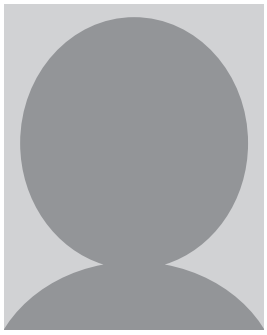
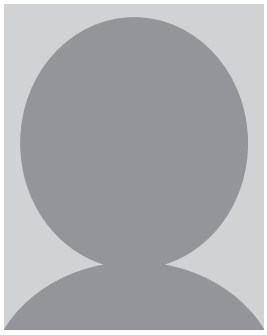
問題解決の数理（’１７）

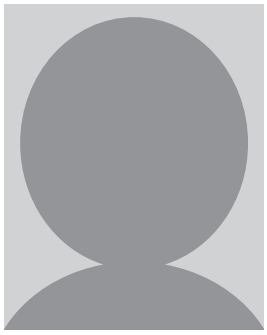
- 収録本番とは多少異なっていることがあります.
- 内容の間違いのご指摘は歓迎します.
- 「完全に無保証」です.



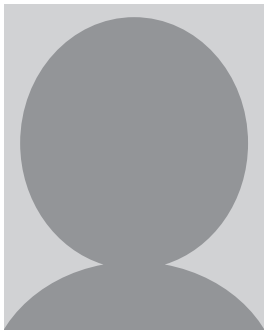
- 今回の講義では，問題の状態空間モデルと，探索法についてお話します．
- この科目で扱う問題のほとんどは，最適解を求めるというタイプの問題ですが，もちろん，異なるタイプの問題もあります．
- 例えば，迷路などのパズルはゴールへ至る経路を発見する問題と捉えることができます．
- パズル以外でも，例えば，数学におけるある種の定理の証明は，与えられた条件から式変形を行い，特定の性質を表す数式を導く，という問題で，これも同様に経路を発見する問題と捉えることができます．



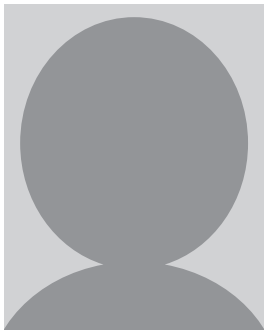
- これらの問題は，少なくとも表面上は最適解を求めるタイプの問題とは異なります．
- 今回お話する状態空間モデルは，これらの問題を表現する方法の一つです．
- 状態空間モデルを用いた問題解決の研究は，人工知能の分野での最初期の研究テーマで，1950 年代から 1960 年代あたりに盛んに研究されました．



- このような随分古い話題を，今回敢えて取り上げるのは，問題の定式化や問題解決の方法にいろいろな考え方があることを知ってもらいたいからです．



- 状態空間モデルで定式化された問題を解くのは，グラフの探索に帰着されます.
- グラフの探索は汎用的な方法で幅広く応用されていて，最適化問題を解くのに利用されることもありますので，基本的な方法について紹介します.
- それでは，まず状態空間モデルについてお話します.

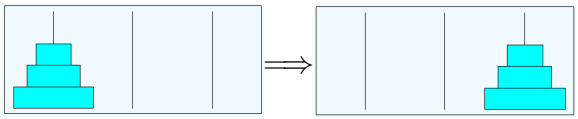


- 状態空間モデルを説明するための例としてハノイの塔と呼ばれるパズルを用います.
-

状態空間モデル

ハノイの塔

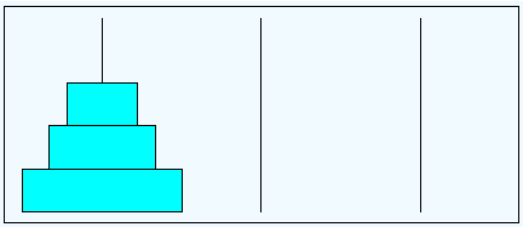
- 3 本の柱と大きさの異なる複数の円盤
- 最初はすべての円盤が左端の柱 (大きさの順)
- 円盤を一回に一枚ずつ別の柱に移動させる
- すべての円盤を右端の柱に大きさの順に置く
- 小さな円盤の上に大きな円盤を置くことはできない



