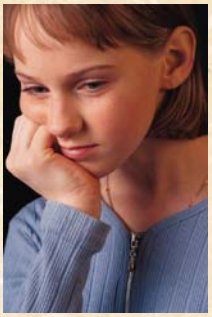
知能情報処理　制約充足(1)

制約をみたす意志決定をするエージェント

制約充足問題

(Constraint Satisfaction Problems)

- 制約充足問題
- 制約充足アルゴリズム
  - バックトラック法
  - フォワードチェック
  - 動的変数順序



今回の授業では，**制約充足問題**と呼ばれる広い範囲に適用可能な問題群の定義とその実例をいろいろ学ぶ．また，そのような問題の解を求める基本的なアルゴリズムとして**バックトラック法**を理解し，その効率を向上させるヒューリスティックな手法として，**フォワードチェック**と**動的変数順序**について学ぶ．

制約充足問題
(Constraint Satisfaction Problems)

制約充足問題(constraint satisfaction problem: **ＣＳＰ**)は，つぎの3項目

1) **変数**の集合
2) 各変数の**領域**
3) 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.

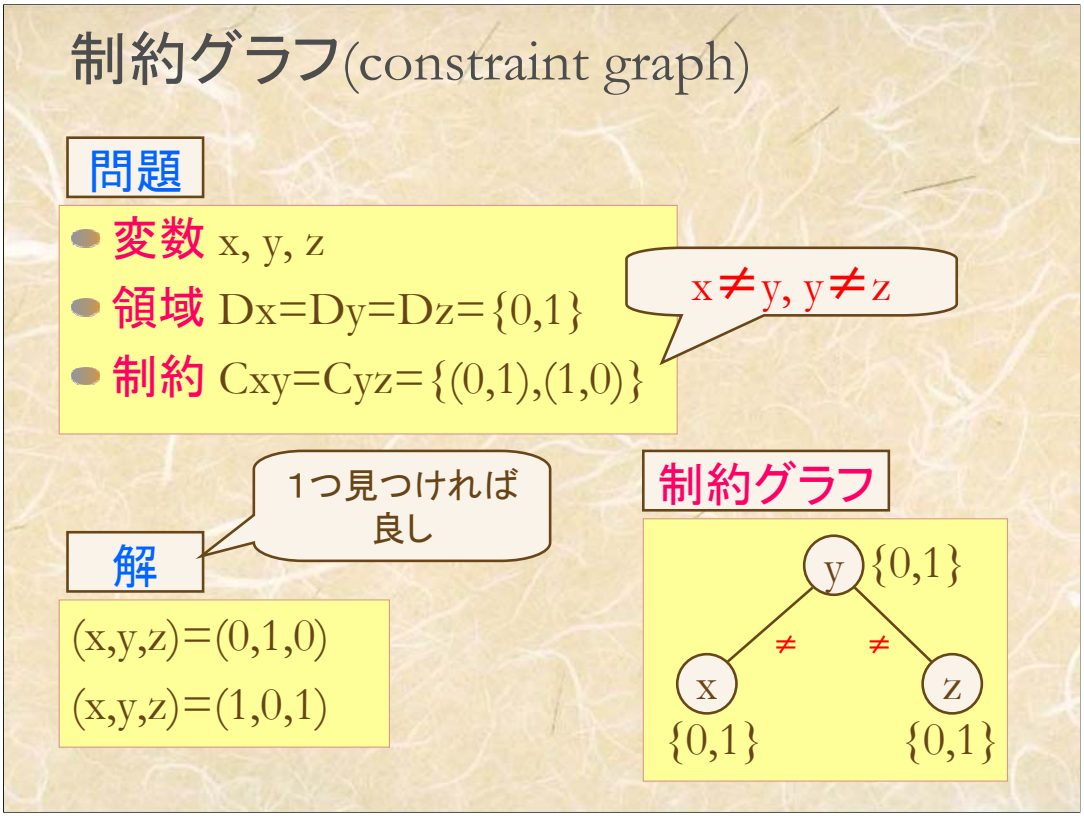
領域 D_i は変数 x_i の取りうる値の集合を表している (i=1,2,...,n). 領域に含まれる要素は有限個であると仮定する.

制約とは複数の変数の間が満たすべき条件であり，取ることが許される値の組合せで表現される. たとえば，

Cxy = {(a,b),(c,d)}

という制約は，変数xとyの間の制約を表していて，(x=a,y=b)と(x=c,y=d)という値の組合せだけが許される.

すべての制約を満たすように変数に値を割当てることを，「制約充足問題を解く」という. そのような割当てのことをその問題の**解**という.



このスライドは簡単なCSPの例である．変数はx,y,zの3個であり，その領域の定義からわかるように，どの変数も0または1の値を取り得る．

制約はx≠y, y≠z の2つであるが，1つ前のスライドの表現方法にしたがって，許される値の組合せである (0,1)と(1,0)を列挙して集合とし，共通にDxyおよびDyzとしている．

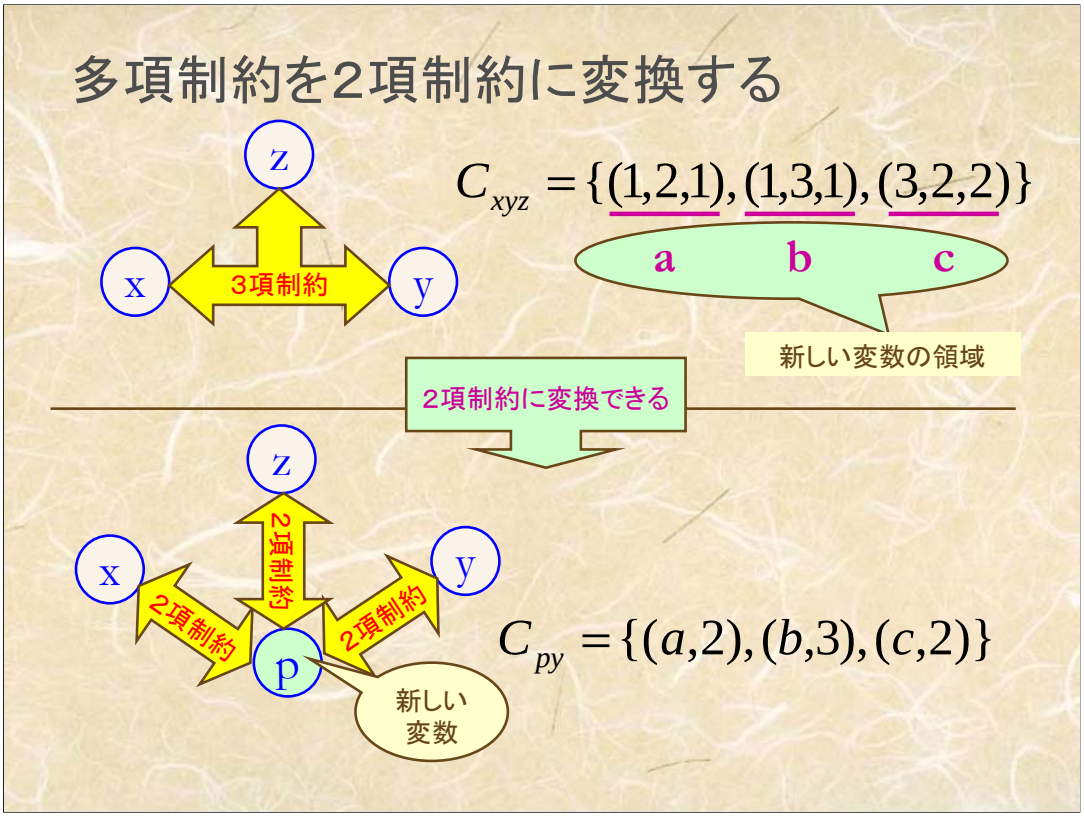
数学の得意な人は，Cxyは直積集合Dx×Dy={(0,0),(0,1),(1,0),(1,1)}の部分集合になっていると理解してよい．もっと得意な人向けには，CxyはDxとDyの間の**二項関係**を表していると言っておこう．

CSPを視覚的にわかりやすく表現するために，**制約グラフ**が用いられる．これは，各変数をノード（頂点）とするグラフであり，変数間に制約が存在するときには，対応するノードを辺で結ぶ．ノードxを領域Dxでラベル付けし，辺(x,y)を制約Cxyでラベル付けして，このスライドのように図示しておくとわかりやすい．

この問題の解は(x,y,z)=(0,1,0)と(x,y,z)=(1,0,1)の2つある．CSPを解くアルゴリズムは，理論的には解をすべて見つけることが望ましいが，それは一般的に計算時間が非常にかかることが多いので，実用的には解を1つだけ見つければ良しとする．

2つの変数間の制約を**２項制約**，3つの変数間の制約を**３項制約**という．一般に，３つ以上の変数間の制約を**多項制約**という．

多項制約を考慮しないで，2項制約のみを対象としても理論的な一般性は失われない．なぜなら，新しい変数を導入して，多項制約をそれと等価な2項制約に変換するテクニックが知られているからである．ただし，実際には，多項制約を直接扱えるようにプログラムを作成するのがよい．この授業では，2項制約のみを扱う．



多項制約を2項制約に変換する方法の一例を示す．このスライドの例では，x,y,zの間の3つの変数を含む問題に対して，pという新しい変数を導入して，3項制約Cxyzを3つの2項制約Cpx,Cpy,Cpzで表現している．

