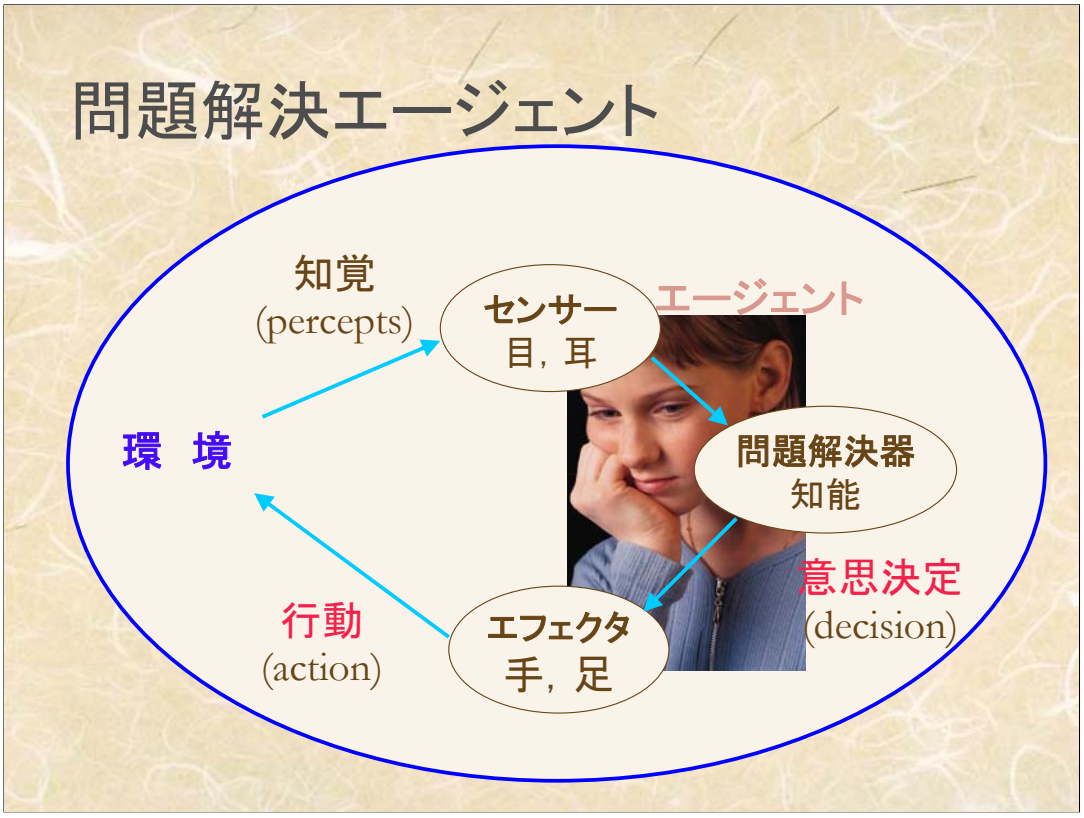
知能情報処理　探索（1）
先を読んで知的な行動を選択するエージェント

探索による問題解決

- 探索問題の定式化
- 探索問題の例
- 探索木とそのデータ構造
- 一般的探索アルゴリズム

人工知能のソフトウェアの特徴の1つは，与えられた問題の解を「**試行錯誤**」によって見つけるタイプのものが多いことである．その技術は「**探索**」(**search**) と呼ばれており，今回から数回にわたってその基本的な考え方とアルゴリズム，およびそれを実行するためのデータ構造の概要について学ぶ．



人間に模して設計された特定の目的をもつソフトウェアを「**エージェント**」と呼ぶことがある．

エージェントは，**センサー**によって自分の周りの環境の状態を**知覚**し，それに応じて**問題解決器**と呼ばれる**自動意思決定**ソフトウェアモジュールによって，自分のとるべき行動を決定する．そして，**エフェクタ**を通して**行動**し，結果的に環境の状態を変えることになる．

人間でいえば，センサーは目や耳，問題解決器は知能，エフェクタは手や足に相当するはたらきをすることになる．

これから数回にわたって取り扱う**探索問題**とその**解**とは何かを厳密に定義する．

このスライドはその入門的な図である．

探索問題とは，**状態(state)**と呼ばれるものの集合と**行為（行動,action）**と呼ばれるものの集合を具体的に定めることによって規定される．エージェントは常にいずれかの状態にあり，いずれかの行為を実行することによって，他の状態に移行することができる．どの行為を実行すればどの状態に移行するかは事前に問題の一部として与えられるものとする．

それに加えて，**初期状態**と呼ばれる状態がただ1つ指定され，**ゴール**と呼ばれる状態が任意個指定されるものとする．

そこで問題は，最初，エージェントが初期状態にあるとして，その後どのような行為を次々と実行すればゴール状態に達するかということである．そのような行為の列をその探索問題の**解(solution)**という．

探索問題は，この図のように**有向グラフ**で表すこともできる．状態を**ノード**で表す．また，状態sで行為Aを実行したときに状態 t になることを，ノードsからノード t への**有向辺**で表し，この辺への**ラベル**としてAを付しておく．探索問題の解は，初期状態ノードからゴールノードへの**経路**である．

探索問題に**経路コスト**を導入することもある．たとえば，行為の実行回数（経路の長さ）をコストとし，なるべくコストの小さな解を求めることを要求してもよい．

探索問題の定式化

探索問題とは以下の4つの情報の集まりである

- **初期状態**
- **オペレータ(行為)**
- **ゴール検査(アルゴリズム)**
- **経路コスト**

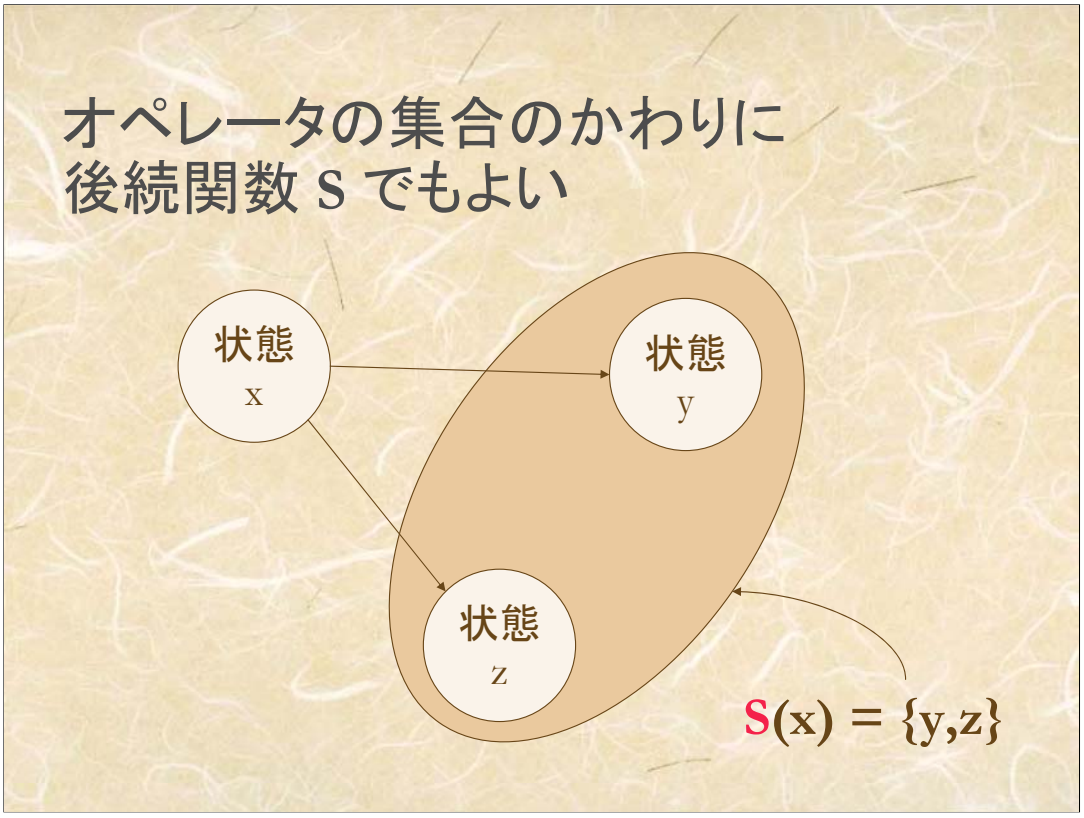
探索問題を正式に定義したのがこのスライドである.