- ハノイの塔は次のようなパズルです.
- ハノイの塔は，3 本の柱と中心部に穴の空いた大きさの異なる複数の円盤から成ります.
- ここでは 3 枚の円盤とします.
- 最初は，左の図のようにすべての円盤が左の柱に大きさの順に積まれています.
- 円盤を一回に一枚ずつ別の柱に移動させ，すべての円盤を右の柱に大きさの順に置けば終了です.
- ただし，円盤の上により大きな円盤を置くことはできません.
- このパズルの解は次のようになります.

状態空間モデル

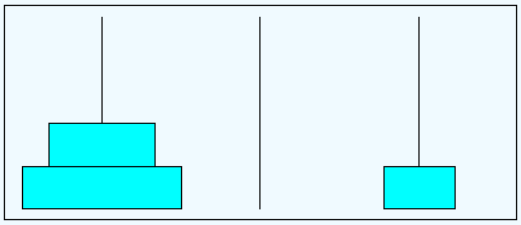
0



- 最初は，左の柱に 3 枚の円盤が置かれています.
- （ポーズ）
- まず，小さな円盤を右の柱に移します.

状態空間モデル

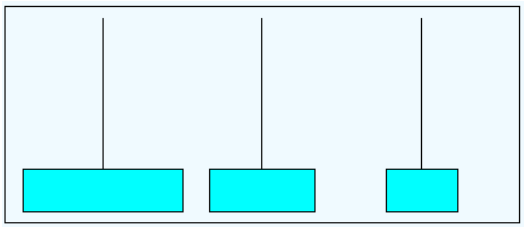
1



- そうしますと，このようになります.
- （ポーズ）
- 次に，左の柱にある，中間の円盤を，中央の柱に移します.

状態空間モデル

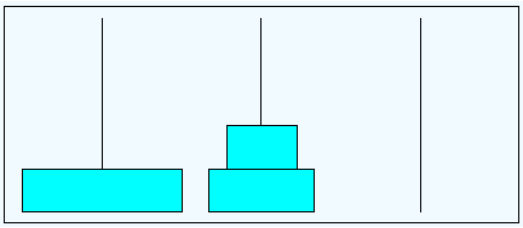
2



- そうしますと，このようになります.
- （ポーズ）
- 次に，右の柱にある，小さな円盤を，中央の柱に移します.

状態空間モデル

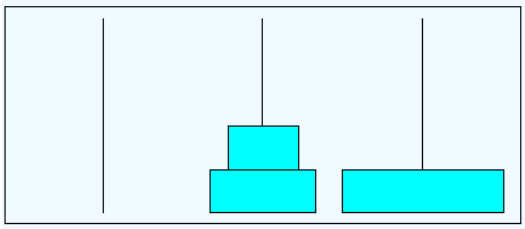
3



- そうしますと，このようになります.
- （ポーズ）
- 次に，左の柱にある，大きな円盤を，右の柱に移します.

状態空間モデル

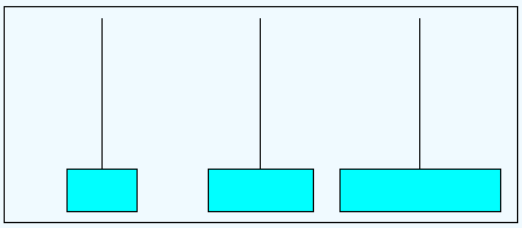
4



- そうしますと，このようになります.
- （ポーズ）
- 後は簡単で，中央の柱にある，小さな円盤を，左の柱に移します.