しかし，この話題はこの授業の重要な学習項目ではないので細かいことは気にしなくてよい．

制約充足問題の例

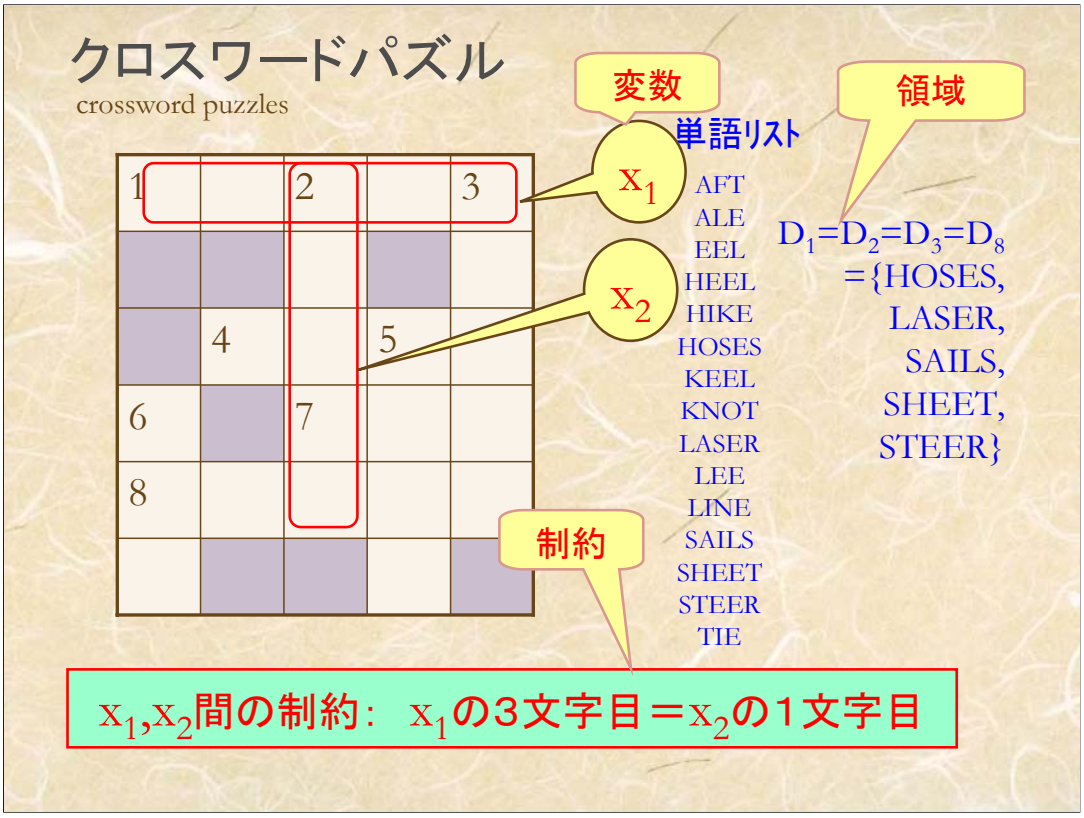
- ***n*** **クイーン問題** (n queens problem)
- **クロスワードパズル**(crossword puzzles)
- **グラフ彩色問題** (graph coloring)
- **線画解釈** (interpretation of line drawings)
- **レイアウト** (layout)
- **スケジューリング** (scheduling)

CSPの例を6つ紹介する.

nクイーン問題は8クイーン問題を一般化したもので，n×nのチェスボード上に，n個のクイーンを互いにアタックしないように置くことを要求するCSPである．

nクイーン問題が確かにCSPであることを確認するには，変数，領域，制約の3項目をそれぞれこのスライドのように決めればよい．変数xi (i=1,2,...,n) は第i列に置かれるクイーンの位置を表す行番号を表す．したがって，どの変数の領域も{1,2,...,n}である．

制約Cijは，第i列に置かれるクイーンと第j列に置かれるクイーンが互いにアタックしないような位置の対を列挙する方法で表現される．それをこのスライドのように数式で表現してもよい．



クロスワードパズルも，明らかにCSPである．ただし，これは雑誌等でよく見られるような「タテのカギ」とか「ヨコのカギ」と呼ばれるヒントがないものとする．（そういう自然言語を理解するのはまだAIの研究途上のテーマである．）そのかわりに，候補となる単語が辞書のように数多くリストとして列挙されていることにする．

この問題がCSPであることを確認するために，単語が入るべきタテあるいはヨコの一つながりの枠を各変数に対応付ける．その枠の長さに合った文字数の単語のすべてを集めて，対応する変数の領域とする．たとえば，「ヨコの1」を変数x1とすると，その領域D1は5文字からなる単語の集合となる．「タテの2」，「タテの3」，「ヨコの8」を表す変数の領域も，文字数5なのでこれと同じものとなる．

タテ，ヨコの2つの枠が交差するような2つの変数の間には，交差部での1文字が一致することを制約として記述する．この図では，x1,x2間の制約は，「x1の3文字目とx2の1文字目が等しい」ことを表すもので，その組合せを列挙すると，

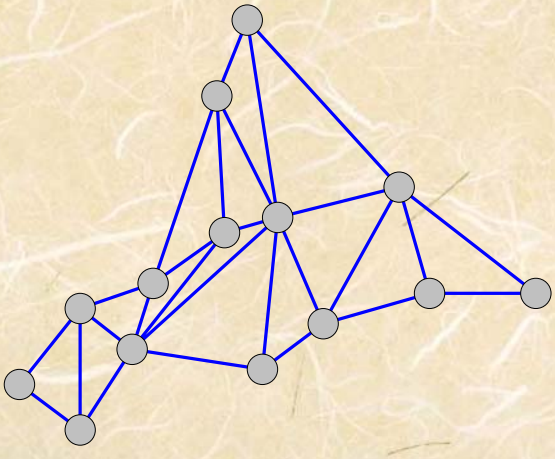
C12={(HOSES,SAILS),(HOSES,SHEET),(HOSES,STEER),
(LASER,SAILS),(LASER,SHEET),(LASER,STEER)}

となる．

グラフ彩色問題(1)

