知能情報処理　制約充足(2)
制約をみたす意志決定をするエージェント

局所整合と局所探索
(Local Consistency and Local Search)

- 局所整合アルゴリズム
- 局所探索アルゴリズム

制約充足問題(CSP)とは(復習)

問題

- 変数(variable)の集合　X
- 各変数の領域(domain)　D
- 変数間の制約(constraint)の集合　C

$x_1\ x_2\ \ldots\ x_n$

$D_1\ D_2\ \ldots\ D_n$

$C_{xy} = \{(a,b), (c,d), \ldots\}$

変数 x-y 間で許される値の組

解

- すべての制約を満たすような変数への値の割当て

$x_1=a_1\ x_2=a_2\ \ldots\ x_n=a_n$

前回の復習

制約充足問題(CSP)は，変数の集合，各変数の領域，変数間の制約の集合を具体的に示すことにより定義される．CSPの解は，すべての制約を満たすような変数への値の割当てである．

制約充足問題の例（復習）

- *n* クイーン問題 (n queens problem)
- クロスワードパズル(crossword puzzles)
- グラフ彩色問題 (graph coloring)
- 線画解釈 (interpretation of line drawings)
- レイアウト (layout)
- スケジューリング (scheduling)

前回の復習

CSPの例として，このようなものがある．

局所整合と局所探索

制約をすべて（**大域的**に）満たすのは困難
global

→ **局所的**に満たしながら大域に拡大する
local

- **局所整合**アルゴリズム
  local consistency
  ⇒ 局所的に**バックトラック不要**になるように不適切な値を事前に削除
  backtrack-free

- **局所探索**アルゴリズム
  local search
  ⇒ 局所的に制約違反を**修復**し，**バックトラックをしない**
  repair

一般に，制約をすべて（大域的に）満たすのは計算量的に困難（NP困難：最悪のケースは指数オーダーの計算時間となる）なので，何らかの意味で局所的に制約を満たしながら，それを大域的に拡大する方法をとる．

その方法は，基本的につぎの2つに大別できる．

局所整合アルゴリズムは，制約グラフの局所的な制約違反を事前に解消するために，変数の領域から明らかに不適切な値を削除して，局所的にバックトラック不要になるように問題を変換する．

局所探索アルゴリズムは，局所的に制約違反を修復し，バックトラックを（する必要があっても）しないで，すべての制約を満たすことを目指す．その結果，解があってもそれを発見する保証（**完全性**）は失われる．しかし，たとえ理論的には完全性があっても，事実上は実用的な時間内で解を発見できないような非常に難しい問題において，（いちかばちか）実用的な時間内で解を発見する可能性を追求する．

局所整合アルゴリズム
(Local Consistency Algorithms)

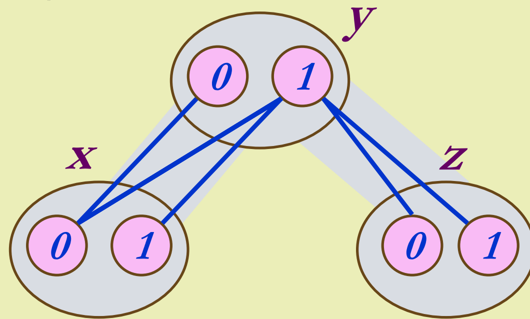
1. **アーク整合**
   arc consistency
2. **制約伝播**
   constraint propagation

制約ネットワークと制約グラフ

constraint network　　constraint graph

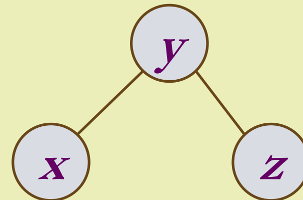
$$C_{xy} = \{(0,0), (0,1), (1,1)\} \quad C_{yz} = \{(1,0), (1,1)\}$$

制約ネットワーク



許された値の組を
辺で結ぶ

制約グラフ



制約のある変数ノード
を辺で結ぶ

X,D,Cからなる制約充足問題(CSP)をこのスライドのような図で表現した場合に，それを**制約ネットワーク**と呼ぶことにしよう．すなわち，各変数の取りうる値をグラフのノードとし，変数はその領域の要素全体を包含するスーパーノードとして表現する．さらに，制約の記述によって，同時に値を取りうる要素どうしはグラフの辺で結ぶこととする．

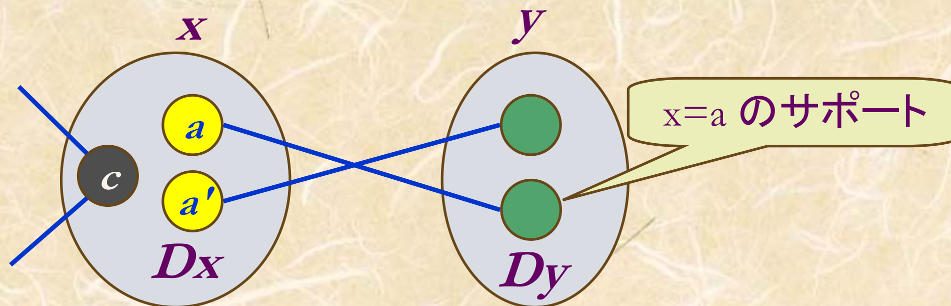
制約グラフとは，変数をノードとし，制約のある変数ノードを辺で結ぶことによって得られるグラフである．制約ネットワークよりも大まかにCSP全体の構造を把握するのに役立つ．

1. アーク整合(1)

arc consistency

■ 値 x=a は y にサポートをもつ $=\left(\exists b\in D_y\right)(a,b)\in C_{xy}$

support



x=c は y にサポートを<u>もたない</u>

⇨ Dx から値c を削除 (**アーク整合アルゴリズム**)

変数 x,y 間に制約 Cxy があるものとする. x への値の割当て x=a に対して, y=b が**整合する**(制約違反とならない)とき, y=b を「y における x=a の**サポート**」と呼ぶ. そのような y=b が y の領域に存在するとき, 「値 x=a は y にサポートをもつ」という.