状態空間モデル

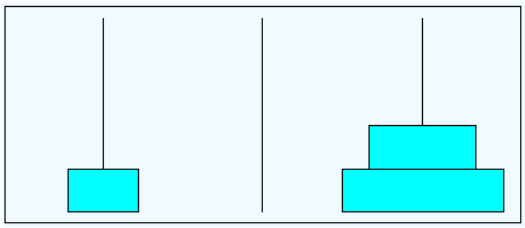
5



- そうしますと，このようになるので，中央の柱にある，中間の円盤を，右の柱に移します.

状態空間モデル

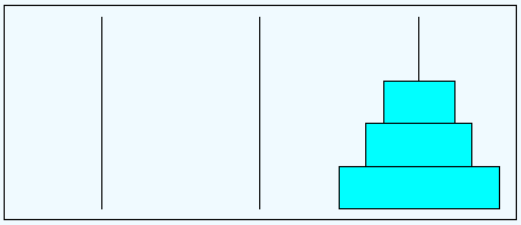
6



- そうしますと，このようになるので，最後に左の柱にある，小さな円盤を，右の柱に移します.

状態空間モデル

7



- これで完成です.
- （ポーズ）
- それでは，状態空間モデルの説明に移ります.

状態空間モデル

問題

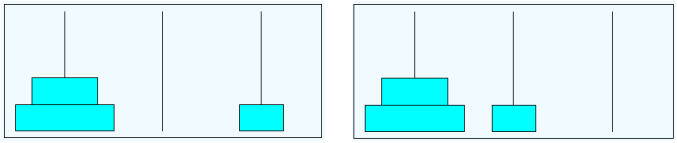
- 状態の集合
- 作用素の集合
- 初期状態
- 目標状態の集合

- 状態空間モデルにおいて，問題は，
  - 状態の集合
  - 作用素の集合
  - 初期状態
  - 目標状態の集合

  から構成されます.
- 各構成要素についてハノイの塔を例に説明します.

状態空間モデル

状態 問題が解かれていく各ステップの状況



(L, L, R)　　(L, L, C)

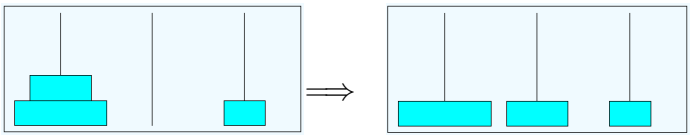
状態の集合 とり得るすべての状態

- まず，状態は，問題が解かれていく各ステップの状況を表現するものです．
- ハノイの塔においては，例えばここに示したような状況を表現したものが状態です．
- このような図は，我々人間が見る分にはよいのですが，数理的にあるいは計算機で扱うには，状況を的確に表しているとは言えません．
- ハノイの塔の状態の表現には，各円盤がどの柱に置かれているかを表わすのが適切です．
- 例えば，左，中央，右の柱を各々L, C, R として，大きな円盤，中間の円盤，小さな円盤の位置の組で状態を表します．
- 左の図では，大きな円盤と中間の円盤が左の柱，小さな円盤が右の柱にあるので，(L, L, R)，
- 右の図では，大きな円盤と中間の円盤が左の柱，小さな円盤が中央の柱にあるので，(L, L, C) と表わします．
- 状態の集合とは，とり得るすべての状態の集合です．

状態空間モデル

作用素（オペレータ） ある状態から別の状態へ遷移するための手段

・ 左の柱にある一番上の円盤を中央の柱に移動



(L, L, R) → (L, C, R)，(L, L, R) − LC → (L, C, R)

LC(x), LR(x), CL(x), CR(x), RL(x), RC(x)