graph coloring

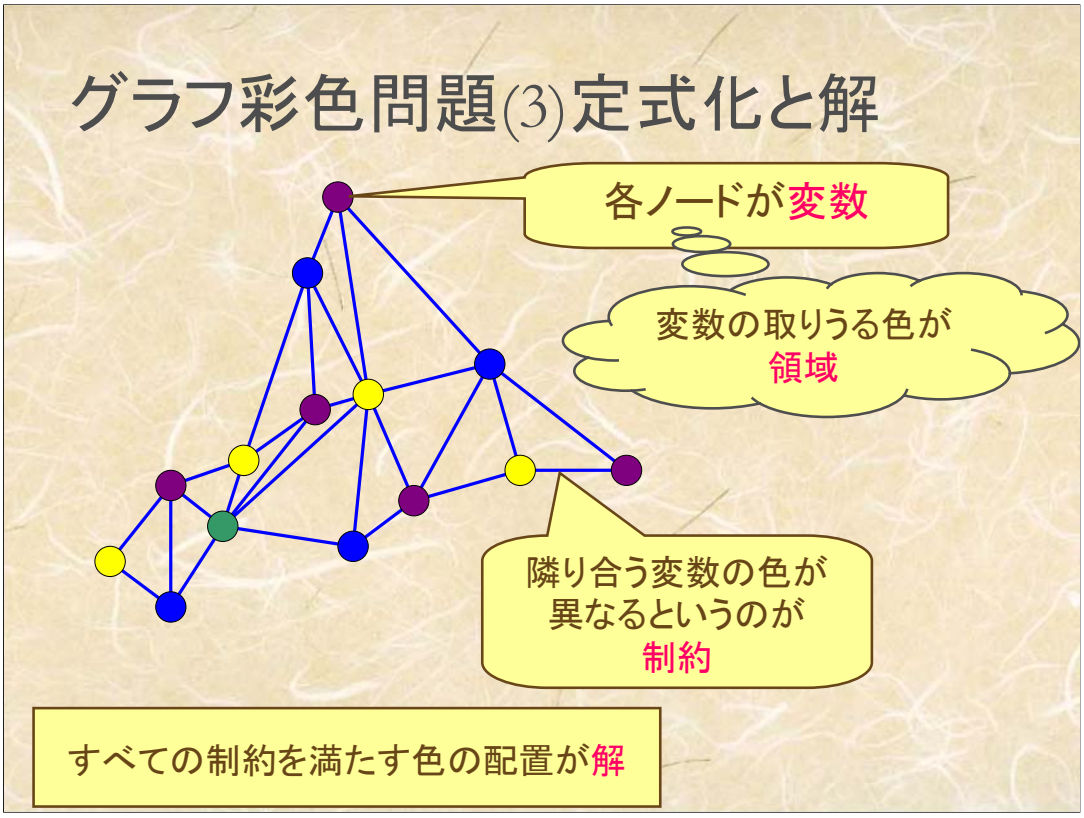
- 辺で結ばれたノードが異なる色になるように
  4色で塗り分けよ



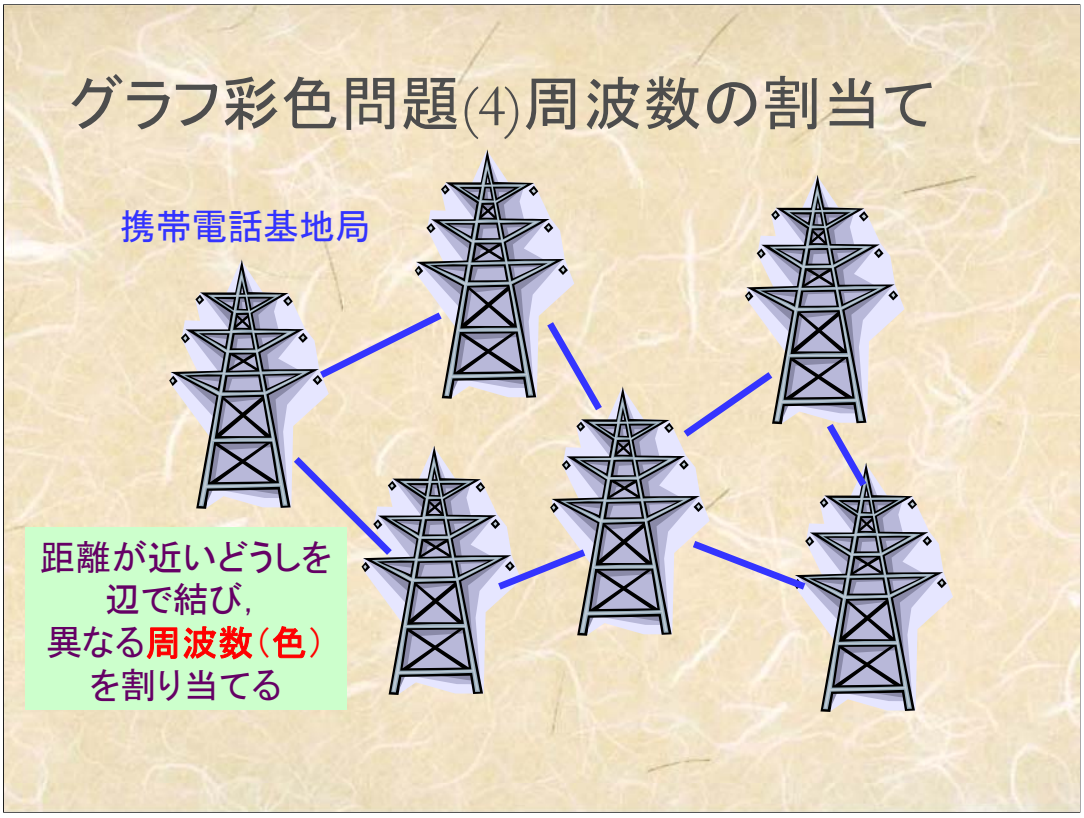
グラフ彩色問題は，頂点の集合Vと辺の集合Eからなるグラフ$G=(V, E)$と整数c（色の数）が与えられたとき，Vの各頂点のそれぞれに1からcまでのどれかの値（色）を対応づける問題である．制約として，辺で結ばれた（隣接した）ノードを互いに異なる色とすることが求められる．

グラフ彩色問題(2)地図の塗り分け

グラフ彩色問題は，**地図の塗り分け問題**と直接の対応関係がある．これは，地図上の隣接した地域に異なる色を塗る問題で，地域をグラフの頂点とし，隣接した地域を表す2つの頂点を辺で結んだグラフの彩色問題に帰着できる．



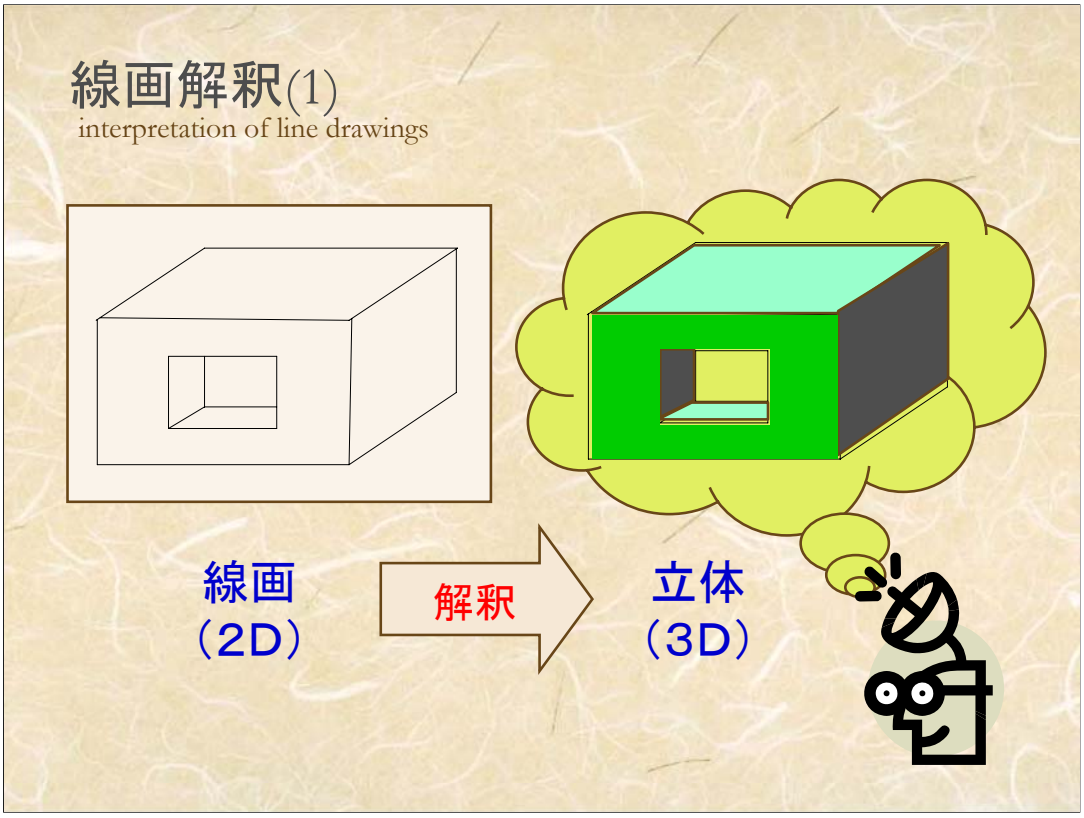
グラフ彩色問題が確かにCSPであることを，このスライドで確認してほしい．



グラフ彩色問題は，携帯電話の基地局(アンテナ)への**周波数割当て問題**とも関係がある．

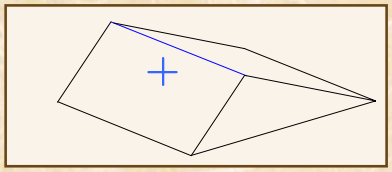
近接した基地局に同一の周波数を割り当てると，電波が干渉し合って混信の原因となるので，異なる周波数帯(チャネル)を割り当てる必要がある．

そこで，基地局をグラフの頂点とし，異なるチャネルを割り当てるべき基地局どうしは辺で結ぶ．各チャネルを色とみなせば，これはグラフ彩色問題となる．

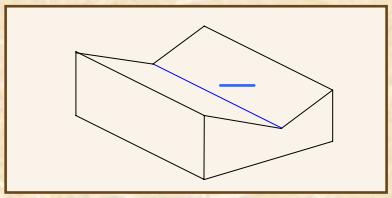


線画解釈の問題は，CSPを一躍有名にした問題である．この問題には，入力として，2次元平面上に描かれた線画が与えられる．これは利用者が直接描いた線画かもしれないし，カメラで撮影した画像から色や濃淡の変化を検出する画像処理によって辺（エッジ）を抽出したものかもしれない．線画解釈とは，このような線画を3次元空間内の立体として解釈する問題である．

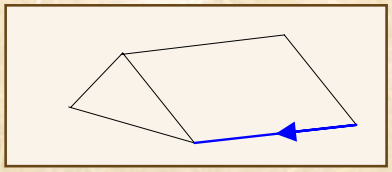
線画解釈(2)
線分のラベル付けによる空間表現



両側に物体の表面が見える.
前方に凸.



両側に物体の表面が見える.
前方に凹.

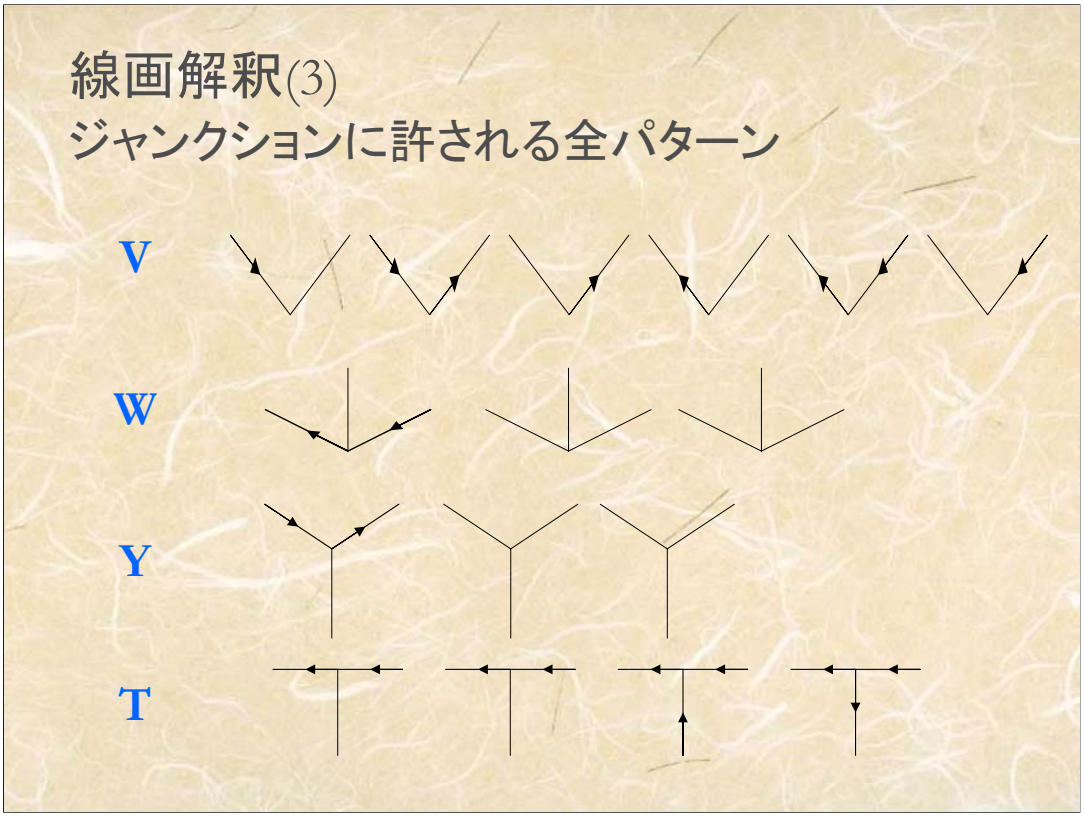


矢線の右側だけに物体の表面が見える.