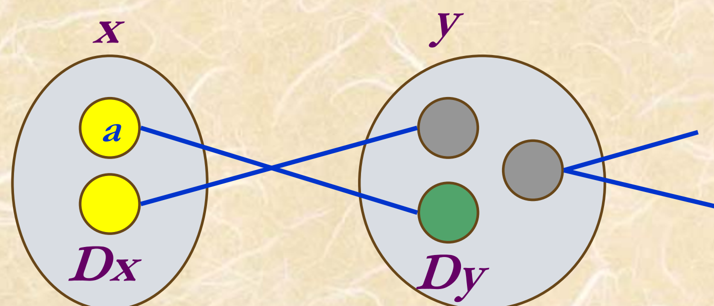
このスライドの状況では, 値 x=a および x=a' は y にサポートをもつ.

しかし, x=c は y にサポートを<u>もたない</u>ので, x=c がCSPの解に含まれることはあり得ない. yの値が何であっても制約違反となるからである. したがって, x=c を x の領域から削除してよい. その処理を行うのが, **アーク整合アルゴリズム**である.

アーク整合(2)

■ アーク (x, y) はアーク整合 している
arc consistent

＝すべての x=a が y にサポートをもつ



(y, x) はアーク整合 していない

(x, y) が制約グラフのアーク（有向辺）であるとする.

x の領域に含まれるすべての x=a が y にサポートをもつとき,「アーク (x, y)は**アーク整合** している」という.

アークには向きがあるので，アーク (x, y)はアーク整合 していることと，その逆向きのアーク(y, x)はアーク整合していることは別である. このスライドの状況では，(x, y)はアーク整合 しているが，(y, x) はアーク整合 していない.

アーク(x, y)の整合アルゴリズム

```
boolean REVISE(x, y) {
  changed ← false;
  for each a in Dx {
    if (x=a が y にサポートをもたない) {
        Dx から a を除去する.
        changed ← true;
    }
  }
  return changed;
}
```

値を1つでも除去したら true

計算量（制約チェック回数）

$$O(d^2)$$

d は領域Dx,Dyの要素数の最大値

アーク(x, y)の整合アルゴリズムは，ここに示したとおりである．このアルゴリズムは，アーク (x,y)が与えられたとき，それがアーク整合するように，必要に応じて x の領域から値を削除する．戻り値は，値を1つでも除去したら true，1つも削除しなければ false である．

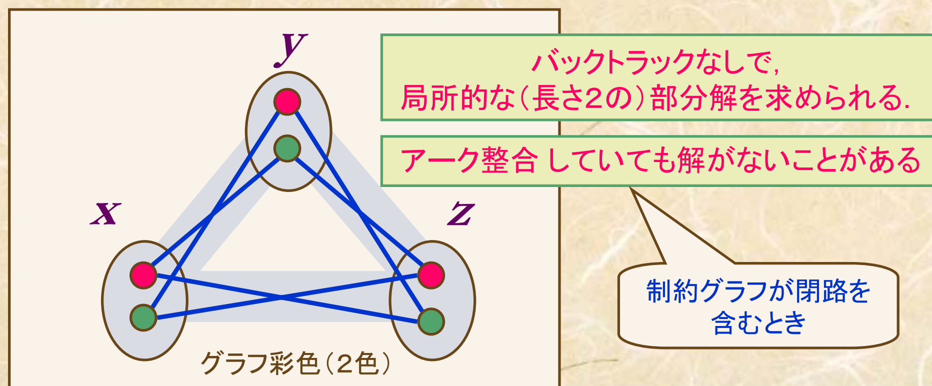
手続きは，すでに学んだアーク整合の定義のとおりである．各 x=a について， x=a が y にサポートをもつかどうかを判定し，サポートをもたなければ x=a を領域から除去する．サポートをもてば，単にその x=a をスキップする．

このアルゴリズムの計算量を考察しよう．ここでは，制約チェックの実行回数によって時間計算量を評価する．領域 Dx, Dy のサイズ（要素数）が高々 d だとしよう．たとえば，3色によるグラフ彩色問題では，色の数が3なので d=3 である．

if 文の条件部でx=a が yにサポートをもつかどうかの判定には O(d)回の制約チェックが必要である．この if 文はO(d)回実行されるので，制約チェックの総回数はO(d^2)（d の2乗のオーダー）となる．

アーク整合(3)

- 制約ネットワーク はアーク整合 している
  ＝すべてのアーク(x, y) がアーク整合 している
- アーク整合アルゴリズム
  ＝アーク整合 していない制約ネットワークから
  最小限の要素を削除してアーク整合させる．
  空の領域が生じたら，CSPには解がない．