行為のことを, ここでは**オペレータ**と呼んでいる.

ゴールは無数にあるかもしれないので, 1つ1つ列挙できないかもしれない. そこで, ここでは, 任意の状態が与えられたとき, それがゴールかどうかを検査する**ゴール検査**手続き(アルゴリズム)によって, ゴールを指定している.

オペレータは，数学的に，**状態遷移関数**と呼ばれることもある．

オペレータAを状態集合から状態集合への写像（関数）とみなし，スライドの図のようにy=A(x)の形で表現することができる．



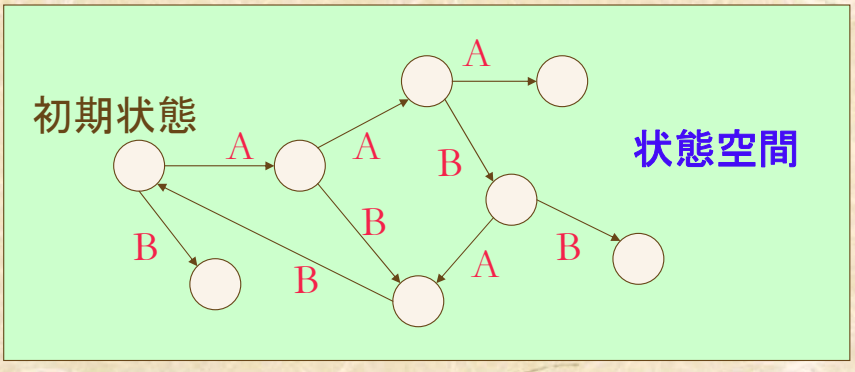
あるいは，オペレータを**後続状態**の集合を表す関数として数学的に表現することもできる．そのような関数に S という名前を付けることにしよう．この図の場合，状態 x から状態 y または z に遷移できるので，S(x)={y,z} である．

この方式の便利な点は，オペレータに明示的に名前を付けなくてよいことである．実際，この例では，2つのオペレータA,Bを用いて，y =A(x), z=B(x)とすることもできるのだが，後続状態集合を用いることによって，オペレータ名は表面に登場せず，Sという関数名だけで済んでいる．Sのことを**後続関数(successor function)**ということもある．

この場合，探索問題の解は，オペレータ名の列ではなく，状態の列ということになる．

状態空間

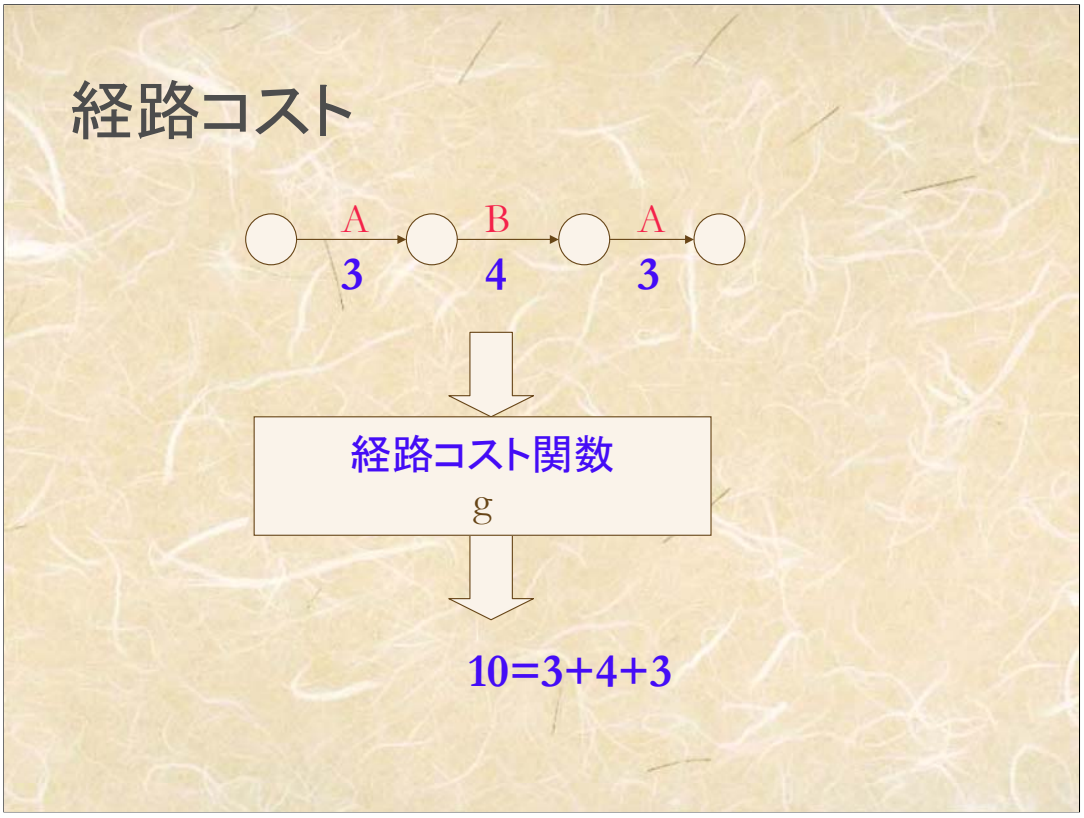
- 初期状態から到達可能なすべての状態の集合
- 初期状態とオペレータの集合から定義される
- 無限集合であってもよい



初期状態から到達可能なすべての状態の集合を**状態空間**という.

ゴール検査は，たとえば，GoalTest(x) というような関数として数学的にモデル化できる．

この関数は，状態 x を引数として受け取り，x がゴールなら真，そうでなければ偽を値として返すようにユーザが定義して，探索問題の記述の一部として与えるものである．



経路コストは，ふつう，経路を構成するそれぞれの有向辺のコストの和として与える．有向辺のコストは，たとえば，オペレータに依存して一意に決めるなどする．

この例では，オペレータAのコストは3，Bのコストは4としており，全体の経路コストは10となっている．

探索問題の定式化（再掲）

探索問題とは以下の4つの情報の集まりである

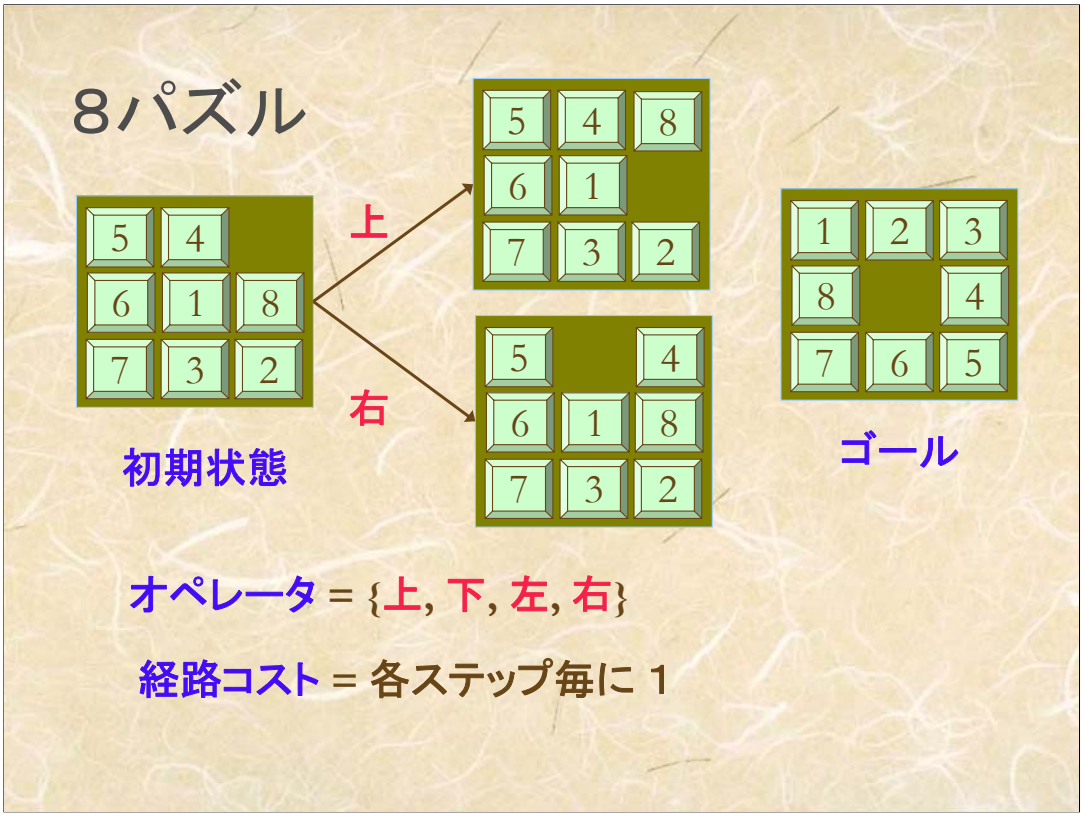
- **初期状態**
- **オペレータ**
- **ゴール検査**
- **経路コスト**

} **状態空間**

探索問題として定式化できる問題にはさまざまなものがある.