3次元での解釈は，具体的には，線画に現れる線分に，＋，－，←，→のいずれかの**ラベル**を割り付けることによって表現される.

各ラベルの意味は，スライドに示してある通りである.「＋」ラベルの辺の両側には物体の表面が見え，その辺自体は前方（見ている側，手前）に凸になっている.「－」ラベルの辺もやはり両側に物体の表面が見えるが，その辺自体は前方に凹（後方に凸）になっている点が異なる. 矢印（←，→）の辺は，その進行方向の右側だけに物体の表面が見えており，左側は空間になっている.（ただし，スライドの3つめの例は，空間中に浮いている立体とする. 平面上に置いてある立体のときには，矢印が割り当てられている辺のラベルは「－」となる.）

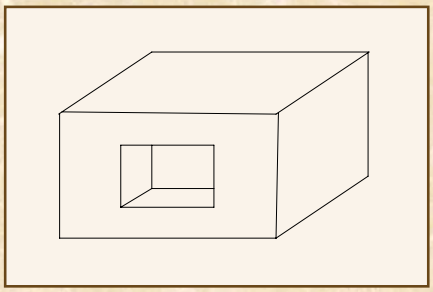
すべての辺にこのようなラベルを正しく割り付けることができれば，それで線画を立体として解釈したものとみなす.

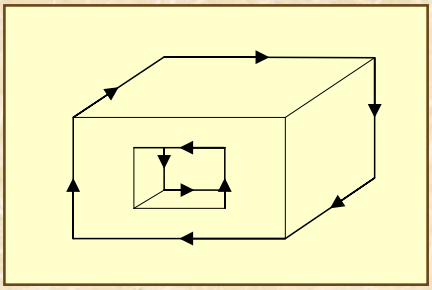


線画を構成している線分が互いに端点で接しているパターンを**ジャンクション**という．ジャンクションには，V, W, Y, T という4種類がある．Vは2本の線分が結合したものである．W, Y, T は3本の線分の結合であるが，各線分間の角度が鋭角か鈍角か（90度との大小）による違いにより，分類されている．

この応用を考えた研究者が綿密に分析した結果，ジャンクションを3次元立体の辺の交わりと解釈するには，各線分のラベル付けは，このスライドに示される少数個のパターンのいずれかでなければならないことに気がついた．つまり，ラベル付けにはこのような制約があるのである．したがって，線分に変数を対応付け，ラベルをその領域の要素としてCSPを定義した場合，ジャンクションの各パターンが制約となる．Vは2項制約だが，W, Y, T は3項制約となる．

線画解釈(4)
制約充足による解釈の生成





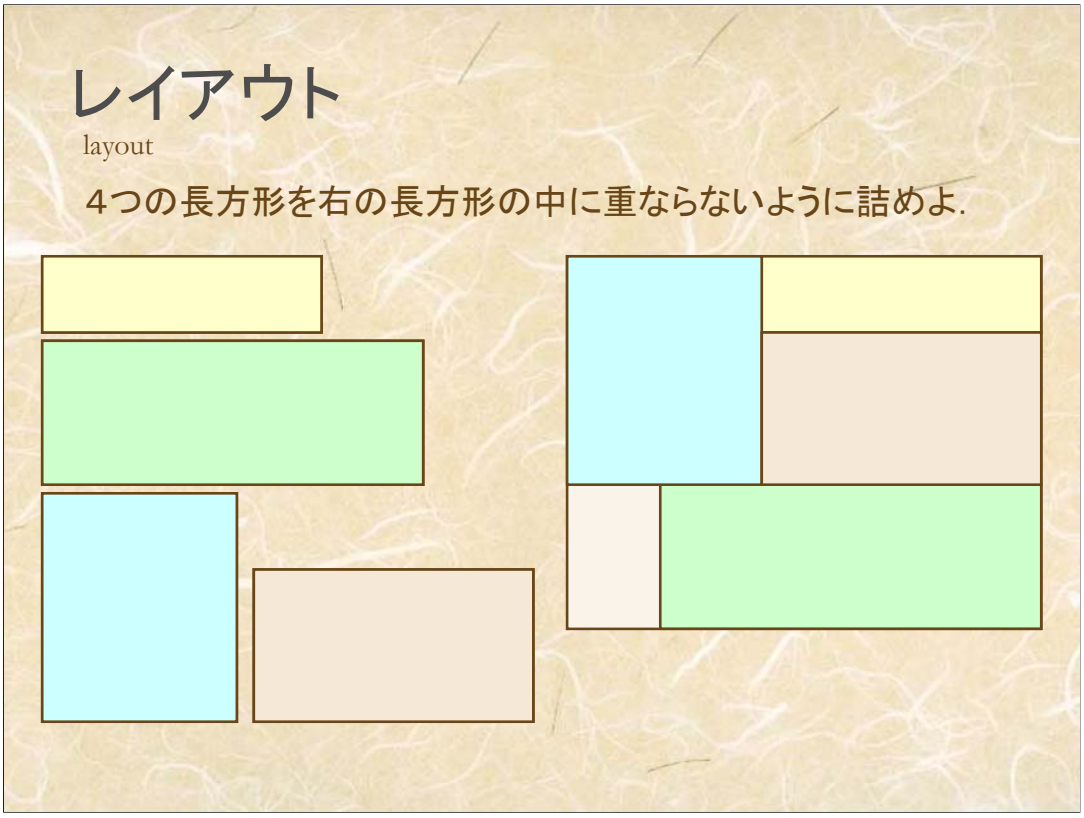
変数：　ジャンクション
領域：　ジャンクション
　　のパターン
制約：　辺に同じラベル
　　が付くこと

制約充足

解：　解釈

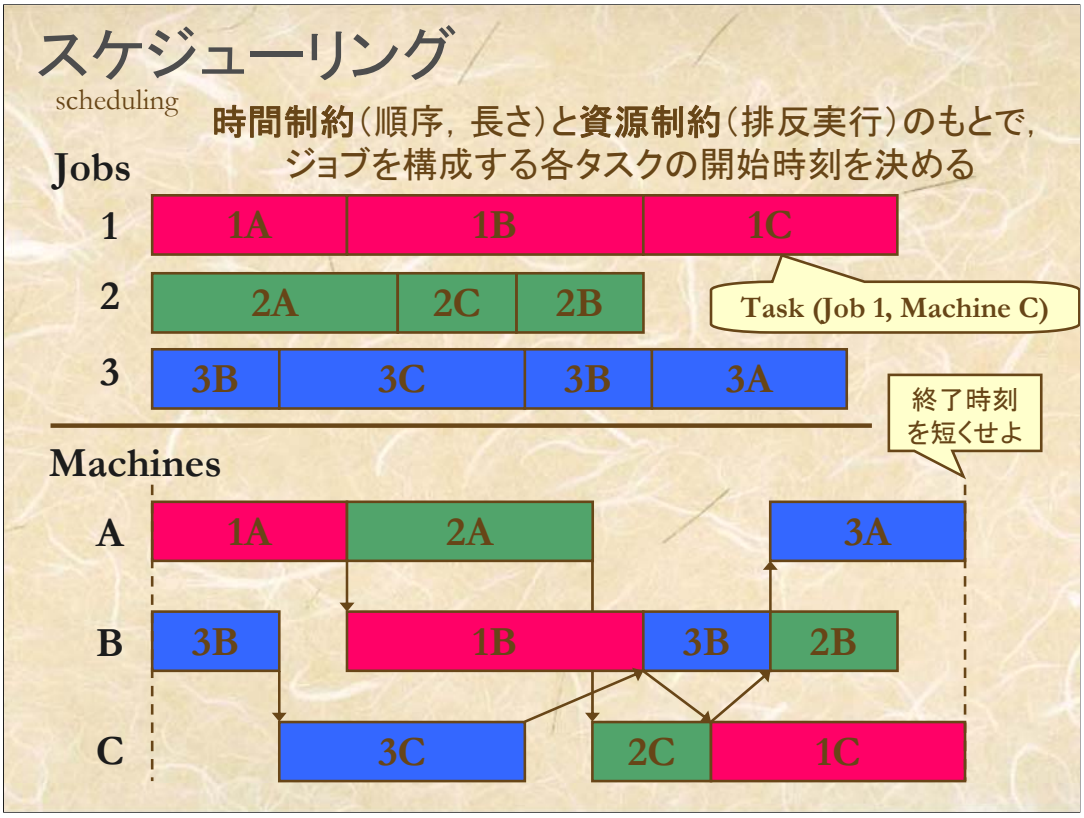
このスライドのように，この問題を2項制約のみからなるCSPとして定式化することもできる．すなわち，ジャンクションを変数とみなし，その取りうる値は前のスライドで示されたパターンのいずれかとする．その場合，各ジャンクションのパターンを独立に決めることはできず，当然，各辺はそれを含むジャンクションによって同じラベル付けがなされなければならない．それが制約となる．

そのスライドの右半分は，可能な解の一つを示している．



レイアウト問題は，建築設計においてフロアにいろいろな家具や機器を配置したり，VLSIのような電子回路に素子を配置するような問題である.

最も簡単な例だけをこのスライドで示す.