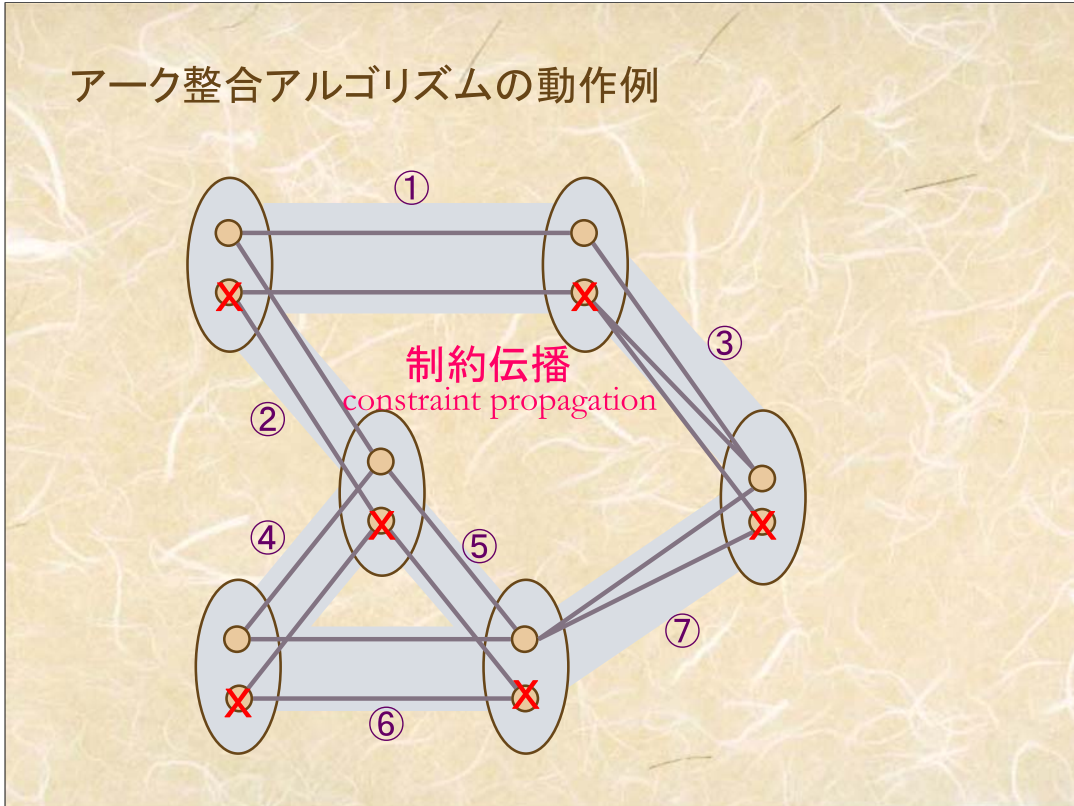


さきほど学んだ「アーク整合する」の主語は「アーク」だったが，こんどは，主語が「制約ネットワーク」のときにも使える言葉として定義する．

すべてのアーク(x, y) がアーク整合しているとき，「制約ネットワーク は**アーク整合** している」という．アークには向きがあるので，制約グラフのそれぞれの無向辺 (x, y) ごとに，(x, y) と (y, x) のアークがあることに注意しよう．

アーク整合アルゴリズム は，アーク整合していない制約ネットワークから最小限の要素を削除してアーク整合させる．その結果，空の領域が生じたら，CSPには解がないことがわかる．

逆は，一般に成り立たない．CSPに解がなくても，空の領域が生じないことがあるからである．このスライドの2色のグラフ彩色問題はそのような例である．この例では，グラフに閉路が含まれていることに注意しよう．閉路が含まれていないときには，さきほどの「逆」が成り立つことを後に説明する．



このスライドは，アーク整合アルゴリズムの動作例をアニメーションで示している．

領域から要素を1つ削除したことの影響が，次々と隣接ノードに伝わっていく．これを**制約伝播**という．

アーク整合アルゴリズム　AC-1

```
AC-1(CSP G) {
  Q ← Gのすべての有向アークの集合.
  do {
      changed ← false;
      for each (x, y) in Q
         if ( REVISE(x, y) ) changed ← true;
  } while ( changed );
}
```

全アークをそれぞれ整合

1か所だけ変化しても，全アークをスキャンしなおすので効率が悪い

これは最も素朴なアーク整合アルゴリズムであり，ふつう，AC-1と呼ばれている．

これはすべてのアーク (x, y) をスキャンしながら，すでに学んだ REVISE(x, y) によって，アークの整合をとる．すべてをスキャンしても全く変化がなかったら終了する．

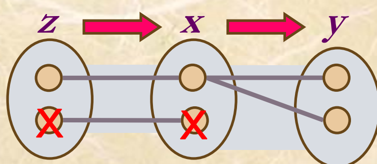
このアルゴリズムが非効率的なのは明らかである．スキャンのほとんどでは変化がなく，終わりの方で1つだけ変化があったとしよう．それでもこのアルゴリズムは，全アークをスキャンしなおすという不要な動作をしてしまう．

アーク整合アルゴリズム　AC-3

Qは First-In First-Out のキュー（待ち行列）

```
AC-3(CSP G) {
  Q ← Gのすべての有向アークの集合.
  while ( Q が空でない ) {
    Qから先頭のアークを取り出し，(x, y) とする.
    if ( REVISE(x,y) )
       for each z in y を除く xの隣接ノードのすべて{
          Qの末尾にアーク (z,x) を追加する.
       }
  }
}
```

(x, y) の整合の結果，x の領域から要素が削除されたら x へ向かうすべての有向アーク (z, x) （ただし，z≠y）をQに追加する.



高々 d 回Qに追加

計算量（制約チェック回数）

$$O(d^2 \cdot de) = O(d^3 e)$$

d は領域の要素数の最大値
e は制約グラフのアーク数

AC-3 は，AC-1 の改良版で，1つの変更が周辺に影響を及ぼす制約伝播の様子を素直に表現したもので，じゅうぶんに実用的なアルゴリズムである．ポイントは，将来に整合性をチェックすべきアークを**待ち行列**Qに保持しておくことである．

最初は，全アークをQに保持する．アルゴリズムのメインループは，Qからアーク (x, y) を取り出し，REVISE(x, y) で整合させる．もし，x の領域に変化がなければ，スキップ．

もし，x の領域から値が削除されたなら，その影響を受ける可能性のあるアーク (z, x) をQに追加する．これによって，整合性をチェックする必要性のあるアークのみがQに保持され，順々に取り出されてはチェックされることになる．

このアルゴリズムの計算量を考察してみよう．ここでは，制約チェックの回数によって時間計算量を評価する．このアルゴリズムで制約チェックが行われるのは，REVISE(x, y) の実行の中だけである．すでに学んだように，その計算量はO(d^2) なので，あとは，REVISEが最大で何回実行されるかを求めて，両者の掛け算をすればよい．

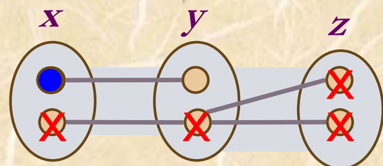
REVISEの実行回数は，その直前に，Qからアークを取り出す回数と一致する．アルゴリズムが終了するときにはQは空なので，その回数は，Qにアークを追加する回数と一致する．

これを求めよう．まず，初期設定のときに，Qに 2e 本のアークを追加している．（アークは向きが2通りあるので2倍している．）つぎに，特定のアーク (z, x) について考えると，これがQに追加されるのは高々 d 回である．なぜなら，x の領域（要素数は高々 d）から値を1個削除するときに限って追加されるからである．これは他の2e本のすべてのアークについても当てはまる．

