論理と計算

第９回
論理プログラム：発展

担当：尾崎　知伸
ozaki.tomonobu@nihon-u.ac.jp

講義予定　※一部変更（前倒し）になる可能性があります

09/22	01. オリエンテーション と 論理を用いた問題解決の概要
09/29	02. 命題論理：構文・意味・解釈
10/06	03. 命題論理：推論
10/13	04. 命題論理：充足可能性問題
10/20	05. 命題論理：振り返りと演習（課題学習）
10/27	06. 述語論理：構文・意味・解釈
11/03	07. 述語論理：推論　※文化の日，文理学部授業日
11/10	08. 述語論理：論理プログラムの基礎
11/17	09. 述語論理：論理プログラムの発展
11/24	10. 述語論理：振り返りと演習（課題学習）
12/01	11. 高次推論：発想推論
12/08	12. 高次推論：帰納推論の基礎
12/15	13. 高次推論：帰納推論の発展
12/22	14. 高次推論：振り返りと演習（課題学習）
01/19	15. まとめと発展的話題

目次：今回の授業の内容

- 標準論理プログラムに対する安定モデルの導出アルゴリズム

- 解集合プログラミング
  - 一貫性制約 と その標準論理プログラムへの変換
  - 選択ルール と その標準論理プログラムへの変換
  - 基数制約 とその標準論理プログラムへの変換
  - 条件付きリテラル
  - 短縮表記
  - 算術計算
  - 最適化

- 例題
  - グラフの頂点彩色（SATとの違いを体験しよう）
  - N人の女王（SATとの違いを体験しよう）
  - 数独
  - クリーク抽出（安定モデル＝問題の解を意識しよう）
  - ハミルトン閉路（安定モデル＝問題の解を意識しよう）

安定モデルの導出アルゴリズム

標準論理プログラムの安定モデルの導出

- ナイーブな方法：エルブラン基底のべき集合を一つずつ調べる
  - エルブラン基底のサイズが|B|のとき，べき集合のサイズは$2^{|B|}$．非現実的

  →安定モデルとReductの最小モデルの関係を利用する

- 性質１：標準論理プログラムP とアトム集合S1，S2に対し，以下の関係が成立
  - $S1 \subseteq S2 \Rightarrow P^{S2} \subseteq P^{S1} \Rightarrow Cn(P^{S2}) \subseteq Cn(P^{S1})$
  - （S1がS2の部分集合であれば， P^{S2}はP^{S1}の部分集合となり，

    P^{S2} の最小モデルはP^{S1} の最小モデルの部分集合となる）

- 性質２：性質１より，Pの安定モデルXに対し，以下の関係1-3が成立
  1. $L \subseteq X \Rightarrow X \subseteq Cn(P^L)$
  2. $X \subseteq U \Rightarrow Cn(P^U) \subseteq X$
  3. $L \subseteq X \subseteq U \Rightarrow (L \cup Cn(P^U)) \subseteq X \subseteq (U \cap Cn(P^L))$
     - 意図：安定モデルXがLとUの間 ⇒ 安定モデルXは $(L \cup Cn(P^U))$ と $(U \cap Cn(P^L))$の間
     - なお$L \subseteq (L \cup Cn(P^U)) \subseteq X \subseteq (U \cap Cn(P^L)) \subseteq U$　なので，

       Xの範囲を（$L \subseteq X \subseteq U$から）さらに絞ることが可能

$X==Cn(P^X)$ に注意

※それぞれの性質を読み解き，また証明してみよう．

標準論理プログラムの安定モデルの導出

- 関数expand(P, L, U)：
  - 性質2-3を繰り返し適用してL,Uを更新：Pの安定モデルXの範囲を絞れるだけ絞る
    - $L \subseteq X \subseteq U \Rightarrow (L \cup \mathrm{Cn}(P^U)) \subseteq X \subseteq (U \cap \mathrm{Cn}(P^L))$
  - 結果がL == U なら，Lが安定モデル
  - 結果が$L \not\subseteq U$ なら，安定モデルは存在しない
  - 結果が上記以外なら，Xの範囲は $L \subseteq X \subseteq U$
    →$L \subseteq X \subseteq U$の範囲を探す＝expandの繰り返し
    ＝アルゴリズムsolve
- アルゴリズム solve(P, L, U)
  - expandを用いた安定モデル導出アルゴリズム
  - 最初の呼び出し：$L = \{\}$, U = P中のアトム集合
  - solve(P, $L \cup \{a\}$, U)： aを含む安定モデルを探す
  - solve(P, L, U-{a})：aを含まない安定モデルを探す

```
def expand( P, L, U )
   while( true ) :
      L_org := L
      U_org := U
      L := L ∪ Cn(P^U_org)
      U := U ∩ Cn(P^L_org)
      if L==L_org and U==U_org
      then break

return ( L, U )
```

```
def solve(P, L, U )
   ( L, U ) := expand(P, L, U)
   if L ⊄ U then return
   if L == U then print L
   else choose a ∈ U-L
      solve(P, L ∪ {a}, U)
      solve(P, L, U-{a} )
```