スケジューリング問題は，**タスク**の集合が与えられたときに，**時間**や**資源**に関する制約のもとで，各タスクに開始時刻を割り当てる問題である．それには，種々のバリエーションがあるが，このスライドは，**ジョブショップ・スケジューリング**(jobshop scheduling)と呼ばれる問題の例である．

この問題では，m個の**ジョブ**とn個の**マシン**が与えられる．各ジョブは一連のタスクから構成されている．（このスライドではジョブを1,2,3，マシンをA,B,C，タスクを1Aなどとラベル付けされた長方形で表している．ラベルの整数がジョブの番号，アルファベットがそのタスクを実行できるマシンの名前を表している．）

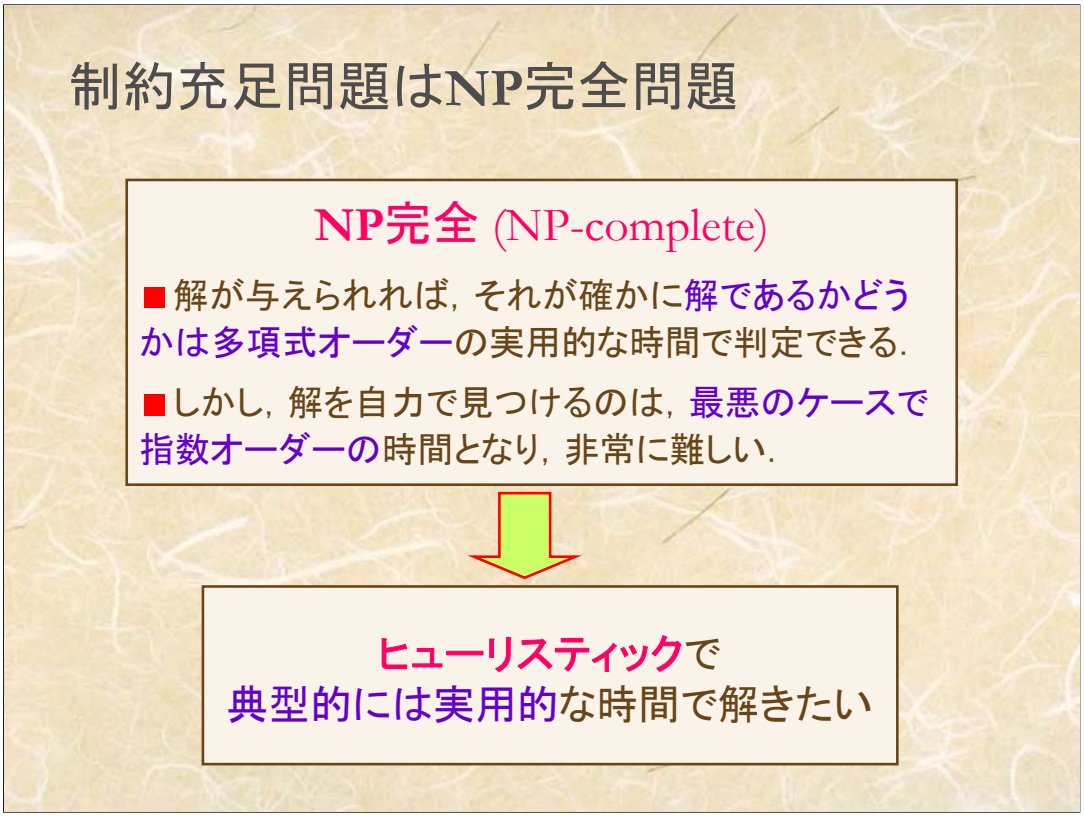
タスクの実行には次のような2種類の制約がある．

時間制約：タスクの**実行順序**と**所要時間**が決まっている．このスライドでは，長方形の並び順（左から右へ）が実行順序，長方形の長さが所要時間を表す．

資源制約：この問題ではマシンをタスク実行のための「資源」とみなす．各タスクを実行できるマシンは特定のマシンに制限されている．また，複数個のタスクを並列に実行することができるが，1つのマシンで同時に実行できるタスクは高々1つである．

このような問題は，工場などでの生産計画で必ず生じる問題であるが，情報システムにおいてもよく見られる．たとえばコンピュータシステムでは，ジョブは特定の利用者が投入した特定の処理を行なう単位であり，それがいくつかのタスク（たとえば，コンパイル，リンク，ロード，実行，入出力など）からできている．マシンというのは，CPU，ハードディスク，プリンタなど，タスク実行に必要なハードウェア資源である．

このような条件設定のもと，すべてのジョブを特定の時間内にすべて実行することが要求されたり，できるだけ全体が短時間で終了するようなスケジュールの作成が求められる．



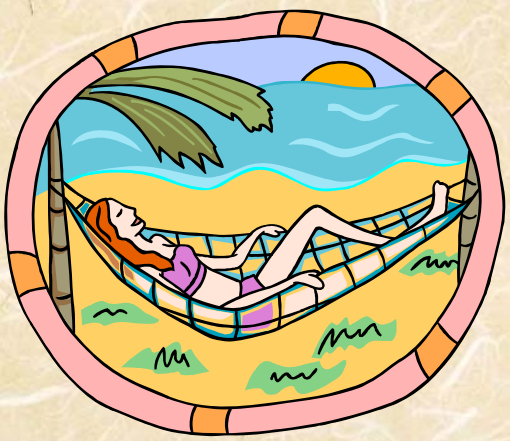
CSPは一般に**ＮＰ完全問題**と呼ばれる解くのが難しい部類の問題に属している．NP完全問題は，アルゴリズム理論における重要な概念であるが，ここでは一般的な説明を避け，CSPの場合に限定した簡単な説明をする．

CSPを，解が存在すればYES，存在しなければNOを出力する問題（**判定問題**）として考えよう．すると，YESの問題に対して，外部から解を教えてもらえれば，それが確かにYESであることを簡単に確認するアルゴリズムを作ることができる．YESであることは，教えられた解がすべての制約を満たすことを確認すればよい．また，「簡単に」の正確な意味は，変数の数や制約の数などで表現される問題のサイズに関して**多項式オーダー**の時間で確認できるという意味である．そのような問題を**ＮＰ問題**という．外部から解が与えられない場合，NP問題を解くのは難しかったり，易しかったりする．

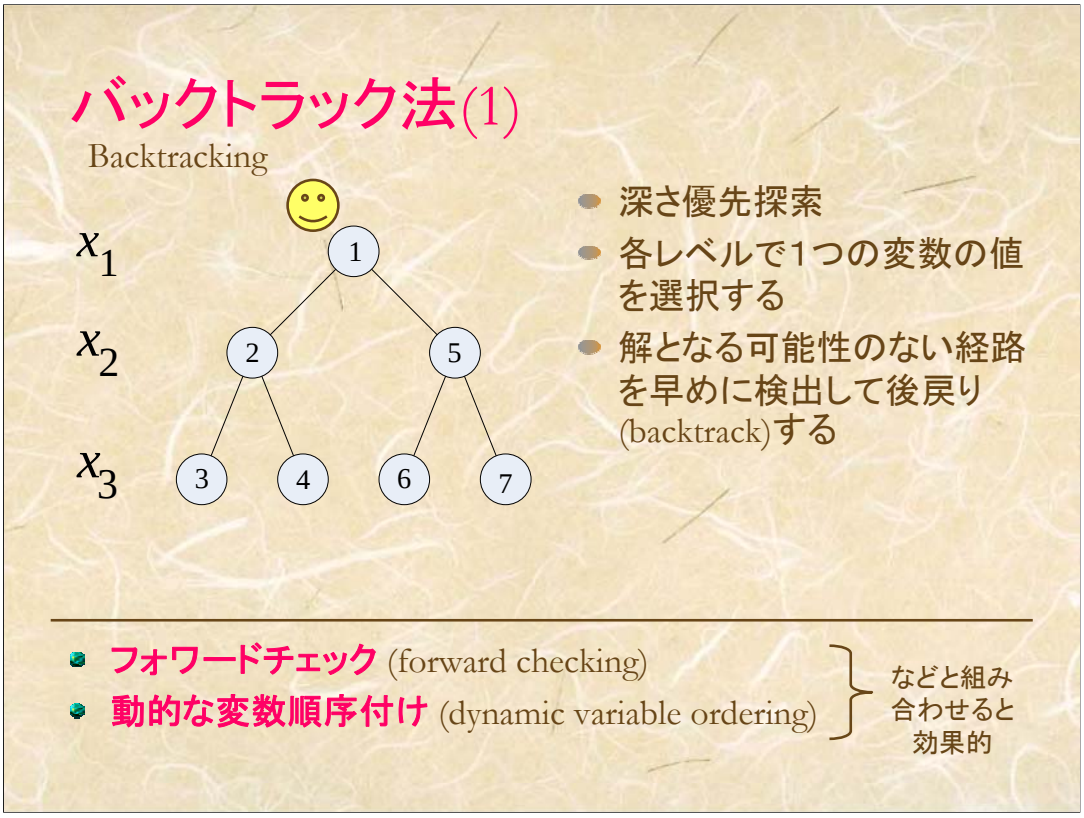
厳密な定義は避けておくが，**ＮＰ完全問題**とは，NP問題のうちで，ある意味で最も難しい（計算量が多い）問題群のことである．そのような問題を効率良く（多項式オーダーで）解くアルゴリズムは存在しないと考えられている．（ただし，証明はされていない．証明すれば，間違いなくノーベル賞級である．）したがって，一般には，CSPを解くには，最悪のケースで**指数オーダー**の計算量となるのは避けられない．

しかし，AIの立場は，問題を効率良く解くのをあきらめるのではなく，問題分野特有の**ヒューリスティックな知識**をうまく利用して，典型的には，あるいは，平均的には，実用的な時間で解くようなアルゴリズムを開発することである．CSPを解くソフトウェアもそのような観点から開発されている．

