

Programowanie z więzami oraz przeszukiwanie lokalne

Paweł Rychlikowski

Instytut Informatyki UWr

10 kwietnia 2025

Wykorzystanie CLP w rozwiązywaniu więzów

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

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. Przypomnienie

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

Składowe zadania w CLP

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

Zmienne

Zmienne są zmiennymi prologowymi, piszemy je **wielką literą** (to nie jest konwencja, tylko silne wymaganie!).

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.
```

Uwaga: przecinek oddziela **atomy**, na końcu **klauzuli** musi być kropka.

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]).
```

(nie ma spacji po krzyżykach!)

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]).
```

(nie ma spacji po krzyżykach!)

- Czyli mamy obsługę **zmiennych i dziedzin**, **ustanowienie więzów** oraz wywołanie przeszukiwania z nawrotami (labeling).

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]).
```

(nie ma spacji po krzyżykach!)

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

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]).
```

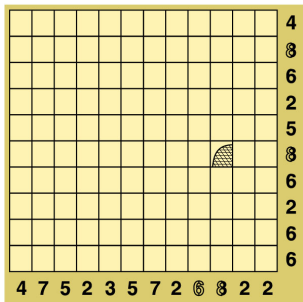
(nie ma spacji po krzyżykach!)

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

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

Przykład 2: burze

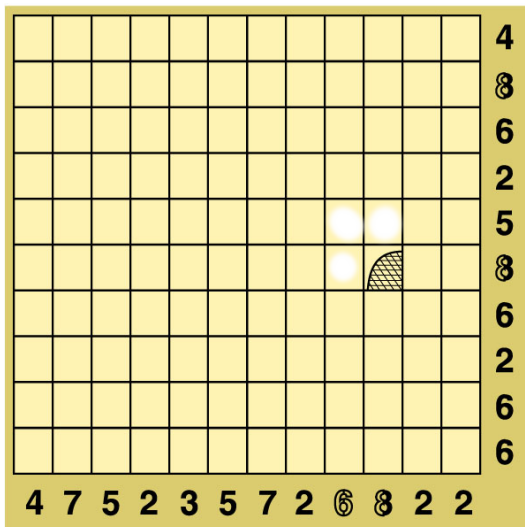
Zapewne pojawią się na liście P3...



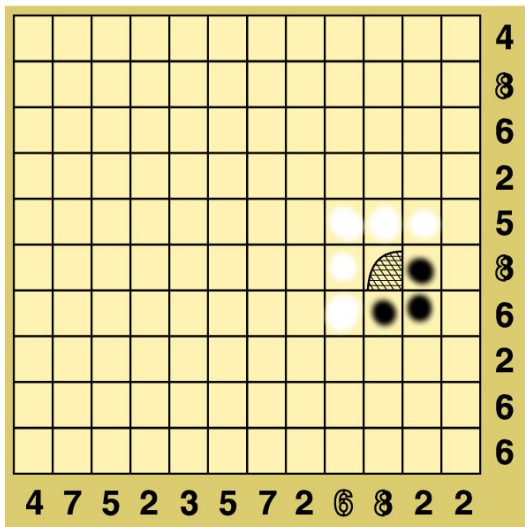
Zasady

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 .

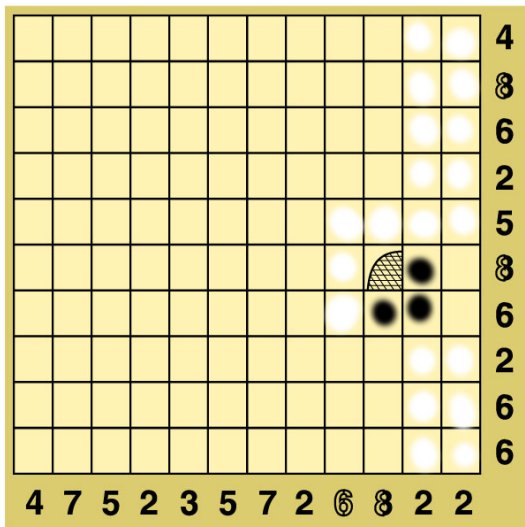
Burze: wnioskowanie



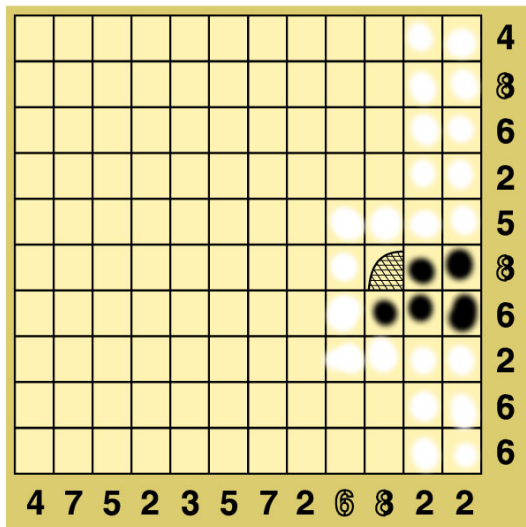
Burze: wnioskowanie



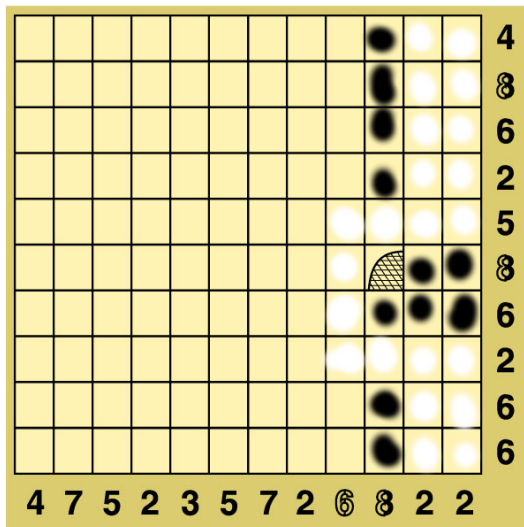
Burze: wnioskowanie



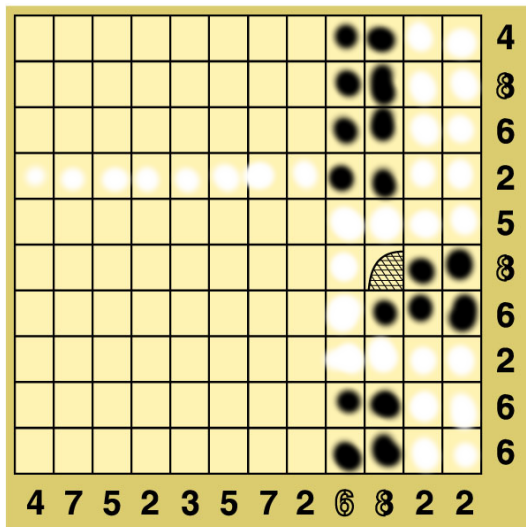
Burze: wnioskowanie



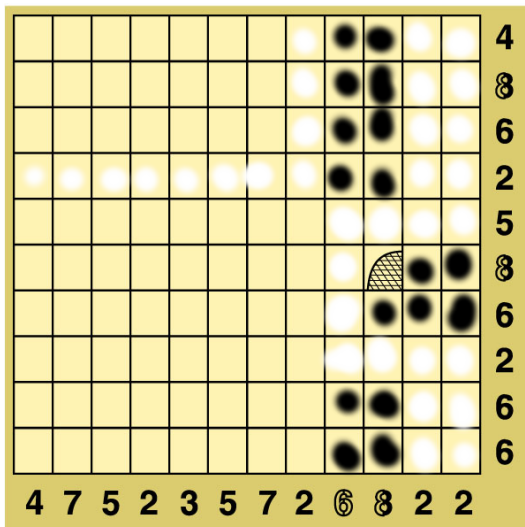
Burze: wnioskowanie



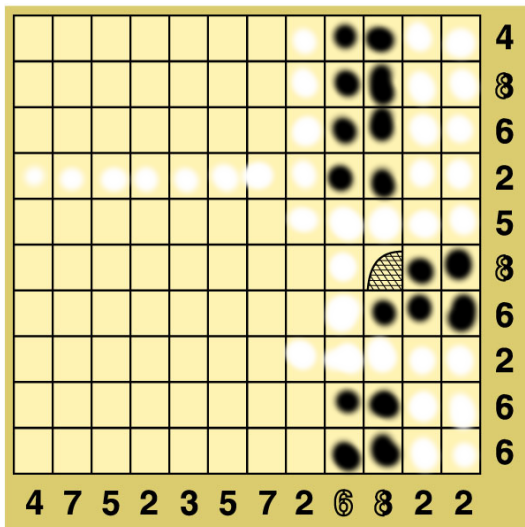
Burze: wnioskowanie



Burze: wnioskowanie



Burze: wnioskowanie



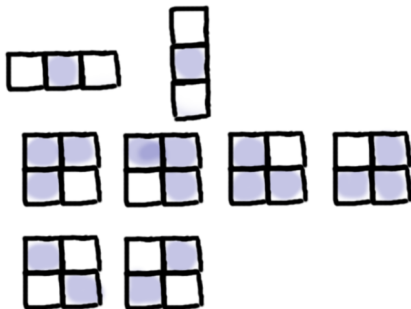