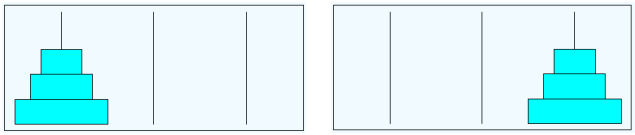
作用素の集合 とり得るすべての状態遷移

- 次に，作用素，オペレータとも言いますが，作用素はある状態から別の状態へ遷移するための手段のことです．
- 左の図の状態から右の図の状態への遷移の手段は，例えば，ここに示すように遷移前の状態と遷移後の状態を並べて表わすことができます．
- どうやって状態を遷移したかの情報が必要なら，例えば，右のようにLCという移動の仕方を加えてやります．
- ここで，LCとは，左の柱の一番上の円盤を中央の柱に移すという意味です．
- ただ，状態遷移を書きならべるのは，表現のコンパクトさに欠けますので，下のように関数の形で書くというのもいいでしょう．
- x は変数で遷移前の状態を表し，関数は遷移後の状態を返すとします．
- 作用素の集合は，とり得るすべての状態遷移を過不足なく記述する必要があります．

状態空間モデル

初期状態 最初に与えられた状態

目標状態 達成すべき状態（ゴール）



初期状態　　目標状態

初期状態 1 個

目標状態 1 個ないしは複数

- 最後に初期状態と目標状態です.
- 初期状態は，最初に与えられた状態のことで，ハノイの塔においては左の図に表わす状態が初期状態です.
- 目標状態は，達成すべき状態，すなわちゴールのことで，右の図に表わす状態が目標状態です.
- 問題の初期状態は 1 個にしないと，どこから始めればよいか分からないので，1 個のみです.
- 目標状態は，一般には複数の場合もあり，その場合いずれかの目標状態に達すれば，問題は解けたということになります.

状態空間モデル

問題 (状態の集合，作用素の集合，初期状態，
目標状態の集合)

状態 問題が解かれていく各ステップの状況

作用素（オペレータ） ある状態から別の状態へ
遷移するための手段

初期状態 最初に与えられた状態

目標状態 達成すべき状態（ゴール）

- 状態空間モデルでは，これらの構成要素から問題が表現されます．
- すなわち，問題は，状態の集合，作用素の集合，初期状態，目標状態の集合の組として定式化されます．
- ハノイの塔の状態の数は有限ですが，問題によっては状態が無限集合になる場合もあります．
- その場合，集合の要素を書きならべることは不可能ですが，状態の満たすべき条件を指定すれば，状態の集合を定義できます．
- 作用素の集合についても，先ほど少し触れましたように同様です．