トイプロブレム(おもちゃの問題)は, 実社会ではあまり役に立たないが, 問題として簡潔でわかりやすく, 探索問題の本質を突いているものが多いので, さまざまな探索アルゴリズムの性能を実験的に評価するための標準的な例題として研究目的に使用されることが多い. この授業では, 後にこれらを1つずつ説明する.

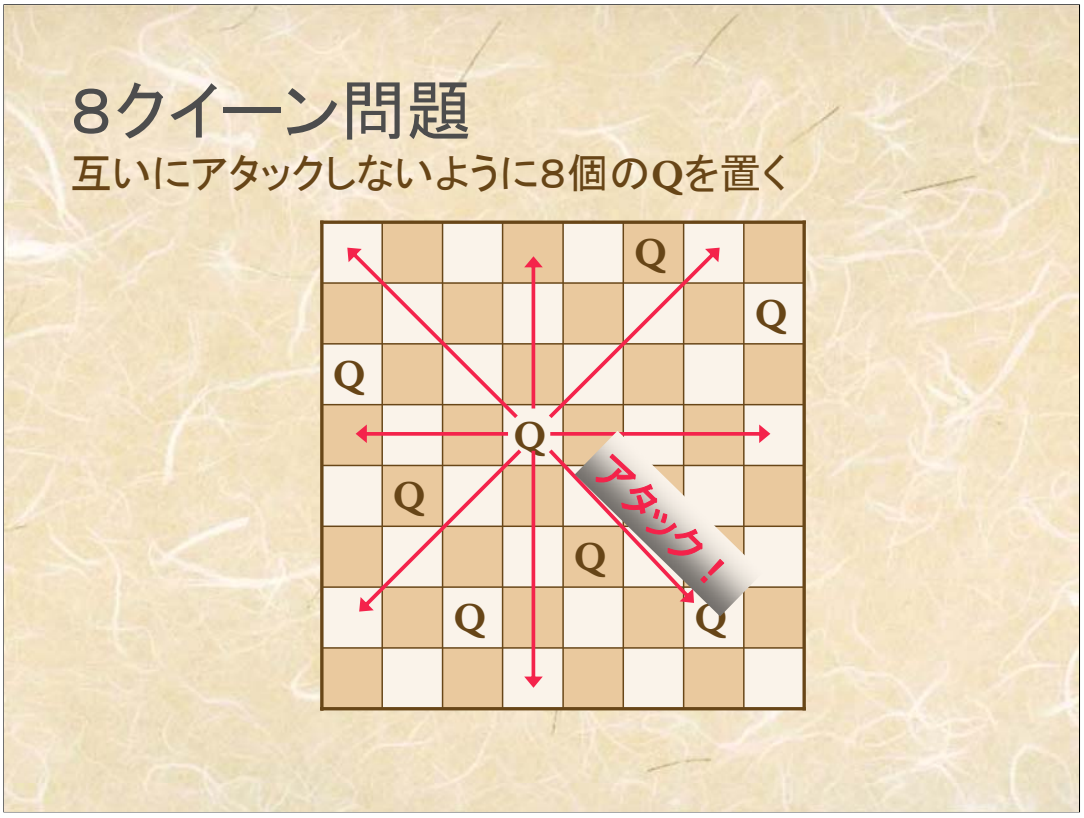
リアルワールド・プロブレム(現実世界の問題)は, 実用的に社会で役立っている問題の例である. この授業では, これらを1つずつ説明する時間はないが, カーナビで使用されている**ルート発見**などは最も身近に感じる探索問題の例であろう.



８パズルは，おもちゃ屋などでよく見かけるパズルである．ただし，市販されているのは，**１５パズル**と呼ばれていて，４×4のマス目に15個のピースである．8パズルは，教科書的な説明の都合上，それを一周り小さくしたものである．

これは人間なら小学生高学年程度で解ける簡単な問題である．しかし，最短手数で動かせといわれると，人間では解くのが難しい．

初心者的な簡単な探索アルゴリズムでは8パズルも15パズルも解くのが結構難しい．しかし，先進的な探索アルゴリズムでは，もっと大きなパズルをテスト問題として使っている．一般に，マス目をn×nとすれば，そのパズルはn^2-1パズルということになる．

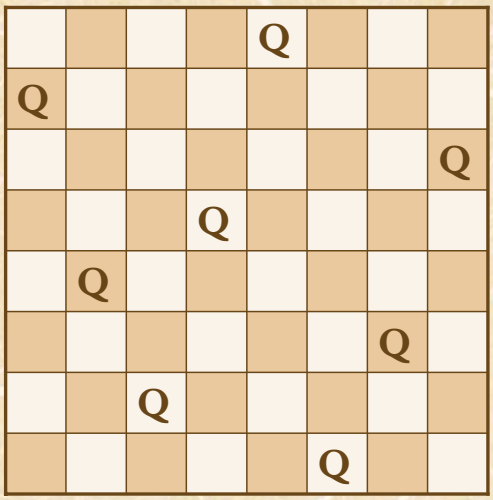


８クイーン問題は，チェスボードの上に，８個のクイーンを互いにアタックし合わないように置くパズルである．チェスを知らない人のために少し解説しておこう．クイーンは将棋の飛車と角を合わせた動きが可能である．つまり，前後，左右，斜めの８方向に限りなく進むことができる．その進路上に他のクイーンがあれば，この2つのクイーンは**アタック**し合っているという．この問題は8個のクイーンを互いにアタックし合わないように置くことを要求する．

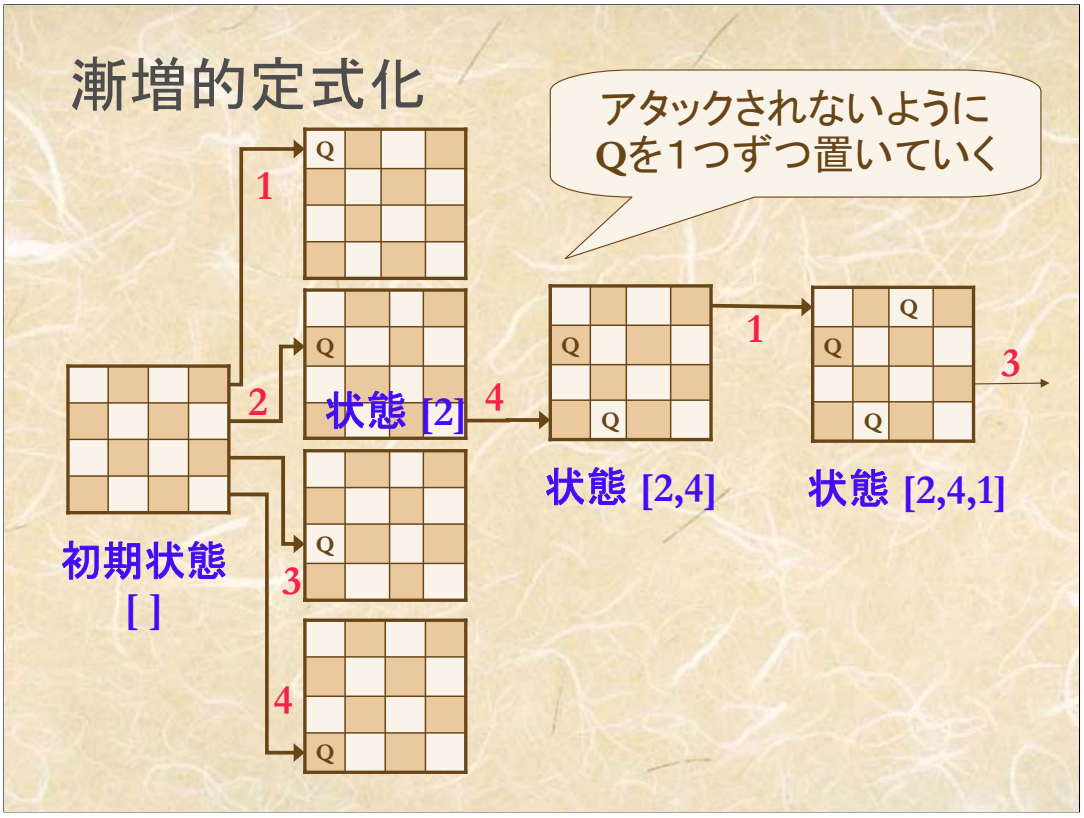
クイーンが8個なのは，伝統的にチェスボードが８×８だからだが，探索アルゴリズムのためのテスト問題としては，一般にｎ×ｎのボードにｎ個のクイーンを置く問題に拡張し，**ｎクイーン問題**と呼んでいる．

8クイーン問題の解

ゴール
（の1つ）



これが8クイーン問題の解の1つである.



