知能情報処理　探索(2)

先を読んで知的な行動を選択するエージェント

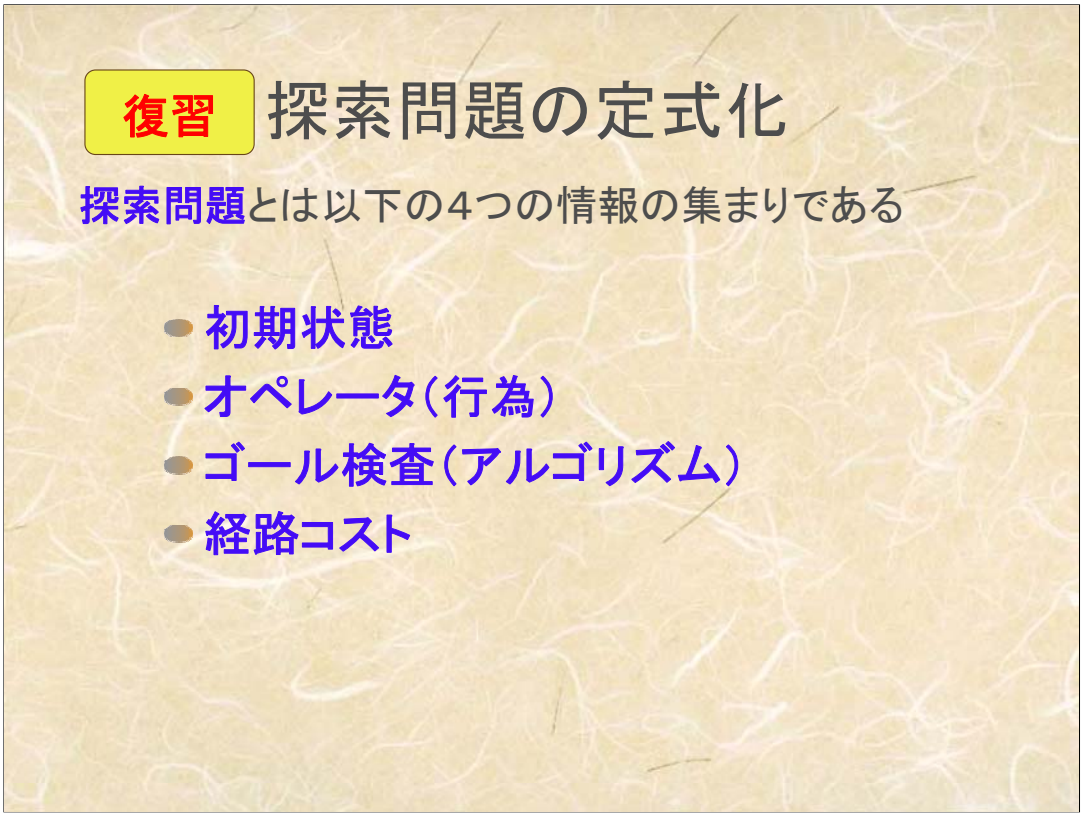
知識なしの探索
― しらみつぶしの探索 ―
(Uninformed Search)

- 探索戦略とその評価基準
- 幅優先探索(BFS)
- 深さ優先探索(DFS)
- 反復深化探索(ID)

前回の授業では，一般的な探索アルゴリズムを学んだが，今回はさらに具体的なアルゴリズムとして，**幅優先探索**，**深さ優先探索**，**反復深化探索**を学ぶ．

これらのアルゴリズムは，与えられた探索空間の性質について特別な知識がないことを前提としたもので，基本的に，ありとあらゆる可能性を**しらみつぶし**に探すものである．それでも，探す順序などの**戦略**が異なるので，アルゴリズムの性質がかなり異なっている．そこで，アルゴリズムの良さを評価するための**評価基準**を4つ決め，それで各アルゴリズムの良し悪しを評価する．

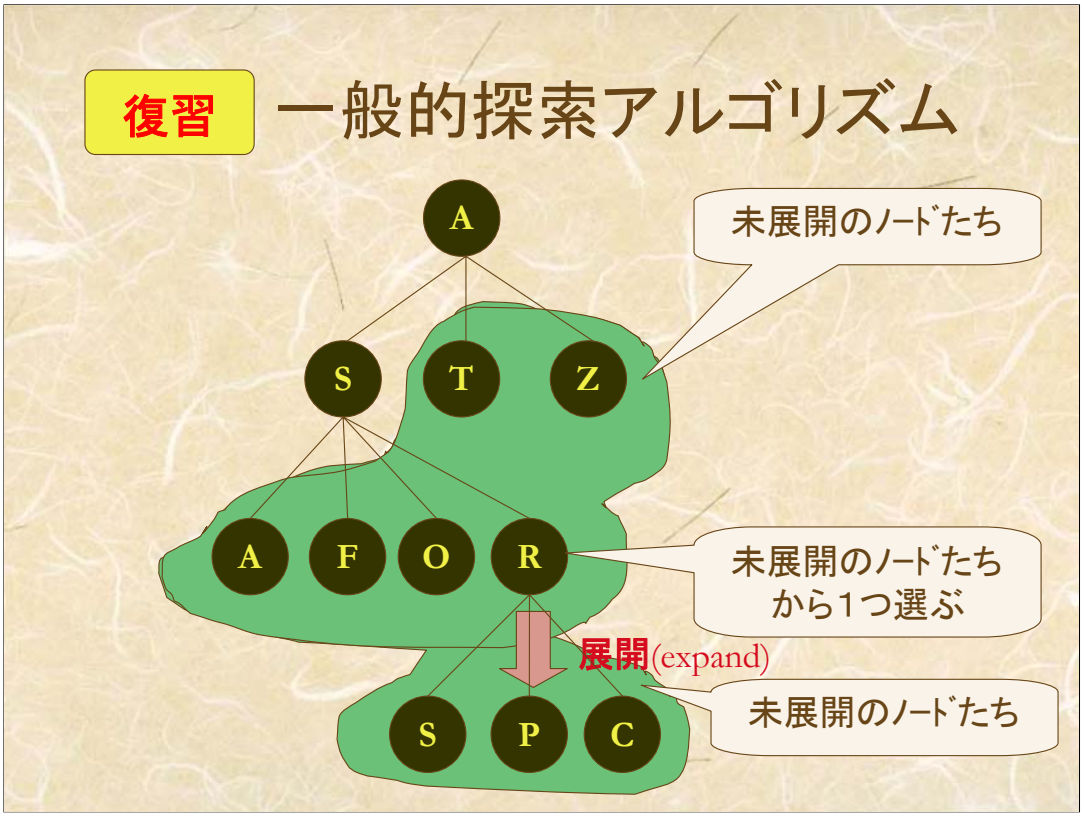
典型的な実験結果によると，総合的には，反復深化探索が優秀であるというのが結論である．



これは探索(1)の復習.
探索問題とは,
　初期状態,
　オペレータの集合,
　ゴール検査手続き,
　経路コストの定義
という4つの情報の集まりとして定義される.

これも探索(1)の復習.

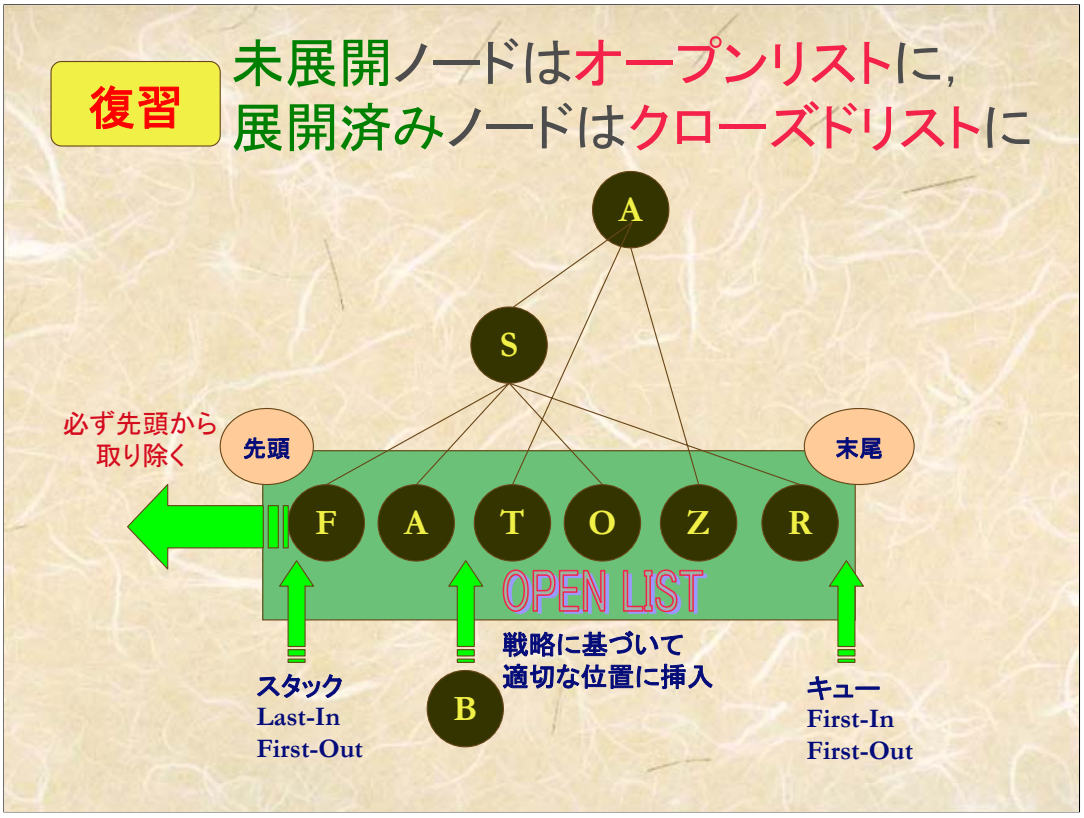
このようなルート発見(ナビゲーション)の問題が典型的な探索問題の例である.