以上から，Qに追加されるアーク数は，高々 2e+d*2e = O(de) となる．したがって，制約チェックの総数は，O(d^d)*O(de) = O(d^3 e) となる．d が定数のときには，アーク数に比例した線形時間で制約ネットワーク全体のアーク整合が完了することになる．

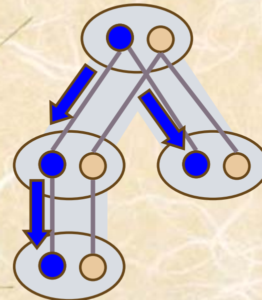
アーク整合の利用法

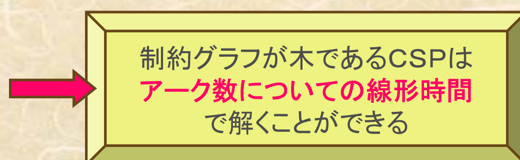
- 前処理
- バックトラック法で，変数の値を選択したときに，フォワードチェックのかわりに使う



この例では，z の2つの値を削除して，失敗を早期に検知できる

- 制約グラフが木のときに，バックトラックなしで解を求める
  backtrack-free





アーク整合は，単独で用いるより，他のアルゴリズムの補助として用いられることが多い．CSPは一般にNP完全問題であるため，それを解くアルゴリズムの計算量は最悪で指数オーダーである．したがって，線形オーダーで処理の済むアーク整合は，効率を改善するための補助手段として手軽である．

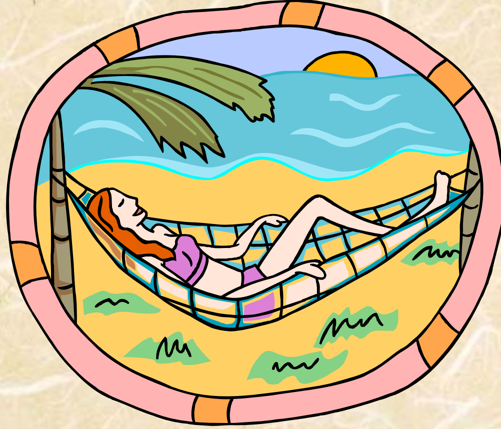
まず，いかなる制約充足アルゴリズムを使うにしても，アーク整合をその**前処理**に使うのが得策である．簡単な問題の場合には，この処理だけで空の領域が生じて「解なし」と判定できることもある．

また，バックトラック法で，変数の値を選択したときに，フォワードチェックのかわりに使うこともできる．このスライドの図では，xの値を選択したときに，他のxの値を領域から削除し，アーク整合アルゴリズムを実行した様子である．x－z間に直接の制約はないので，フォワードチェックだと，yの要素を削除できるが，zの要素は削除できない．しかし，フォワードチェックのかわりにアーク整合を使うと，zの値をすべて削除することができるので，xの値の選択が間違いであることがすぐにわかる．

ただし，フォワードチェックは，アーク整合よりもさらに軽い処理なので，どちらが性能向上に寄与するかは一概に判断はできない．

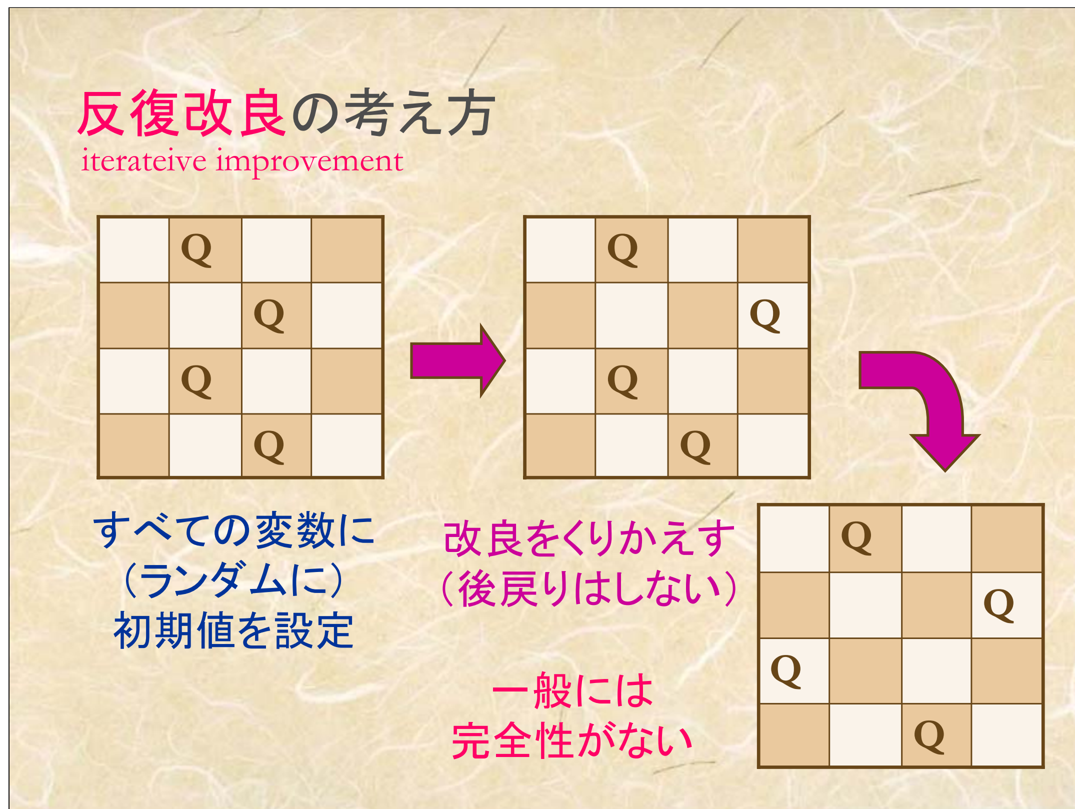
特別な場合として，制約グラフが**木**（閉路をもたないグラフ）のときには，アーク整合アルゴリズムを単独で用いて解が求められる場合がある．アーク整合の後に空の領域が生じていなければ，任意の変数を1つ選び，その領域から任意の値を1つ選ぶ．この変数を根とする木を作る感じで，選択された値のサポートを次々とたどっていけば，アーク整合の定義により，**バックトラックなし**(backtrack-free)で，O(e)の時間ですべての変数の値を制約違反なしで決定することができる．したがって，制約グラフが木であるCSPはアーク数についての線形時間で解けることがわかる．