さきほど定式化した「探索問題」として，4クイーン問題を定式化してみよう．

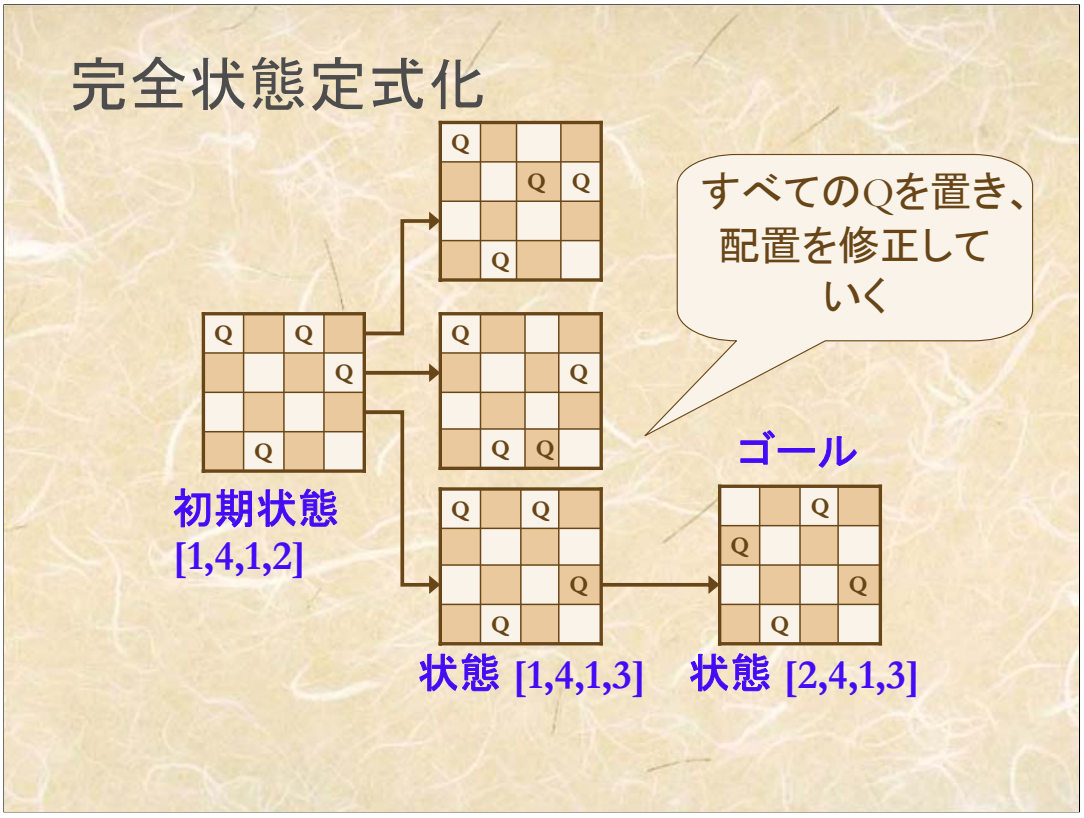
少なくとも2つの方法がある．

このスライドでは1つ目の方法として，**漸増的定式化**と呼ばれるものを示している．

初期状態をクイーンが1個も盤面にない状態とする．

左の列から逐次的に1個ずつ右の列に向かってクイーンを置いていくオペレーションを考えることとする．上から i 番目の行にクイーンを置くオペレータを i とする．i =1,2,3,4 の4つのオペレータがある．それぞれのオペレータによって状態（盤面）がどう遷移するのかは自明である．

適用してきたオペレータの列 [2,4,1]で盤面の状態と経路の両方をコンピュータ向きに記述できる．この問題の場合，解は[2,4,1,3]となる．



完全状態定式化では，状態として，盤上に必ず4個のクイーンが置いてある盤面だけを考える．そのうちの任意の1つを初期状態とする．

オペレータとして，アタックされている任意のクイーンを1つ選び，同じ列の中の他の行に移動するものをいろいろ考える．

つまり，まずはクイーンをランダムに配置し，その後，アタックの数ができるだけ少なくなるように，局所的な改善を少しずつ繰り返して解を探索する考え方の定式化である．

覆面算も昔からよく知られたパズルである.
これもやはり漸増的定式化と完全状態定式化が可能なのは自明である.

掃除機ロボットを考える.

簡単のため, 部屋がいくつかのマス目に区切られていて, 各マス目にはゴミがあるかないかのどちらかの状態で区別できるとしよう.

探索問題の状態は, 各部屋のゴミの有無, ロボットの位置などを総合的に表現した構造データとなる.

オペレータとしては, ロボットを隣のマス目に移動させるオペレーション, ゴミを吸い込むオペレーションなどがある.

もちろん, ゴールはすべてのマス目からゴミを除去した状態に遷移することである.

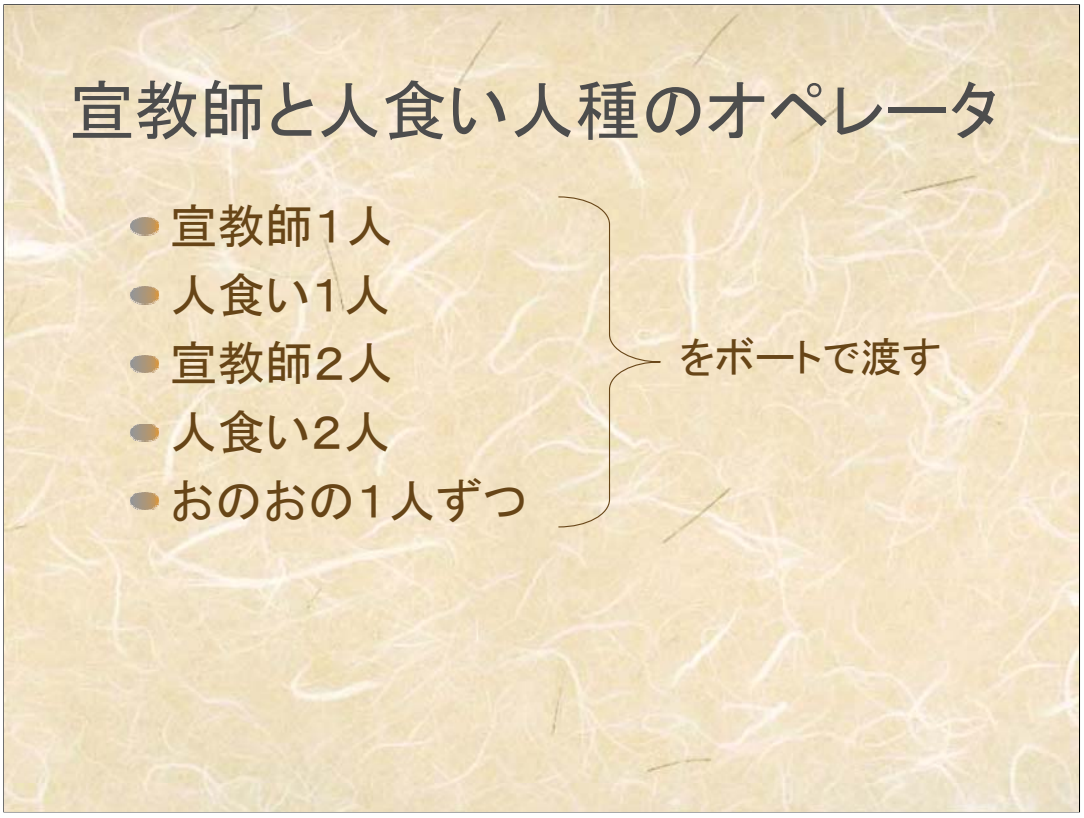
「**宣教師と人食い人種**」というパズルがある．（ただし，「人食い人種」という言葉が人種差別的に感じられる場合には，パズルの本質を変えずに別な登場人物とすることもある．）

