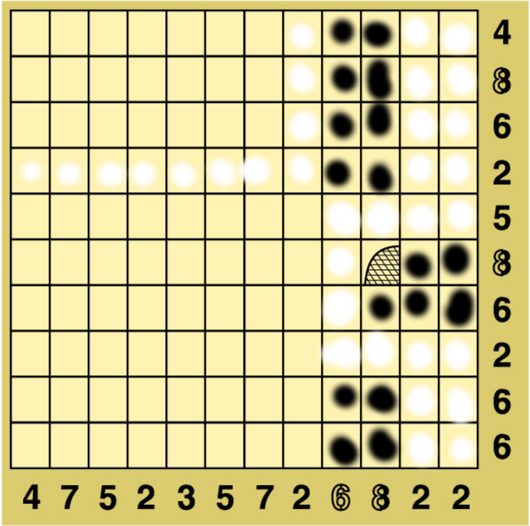
Burze: wnioskowanie



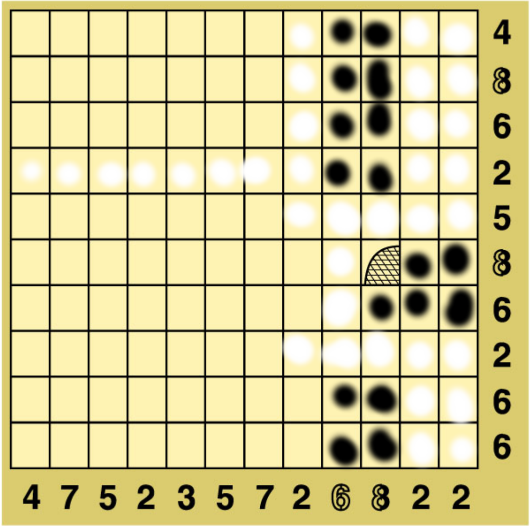
Burze: wnioskowanie



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.

Kodowanie burz

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

Zabronione układy

010	11	11	01	10	01	10	0
	01	10	10	01	11	11	1
							0

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.