休憩



局所探索アルゴリズム
(Local Search Algorithms)

1. **山登り法**
   hill climbing
2. **制約違反最小化**
   min-conflicts
3. **焼きなまし法**
   simulated annealing



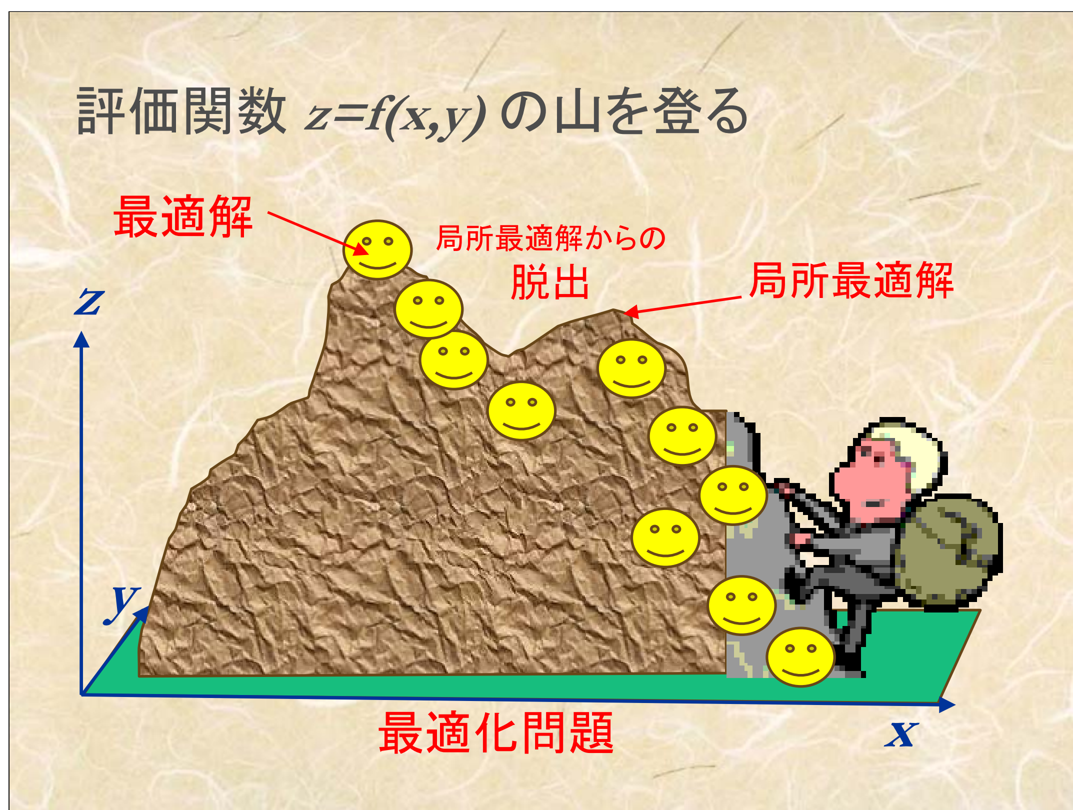
局所探索(local search)アルゴリズムに共通した考え方は，**反復改良**というものである．バックトラック法のように変数への割当てが部分的にしか決まっていない状態を考えるのではなく，制約違反があってもよいから，すべての変数に値を割り当てる．

ふつうは，初期状態として，すべての変数にランダムな初期値を設定する．

それを反復的に改良していくプロセスがそのアルゴリズムである．そのプロセスではバックトラックを行わない．

そのため，特別なメカニズムを用意しない限りは，基本的に，反復改良アルゴリズムには**完全性**はない．すなわち，解があっても見つける保証はない．また，解がない場合でも，そのことを判定できず，限りなく反復が継続するか，または，それ以上の改良ができない状態で止まってしまう．

しかし，非常に難しい問題で，かつ，必ず解が存在するような問題では，バックトラック法やそれを多少改善したものよりも効率よく解が求められることが多い．



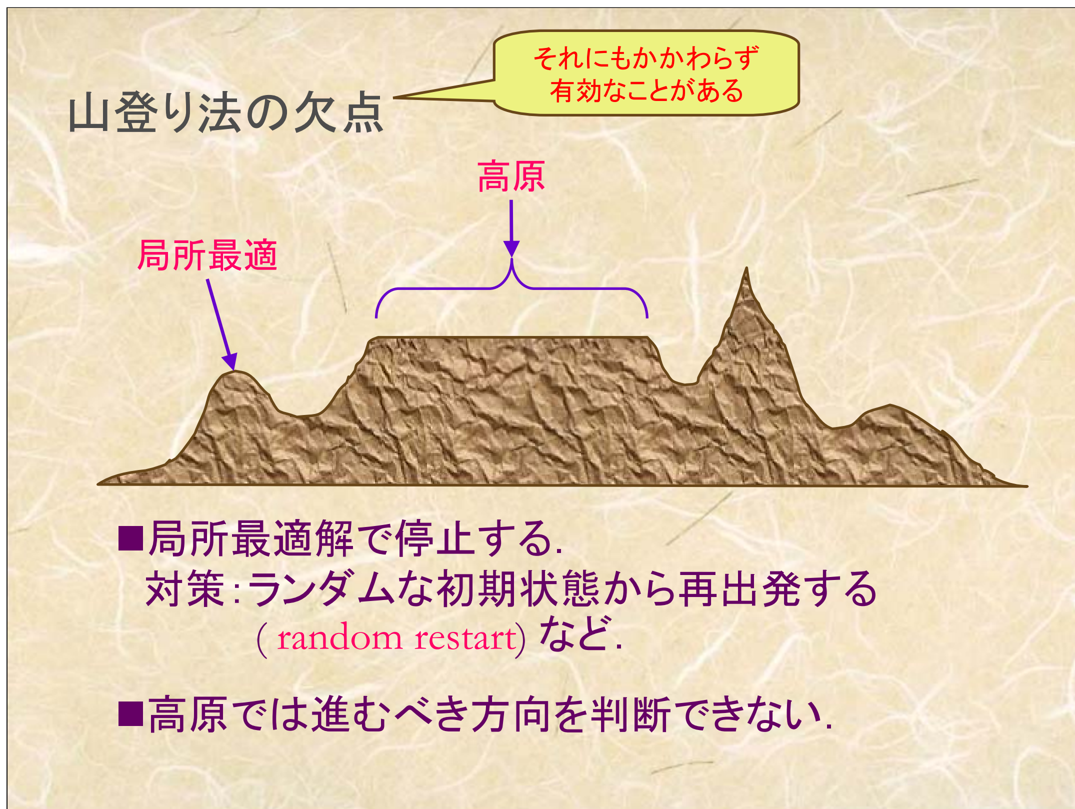
反復的に「改良」するということは，数学的には，現在の状態（仮に (x, y) とする）の良さを表す**評価関数** z =f(x, y) が与えられており，(x, y) を少しずつ変更しながら，z の最大値を探す問題となる．