川の西岸に3人の宣教師と人食い人種がいる．2人乗りの1台のボートを使って全員を東岸に渡すにはどうすればよいだろうか．ただし，西岸，東岸，ボート内のいずれにおいても，宣教師数が人食い人種の数を下回ると，宣教師は人食い人種に食べられてしまう．もちろん，食べられずに全員が無事に東岸に渡りたいのだが．．．

これを探索問題として定式化するために，状態として，[3,2,西]のような構造データを導入する．

最初の数字は西岸の宣教師の数，中央の数字は西岸の人食い人種の数，右端の文字はボートが東西のどちらに位置しているかを表す．

初期状態は [3,3,西]，ゴールは[0,0,東]または[0,0,西]である．

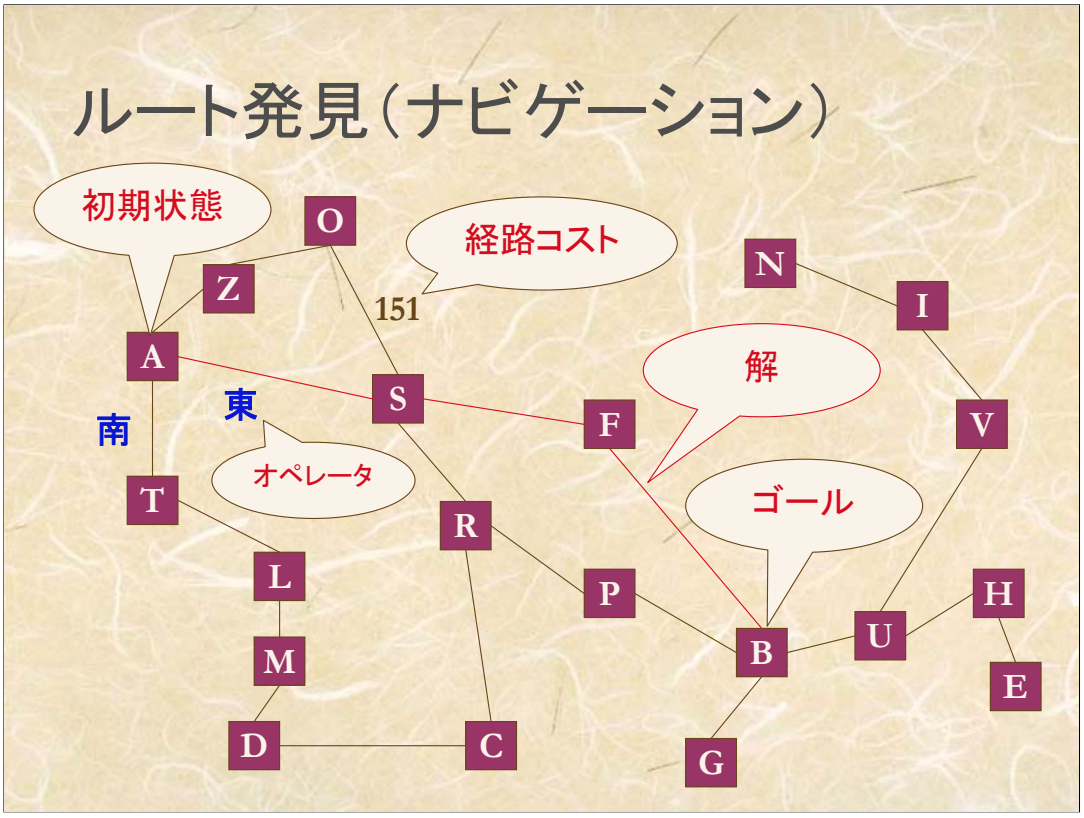


オペレータとして，このスライドの5つを考える．各オペレータによって，状態がどう遷移するかは自明なので省略する．

実は，ここで人間の知能のすばらしさにふと気付く．

この定式化の際に，状態としては，宣教師や人食い人種の数だけが問題で，個人を区別する必要はないと判断している．これは我々通常の知能の持ち主にとっては極めて自明なことと思われる．宣教師に，エドワード，ミカエル，イサクなどと名前を付け，西岸に誰々がいるかによって状態を区別する必要がないことは自明に思われる．

しかし，それはコンピュータにとっては極めて難しい判断である．こういうところは今のところ人間が定式化するしかないと考えられている．



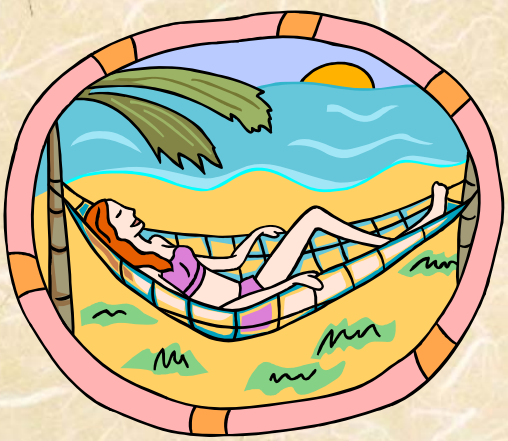
探索問題をもっともわかりやすく表現していて，しかも，現実世界で応用されているのが，**ルート発見**あるいは**ナビゲーション**の問題である．

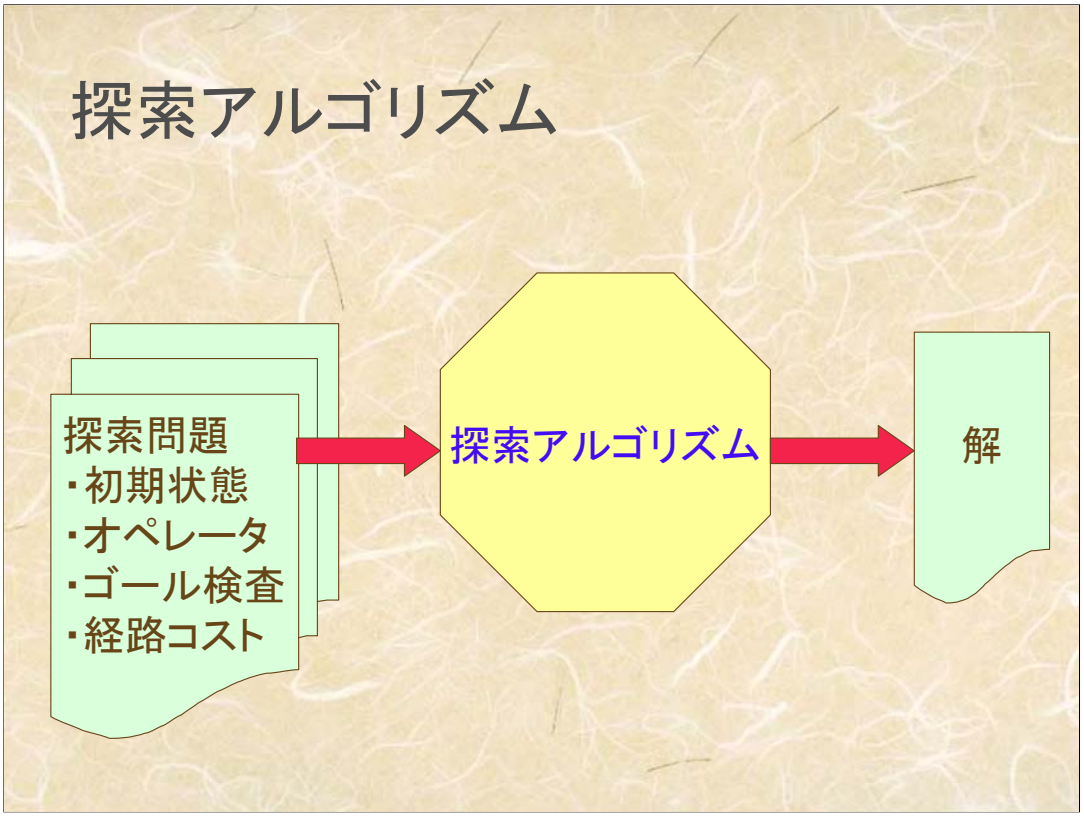
その例として，このスライドのようなものを考えよう．これは参考文献（S. Russell, P. Norvig著の教科書）にある例で，ブルガリア（という読者の大部分にとっては行ったことのない国）の地図の簡略版を抽象化し，「都市Aから都市Bまでの経路を見つけよ」というナビゲーションの問題である．（ただし，この図では都市名はその頭文字のアルファベット1文字で表示している．その頭文字が重複しない点でこの例はよくできている．）