- Strategia 1: jak obrazki logiczne, wnioskowanie + ew. backtracking
- Strategia 2: wykorzystujemy SWI-Prolog

- Zmienne, dziedziny: piksele, 0..1
- Radary: $b_1 + b_2 + \dots + b_n = K$
- 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



Pytanie

Jak wyrazić to językiem relacji arytmetycznych?

Mamy zmienne A , B , C

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

Naturalne sformułowanie

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

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

- Inny przykład: $A \#<==> (B \#> C)$ (nawias dla czytelności)
- Naturalna propagacja:
 - Ustalenie A dorzuca więz
 - Jak wiemy, czy prawdziwy jest $B \#> C$, to znamy wartość A

(więcej przykładów: zob. *SWI Prolog Reification predicates*)

Więź uniwersalny w SWI-Prologu

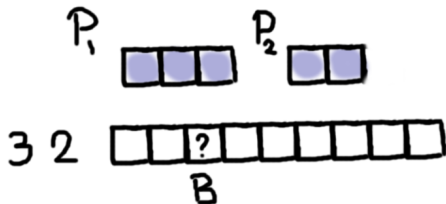
tuples_in

Wymieniamy explicite krotki wartości, jakie może przyjmować krotka zmiennych

Uwaga

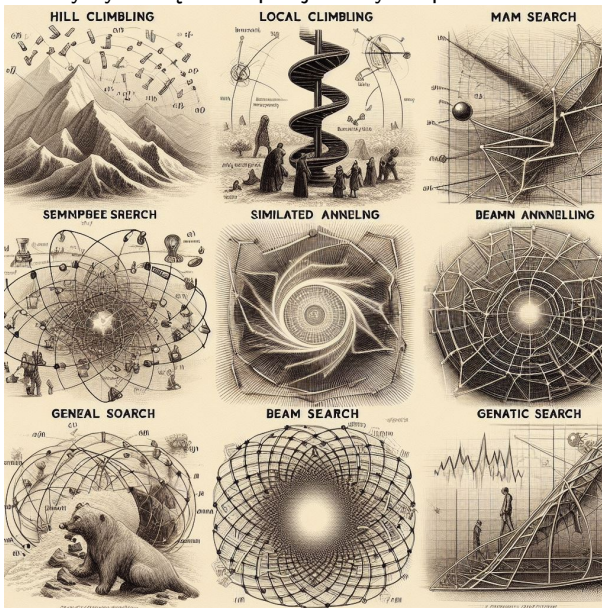
Zauważmy, że ten więź pasuje do lokalnych warunków dla burz, na przykład dla prostokątów 3×1 :