これも探索(1)の復習.

探索アルゴリズムは，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．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**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．

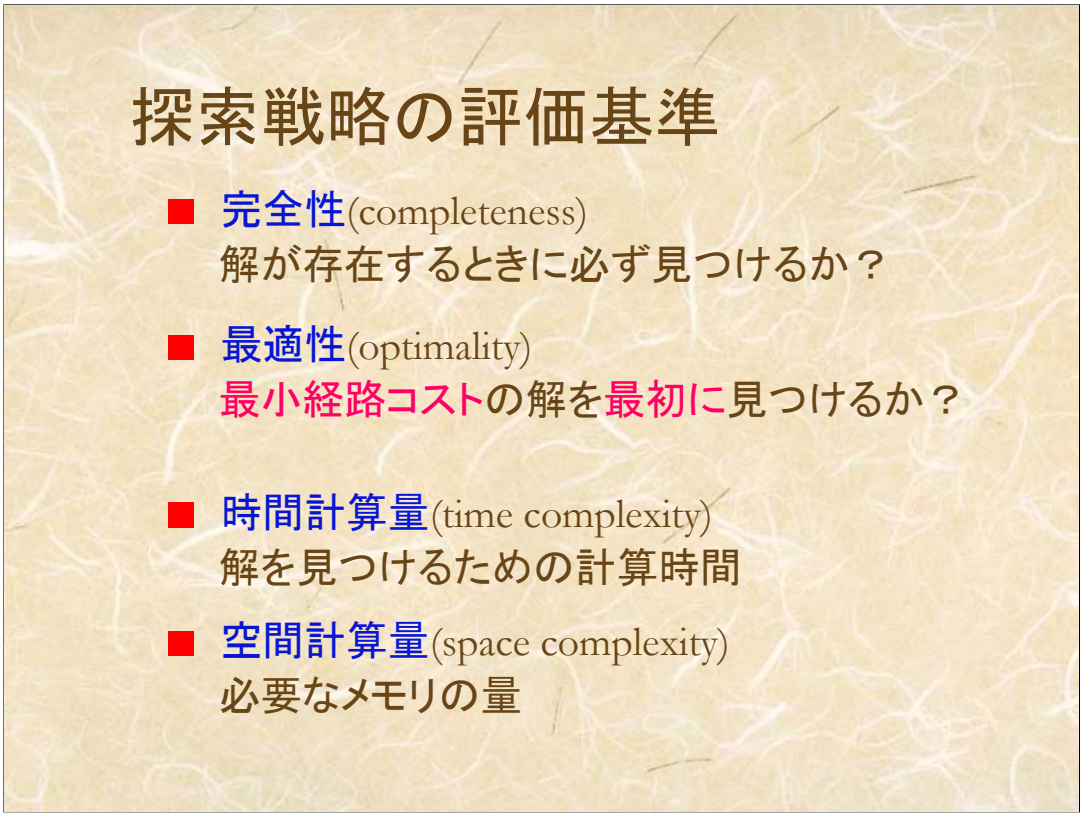


これも，また探索(1)の復習．

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)といい，それら2種類のノードをそれぞれ，**オープンリスト**(open list)と**クローズドリスト**(closed list)に保持する．

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，**戦略**(strategy)に基づいて適切な位置に挿入する．特別な場合として，常に子ノードを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)のデータ構造となり，子ノードを常に末尾に挿入するというなら，**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．



探索戦略の評価基準として，つぎの4つを考える．

■完全性　解が存在するときに，この戦略を用いた探索アルゴリズムはそれを必ず見つけるか？　もちろん，見つけることが望ましく，そのとき，その戦略は完全性があるという．

■最適性　解が複数個存在するときに，この戦略を用いた探索アルゴリズムが最初に見つける解は，経路コストが最小のものか？　もしそうなら，その戦略は最適性があるという．

■時間計算量　解を見つけるための計算時間．理論的には，具体的に何秒かかるということよりは，問題のサイズ n が大きくなるとともに，計算時間がどのような速さで増大するかという漸近的な性質を問題にすることが多い．

■空間計算量　解を見つけるために必要な記憶の量．時間計算量と同様に，理論的には，具体的に何キロバイト必要かということよりは，問題のサイズ n が大きくなるとともに，記憶量がどのような速さで増大するかという漸近的な性質を問題にする．

探索戦略は，ここに書いてある2つに分類できる．

知識に基づく探索あるいは**ヒューリスティック探索**（**発見的探索**ともいう）と呼ばれるものは，図にあるように，各状態からゴールまでの近さに関する知識が経験的にうすうす知られているものをいう．その場合には，探索空間をしらみつぶしに探す必要はなく，ゴールがありそうな方向に向かって効率よく探索する方法が考えられている．

それについては次回に学ぶとして，今回はそのような**知識なしの探索**，あるいは，**しらみつぶしの探索**と呼ばれるアルゴリズムのみを考える．

しらみつぶしの探索のアルゴリズムとして，つぎの3つを学ぼう．

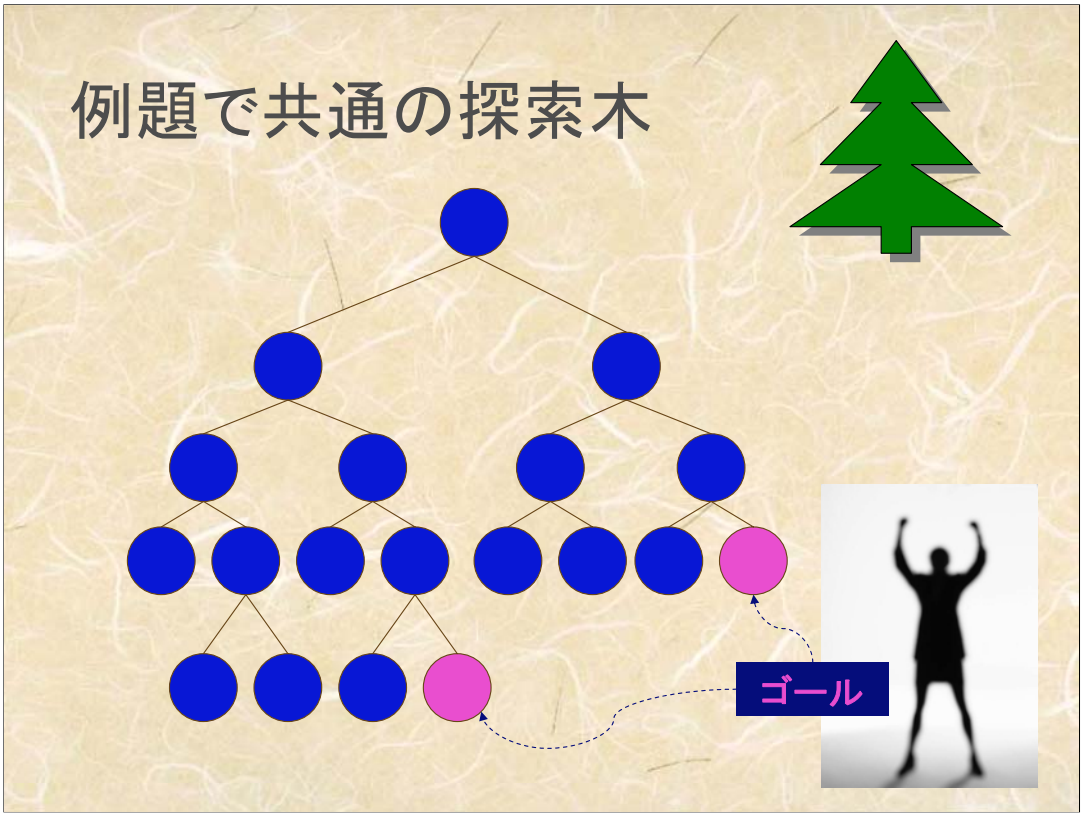
1. **幅優先探索**
2. **深さ優先探索**
3. **反復深化探索**