このような問題は一般に**最適化問題**と呼ばれている．

このような問題を解くときの難しさは，**局所最適解**の存在である．これは最大値ではない，いわゆる，極大値のことで，局所的なピークを形成しているものである．その近傍では z の値が相対的に小さくなっているので，(x, y)を少し改善したくらいでは，zの値は現在より大きくならないので，改良が難しい状態になっている．

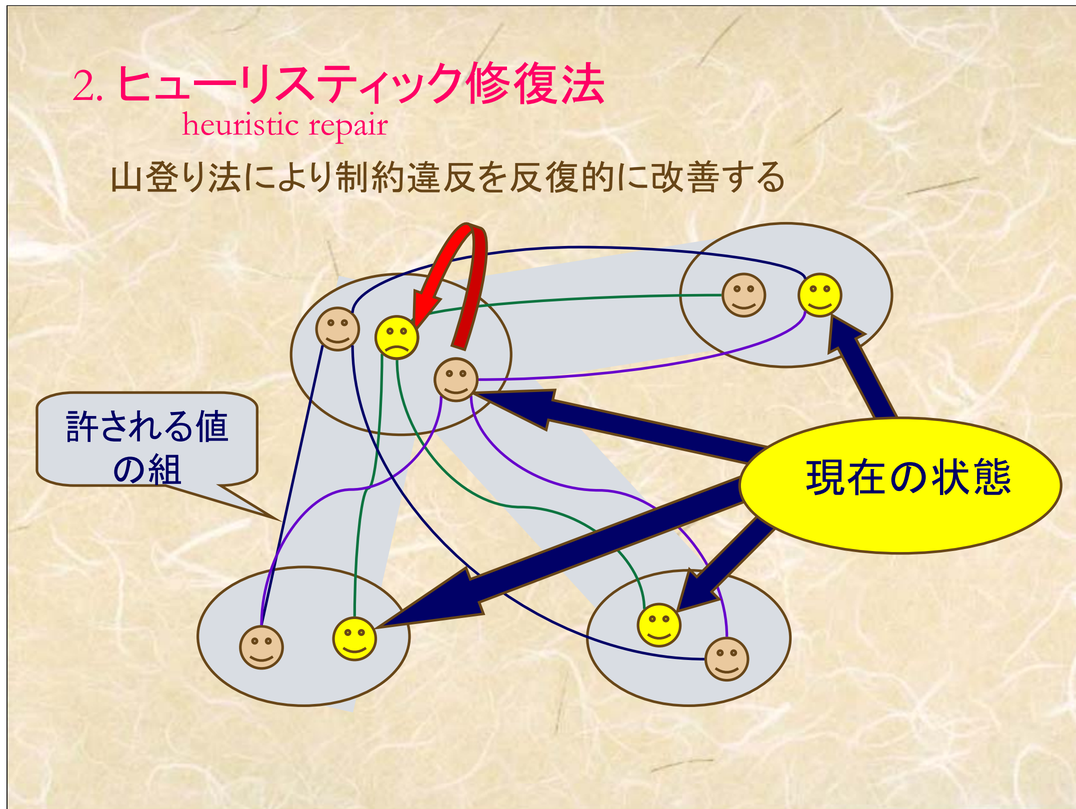
局所最適解からいかに**脱出**するかが，アルゴリズム設計のポイントとなる．

山登り法は，（何らかの意味での）「近傍」の状態のうち評価値が最大の状態に進む．決して下り坂を降りることはしない．

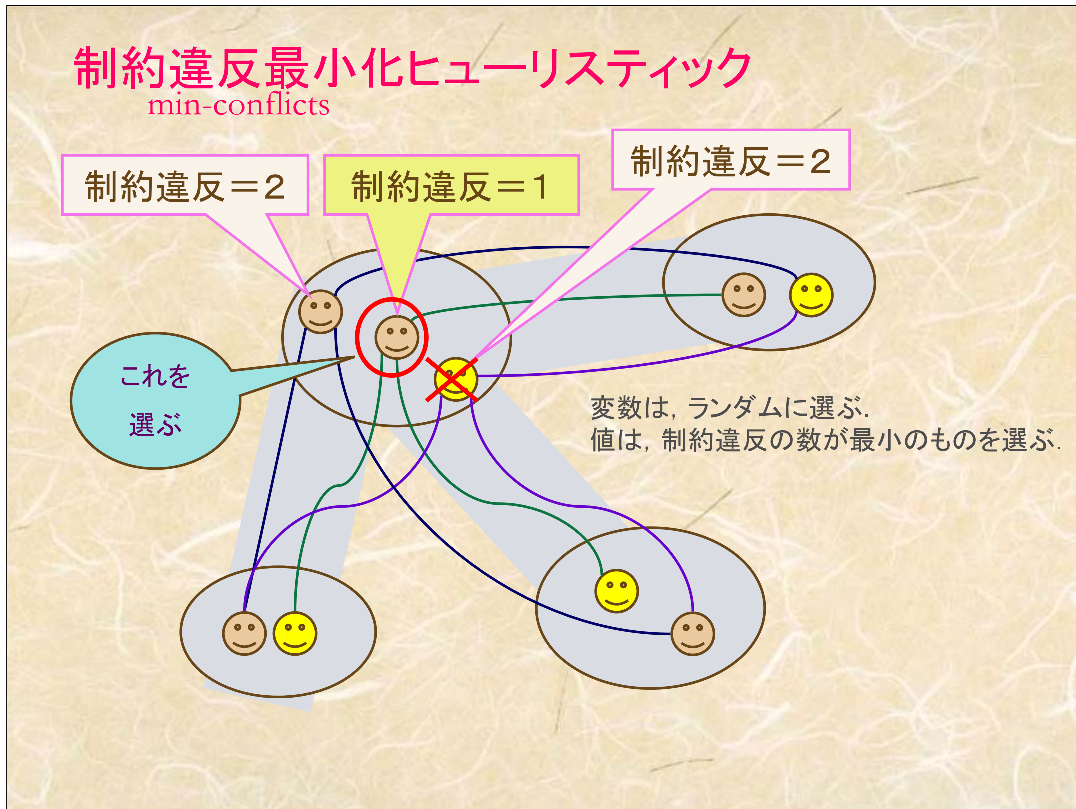


山登り法には，局所最適解で停止したり，評価値が一定の「高原」と呼ばれる広い範囲においては進むべき方向を判断できないというような欠点がある.

しかし，それにもかかわらず，**ランダム再出発**（random restart）などと組み合わせて，非常に難しい問題の解を求められることがある.



ヒューリスティック修復法は，山登り法の一般的な考え方を制約充足問題に適用したもので，ただ1つの変数の値を変えることにより制約違反を反復的に改善する．