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

- 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]]
```

Struktura problemu

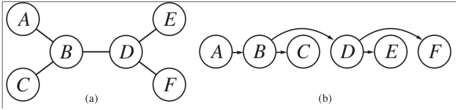
Uwaga

Patrząc na graf więzów, możemy zauważyć pewne właściwości. Na przykład podzielić więzy graf na spójne składowe i rozwiązać je osobno (Tasmania!).

Uwaga 2

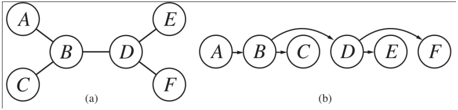
Inną ważną klasą grafów są drzewa. CSP o strukturze drzewiastej da się łatwo rozwiązać. Jak?

Drzewiaste CSP



1. Sortujemy topologicznie drzewo
2. Osiągamy spójność łukową (algorytm AC-3)
3. Rozwiązujemy szybko taką sieć więzów, zaczynając od korzenia.

Drzewiaste CSP



Algorytm

Trywialny: idziemy od lewej do prawej, wybieramy dowolne wartości z dostępnych w danym momencie.

Pamiętajmy o gwarancji, jaką daje AC-3!

Udrzewianie

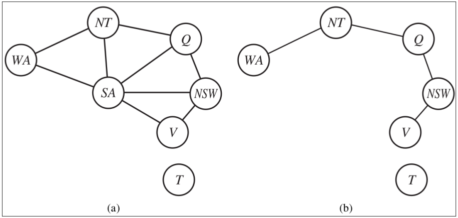
- Cieszymy się z algorytmu na drzewach, ale jak sprawić, by CSP stało się drzewem?
- I czy zawsze się da?

Uwaga

Pamiętajmy, że CSP jest NP-zupełne, a nie spodziewamy się wielomianowego algorytmu dla takich problemów!

Udrzewianie (2)

Można usunąć jakiś węzeł (węzły)



Popatrzmy na Australię:

- Usunięcie węzła to przypisanie wartości zmiennej (i sprawdzenie innych wartości w kolejnych nawrotach)
- Problem **cycle cutset** (na liście c3)
- Używane również w przetwarzaniu sieci bayesowkich

Obrazki logiczne z P1

- Jedno z pierwszych zadań na naszej pracowni to były obrazki logiczne (inspirowane algorytmem WalkSat)
- Spróbujemy uogólnić sobie te idee na dowolne CSP.

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:

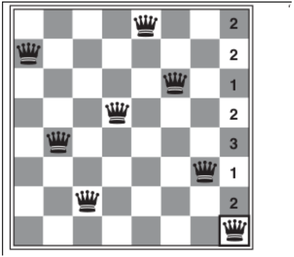
- Zmieniamy tę zmienną, która powoduje niespełnienie największej liczby więzów.
- Wybieramy dla niej wartość, która owocuje najmniejszą liczbą konfliktów.

Przykład: 8 hetmanów

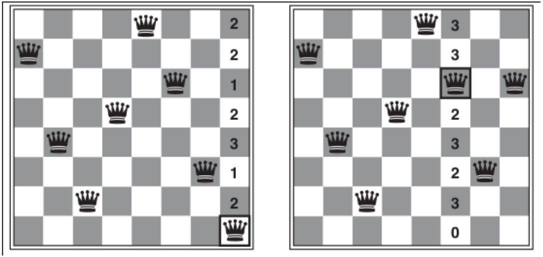
- Jak wybrać stan? (Wskazówka: powinniśmy umieć łatwo przejść ze stanu do stanu)
- Stan: w każdej kolumnie 1 hetman, Ruch: przesunięcie hetmana w górę lub w dół

Popatrzmy, jak działa min-conflicts dla hetmanów.

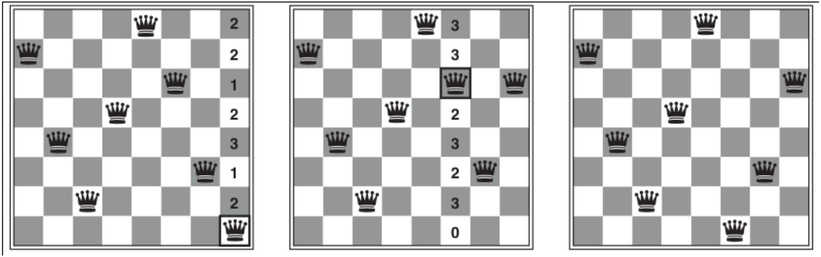
Min-conflicts dla hetmanów



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 możemy poprawić, ale możemy nie pogorszyć, 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, ...

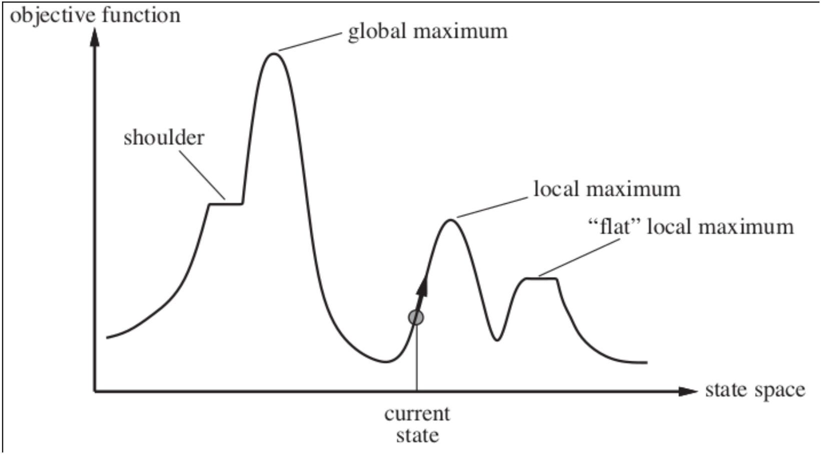
On-line CSP

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

Przeszukiwania lokalne (ogólnie)

- Powiemy sobie jeszcze o paru ideach związanych z przeszukiwaniem lokalnym.
- Można je wykorzystywać w zadaniach więzowych, ale nie tylko.

Krajobraz przeszukiwania lokalnego



Hill climbing jest chyba najbardziej naturalnym algorytmem inspirowanym poprzednim rysunkiem.

- Dla stanu znajdujemy wszystkie następniki i wybieramy ten, który ma największą wartość.
- Powtarzamy aż do momentu, w którym nie możemy nic poprawić

Problem

Oczywiście możemy utknąć w lokalnym maksimum.