「しらみつぶし」というのは，英語のblind（盲目）という単語を，身体障害者に対する差別的なニュアンスが出ないように意訳した言葉である．

しかし，この言葉も「しらみ」をつぶすという語源を考えると下品な感じがあるので，「系統的(systematic)」あるいは「網羅的(exhaustive)」という言葉が良いかもしれない．要するに，探索空間を漏れなく系統的に隅から隅までずずーぃと探し尽くすアルゴリズムである．

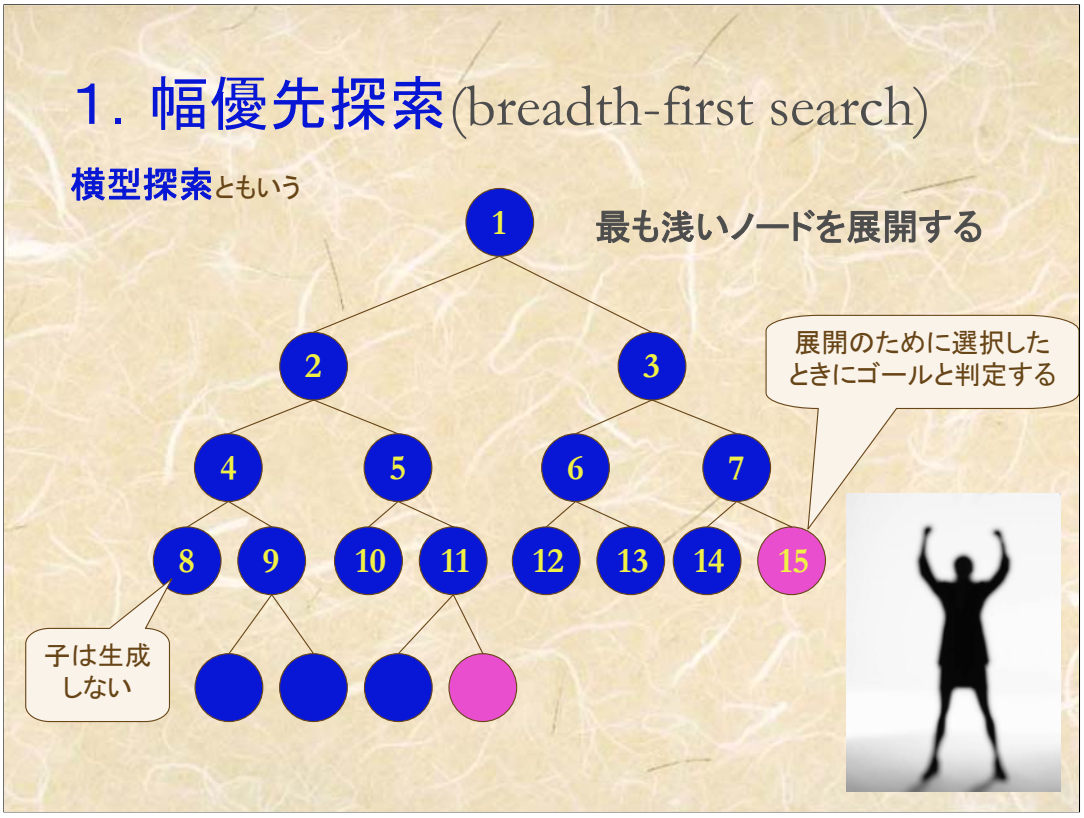
最初の2つのアルゴリズム（幅優先探索，深さ優先探索）は，人工知能というよりは，ふつうのコンピュータサイエンスの科目である「データ構造とアルゴリズム」とか「グラフアルゴリズム」というような授業で学ぶことが多い基本的なアルゴリズムである．

最後のアルゴリズム（**反復深化探索**）は，人工知能研究者が見つけたアルゴリズムで，総合的な観点から，最初の2つのアルゴリズムより優れていると評価される場合が多い．



今回の例題では，この図のような抽象的な探索木を仮定する．

ゴールが2つある．どちらを見つけてもよいのだが，できれば深さの浅い方のゴールを見つけてくれればうれしい（つまり，経路コストが小さいほど良い解である）と仮定する．

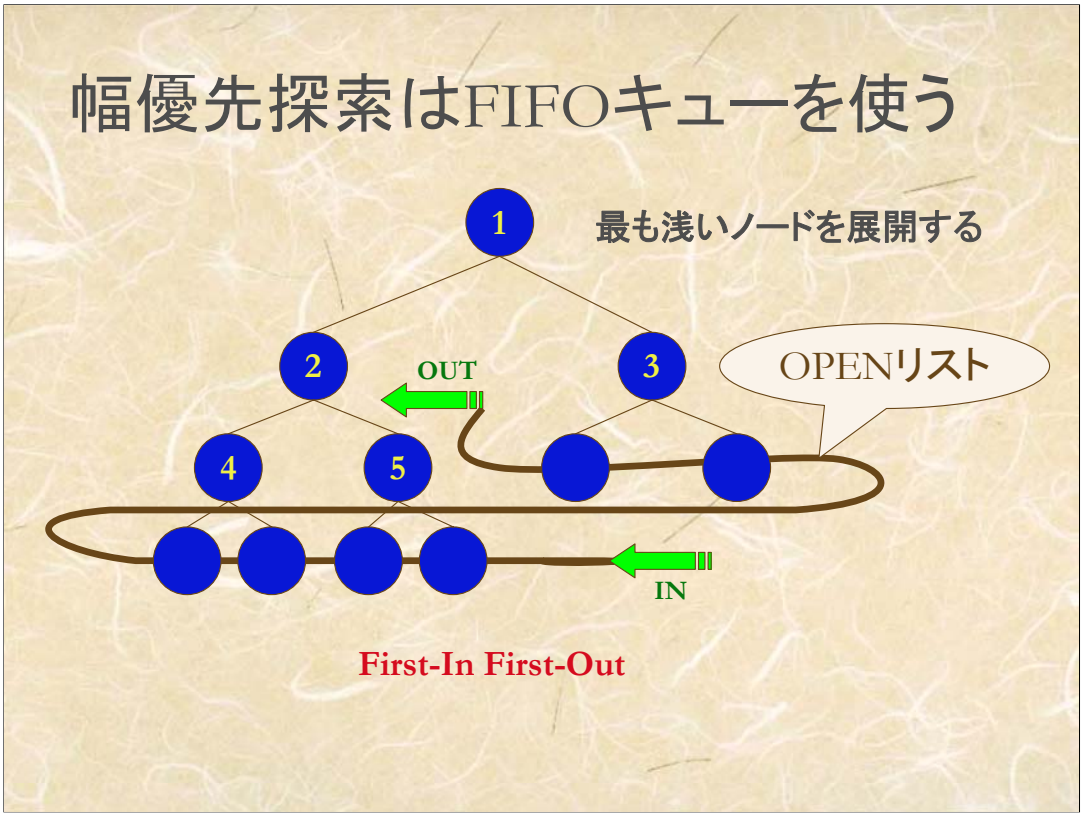


基本的な探索アルゴリズムは，未展開のノードを1つ選んで，それを親ノードとして展開して子ノードを生成するということをゴールノードが見つかるまで繰り返すというものだった．ここで問題となるのは，未展開のノードのうちどれを選べば良いかということである．その決定方法を**戦略**という．

幅優先探索(breadth-first search)は，「未展開のノードのうち，最も浅い位置にあるノードを選んで展開する」という戦略をもつアルゴリズムである．深さが同じノードどうしでは，どちらを選んでもよいが，このスライドでは左側に描かれたノードを選んでいる．このスライドはアニメーションになっているので，具体的にノードが展開されて探索木が成長するようすを確認してほしい．

前回学んだ一般的探索アルゴリズムによると，展開のためにノードが選ばれたときに，それがゴールであることが判定されることに注意しよう．これはこのアルゴリズムの最適性を保証するために重要である．したがって，この例の場合，ノード7を展開してノード15というゴールを生成した時点では，ノード15がゴールであることはまだ判定されない．その後，ノード14までの展開が進み，つぎにノード15を展開しようとする直前にそれがゴールであることがわかり，アルゴリズムは終了する．