もちろん，コンピュータには都市間の隣接関係だけがオペレータあるいは後続関数として与えられるものなので，人間のようにこの図を一目見て，解を瞬時に見つけるということはできない．隣の都市を順々にたどりながら，試行錯誤的に経路を見つける探索アルゴリズムを構成する必要がある．

また，この問題の場合は，隣接する都市間の距離がコストとして与えられている．初期状態からゴールまでの経路コスト（経路を構成する辺のコストの総和）がなるべく小さい解を探すことが求められる．

休憩





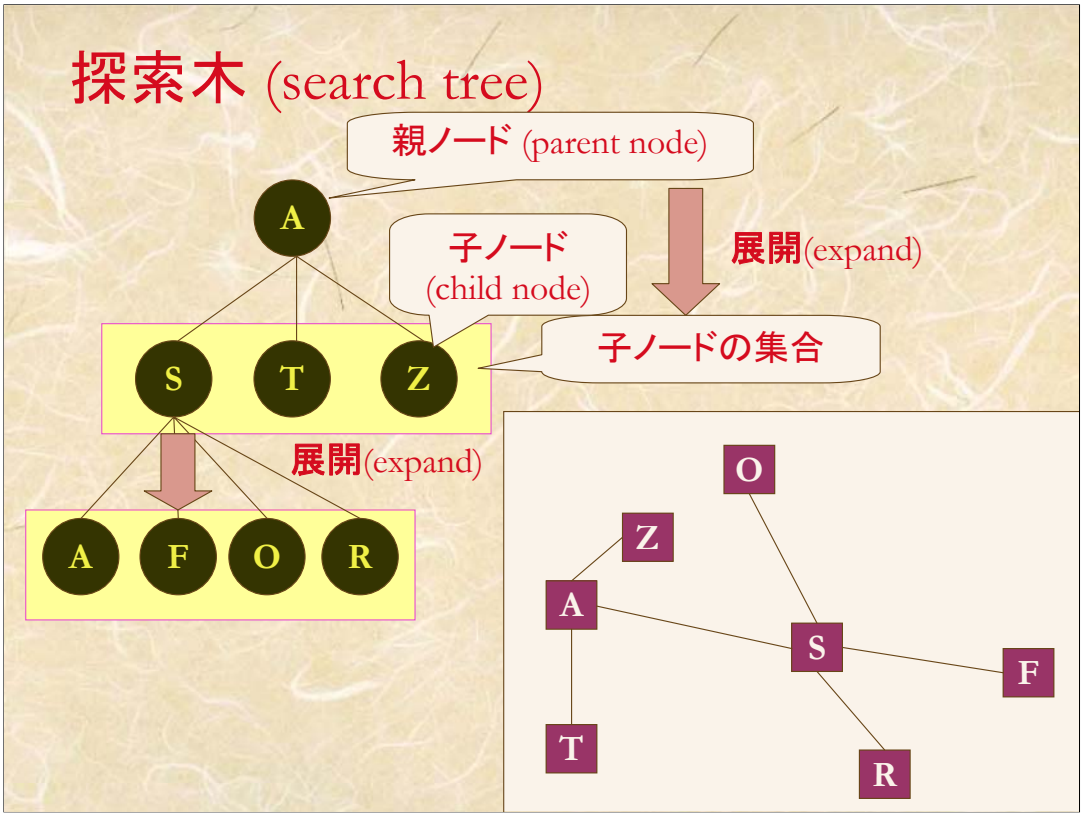
すでに定式化されたような任意の探索問題が与えられたときに，その解を求めるアルゴリズムを**探索アルゴリズム**という．ここからは探索アルゴリズムについて，一般的な考え方を学ぼう．

まず重要なことは，探索アルゴリズムが1つあれば，それを適切な形でプログラムとして作成しておくと，それでどのような探索問題でも解けるということである．8パズル用のプログラム，8クイーン用のプログラム，...などと個別のプログラムを別々に作成する必要はない．一般的な探索問題を解く1つのプログラムを作成しておけば，個別の問題はそのプログラムへの入力を変えたり，せいぜい，それを多少カスタマイズする程度で解くことができる．つまり，探索アルゴリズムはかなり広い範囲の問題を解決できることになるので，歴史的に，この技術は**一般的な問題解決(general problem solving)**を行うための知能の基盤となる技術と考えられてきたのである．

実際，高度なプログラミング言語を使えば，そのようなプログラムを書くことは容易である．

たとえば，人工知能用の言語として古くから開発されている**Lisp**では，構造的なデータを表現するための「リスト」や，手続きを表現するための「関数」が基本データ型（「第1級のオブジェクト」とも呼ばれる）として用意されていて，それらを手続きの引数として渡したり，戻り値として戻したり，入出力の対象として自由に操作することができる．したがって，探索問題の一部に含まれる「ゴール検査」というアルゴリズムのプログラムコードさえも関数として外部から実行時に入力できるのである．

また，現代的なプログラミング言語である**Java**などのオブジェクト指向言語では，探索問題を記述するための「状態」，「オペレータ」，「ゴール検査」，「経路コスト」などを**クラス**あるいは**インタフェース**と呼ばれる機能を用いて**抽象データ型**として定義し，それを用いた探索プログラムを作成しておくことができる．個別の探索問題を解くときには，一般的な探索プログラムを変更しなくても，個別問題の抽象データ型を具体的に**実装**してコンパイルしておけば，実行時に動的にリンクして実行することができる．



探索アルゴリズムのポイントは，**探索木**と呼ばれるデータ構造（**木**）を用いて，探索をコントロールする点にある．以下の説明では，「**グラフ理論**」の基本用語に関する知識を前提とする．

探索木の**ノード**（頂点）は1つの状態を表す．このスライドは，ルーマニアのルート発見問題の場合を使った例で，ノードは1つの都市を表している．

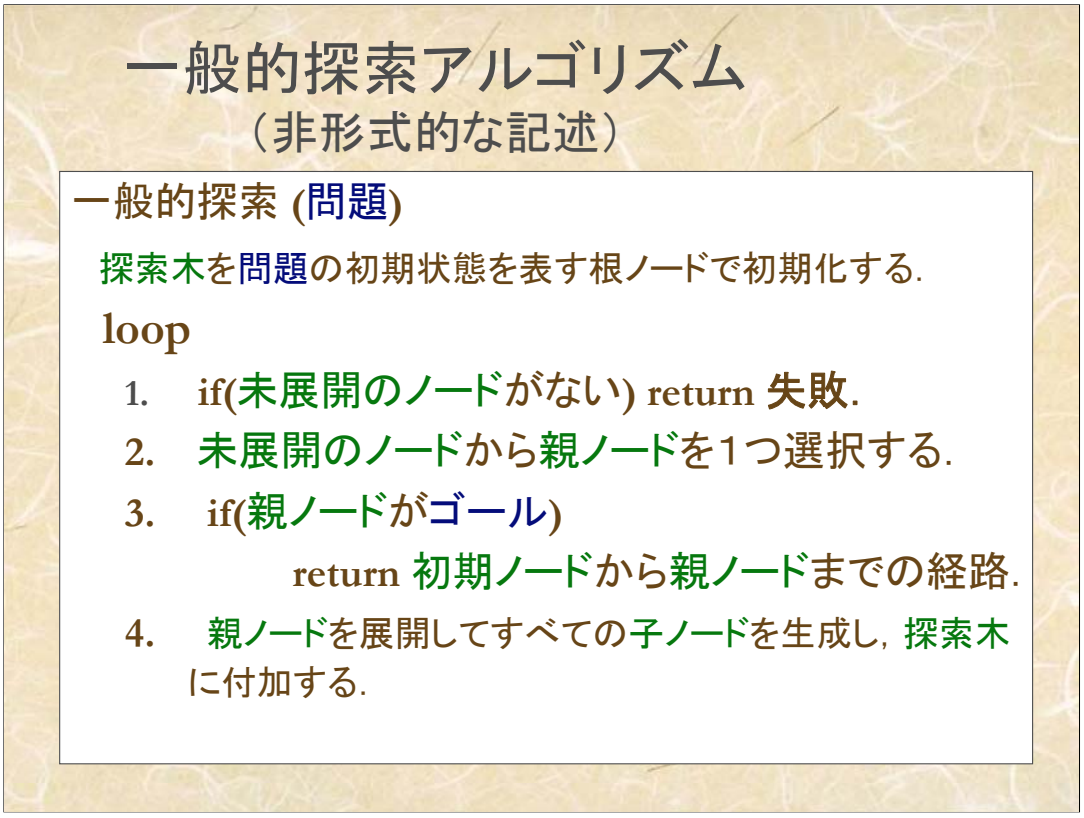
探索アルゴリズムの基本動作は簡単なものである．

まず，アルゴリズムの実行開始時点で，初期状態を表すノードを生成し，探索木の**根ノード**とする．

つぎに，アルゴリズムは，その根ノードに対して適用可能な**オペレータ**をすべて適用してすべての**後続状態**を生成し，その1つ1つを表すノードを生成する．そして，根ノードを**親ノード**，生成された各ノードを**子ノード**とする親子関係を表す**辺**で結ぶ．