導出の例

a :- not b.
b :- not a.

solve(P, { }, {a, b})
  expand(P, L={ }, U={a, b})
    L_org = { }, U_org = {a,b}
    P^{L_org}={a. , b.}, Cn(P^{L_org}) = {a, b}
    P^{U_org}={ }, Cn(P^{U_org}) = { },
    L = { } ∪ { } = { } , U = {a,b} ∩ {a,b} = {a, b },

  L = { }, U = {a,b}

  aを選択①：L = {} ∪{a}, U={a,b}
  solve(P, {a}, {a,b})
    expand(P, L={a}, U={a,b})
      L_org = {a}, U_org = {a,b}
      P^{L_org} = {a. }, Cn(P^{L_org}) = {a}
      P^{U_org}={ }, Cn(P^{U_org}) = { }
      L={a} ∪ {} = {a}, U = {a,b} ∩ {a} = {a}

      L_org = {a}, U_org = {a}
      P^{L_org} = {a}, Cn(P^{L_org}) = {a}
      P^{U_org} = {a}, Cn(P^{U_org}) = {a}
      L = {a} ∪ {a} = {a} , U = {a} ∩ {a} = {a},

    L ={a}, U={a} よりprint { a }

  aを選択②：L = {}, U={a,b}-{a}={b}
  solve(P, {}, {b})
    expand(P, L={}, U={b})
      L_org = {}, U_org = {b}
      P^{L_org} = {a., b.}, Cn(P^{L_org}) = {a, b}
      P^{U_org}={b}, Cn(P^{U_org}) = {b}
      L={} ∪ {b} = {b}, U = {b} ∩ {a,b} = {b}

      L_org = {b}, U_org = {b}
      P^{L_org} = {b}, Cn(P^{L_org}) = {b}
      P^{U_org} = {b}, Cn(P^{U_org}) = {b}
      L = {b} ∪ {b} = {b} , U = {b} ∩ {b} = {b},

    L ={b}, U={b} print {b} よりprint { b }

導出の例

a :- not a.

solve(P, { }, {a})
  expand(P, L={ }, U={a})
    L_org = { }, U_org = {a}
    P^{L_org}={a}, Cn(P^{L_org}) = {a}
    P^{U_org}={ }, Cn(P^{U_org}) = { },
    L = { } ∪ { } = { } , U = {a} ∩ {a} = {a},

  L = { }, U = {a}

  aを選択①：L = {} ∪{a}, U={a}
  solve(P, {a}, {a})
    expand(P, L={a}, U={a})
      L_org = {a}, U_org = {a}
      P^{L_org}={}, Cn(P^{L_org}) = {}
      P^{U_org}={ }, Cn(P^{U_org}) = {},
      L = {a} ∪ { } = {a} , U = {a} ∩ {} = { }

      L_org = {a}, U_org = { }
      P^{L_org}={}, Cn(P^{L_org}) = {}
      P^{U_org}={a}, Cn(P^{U_org}) = {a},
      L = {a} ∪ {a} = {a} , U = {} ∩ {} = { }

    L={a}, U = { } return

  aを選択②：L = {} , U={a}-{a} = { }
  solve(P, {}, {})
    expand(P, L={}, U={})
      L_org = {}, U_org = {}
      P^{L_org}={a}, Cn(P^{L_org}) = {a}
      P^{U_org}={a}, Cn(P^{U_org}) = {a},
      L = {} ∪ {a} = {a} , U = {} ∩ {a} = { }

      L_org = {a}, U_org = { }
      P^{L_org}={}, Cn(P^{L_org}) = {}
      P^{U_org}={a}, Cn(P^{U_org}) = {a},
      L = {a} ∪ {a} = {a} , U = {} ∩ {} = { }