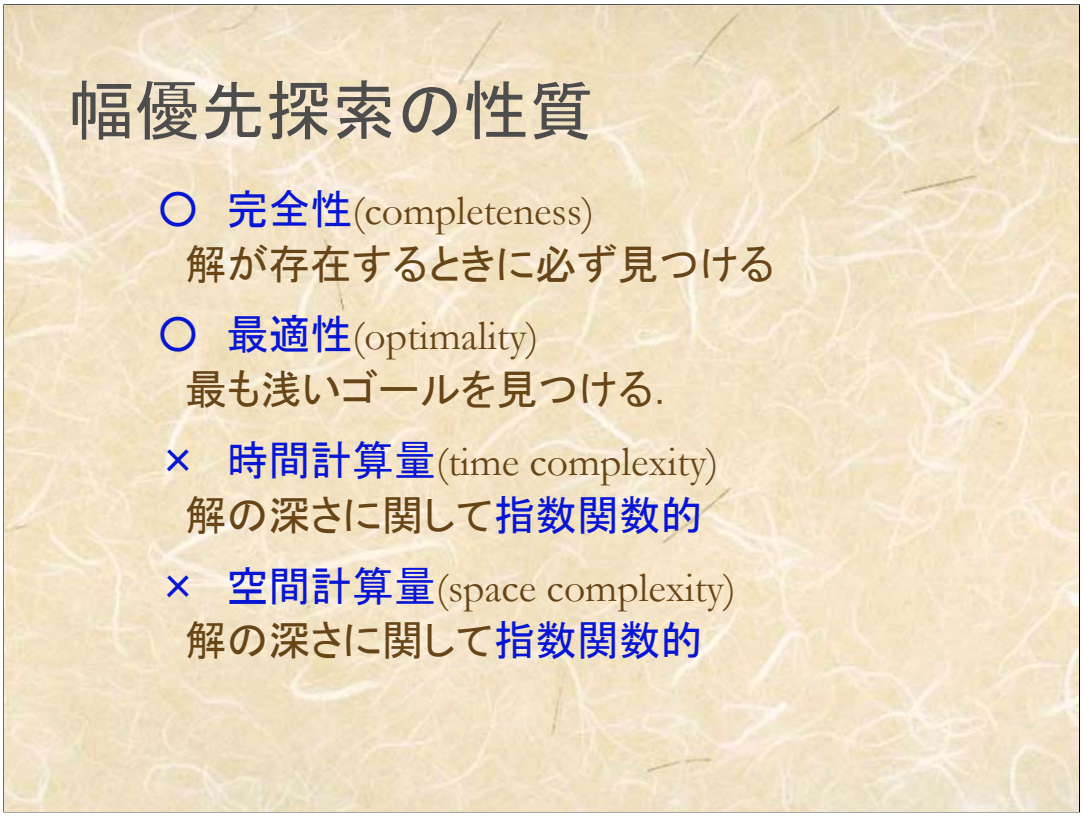
幅優先探索は，この図からわかるように，展開すべきノードが4→5 →6 →7のように木の幅方向すなわち横方向に進んでいくことから，**横型探索**と呼ばれることもある．



未展開ノードたちは，**オープンリスト**に一列に並べておくのだったことを思い出そう．未展開ノードから1つノードを選ぶときには，オープンリストの先頭のものを選ぶことに決めていたのだった．したがって，最も浅いノードを最初に展開するために，幅優先探索では，未展開ノードを深さの浅い順にオープンリストの中に並べておく．

選んだノードを展開して生まれた子ノードは（引き分けも含めて）最も深さが深い．したがって，子ノードはオープンリストの末尾に付加すればよい．

つまり，生まれた子ノードはオープンリストの末尾から入り，徐々に先頭へ移動していき，最後に親ノードとして先頭から取り出されて子ノードを産む．先頭から取り出される親ノードは，現在のオープンリスト中のノードのうち，最も最初に入ってきたものである．つまり，最も最初に入ってきたもの(first-in)が最初に出ていく(first-out)．その意味で，このデータ構造は，**First-In First-Out** (FIFO)という言葉で特徴が説明できる **待ち行列**あるいは**キュー(queue)**というものとなる．

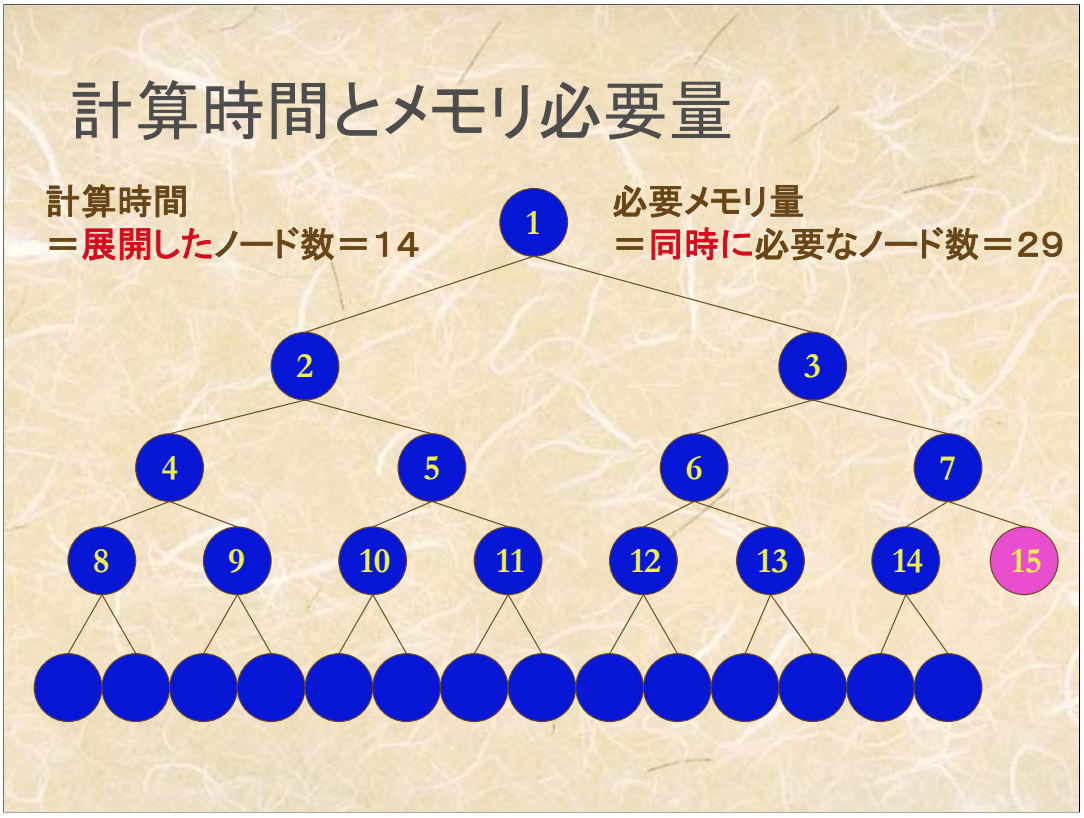


幅優先探索の性質がこのスライドのようになることを確認しておこう.

幅優先探索には完全性と最適性があることは自明である.(自明と感じない人は,このアルゴリズムを良く理解していない.)

幅優先探索の計算量は**指数関数的**である.それは,解の深さ(ゴールノードが存在する位置の深さ)d が1だけ増えたときに,計算量が一定量だけ増えるのではなく,何倍にも増えるということである.その結果,計算量を数式で表すと,ある定数 c を用いて cのd乗 というように d に関して指数関数になるのである.そうなる理由はこの後のスライドで示す.

幅優先探索は計算量が指数関数的なので,ゴールが探索木の深い位置にある問題を実用的に解くことは困難である.ただし,ゴールが浅い位置にある問題の最適解(最も浅いゴール)を確実に求めるのには適している.