ヒューリスティック修復法で良く用いられるのが，**制約違反最小化**ヒューリスティック（**min-conflicts**）である.

この方法は，変数を1つランダムに選び，その値として，制約違反の数が最小となるものを選ぶ.

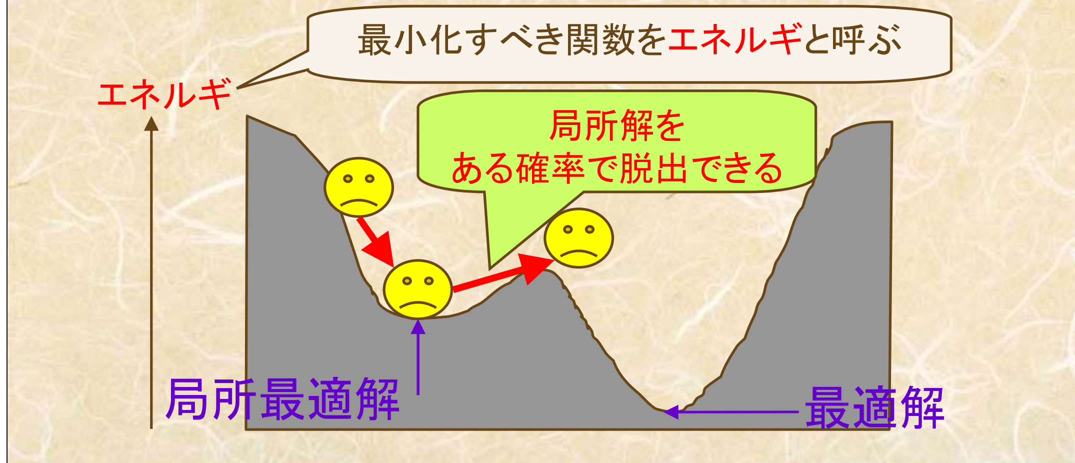
制約違反最小化の実績

百万クイーン問題： 平均50ｽﾃｯﾌﾟ

ハッブル天体望遠鏡の観察スケジュール

3週間から10分に短縮

簡単なヒューリスティックであるにもかかわらず，制約違反最小化ヒューリスティックの実績には，このスライドに書かれているように，目をみはるものがある．



焼きなまし法は，粒子の分子運動のような物理学のモデルからの類推によって設計されたアルゴリズムである．鋼などの固い金属を作るために，金属が溶解している状態から徐々に温度を冷やして固化させていく「焼きなまし（annealing）」という技術がある．それのシミュレーションから発想されたアルゴリズムなので，**simulated annealing** (SA) と呼ばれている．

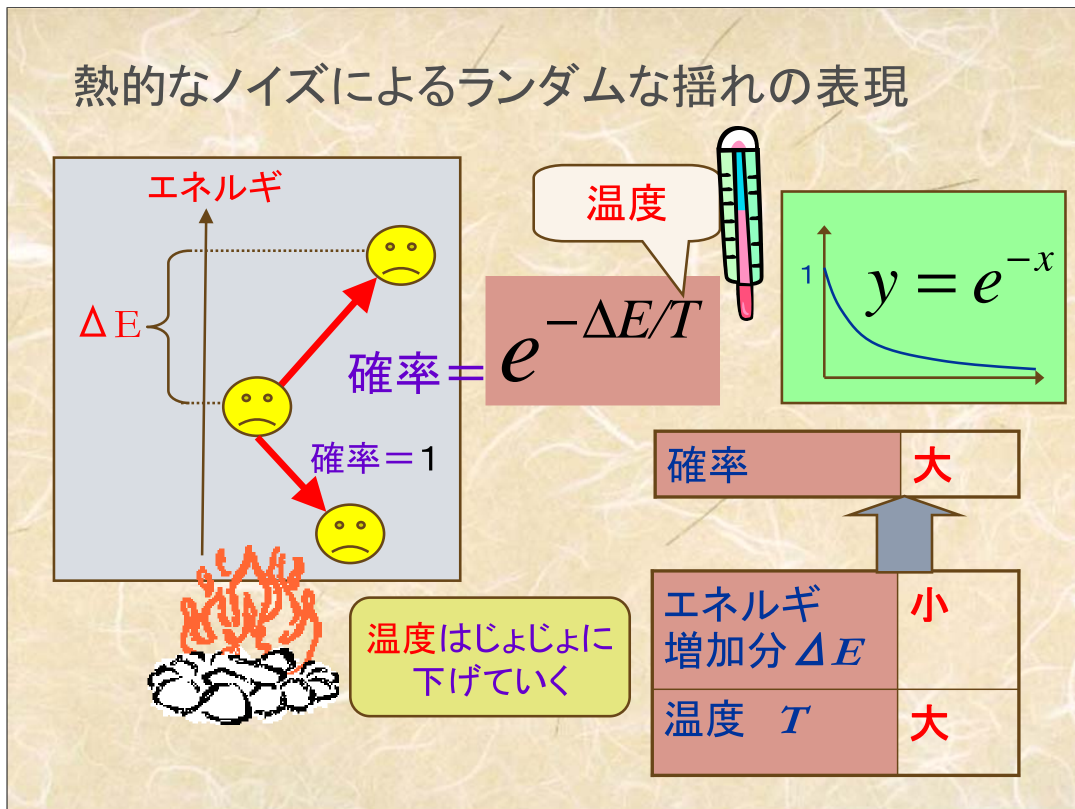
焼きなまし法の慣例では，評価関数による評価値は「**エネルギ**」と呼ばれ，これを最小化する最適化問題になっている．