    L={a}, U = { } return

解集合プログラミング

解集合プログラミング（Answer set programming）

- Answer Set Programming（ASP）
  - ASP = Database + Logic Program + Knowledge Representation + SAT
  - 解集合プログラミング = データベース +論理プログラム + 知識表現 + 充足可能性問題
- 論理に基づくプログラミング
  - 入力：（関数フリーで安全な）一般拡張選言プログラム ＋α（数量を扱うための拡張など）
    - 関数フリー：項に関数記号を含まない
    - 安全：頭部リテラル中の変数 及び 本体部の負リテラル中の変数は，本体部の正リテラルにも現れる
      - 非安全な節の例：p(A, B, C) :- q(B), not r(C), not s(B,D). #A, C, Dが非安全
  - 出力：解集合（の集合）
- 基本構文の要素
  - 一般拡張選言プログラム
  - 一貫性制約（→標準論理プログラムへ変換）
  - 選択ルール（→標準論理プログラムへ変換）
  - 基数制約（→標準論理プログラムへ変換）
  - 条件付きリテラル
  - 短縮表記（→表記上の工夫）
  - 算術計算
  - 最適化

一貫性制約（integrity constraint）

- 以下の形式をしたルール（＝ヘッドが空のルール）を一貫性制約と呼ぶ
  - $\leftarrow A_{l+1}, \ldots, A_m,\ not\ A_{m+1},\ \ldots,\ not\ A_n \quad (A_1, \ldots, A_n$はアトム$)$

    - ボディが成り立つとヘッド＝矛盾が成り立つ →ボディが成り立ってはいけない
      - :- color(X, C1), color(X, C2), C1 != C2. （Xの色が，C1, C2の両方であってはいけない）
    - デフォルトの否定notと組合せ
      - :- not p.（pが成り立たないとNG → pが成り立たなければいけない）

    pを含む解集合が得られる（クエリの役割を果たす）

    - 注意：変数を含む制約には注意が必要
      - 変数を含むルール・制約は基礎化される
      - 基礎化された制約すべてを満たす必要がある（一つでも違反すると矛盾が導かれる）
        - クリークの例で確認します（後述）

- 標準論理プログラムへの変換
  - 変換前：　$\leftarrow A_{l+1}, \ldots, A_m,\ not\ A_{m+1},\ \ldots,\ not\ A_n$
  - 変換後：x $\leftarrow A_{l+1}, \ldots, A_m,\ not\ A_{m+1},\ \ldots,\ not\ A_n, not\ x$　（xは新たなアトム）

p ← not p. は安定モデルを持たない.
x ← y，not x は，yが真のときに機能し，モデルを破棄する

※前処理：標準論理プログラムへの変換
※後処理：モデル表示時に，新たに導入したアトムを除去（制約の場合は不要）

選択ルール（choice rule）

- 以下の形式をしたルール（{ }でヘッドを囲んだルール）を選択ルールと呼ぶ
  - $\{A_1; \ldots; A_m\} \leftarrow A_{m+1}, \ldots, A_n,\ not\ A_{n+1},\ \ldots,\ not\ A_o$　$(A_1, \ldots, A_o$はアトム$, 0 \leq m \leq \mathrm{n} \leq \mathrm{o})$
  - ボディが成り立つとき$\{A_1; \ldots; A_m\}$ の部分集合のうち少なくとも一つがモデルに含まれる
    - すなわち，$\{A_1; \ldots; A_m\}$ のうち 0 個以上のアトムがモデルに含まれる
    - 後ほど，サイズL以上U以下の部分集合が成り立つ

- 例：{ color(X,red); color(X,blue); color(X, green)} :- node(X).
  - ノードXの色は，赤，青，緑　（複数の色を持っていても構わない）

- 標準論理プログラムへの変換
  - 部分集合を考える⇒「それぞれのA_iが成り立つ，もしくは成り立たない」
  - 変換前：$\{A_1; \ldots; A_m\} \leftarrow A_{m+1}, \ldots, A_n,\ not\ A_{n+1},\ \ldots,\ not\ A_o$
  - 変換後：$A' \leftarrow A_{m+1}, \ldots, A_n,\ not\ A_{n+1},\ \ldots,\ not\ A_o$

    $A_1 \leftarrow A',\ not\ A'_1.$　　$A'_1 \leftarrow\ not\ A_1.$

    …

    $A_m \leftarrow A',\ not\ A'_m.$　　$A'_m \leftarrow\ not\ A_m.$

※ $A', A'_1 \cdots A'_m$は新たなアトム（ルール毎に準備する）

{ A ← not B.
 B ← not A. } は，2つの安定モデル{A}, {B}を持つ．すなわち{A,B}から一つを選択する
上記の変換では，各頭部アトムA_iに対し，条件A'の下で$\{A_i, A'_i\}$から一つを選択する