使用するコンピュータの種類や，プログラミングのしかたによって，実際の計算時間やメモリ必要量は異なる．そこで，そういう環境条件に依存しないで計算時間やメモリ必要量を定義するために，

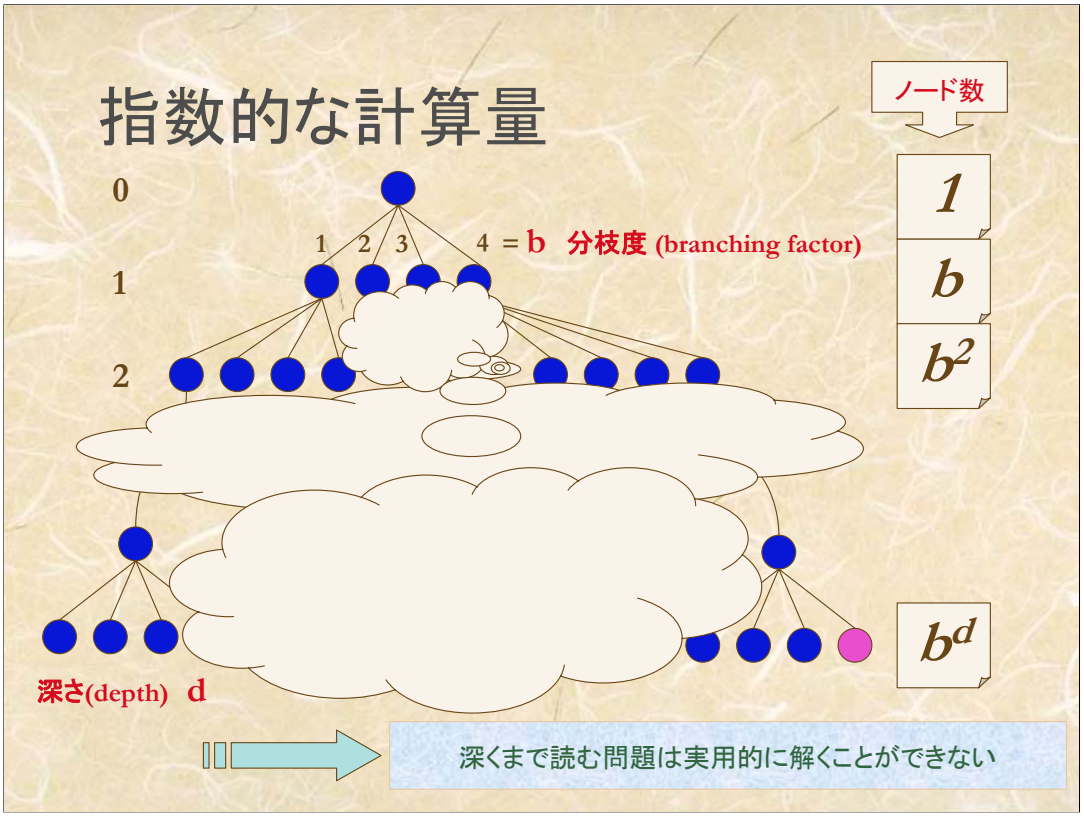
計算時間＝展開したノード数

必要メモリ量＝(同時に)必要なノード数

という単位で測ることにする．

多くの場合，探索アルゴリズムの実行時間の大部分がノードの展開に費やされ，かつ，ノードを展開する時間が一定と考えられるので，展開したノード数として定義された計算時間は現実の秒単位の物理的な時間にほぼ比例した値となる．同様に，必要な記憶量の大部分はノードを記憶するために使われ，かつ，ノード1つの記憶量が一定と考えられるので，ノード数で定義された必要メモリ量は，現実のビット単位の物理的な記憶量にほぼ比例した値となる．

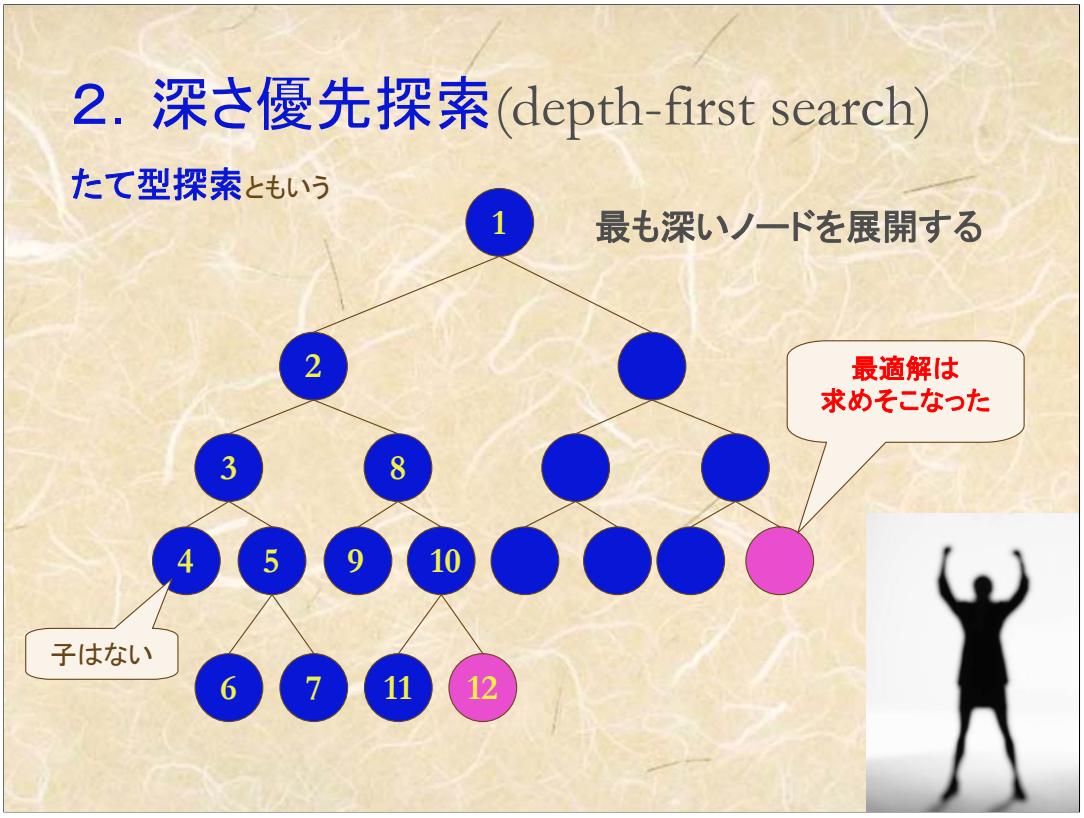
なお，必要メモリ量に書かれている「同時に」の意味は，「時間を任意に固定したときに」という意味である．幅優先探索では該当しないのだが，後に出てくる探索アルゴリズムでは，ノードを生成する一方で，不要になったノードは削除することができるので，メモリ中に保持されるノード数は時間とともに増減する．したがって，必要メモリ量は，単に生成されたノード数の総計ではなく，時間を固定したときに保持されているノード数で測るということである．より正確に述べると，その固定する時間をいろいろ変えてみたときの最大ノード数分だけのメモリが必要となる．



幅優先探索の時間計算量と空間計算量が指数関数的になる理由はこの図からすぐわかるだろう.

分枝度bとは枝分かれの平均数である. この図では, 枝はみな4分枝しているので, b=4である. また, 解の深さ(ゴールノードの深さ)をdで表している.

このように, 幅優先探索はdに関して指数関数的な計算量をもつので, ゴールが深い位置になるような問題を実用的に解くことは困難である.



深さ優先探索(depth-first search)は，「未展開のノードのうち，最も深い位置にあるノードを選んで展開する」という戦略をもつアルゴリズムである．深さが同じノードどうしでは，どちらを選んでもよいが，このスライドでは左側に描かれたノードを選んでいる．このスライドはアニメーションになっているので，具体的にノードが展開されて探索木が成長するようすを確認してほしい．

これまでと同様に，展開のためにノードが選ばれたときに，それがゴールであることが判定される．

この例では，最初に見つかったゴールノード(12)は最適解ではない．もっと浅い位置に別なゴールが存在するからである．

深さ優先探索の特徴は，必要とするメモリ量が少なくて済むことである．適切な方法で実装(プログラミング)すれば，これより深い位置にはゴールはないとわかったノードを削除して，記憶量を節約することができる．主な実装方法はつぎの3つである．