今度は，いま生成された（一般に複数の）子ノードのそれぞれが根ノードの役割を果たして同様な処理を続ける．すなわち，一般に，アルゴリズムは探索木の先端（**葉ノード**）のどれか1つを親ノードとして選び，そのノードが表す状態に対して適用可能なオペレータをすべて適用してすべての後続状態を生成し，その1つ1つを表す子ノードを生成し，親と子を辺で結ぶことによって，**探索木を成長**させていく．

親ノードから子ノードの集合を作る操作を**展開(expand)**という．アルゴリズムの基本動作は，このようにノードを次々と展開して探索木を成長させていき，ゴールノードが生成されるのを待つことである．



探索アルゴリズムの一般的な考え方を手順として記述したのがこのスライドである.

初期設定として，初期状態を表すノードを生成して，**探索木**の唯一のノードとする．これが今後成長する探索木の**根ノード**となる.

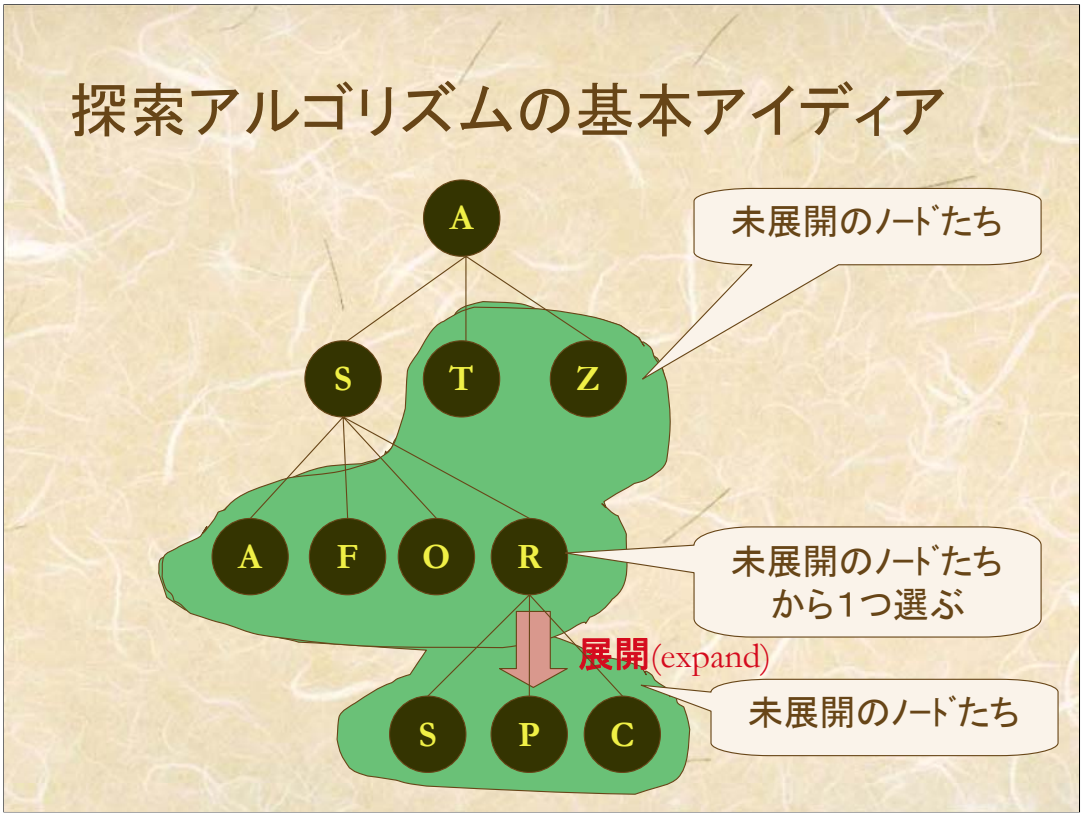
探索アルゴリズムの中心はつぎのようなことを繰り返すループの処理である.

1. まず，（まだゴールが見つかっていないときに）**未展開のノード**がない（つまり，探索木に含まれるすべてのノードを展開しつくした）ということになれば，探索は「**失敗**」となり，終了する.
2. **未展開のノード**が1つでもあれば，それらから任意の1つを選び，**親ノード**とする.
3. いま選んだ**親ノード**がゴール検査によって**ゴール**であると判明したら，このアルゴリズムは終了である．そのゴールから探索木を上へ上へ（親へ親へ）と根ノード（初期状態ノード）までたどり，その経路の逆順を出力として返す．これが探索問題の**解**となる.
4. **親ノード**がゴールでなければ，そのノードを**展開**してすべての**子ノード**を生成し，探索木に付加する.

このような処理を繰り返すのが探索アルゴリズムの基本である.

途中で解がわかればそれを返すし，解に至る経路がないとわかれば「失敗」という情報を返す.

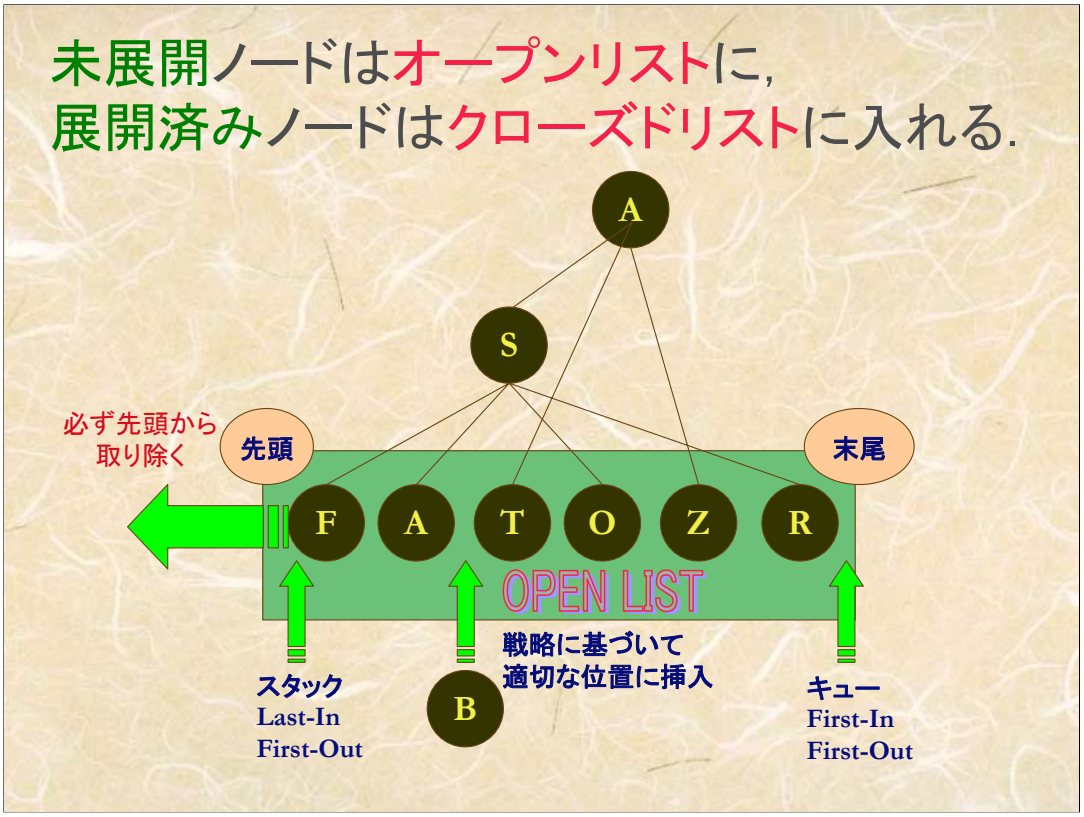
また，いつまでも解を探し続けて，探索木が限りなく大きくなり続け，アルゴリズムが停止せずに動き続けることもあり得る.