休憩



ここからは，CSPを解くアルゴリズムを学ぶ．そのアルゴリズムは基本的にはすでに学んだ探索アルゴリズムの一種であるが，CSP特有の問題設定を利用して，効率化がはかられている．このようなアルゴリズム（あるいはそれを実装したプログラム）は，制約解消器あるいは**制約ソルバー**(constraint solvers)と呼ばれることもある．

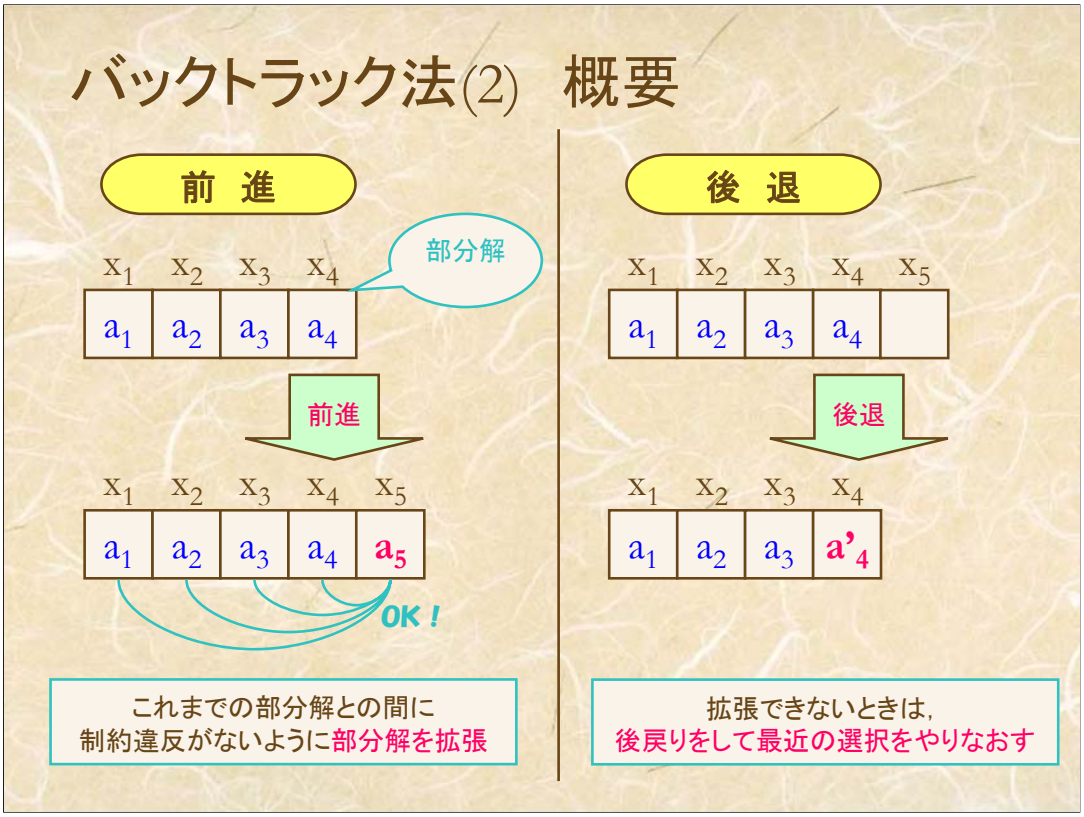


CSPを解くための最も基本的なアルゴリズムは**バックトラック法**(backtracking)である．これは基本的には深さ優先探索と同じであるが，CSPの場合には，つぎの2つの点でやや特殊化され，効率が改善されている．

(1) 各レベル毎に値を割当てるべき変数を1つに決めている．たとえば，第1レベルではx1に値を割り当てるとして決めてしまい，このレベルでx2などの他の変数に値を割り当てるような探索はしない．その理由は，CSPでは変数に値を割り当てる順序は任意だからである．変数がn個あれば，値を1つずつ割り当てていく順序はn！通りもあるのだが，そのうち1通りだけ試せば十分である．

(2) 解となる可能性のない経路を早めに検知して後戻りする．これは全変数に値を割り当てなくても，それ以前に部分的に割り当てた時点で1つでも制約違反が生じていれば，そこから先にはゴールがないことがわかるからである．

深さ優先探索自体が結構効率が良いことに加えて，上記の工夫により，バックトラック法はかなり効率的で実用的である．それでも難しいNP困難問題には歯がたたないこともある．そのようなときには，**フォワードチェック**や**動的変数順序**という工夫を組み合わせると効果的である．これについては後に説明する．

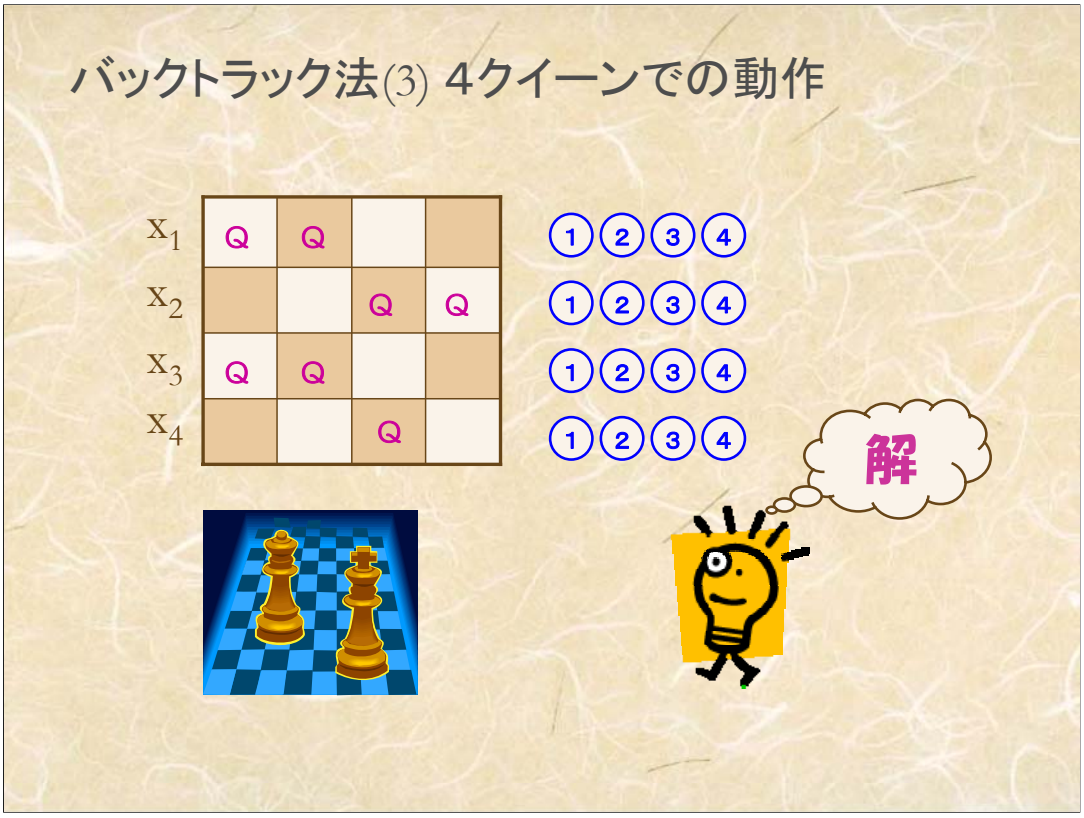


前のスライドでは，探索木を深さ優先でたどるという観点からバックトラック法の概要を説明したが，このスライドでは，もう少し細かく見ておく．

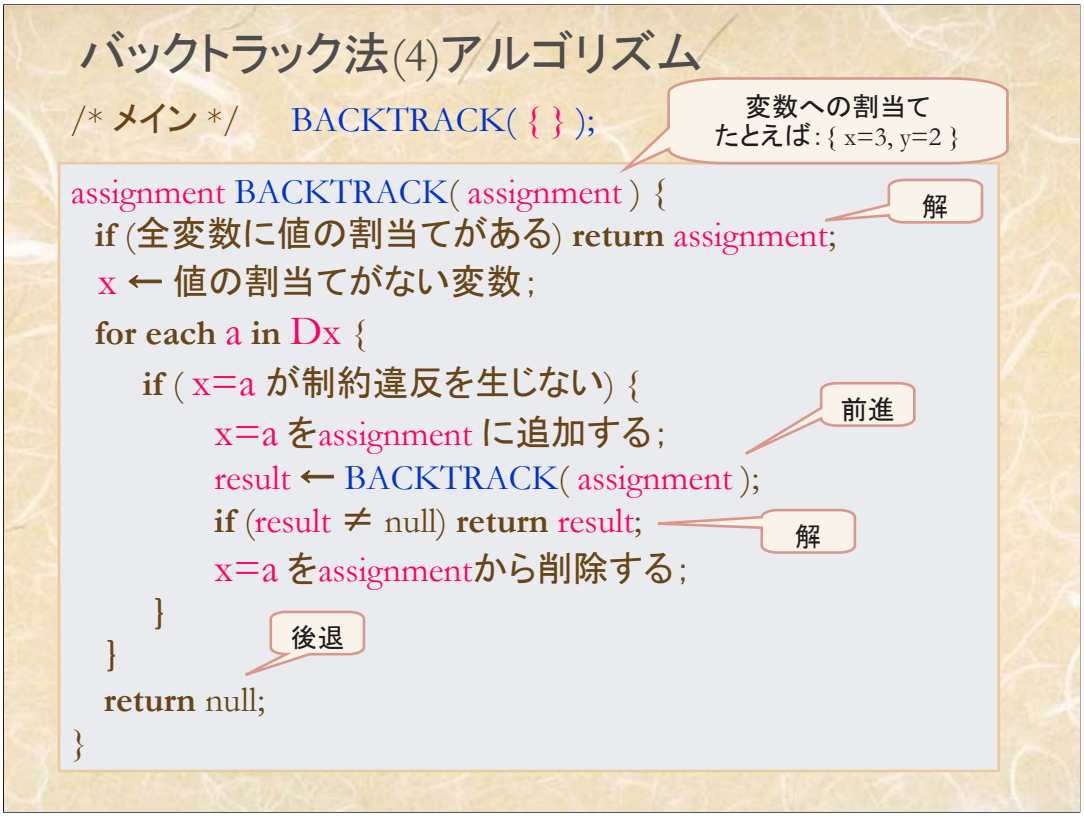
バックトラック法は，**前進**と**後退**の2つのステップを適切に織り交ぜて実行する．