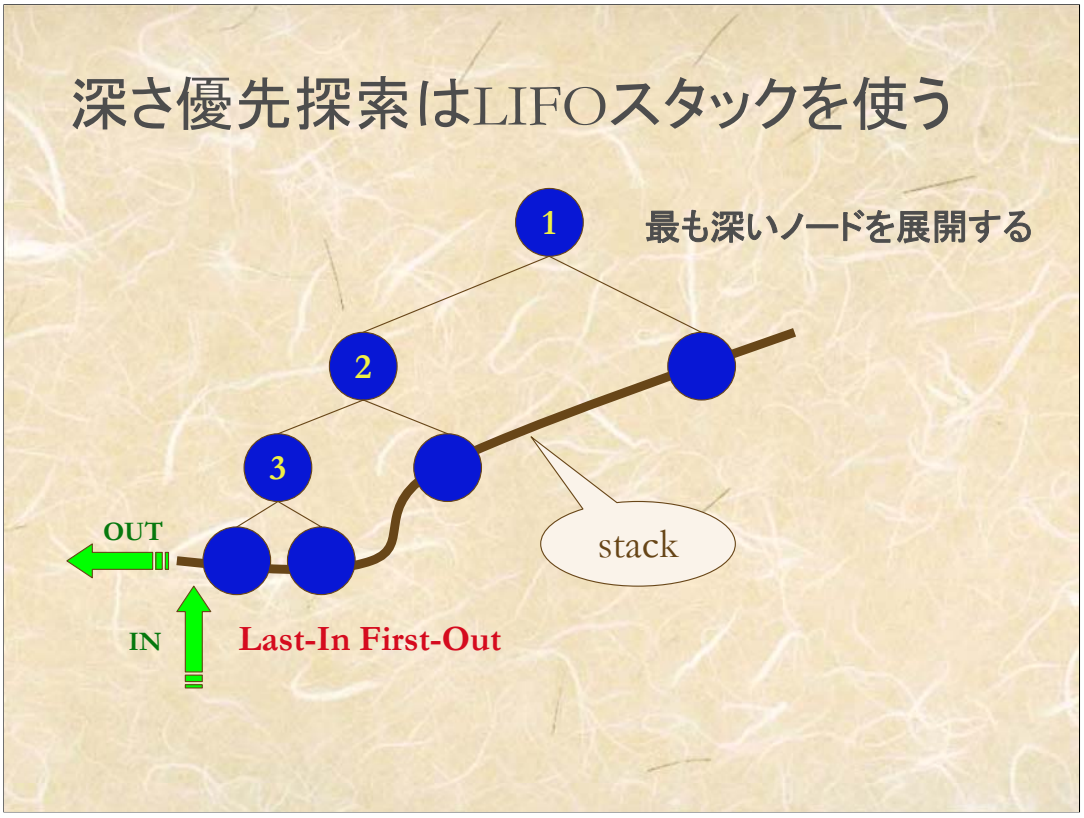
(1) 通常のプログラミング言語(CやC++など)では，動的に生成したメモリ領域(構造体やオブジェクト)が不要になったら，プログラマの責任で，プログラミング言語が提供する基本機能を用いて明示的にメモリ領域を解放するようにコーディングする．

(2) ガベージコレクション(ゴミ集め)の機能がある言語(Java，Lisp，Prologなど)や実行環境(Microsoft .NET フレームワークなど)では，放っておけば，不要になったメモリ領域をシステムが自動的に判断して再利用可能とする．

(3) 探索木を直接作らず，スタックや再帰的アルゴリズムをうまく用いて，間接的にメモリ内に探索木ができるようにする．これを正確に実装できるのは，やや上級のプログラマに限られる．また，このテクニックは深さ優先探索では可能だが，他の大多数の探索アルゴリズムの実装には適していない．

このスライドの例の場合，ゴールノード12が見つかった時点でメモリ中に保持されているノードは，1，2，8，10，12および番号が付いていないがノード1のもう1つの子ノードの6個のみである．ノード3～7および9，11のノードは削除されている．

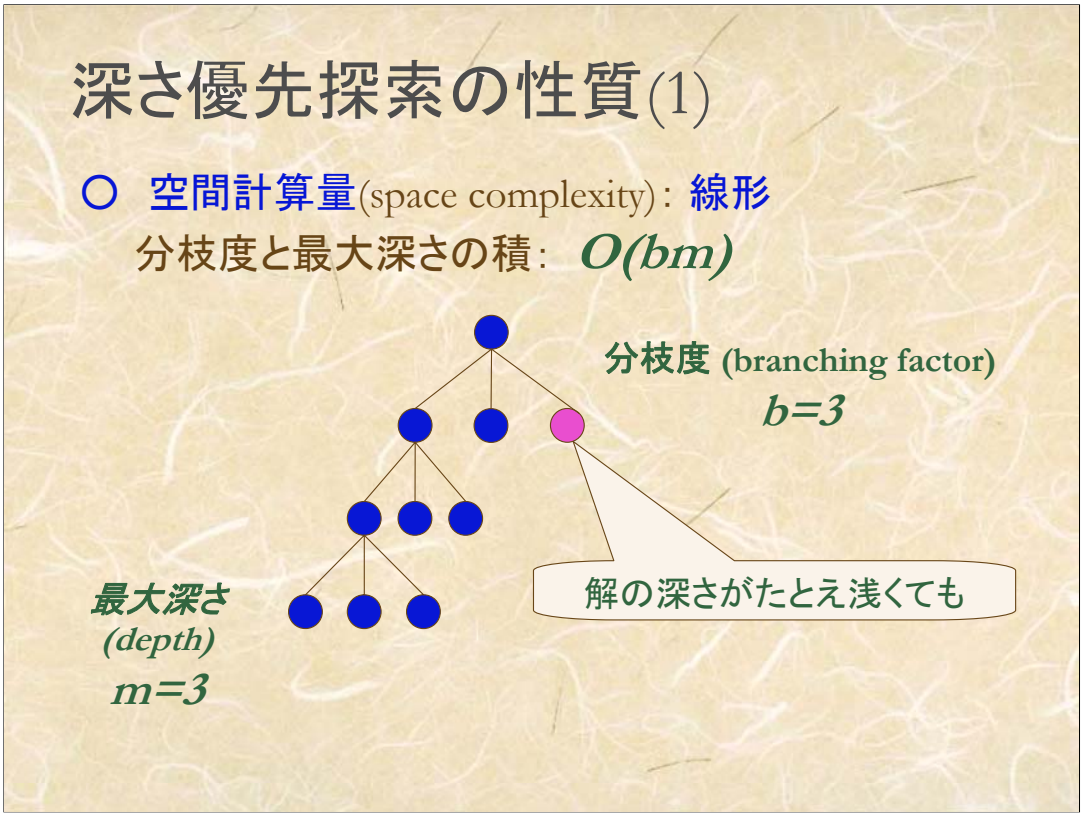
展開すべきノードがこの図の1→2 →3 →4のように木の深さ方向すなわち縦(たて)方向に進んでいくことから，深さ優先探索は**たて型探索**と呼ばれることもある．



未展開ノードは**オープンリスト**に一列に並べておき，それらから1つ親ノードを選ぶときには，リストの先頭のものを選ぶことに決めていたのだった．したがって，最も深いノードを展開するために，深さ優先探索では，未展開ノードを深さの深い順にオープンリストに並べておく．

最も深い位置にあるノードを展開して生まれた子ノードはやはり最も深さが深い．したがって，子ノードはオープンリストの先頭に付加すればよい．

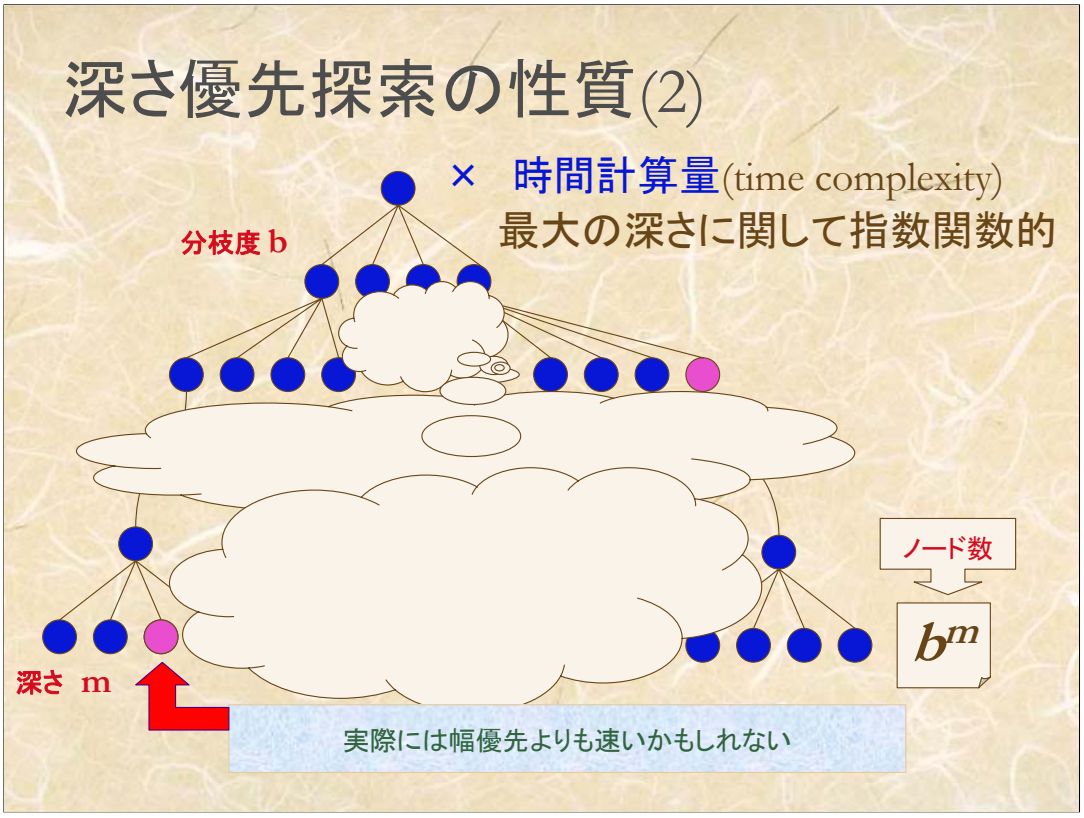
つまり，生まれた子ノードはオープンリストの先頭から入り，短時間のうちに，再び先頭から取り出されて子ノードを産む．先頭から取り出されるノードは，現在のオープンリストのノードのうち，最も最後に入ってきたものである．つまり，最も最後に入ってきたもの(last-in)が最初に出ていく(first-out)．その意味でこのデータ構造は，**Last-In First-Out** (LIFO)という言葉で特徴が説明できる**スタック**（**stack**）というものとなる．