このアルゴリズムを効果的に実行するために，**未展開ノード**をひとまとめに集めて集合として保持しておくのがよい．

それらの中から**親ノード**を選んで展開する．展開された親ノードはその集合から取り除き，かわりに，生まれてきた子ノードをその集合に入れる．

このスライドは，ノードRを展開する前後の様子をアニメーションで示している．（印刷媒体ではその雰囲気は伝わらないので注意．）



探索木に含まれるノードは**未展開**か**展開済み**かのいずれかである．未展開であることを**オープン**(open)，展開済みであることを**クローズド**(closed)という英語の形容詞で表すことにしよう．これら2種類のノードは仕分けして，別々の集合に保持するのがよい．また，これらのノードをそれぞれ1列に並べて保持しておくことにして，未展開ノードの列を**オープンリスト**(open list)，展開済みノードの列を**クローズドリスト**(closed list)と呼ぶ．プログラミングの際には，これらのリストは配列を用いれば実現が容易である．また，高度なプログラミング技法として，**リンク**（ポインター）を用いた動的な**リスト構造**を使うこともできる．今後，アルゴリズムの記述の中では，これらのリストをそれぞれ OPEN および CLOSED という名前で表すことにする．

このスライドは探索アルゴリズムがオープンリストをどう扱うかを表現したものである．

オープンリストから**親ノード**を選ぶときには，その先頭から選ぶことにする．（そしてそれをオープンリストから取り除いてクローズドリストに移す．）

一方，生まれた**子ノード**をオープンリストに入れるときには注意が必要である．先頭に近いところに入れれば，今度アルゴリズムによって早めに親として選ばれることになるし，末尾の方だとなかなか親として選ばれる番に回ってこない．

実は，子ノードをオープンリストのどこに挿入するかに応じて，それぞれアルゴリズムに名前が付けられ，その理論的な性質や性能がかなり異なるものとなる．そこで，今のところは，その詳細は気にしないことにしておき，抽象的だが，**戦略(strategy)**に基づいて**適切な位置**に挿入するとだけ理解しておこう．

特別な場合として，子ノードを常に先頭に挿入するというのなら，このオープンリストは一般に**スタック(stack)**と呼ばれる**後入れ先出し**型(**Last-In First-Out, LIFO**)のデータ構造となる．また，子ノードを常に末尾に挿入するというなら，そのオープンリストは一般に**キュー(queue)**と呼ばれる**先入れ先出し**型(**First-In First-Out, FIFO**)のデータ構造となる．

一般的探索アルゴリズム

一般的探索 (初期状態，後続関数，ゴール検査)
OPENに初期状態を表す初期ノードだけを入れておく.
CLOSEDを空にしておく.
loop
1. if(OPENが空) return null.
2. OPENの先頭ノードをNODEとし，CLOSEDに移す.
3. if(ゴール検査（NODE ）) return NODE.
4. 子ノード集合←後続関数（ NODE ）.
5. for each 子ノード in 子ノード集合 do
   5-1. 子ノードとNODEを親子関係を表す辺で結ぶ.
   5-2. 子ノードをOPENに挿入する.

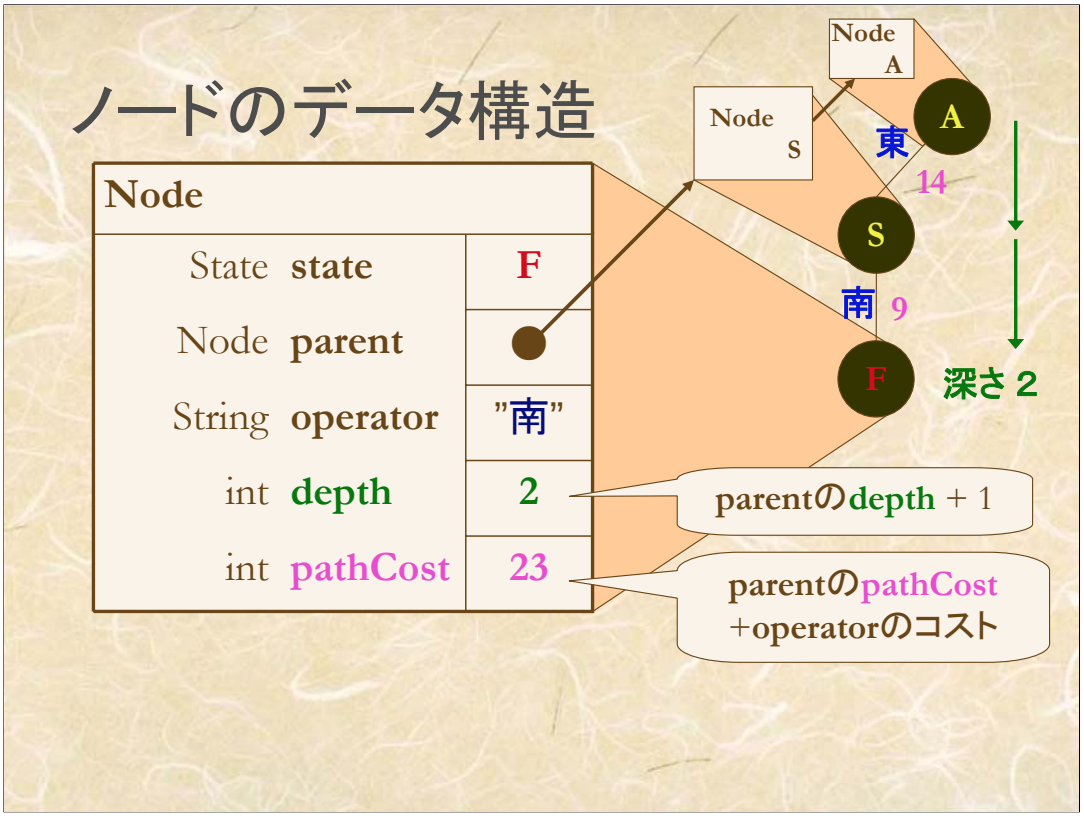
オープンリストを導入して，一般的探索アルゴリズムをもう少し詳細化したのがこのスライドである. **未展開ノード**がOPENに保持され，**展開済みノード**はCLOSEDに移される. 探索木という言葉がなくなっているが，そのかわりに探索木のノードはOPENとCLOSEDに分けて保持されている.

なお，このアルゴリズムでは，ゴールが見つかったら，そのノード（へのポインタ）をそのまま返すことにした. そのノードから根ノードまでたどって解を表す経路を生成する作業はこのアルゴリズムを呼び出した側のソフトウェアの責任とする.

「失敗」を表す情報として，多くのプログラミング言語で利用可能な **null** というポインタを借用した. また，

for each <変数> **in** <集合> **do**
　　<ループ本体>

という構文は，<集合>に含まれる要素を（重複なく）1つずつ<変数>に代入しては<ループ本体>を実行するという反復処理を表すものである. <集合>が具体的にどのようなデータ型で表されるかはプログラミング言語に依存するが，このアルゴリズムを理解する上では知る必要がない. たとえば，簡単に，1次元配列で表されると理解しておけばじゅうぶんである.



ノードは，このスライドのように，5つのフィールドからなる構造データ（プログラミング言語によって，**レコード**，**構造体**，**オブジェクト**などと呼ばれる）として表現できる．

state は**状態**の記述である．ルート発見の例題では都市名．

parent は**親ノード**がどのノードであるかの情報（ポインタ）．

operator は親ノードにどの**オペレータ**を適用してこのノードができたのかを示す情報．

depth は探索木中におけるこのノードの**深さ**（＝親の深さ＋1）．

pathCost は初期状態からこのノードまでの**経路コスト**（＝親までの経路コスト＋オペレータのコスト）．

このスライドの例は，ルート発見の例題で探索がA→S→Fと進んできたことを想定している．このノードは都市Fを表すもので，ノードSに「南へ進め」というオペレータによって生成されたもので，深さは2，経路コストは4であることを表している．