山登り法との違いは，まず，近傍の状態から1つをランダムに選ぶことである．

さらに，選んだ状態へ進むかどうかは，現在の状態とのエネルギー差に依存する．

エネルギが減少するときには，必ずそこに進む．これは山登り法の考え方に通じている．

特徴的なのは，このアルゴリズムは**温度**と呼ばれるパラメータを持っており，次状態でエネルギが増加するなら，温度に応じた確率でそこに進むことである．これによって，局所最適解でストップすることなく，そこからの脱出をはかっている．



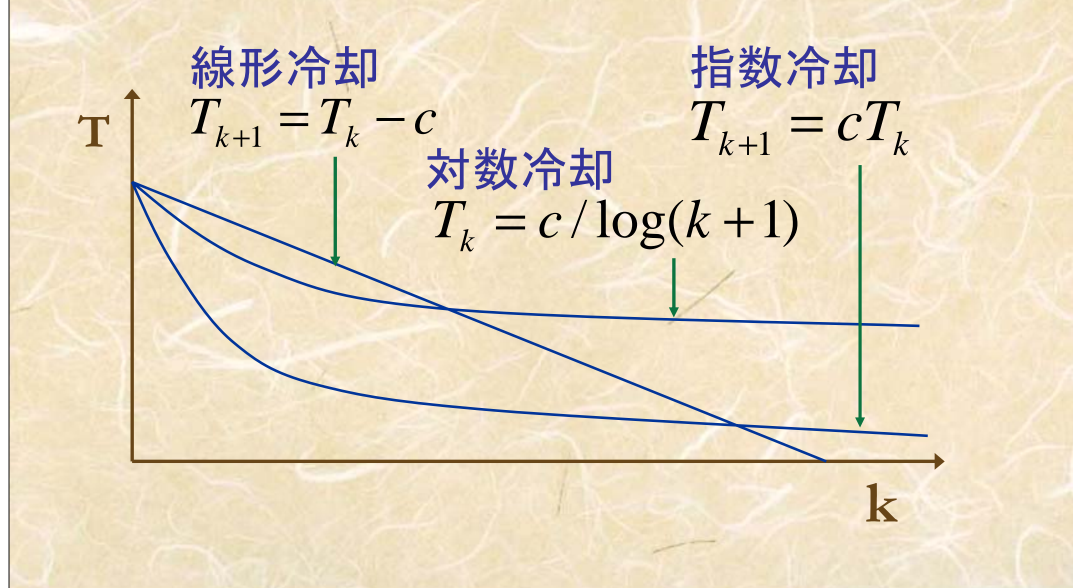
次状態での**エネルギーの増分**をΔE，**温度**をTとする．

すでに述べたように，ΔE≦0ならば，確率1でその状態に進む．

ΔE＞0のときには，物理学（熱力学）で知られている結果からの類推から定義された確率 exp(－ΔE／T)でその状態に進む．その状態に進まないときには，再度近傍から状態を選び直す．

（ここで，exp(－ΔE／T)という式は，自然対数の底 e=2.71... の－ΔE／T乗という意味だが，eであることは重要ではなく，かわりに2を用いてもよい．）

この遷移確率は，エネルギー差ΔEが小さいほど大きい．また，温度Tが大きいほど大きい．それは物理学では，温度が高いほど粒子の運動エネルギーが大きく，活発に運動しており，運動エネルギーを多少減少させることによって位置エネルギーの増大する方向に飛び跳ねる確率が大きいからである．



焼きなまし法の技術的なポイントは温度パラメータTの管理である．温度を低く保てば，山登り法と同じ効果しか得られない．といって，温度を高くしたままでは，粒子の運動が激しく，最適解に収束して停止するのは望めない．

そこで，計算の初期段階では温度を高くしておき，時間とともに，温度を徐々に低くしていく制御がなされる．その具体的な方法を**冷却スケジュール**とし い，代表的にはこのスライドで示したようなものがある．よく使われるのは**指数冷却**である．

焼きなまし法の最適性

- 温度Tを十分ゆっくり下げるならば，確率1で大域的最適解を見つける．
- 対数冷却 $T_k=c/\log(k+1)$ はこの条件を満たすが，収束時間は$O(n!)$より長い．

温度はすごくゆっくり
下げていく

驚くべきことに，ある条件のもとで，焼きなまし法には**最適性**がある．つまり，大域的な最適解があれば，必ずそれを見つけることができる．

その条件は，温度Tを十分ゆっくり下げるということである．これは数学的にあいまいな表現だが，前のスライドで示した**対数冷却**はそのような条件を満たすことが知られている．

しかし，残念なことに対数冷却は温度が十分に下がるまでの収束時間は指数関数的に長いものであり，実用的ではない．ふつうは，指数冷却の範囲でなるべくゆっくり冷却し，最適性をあきらめる．