4つの評価尺度ごとに，深さ優先探索の性質を検討してみよう．

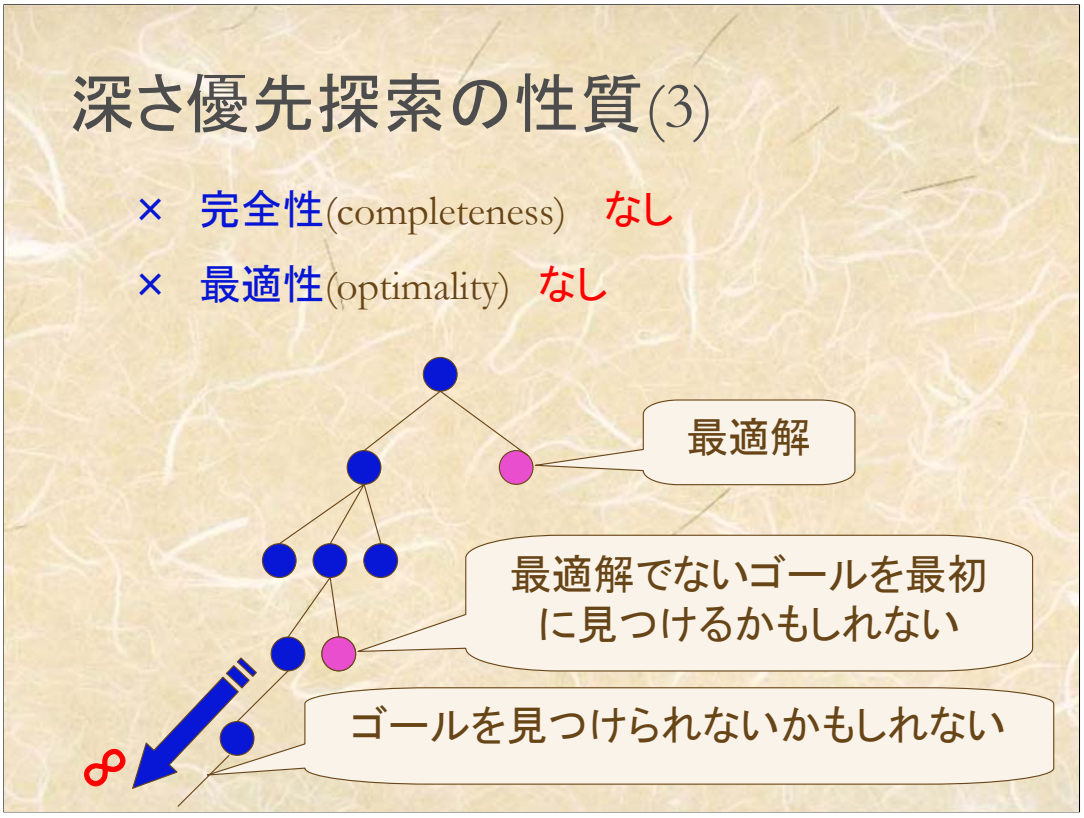
この図からわかるように，深さ優先探索の空間計算量は**線形**である．より正確に述べると，分枝度と最大深さの積に比例する．多くの問題がそうであるように，分枝度の上限が一定の値 b で抑えられる問題では，bを定数とみなすと，空間計算量は深さmに比例することになる．

ここで，mは「探索木の最大の深さ」であることに注意しよう．それに対して，幅優先探索では「解の深さ」あるいは「ゴールノードの深さ」dを用いていた．このスライドの図の場合には，m=3, d=1 となっている．



最悪の場合，深さ優先探索の時間計算量は，最大の深さmに関して指数関数的になる．(ゴールが木の最も右下の場合を考えよ．)式で書くとbのm乗となる．幅優先探索ではbのd乗で，ふつうdはmより小さいので，理論的には幅優先探索のほうが，計算時間が短いようにみえる．

しかし，現実にはそうでないことが多い．深さ優先探索はとにかく深く進んで解を見つけようとするので，実際には幅優先探索よりも早く解を見つけることも珍しくないのである．幅優先探索は，浅い順にすべてのノードを生成するまで解を見つけない．悪いことに，すべてのノードをメモリに記憶させていくので，解を見つける以前にメモリがパンクして，それ以上の探索が不可能になってしまうことが多いのである．