基数制約(1)（cardinality rules）

基数制約では，頭部，本体部それぞれに上限と下限を指定できます

- 以下の形式をしたルール（本体部を{ }で囲んだルール）を基数制約と呼ぶ
  - $A_0 \leftarrow \boldsymbol{l}\,\{A_1; \ldots; A_m;\ not\ A_{m+1};\ \ldots,; not\ A_n\}$　$(A_0, \ldots, A_n$はアトム$, 1 \leq m \leq \mathrm{n},\ l$ は非負整数$)$
  - l個以上の本体部リテラルが成り立つとき，頭部A_0が成り立つ
- 例：pass(c42) :- 2 { pass(a1); pass(a2); pass(a3) }.
  - 課題 a1, a2, a3のうち2つ以上を満たすと，コース c42 に合格する
- $A_0 \leftarrow A_1, \ldots, A_m,\ not\ A_{m+1},\ \ldots,\ not\ A_n.$ は$A_0 \leftarrow n\,\{A_1; \ldots; A_m;\ not\ A_{m+1};\ \ldots; not\ A_n\}$と表現できる

同様に$\{A_1; \ldots; A_m\} \leftarrow A_{m+1}; \ldots; A_n;\ not\ A_{n+1};\ \ldots;\ not\ A_o$
→ $0\{A_1; \ldots; A_m\}m \leftarrow A_{m+1}; \ldots; A_n;\ not\ A_{n+1};\ \ldots;\ not\ A_o$

- 標準論理プログラムへの変換
  - ctrl(i, j) : i番目以降のアトムのうち，少なくともj個が成立する
  - 変換前：$A_0 \leftarrow l\,\{A_1; \ldots; A_m;\ not\ A_{m+1};\ \ldots;\ not\ A_n\}$
  - 変換後：$A_0 \leftarrow ctrl(1, l)$ [本体部の1番目以降のアトムのうち，少なくともl個が成立するならA_0]

$ctrl(n+1, 0).$　[n+1番目以降では（本体部リテラルがないので）0個のアトムが成立]

$ctrl(i, k+1) \leftarrow ctrl(i+1, k), A_i$　$(1 \leq i \leq m,\ 0 \leq k \leq l)$

$ctrl(i, k) \leftarrow ctrl(i+1, k)$　$(1 \leq i \leq m,\ 0 \leq k \leq l)$

$ctrl(j, k+1) \leftarrow ctrl(j+1, k), not\ A_j$　$(\mathrm{m}+1 \leq j \leq n,\ 0 \leq k \leq l)$

$ctrl(j, k) \leftarrow ctrl(j+1, k)$　$(\mathrm{m}+1 \leq j \leq n,\ 0 \leq k \leq l)$

i+1番目以降で最低k個成り立つ(ctrl(i+1, k)) かつ
i番目が成り立つ(Ai) ならば
i番目以降では最低k+1個成り立つ

j+1番目以降で最低k個成り立つ(ctrl(j+1, k))なら
（j番目の成否に関係なく）
j番目以降では最低k個成り立つ

基数制約

変換前：c :- 1 { a; b }. (l = 1, m =2, n = 2)
変換後：

```
c :- ctrl(1,1).
ctrl(3,0).

# k=0, i =1
ctrl(1,1) :- ctrl(2,0), a.
ctrl(1,0) :- ctrl(2,0).

# k=1, i =1
ctrl(1,2) :- ctrl(2,1), a.
ctrl(1,1) :- ctrl(2,1).

# k=0, i =2
ctrl(2,1) :- ctrl(3,0), b.
ctrl(2,0) :- ctrl(3,0).

# k=0, i =2
ctrl(2,2) :- ctrl(3,1), b.
ctrl(2,1) :- ctrl(3,1).
```

基数制約(2)

- 本体部リテラルに対する上限
  - $A_0 \leftarrow \boldsymbol{l}\{A_1; \ldots; A_m;\ not\ A_{m+1};\ \ldots;\ not\ A_n\}\boldsymbol{u}$　$(A_0, \ldots, A_n$はアトム$, 1 \leq m \leq \mathrm{n},\ l \leq u$は非負整数$)$
  - l 個以上 u 個以下の本体部リテラルが成り立つとき，頭部A_0が成り立つ
- 標準論理プログラムへの変換
  - 変換前：$A_0 \leftarrow l\{A_1; \ldots; A_m;\ not\ A_{m+1};\ \ldots;\ not\ A_n\}u$
  - 変換後：$A_0 \leftarrow B, not\ C$

    $B \leftarrow l\{A_1; \ldots; A_m;\ not\ A_{m+1};\ \ldots; not\ A_n\}$

    $C \leftarrow u+1\{A_1; \ldots; A_m;\ not\ A_{m+1};\ \ldots; not\ A_n\}$

※ B, C新たなアトム
（ルール毎に準備する）

- 頭部に対する制約
  - $\boldsymbol{l}\{A_1; \ldots; A_m;\ not\ A_{m+1};\ \ldots; not\ A_n\}\boldsymbol{u} \leftarrow A_{n+1}, \ldots, A_o,\ not\ A_{o+1},\ \ldots,\ not\ A_p$