前進ステップでは，すべての変数にまだ値が割り当てられていない初期状態から探索をスタートして，各変数に1つの値を**選択**して割り当てていく．その結果，一部の変数には値が割り当てられているが，他の変数には値が未設定という状態が生じるが，そのような部分的な割り当てのことを**部分解**という．前進ステップでは，現在の部分解との間に制約違反がないように1つの変数に値を1つ割り当てて，部分解を**拡張**する．この過程が最後まで続き，最終的にすべての変数に値を割り当てることができれば，その時点での部分解がCSPの解となる．

しかし，変数にどの値を割り当てても制約違反となり，それ以上前進できない**行き止まり**(dead end)と呼ばれる状態においては，**後退ステップ**を実行することになる．このステップでは，1つ前の変数の値を割り当てた時点に後戻りをして選択をやりなおす．すなわち，その変数にこれまで割り当てたのとは別な値を割り当て直して，再度，前進ステップを試みる．



バックトラック法を4クイーン問題に適用したときの動作をアニメーションで表現している.



バックトラック法のアルゴリズムの概略である.

assignment は変数への割当て(部分解)を表すデータで, ここでは {x=3, y=2} などと抽象的に書いておくが, 実際のプログラミングでは, 配列などを用いて, 適切に設計されているとする.

メインプログラムは, 再帰的な手続き BACKTRACK に空の割当て { } を渡して実行するだけである.

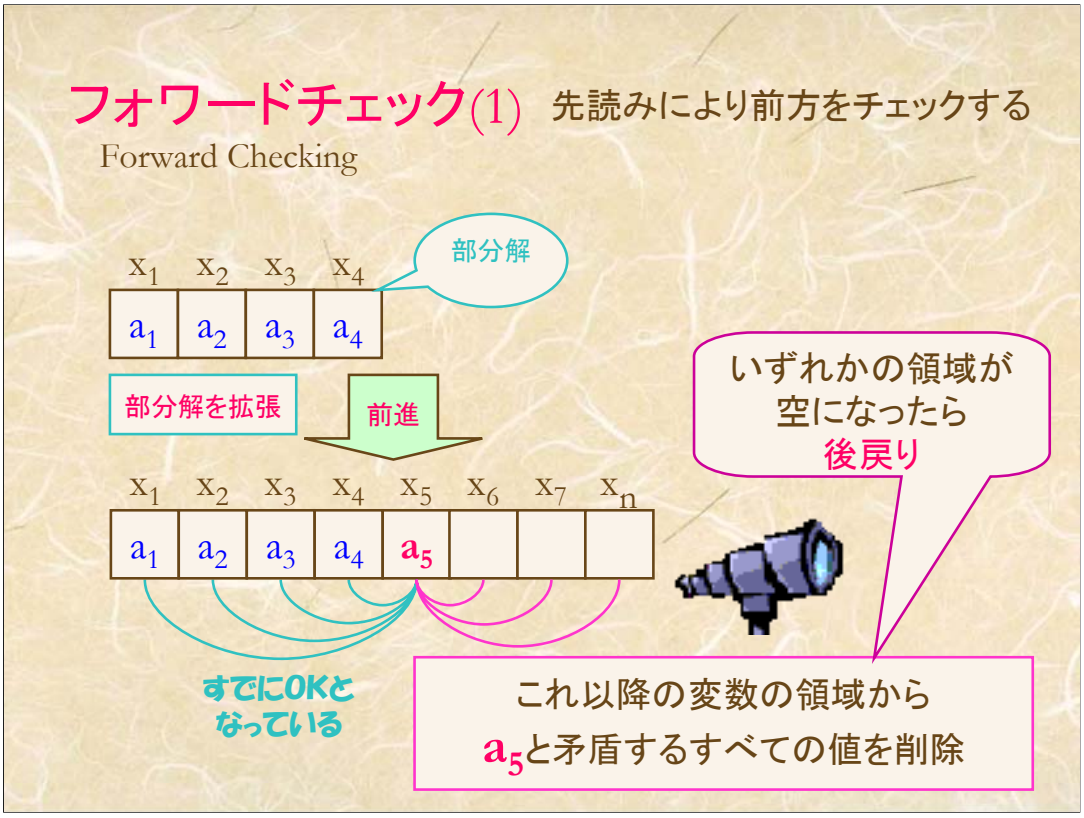
BACKTRACK は, 引数 assignment で与えられた部分解をありとあらゆる方法で拡張して解を探す責任をもっている手続きである. 解を見つけたらそれを返し, 解が見つからなければ「**失敗**」を表す特別な値 null を返すものとする.

その具体的な処理手順をみておこう. まず, 部分解 assignment を受け取ると, まだ値の割当てがない変数 x を1つ選び, それに x の領域 Dx から制約違反を生じないように選んだ値 a を割当てて部分解を拡張し, そこから先の問題を再帰的に BACKTRACK に丸投げして解いてもらう.

その結果, 解が見つかったら, それをそのまま戻して終了する.

解が見つからなかったら, 部分解のうちの x=a を削除し, for ループの機能を利用して, x に別な a ∈Dx を割当てて部分解を拡張する処理を繰り返す.

x に領域 Dx のどの値を割当てても解を見つけることができなければ, null を返すことによって自分の責任範囲の探索を終了する. これが上位レベル(つまり, xより1つ前の変数の処理)へのバックトラック(後戻り)になっている.



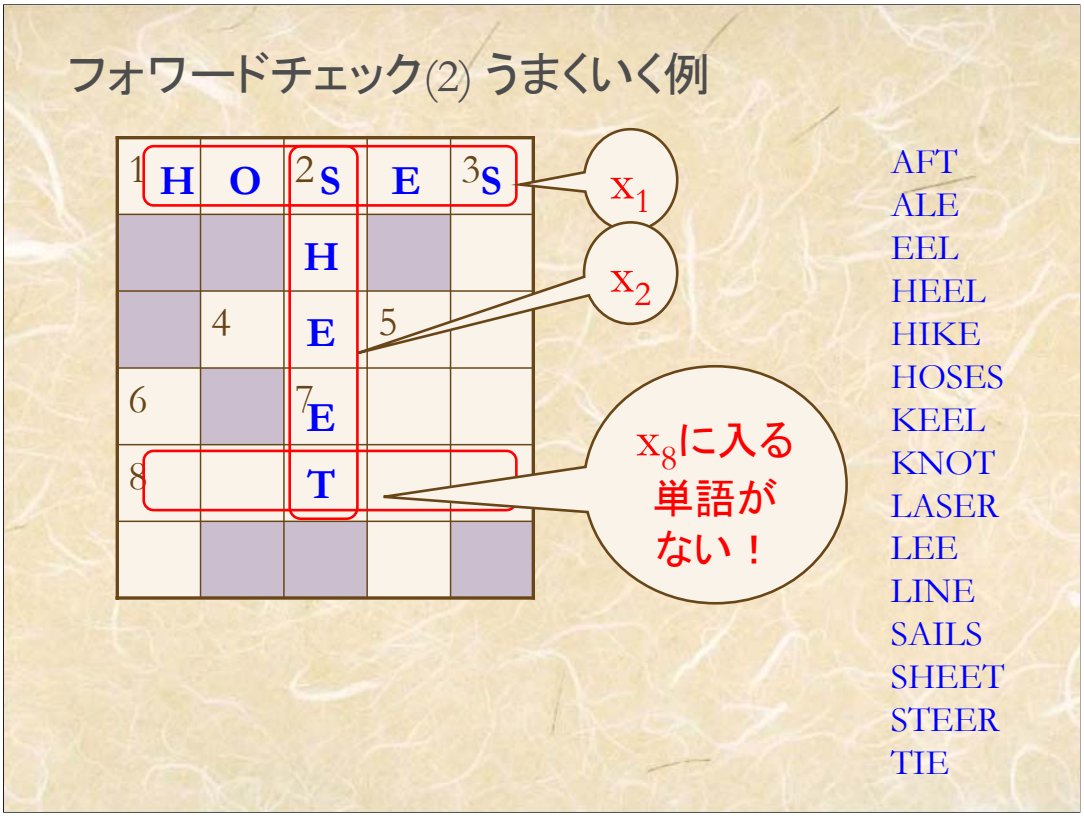
フォワードチェック(forward checking)は，バックトラック法と組み合わされて使用されるテクニックである．これは，先読みによって前方（まだ値を割り当てていない変数群）をチェックし，早い時点で失敗を検知して後戻りをしようと意図している．

スライドの図は，いま変数 x5 に値 a5 を割り当てようとしている状況を表している．以下では x5, a5 を一般的に x, a として考え方を説明する．

変数 x に値 a を与えると，その先の各変数の領域から，この x=a と矛盾するすべての値を削除する．そのとき，いずれかの領域が空になったら，それ以上前進しても解には到達できず，x=a は失敗だったことがわかるので後戻りする．そして，x には別な値を割り当てて再度前進することとする．