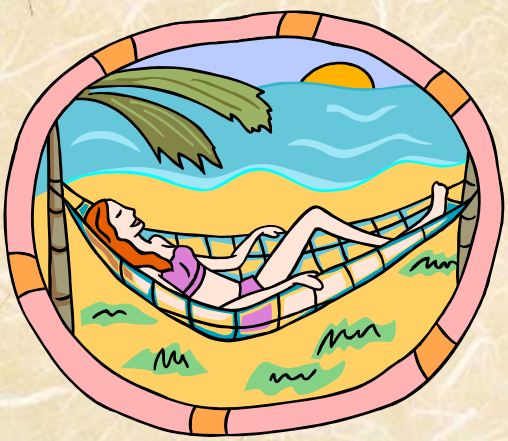


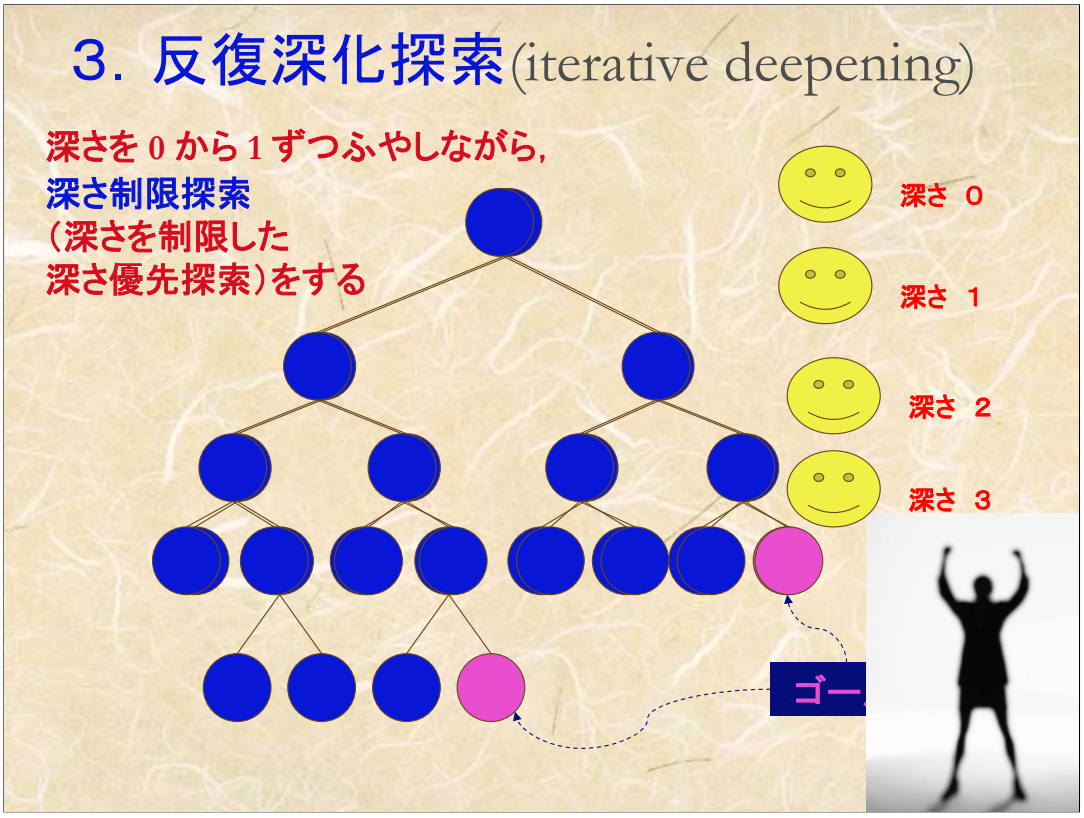
深さ優先探索には完全性も最適性もないのは，このスライドから自明である．

完全性がないのは，その先に進んでもゴールがないような枝を選んで深く深く無限に進むことがあるからである．ただし，探索木の深さが有限のときには完全性がある．

最適性がないのは，この図のように，最適解でない深い位置にあるゴールを最初に見つけるかもしれないからである．

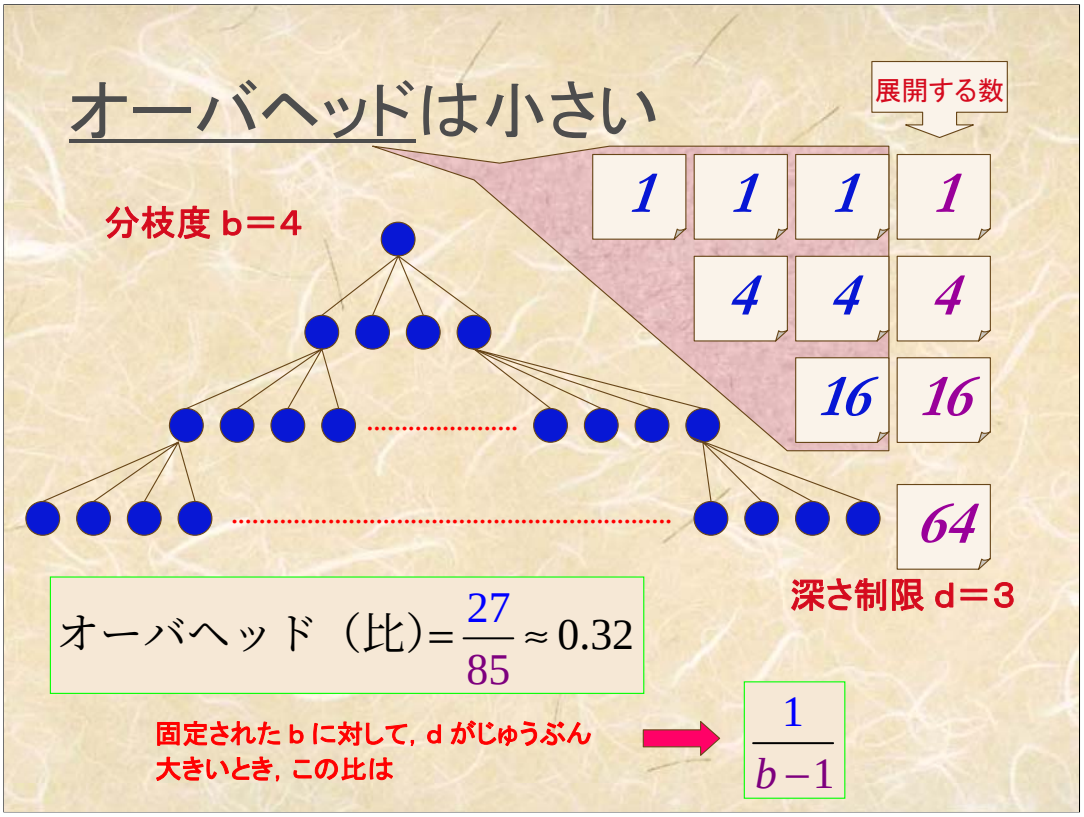
休憩





深さ優先探索は，探索木の深さが無限のときには，完全性がない．そこで，深さを制限した深さ優先探索とでもいうべきものが自然に思いつく．それを**深さ制限探索**と呼ぶ．**深さ制限**dの深さ制限探索は，基本的にはふつうの深さ優先探索と同じなのだが，深さがdに達したら，強制的に，それより深い子ノードを生成しないことにするものである．そうすると，深さd以内にゴールがあるときには，必ずゴールを見つけることができるという意味で制限された完全性の性質をもつ．そのかわり，深さがdより深いところを探索する能力は失われる．

今回の授業のハイライトともいえる**反復深化探索**(iterative deepening)というのは，その名の表しているように，深さ制限を反復的(iterative)に深めること(deepening)を繰り返しながら，深さ制限探索をその都度1回ずつ実行するものである．1回の反復ごとに深さ制限を1ずつ大きく(ゆるく)していく．まず，深さ制限d＝0として，深さ制限探索を実行する．つぎに，d＝1として，深さ制限探索を実行する．つぎに，d＝2として，深さ制限探索を実行する．このように，dの値を1ずつ増やしながら，解が見つかるまで深さ制限探索を実行する．



明らかに，探索木の根に近い部分のノードほど，何度も何度も繰り返し同じものが生成あるいは展開されて，無駄な処理になっている．一般に，何か良い効果（メリット）を得るために支払わなければならない代償（デメリット）を**オーバヘッド**というが，まさにこの反復深化探索の動作はオーバヘッドが大きいように感じられる．しかし，冷静に分析してみると，実際にはオーバヘッドは，ふつうの人が直観的に感じるほど大きくはないのである．

このスライドの探索木は，分枝度をb＝4に固定して深さd＝3まで表示したものである．ノードの個数が1＋4＋16＋64＝85であるから，無駄なくノードを生成すれば，ノードを展開しようとした回数85に比例した時間がかかる．

反復深化探索では，深さ0の部分を4回，深さ1の部分を3回，深さ2部分を2回，深さ3の部分を1回展開する．したがって，オーバヘッド（すなわち，重複して余計に展開した回数）は，

$$(4-1)\times 1+(3-1)\times 4+(2-1)\times 16=27$$

である．これはさきほど求めた85の32%である．

【演習問題】一般に，分枝度が b，最大の深さが d のとき，重複なくノードを生成すると，ノードを展開する回数は $N=1+b+b^2+\cdots+b^d$ となる．また，反復深化探索のオーバヘッド（重複して展開する回数）は $H=d+(d-1)b+(d-2)b^2+\cdots+3b^{d-3}+2b^{d-2}+b^{d-1}$ となる．いろいろなbとdの値について，N，H，およびその比 H／Nを計算せよ．また，もし可能なら，数列の和を求める公式を利用して，それらの値を求める明示的な式を導出し，固定されたbに対して，dがじゅうぶん大きければ，H／Nの値はほぼ1／(b－1)となることを示せ．

ヒント： $bH=db+(d-1)b^2+(d-2)b^3+\cdots+3b^{d-2}+2b^{d-1}+b^d$ からHを引くと

$$(b-1)H=-d+b+b^2+b^3+\cdots+b^{d-2}+b^{d-1}+b^d$$

となり，右辺の第2項以降は等比数列の和になっている．

反復深化探索は完全性と最適性の性質をもつ. これは, 深さ制限を1ずつ増やしていくことにより, 幅優先探索と同様な利点を受け継いでいるためである.

深さ優先探索の利点を受け継いで, 空間計算量は線形である.

残念ながら時間計算量は指数関数的だが, これはある意味で当然である. しらみつぶし探索は, 本質的に, 最悪の場合にはすべての組合せを考慮する運命にあるので, 指数関数的な時間計算量を避けることはできない.

探索戦略の比較

	幅優先 (BFS)	深さ優先 (DFS)	反復深化 (ID)
完全	○	△	○
最適	○	×	○
時間	△ b^d	× b^m	△ b^d
空間	△ b^d	○ bm	◎ bd

b: 分枝度　d: 解の深さ　m: 探索木の最大の深さ

今回学んだ3つのアルゴリズムに対して，4つの評価尺度を検討した結果を表にまとめておく．

この表からは，理論的には**反復深化探索**が最も優れているといえる．とくに，探索木の深さ(m)が無限であったり，有限であっても非常に大きな値のときにはお薦めである．さらに，分枝度bが大きいほど，オーバヘッドは相対的に小さくなるのでその効果は大きい．

深さ優先探索は，探索木の深さが有限のときには完全性があるので，最適性がなくてもよいなら選ぶ理由がでてくる．とくに，探索木全体の深さ(m)があまり大きくないときは，深さ優先でじゅうぶんである．ただし，ゴールが浅い位置にあるにもかかわらず，深さ優先探索はそれを見つけるのに案外手間取るかもしれない．

幅優先探索をお薦めできるのは，ゴールの深さ(d)がきわめて浅い場合か，さもなくば，何か特殊事情があるときに限られる．