    $(A_1, \ldots, p$はアトム$, 0 \leq m \leq \mathrm{n} \leq \mathrm{o} \leq \mathrm{p},\quad l \leq u$は非負整数$)$
  - 本体部が成り立つとき，l 個以上 u 個以下の頭部リテラルが成り立つ
  - 1{ color(X,red); color(X,blue); color(X, green)}1 :- node(X).　ノードの色は赤青緑のどれか一つ
- 標準論理プログラムへの変換
  - 変換前：$l\{A_1; \ldots; A_m;\ not\ A_{m+1};\ \ldots; not\ A_n\}u \leftarrow A_{n+1}, \ldots, A_o,\ not\ A_{o+1},\ \ldots,\ not\ A_p$
  - 変換後：$B \leftarrow A_{n+1}, \ldots, A_o,\ not\ A_{o+1},\ \ldots,\ not\ A_p$

    $\{A_1; \ldots; A_m\} \leftarrow B$

    $C \leftarrow l\{A_1; \ldots; A_m;\ not\ A_{m+1};\ \ldots;\ not\ A_n\}u$

    $\leftarrow B, not\ C.$

本体部が成り立ち（B)かつ頭部の条件を満たさない（not C)はNG

条件付きリテラル（conditional literals）

- 以下の形式のリテラルを条件付きリテラルと呼ぶ
  - $l : l_1, \ldots, l_n$ ($l, l_1 \ldots l_n$はリテラル，$0 \leq n$)
  - リテラルの横に，条件l_1～l_nを：でつなげたもの
  - 集合 $\{ l \mid l_1, \ldots, l_n \}$ の要素の連言を表す
- 例：{ color(blue). color(yellow). color(red). }のとき

  color(v1, C) : color(C). → color(v1,blue); color(v1,yellow); color(v1,red).　#ヘッドは選言

  1 { color(v1, C): color(C) } 1 :- vertex(v1). →

  　　1{ color(v1,blue); color(v1,yellow); color(v1,red) } 1 :- vertex(v1).

  :- color(v1,C) : color(C)　→ :- color(v1,blue), color(v1,yellow), color(v1,red). #ボディは連言

  ※「v1が何かの色ならNG」ではなく「v1がblue,yellow,redを持ったらNG」となるので注意

node(x).　color(a).　color(b).
1 { n(N,C):color(C) } 1:- node(N).
→
1 { n(N,a); n(N,b) } 1:- node(N).
→
1 { n(x,a); n(x,b) } 1:- node(x).

→安定モデル { n(x,a) }, {n(x,b) }

node(x).　color(a).　color(b).
1 { n(N,C) } 1:- node(N), color(C).
→
1 { n(x,a) } 1:- node(x), color(a).
1 { n(x,b) } 1:- node(x), color(b).

→ 安定モデル {n(x,a), n(x,b) }

短縮表記・整数演算

- 短縮表記：述語内の「項」に対するセミコロン，整数のインターバル
  - セミコロンまでを１つ単位とし，それぞれを述語名で囲む
    - p(a ; b). … p(a). p(b).
    - p(a, b ; x, y). … p(a, b). p(x, y).
    - p(a, b ; c ; d, e, f). … p(a, b). p(c). p(d, e, f).
  - ドットx2で，整数範囲を示す
    - p(1..3). … p(1). p(2). p(3).
    - p(1..2, 5..6). … p(1,5). p(1,6). p(2,5). p(2, 6).
  - 両者を組み合わせることも可能
    - p(1 .. 2 ; a ; b ; 10, 11). … p(1). p(2). p(a). p(b). p(10, 11).
- 比較演算：=，!=，<，>，<=，>=
- 整数演算：+，-，*，/，**
  - 整数に対する演算

```
p(1 .. 3).
q(Y):- p(X), Y = X * 2.
r(X,Y):- p(X), q(Y), X > Y.
s(X+Y):- r(X,Y).
t(X/Y):- r(X,Y).
```