```
tuples_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]] )
```



- Użycie zmiennych P_1 i P_2 określających położenie bloku pozwala zmniejszyć dziedziny ($|P_1| + |P_2|$ zamiast $|P_1| \times |P_2|$) (mniejsze zużycie pamięci, niezmnieszona liczba kombinacji)
- Zmienna B ma wartość logiczną:
3 jest przykryte przez blok rozpoczynający się w P_1 lub przez blok rozpoczynający się w P_2

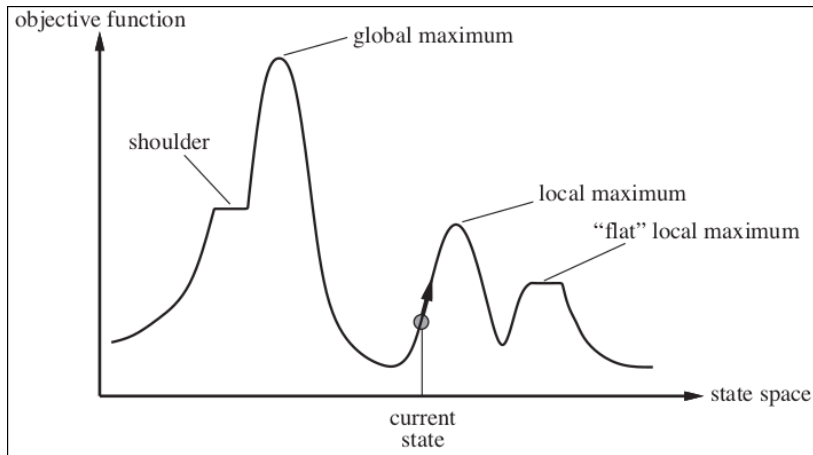
Na tym skończymy o wieżach i przejdziemy do przeszukiwania lokalnego



Przeszukiwania lokalne (ogólnie)

- Powiemy sobie o paru ideach związanych z przeszukiwaniem lokalnym.
- Można je wykorzystywać w zadaniach więzowych (MinConflicts z poprzedniego wykładu), ale nie tylko.

Krajobraz przeszukiwania lokalnego



Czym może być funkcja, którą minimalizujemy?

1. Liczbą **niespełnionych** więzów.

Czym może być funkcja, którą minimalizujemy?

- 1. Liczbą **niespełnionych** więzów.
- 2. **Wagą** niespełnionych więzów. Porównaj więzy:
 - 1. Nauczyciel ma tylko z jedną klasą lekcje na raz
 - 2. nikt nie ma dwóch biologii jednego dnia.

Czym może być funkcja, którą minimalizujemy?

1. Liczbą **niespełnionych** więzów.
2. **Wagą** niespełnionych więzów. Porównaj więzy:
 1. Nauczyciel ma tylko z jedną klasą lekcje na raz
 2. nikt nie ma dwóch biologii jednego dnia.
3. Czymś niezwiązanym bezpośrednio z więzami
 - produktywnością zespołu robotników (maksymalizemy, nie minimalizujemy!)
 - zadowoleniem gości weselnych z towarzystwa przy stolikach,
 - potencjalnym zyskiem sklepu,
 - dopasowaniem modelu do danych uczących

Czym może być funkcja, którą minimalizujemy?

1. Liczbą **niespełnionych** więzów.
2. **Wagą** niespełnionych więzów. Porównaj więzy:
 1. Nauczyciel ma tylko z jedną klasą lekcje na raz
 2. nikt nie ma dwóch biologii jednego dnia.
3. Czymś niezwiązanym bezpośrednio z więzami
 - produktywnością zespołu robotników (maksymalizemy, nie minimalizujemy!)
 - zadowoleniem gości weselnych z towarzystwa przy stolikach,
 - potencjalnym zyskiem sklepu,
 - **dopasowaniem modelu do danych uczących**

Czym może być funkcja, którą maksymalizujemy?

Uwaga

Ważna część uczenia maszynowego dotyczy **maksymalizacji dopasowania do danych uczących**

Hill climbing jest chyba najbardziej naturalnym algorytmem inspirowanym poprzednim rysunkiem.

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

Problem

Oczywiście możemy utknąć w lokalnym maksimum.

Hill climbing z losowymi restartami

Uwaga

Możemy podjąć dwa działania, oba testowaliśmy w obrazkach logicznych:

1. Dorzucać ruchy niekoniecznie poprawiające (losowe, ruchy **w bok**)
2. Gdy nie osiągamy rozwiązania przez dłuższy czas rozpoczynamy od początku.

Hill climbing + random restarts (w trywialny sposób) jest algorytmem zupełnym z p-stwem 1 (bo „kiedyś” wylosujemy układ startowy)

Inne warianty Hill climbing

- a) **Stochastic hill climbing** – wybieramy losowo ruchy w górę (p-stwo stałe, albo zależne od wielkości skoku).
- b) **First choice hill climbing** – losujemy następnika tak długo, aż będzie on ruchem w górę
 - dobre, jeżeli następników jest bardzo dużo

Uwaga

Idee z tego i kolejnych algorytmów można dowolnie mieszać – na pewno coś wyjdzie!

- Motywacja fizyczna: ustalanie struktury krystalicznej metalu.
- Jeżeli będziemy ochładzać powoli, to metal będzie silniejszy (bliżej globalnego minimum energetycznego).
- **Symulowane wyżarzanie** – próba oddania tej idei w algorytmie.

Algorytm

Symulujemy opadającą temperaturę, prawdopodobieństwo ruchu chaotycznego zależy **malejąco** od temperatury.

Symulowane wyżarzanie (2)

- Przykładowa implementacja bazuje na **first choice hill climbing**.
- Jak wylosowany ruch (**r**) jest poprawiający (czyli $\Delta F > 0$), to go wykonujemy (maksymalizacja F).
- W przeciwnym przypadku wykonujemy ruch **r** z p-stwem $p = e^{\frac{\Delta F}{T}}$
- Pilnujemy, żeby T zmniejszało się w trakcie działania (i było cały czas dodatnie)

Komentarze do wzoru

- $\Delta F \leq 0$, $T > 0$, czyli $0 \leq p \leq 1$.
- Im większe pogorszenie, tym mniejsze p-stwo
- Im większa temperatura, tym większe p-stwo.

Problem

Być może płaskie maksimum lokalne.

Rozwiązanie

Dodajemy pamięć algorytmowi, zabraniamy powtarzania ostatnio odwiedzanych stanów.

Local beam search

- Zamiast pamiętać pojedynczy stan, pamiętamy ich k (wiązkę).
- Generujemy następniki dla każdego z k stanów.
- Pozostawiamy k liderów.

Uwaga 1

To nie to samo co k równoległych wątków hill-climbing (bo uwaga algorytmu może przerzucać się do bardziej obiecujących kawałków przestrzeni)

Uwaga 2

Beam search jest bardzo popularnym algorytmem w różnych zadaniach wykorzystujących sieci neuronowe do modelowania sekwencji (np. tłumaczenie maszynowe).