一方，その先のどの変数の領域も空にならなかったときは，解を発見するチャンスが残されているので，前進することにする．

このような事前の制約チェックの結果，x=a を選んだときには，すでに値を与えた変数との間で制約違反が決して生じていないことが保証されていることに注意しよう．バックトラック法はすでに決めた値と矛盾しないように制約を後ろ向きに（過去との関係で）チェックしているのに対し，フォワードチェックは今回決めようとしている値と矛盾しないように制約を前向きに（将来との関係で）チェックしているといえる．その意味で，前者は **look back**，後者は **look forward** という語句で特徴付けられることもある．



この例では，x1,x2 に単語を選んだ時点で，x8 に入る単語がない．フォワードチェックは，x8 の領域が空になることによって，このことをこの時点で検出できる．

ふつうのバックトラック法では，これができず，アルゴリズムは x3,x4,x5,x6,x7 に割り当てるべき単語の組合せを無駄に探索してしまう．

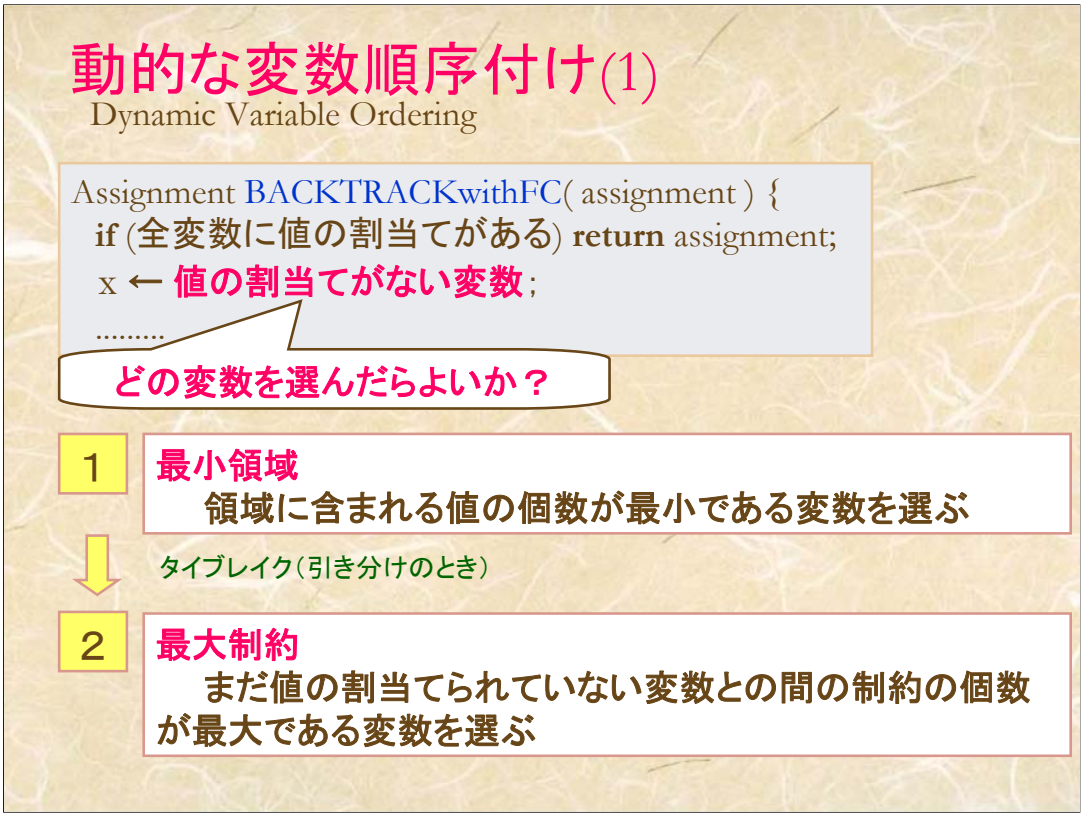
フォワードチェック(3)アルゴリズム

```
assignment BACKTRACKwithFC( assignment ) {
  if (全変数に値の割当てがある) return assignment;
  x ← 値の割当てがない変数;
  for each a in Dx {
      x=a をassignment に追加する;
      フォワードチェック;
      if (空の領域がない) {
          result ← BACKTRACKwithFC( assignment );
          if (result ≠ null) return result;
      }
      変数の領域をフォワードチェック前の状態に戻す;
      x=a をassignmentから削除する;
   }
   return null;
}
```

値の割当てがない変数の領域から x=a と矛盾する値を削除

バックトラック法にフォワードチェックを組み込んだのがこのアルゴリズムである.

x=a を assignment に追加することと, それに続くフォワードチェックはセットになった一体の処理なので, 後戻りする前にその両者の効果を打ち消す処理をおこなっている.



これまで見たアルゴリズムでは，値の割当てがない変数から1つを選んで x としている．そのような変数は一般に複数個あるのだが，アルゴリズムでは特にどれにするかは指定していない．つまり，どれでも良しとしている．

これまでの実行例では暗黙に，変数は x1,x2,x3,... の順番で選ぶことにしていたが，実際にはその必要はない．むしろ，これをうまく選ぶことにより，効率が大きく改善されることが多いのである．

では，どのような順番に変数を選んだらよいのだろうか？

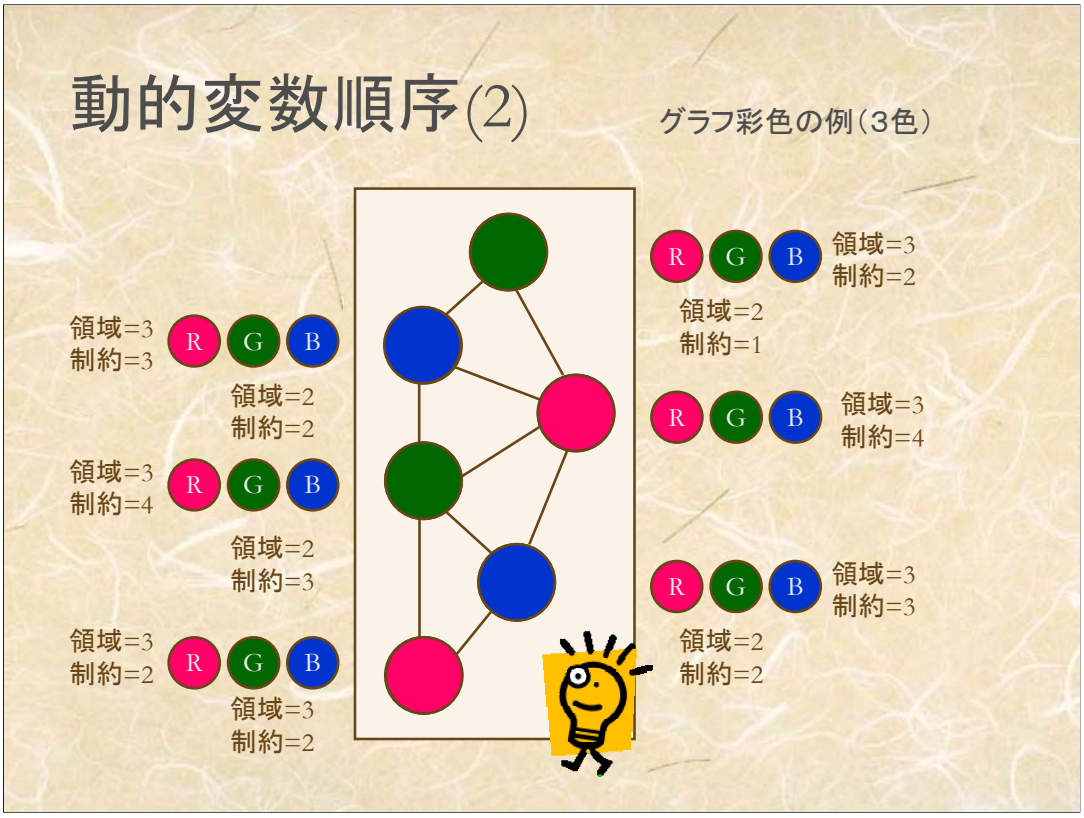
このように実行時に動的に変数の順序を決定する方法を，**動的な変数順序付け** (dynamic variable ordering)と呼んでいる．

お薦めは，つぎの作戦である．

1．**最小領域ヒューリスティック**：領域に含まれる値の個数が最小である変数を選ぶ．領域に含まれる値の個数が多いと，探索木の分岐の数がその数と同じだけ多くなり，探索木が大きくなりがちだからである．特に，領域に含まれる値がただ1個なら，選択の余地なくそれを選ぶしかないことを考えれば，このヒューリスティックの妥当性が納得できるだろう．

そのような変数が複数あるときには，つぎの方法でタイブレイク（引き分けを解消）する．

2．**最大制約ヒューリスティック**：まだ値の割当てられていない変数との間の制約の個数が最大である変数を選ぶ．そのような変数 x の値を決めれば，制約を通して x と結ばれる多くの変数の領域から，フォワードチェックが値を削除し，上記の1の基準をより良く満たす変数を作り出す可能性が高いからである．



バックトラック法(BT)＋フォワードチェック(FC)＋動的変数順序(DVO)の動作を，簡単なグラフ彩色問題(色の数＝3)で示している(アニメーション)．

実験による性能比較

Problem	BT	BT+DVO	BT+FC	BT +FC +DVO
USA	>1,000,000	>1,000,000	2,000	60
n-Queens	>40,000,000	13,500,000	40,000,000	817,000
Zebra	3,859,000	1,000	35,000	500
Random 1	415,000	3,000	26,000	2,000
Random 2	942,000	27,000	77,000	15,000

BT=backtracking FC=forward checking DVO=dynamic variable ordering
数値は制約のチェック回数

5種類のテスト問題を用いた実験によって，BT，FC，DVOの組合せの実行時間を評価している．

すべての問題において，BT＋FC＋DVOが最優秀である．