→ 解集合

{ p(1), p(2), p(3), q(2), q(4), q(6), r(3,2), s(5), t(1) }

最適化

※弱い制約は複数準備することができる

- やりたいこと：安定モデルに対する順序付け
- 弱い制約（weak constraint）：なるべく満たしてはいけない条件
  - cf. 一貫性制約：絶対に満たしてはいけない条件
- 以下の形式をしたルール（＝ヘッドが空のルール）を一貫性制約と呼ぶ
  - $:\sim A_{l+1}, \ldots, A_m,\ not\ A_{m+1},\ \ldots,\ not\ A_n.[\mathrm{w}, t_1, \ldots, t_s]$ ($A_1, \ldots, A_n$はアトム, $t_1, \ldots, t_s$は項)
  - t1…ts に関して，ルール（制約）が成立するとき，コスト（ペナルティ）wが発生する
  - モデルに対するペナルティは，成立する弱い制約のコストの総和（小さい方が嬉しい）

```
hotel(1..3). % 3つのホテルがある
cost(X, X*10):- hotel(X). %各ホテルのコストは ID * 10
noisy(X):- hotel(X), X != 3.  %3番目のホテル以外はnoisy

2{ select(X) : hotel(X) }2.  %ホテルを2つ選ぶ

:~ select(X),noisy(X),cost(X,C). [ C,X ]  % noisyなホテルXにペナルティCを設定
% Xに対してルールが成立するとコストCがかかる
% 変数を含むルールは，（複数のルールに）基礎化されることに注意
```

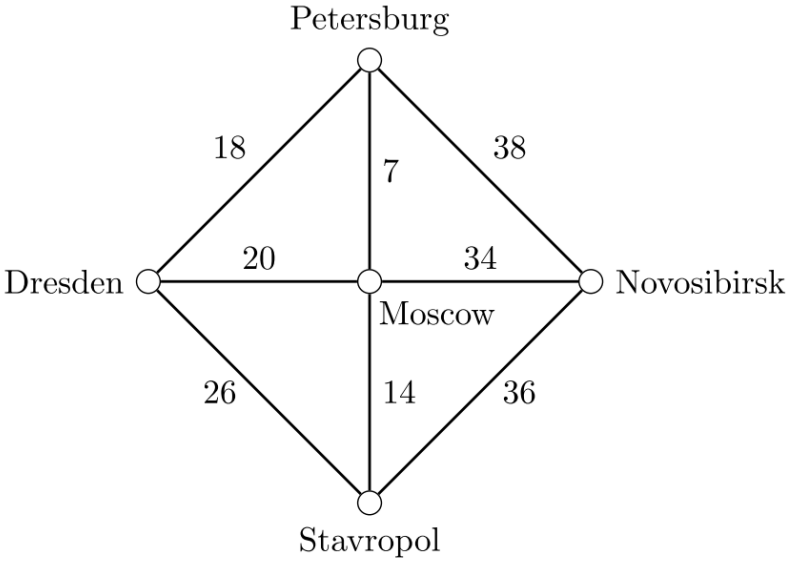
弱い制約がない場合は，以下の3つの安定モデルが得られる．このうち，コスト最小のものが出力される
{ hotel(1) hotel(2) hotel(3) cost(1,10) cost(2,20) cost(3,30) noisy(1) noisy(2) select(2) select(3) }：コスト20
{ hotel(1) hotel(2) hotel(3) cost(1,10) cost(2,20) cost(3,30) noisy(1) noisy(2) select(1) select(2) }：コスト30
{ hotel(1) hotel(2) hotel(3) cost(1,10) cost(2,20) cost(3,30) noisy(1) noisy(2) select(1) select(3) }：コスト10

※clingoではコスト最小のモデルを導出する

最適化

重みの和が最小になるように，隣接する2辺を選択する

```
edge(d,p,18). edge(d,m,20).edge(d,s,26). edge(p,m,7).
edge(p,n,38). edge(m,s,14).edge(m,n,34). edge(s,n,36).

2 { select(X,Y) : edge(X,Y) } 2. %辺を2つ選択
node(X):- select(X,Y). %選択された辺の頂点を導出
node(Y):- select(X,Y). %選択された辺の頂点を導出
3 { node(X):select(X,Y) ; node(Y):select(X,Y) } 3.  %導出ノード数は3

:~ select(X,Y), edge(X,Y,C). [C,X,Y] %導出辺に対するコスト

% select/2とnode/1のみを表示する
#show select/2.
#show node/1.
```



例題

解集合プログラミングシステム clingo

ルール：{ p ; not p.　}
clingo：　p ; not p .

ルール：{ p. ¬ p }
clingo：　p.
　　-p.

ルール：{ p ← not q.
　　q ← not p. }
clingo:　p :- not q.
　　q :- not p.

- 表記
  - " ; "　　選言はセミコロン
  - " , "　　連言はカンマ
  - " :- "　　←はコロンマイナス（メダカマークですね）
  - " not "　デフォルトの否定はnot
  - " - "　　論理否定（負リテラル）はマイナス

  - 注意１：ボディが空の場合は，:- は記述しない.
  - 注意２：各ルールは，"." で終わる.

- 使い方　$ clingo　0　入力ファイル
  - clasp と同じ使い方 (0 はすべてのモデルを表示するためのオプション)
  - Clingoは，基礎化器（grounder）gringoを用いてプログラムを基礎化&変換し，claspを用いて解集合を求めている

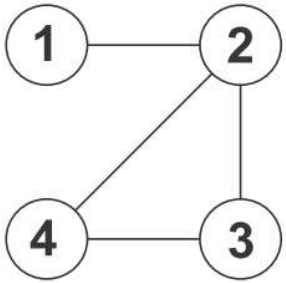
- やってみよう1 : clingoを用いて資料中の「安定モデルの例・解集合の例」を計算してみよう
- やってみよう2：以下のプログラムの解集合を計算し，だれが飛ぶのか確認しよう

fly(X) ← bird(X), *not* abnormal(X).
abnormal(X) ← penguin(X).
bird(john).
bird(tweety).
penguin(tweety).

グラフの頂点彩色

- 隣接する頂点同士が同じ色にならないように全頂点を彩色する
  - どんな（平面）グラフも，４色で塗り分けることができる
- プログラム
  - color(N)：N は色である．　　node_color(X, C)：ノードXの色はCである．
  - node(X)：Xはノードである．edge(X, Y)：XとYの間に辺がある．
  - 条件１：各ノードの色は一つ（チョイスルールで表現）
  - 条件２：隣接のノードが同じ色であってはいけない（一貫性制約で表現）

```
color(1..3).  % 3colors

1{ node_color(X, C) : color(C) }1 :- node(X).
:- edge(X,Y), node_color(X,C), node_color(Y,C).

%%%%%%%%%%%%%%%%%%%%%
node(1..4).
edge(X,Y):- edge(Y,X).
edge(1,2).
edge(2,4).
edge(2,3).
edge(3,4).

#show node_color/2.
```



クリーク抽出

- グラフ中でサイズNの全結合グラフを抽出する
  - node(X) : Xはノードである．edge(X, Y) : XとYの間に辺がある．
  - in_clique(X) : Xがクリークに含まれる．
  - クリークに含まれる（異なる）2頂点間には辺がある．
    - 辺がない場合は矛盾
    - 下記で，X != Y　を忘れると，上手く動作しない
      - XとYが同じ定数に基礎化された場合，自己辺がないために，制約違反になる

```
3 { in_clique(X) : node(X) } . %今回は頂点数3のクリークを考えている
:- in_clique(X), in_clique(Y), X != Y, not edge(X, Y).

%%%
node(1..4).
edge(X,Y):- edge(Y,X).
edge(1,2).
edge(2,4).
edge(2,3).
edge(3,4).

#show in_clique/1.
```

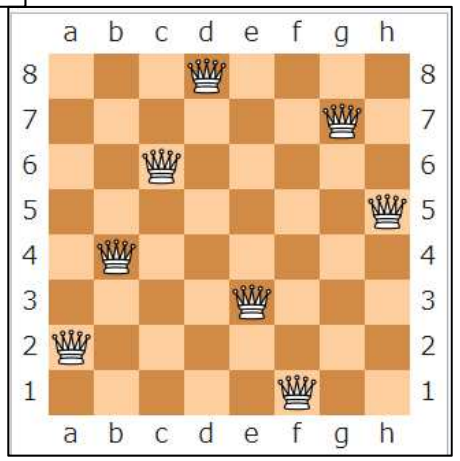
N人の女王

- n 個のクイーンを，n × n のチェス盤に，お互いに取られないように並べる
  - queen(X, Y) : セルX, Y にクイーンがいる
  - 制約１：同じ行(X)，違う列(Y1, Y2)にクイーンを配置してはいけない
  - 制約２：同じ列(Y)，違う行(X1, X2)にクイーンを配置してはいけない
  - 制約３：異なる行（X1,X2），異なる列(Y1,Y2)でも，斜めに2つのクイーンを配置してはいけない

```
n { queen(1..n, 1..n) } n.

:- queen(X, Y1), queen(X, Y2), Y1 != Y2.
:- queen(X1, Y), queen(X2, Y), X1 != X2.
:- queen(X, Y), queen(X1, Y1), X != X1, Y != Y1, |X-X1| = |Y-Y1|.
```

実行時に-cオプションを用いて定数nを指定する．

```
% clingo –c n=8 queen.lp
```



https://ja.wikipedia.org/wiki/エイト・クイーン より

数独

- 9つある各行，9つある各列，9つある3x3の各ブロックに1..9の数が1回ずつ埋める

  number(X) : xは数　　row(X): Xは行　　col(Y) : Yは列　　cell(X, Y, N) : セルX, Yの数はN

  square(S, X, Y) : セルX, Y はブロックSに属する　in_square(S,N) : ブロックSは数Nを含む

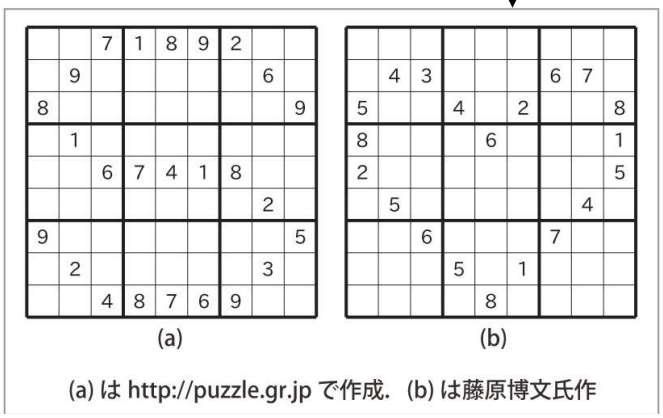
  制約１：同じ行(X)，違う列(Y1, Y2)にあるセルの数(N)が同じではいけない

  制約２：同じ列(Y)，違う行(X1, X2)にあるセルの数(N)が同じではいけない

  制約３：ブロックSは数Nを含んでいないといけない

```
number(1..9).
row(0..8).
col(0..8).

square(s0, 0..2, 0..2). square(s1, 0..2, 3..5). square(s2, 0..2, 6..8).
square(s3, 3..5, 0..2). square(s4, 3..5, 3..5). square(s5, 3..5, 6..8).
square(s6, 6..8, 0..2). square(s7, 6..8, 3..5). square(s8, 6..8, 6..8).

1 {cell(X,Y,N) : number(N)} 1 :- row(X), col(Y).

:- cell(X, Y1, N), cell(X, Y2, N), Y1 != Y2.
:- cell(X1, Y, N), cell(X2, Y, N), X1 != X2.

in_square(S, N):- square(S, X, Y), cell(X,Y,N).
:- number(N), square(S, X, Y), not in_square(S, N).

#show cell/3.
```

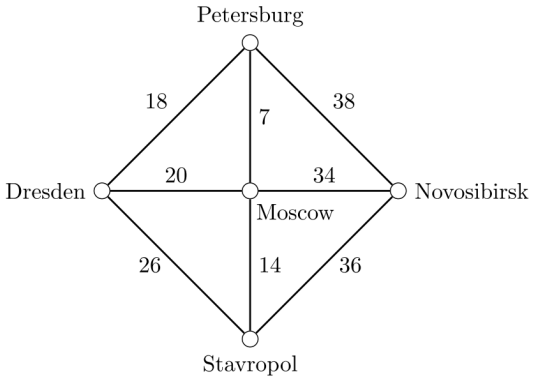
部分的に数が与えられている場合は，
与えられている cell(X,Y,N)も記述する



田村他「SATとパズル」情報処理Vol.57, No.8, pp.710-715,2016.　より

ハミルトン閉路

- 起点へ戻る一筆書き
  - node(X) : Xはノードである．edge(X, Y) : XとYの間に辺がある．
  - cycle(X,Y) : XからYへ移動する
  - reachable(X) : ノードXは到達可能
    - 起点 sからcycleでつながっているYは到達可能
    - 到達可能なXから，cycleでつながっているYは到達可能
  - 制約１：Xから移動できる場所はちょうど一ヶ所（出次数=1）
  - 制約２：Xへ移動できる場所はちょうど一ヶ所（入次数=1）
  - 制約３：すべてのノードは到達可能でなければいけない．




```
1 { cycle(X,Y) : edge(X, Y) } 1 :- node(X).
1 { cycle(X,Y) : edge(X, Y) } 1 :- node(Y).

reachable(Y):- cycle(s, Y).
reachable(Y):- reachable(X), cycle(X,Y).
:- node(X), not reachable(X).
#show cycle/2.
```

実行時に-cオプションを用いて定数sを指定する．
% clingo –c s=dresden hamilton.lp

```
node(dresden).
node(petersburg).
node(novosibirsk).
node(stavropol).
node(moscow).

edge(stavropol, novosibirsk).
edge(dresden, moscow).
edge(moscow, petersburg).
edge(dresden, petersburg).
edge(moscow, stavropol).
edge(dresden, stavropol).
edge(moscow, novosibirsk).
edge(petersburg, novosibirsk).
edge(Y,X) :- edge(X, Y ).
```