状態空間モデル

(ある種の) 定理証明

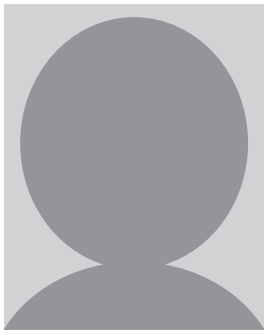
初期状態 前提となる数式

目標状態 定理に相当する数式

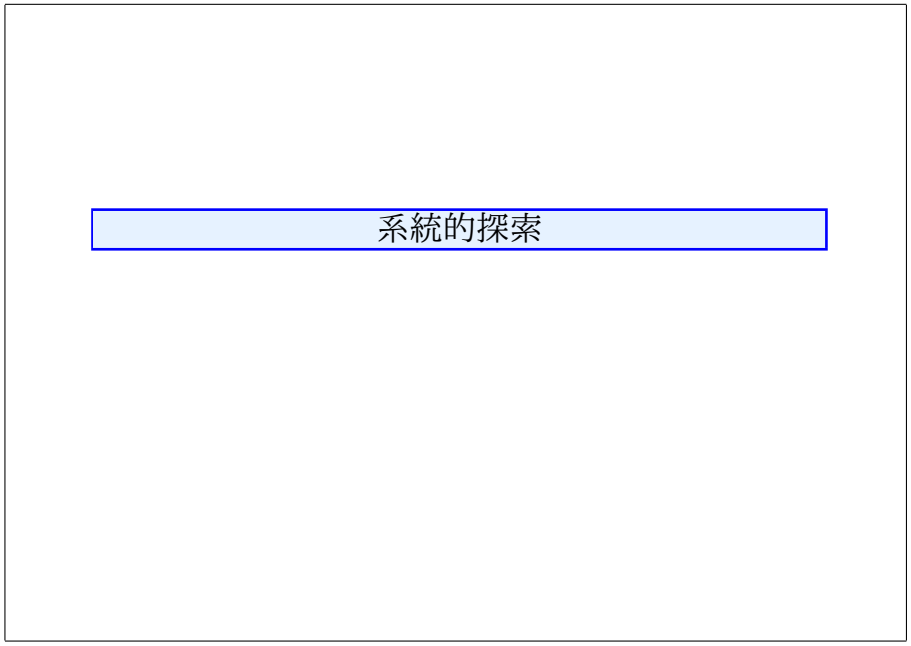
作用素 式変形

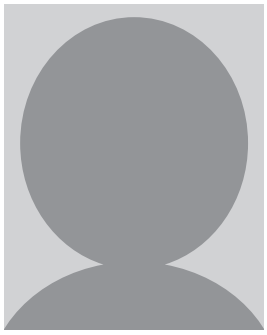
状態 式変形の過程の数式

- ハノイの塔のようなパズルだけでなく，例えば数学におけるある種の定理証明のように，条件が与えられていて，そこから定理に相当する数式まで，式変形を繰り返すというような場合，
- 初期状態を前提となる数式，
- 目標状態を定理に相当する数式，
- 作用素を数式の変形，
- 状態を式変形の過程で現れる数式とすれば，証明問題を状態空間モデルで定式化することができます．



- 状態空間モデルが，直面する問題のベストな定式化であるということは，それほど多くないとは思いますが，このような問題の捉え方もあるということは知っておいて損はないと思います．


系統的探索



- 状態空間は，状態を点，作用素を枝とするグラフとして表現されます.
- したがって，問題解決はグラフ上の探索に帰着されます.
- グラフの探索は，状態空間の探索に限らず幅広く応用されています.
- ここでは，代表的な探索の方法について紹介します.
- 探索の方法は，問題に関する知識を用いず虱潰しに探索する「系統的探索」と，問題に関する知識を利用する発見的探索に分けられます.
- まず，系統的探索についてお話します.

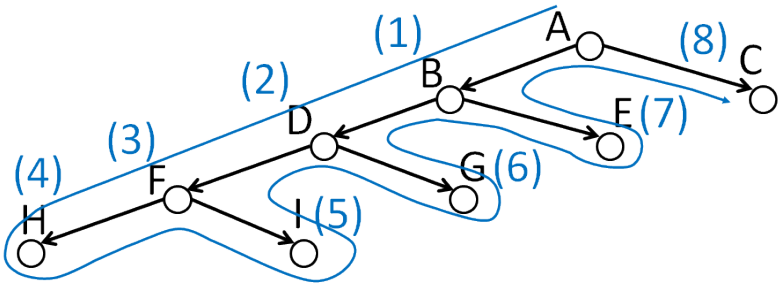
系統的探索

- 問題に関する知識を用いずシラミ潰しに探索
- 汎用的
- 原理的には解（目標）の発見が保証される

- 繰り返しになりますが，系統的探索は問題に関する知識を用いずに，虱潰しに探索をしていきますので，
- 問題によらない汎用的な手法で，原理的には解の発見が保証されます．
- ここで,「原理的には」と断ったのは，現実的には計算量が大きすぎて解けない問題があるからです．
- それでは，深さ優先探索から見ていきましょう．

系統的探索

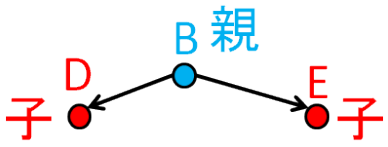
深さ優先探索（縦型探索）



- この図において，点 A を初期状態として，点 A から探索を行うとします．
- 深さ優先探索は，縦型探索とも呼ばれますが，点 A から深く伸びていく方向に探索を行い，一番奥の点に行き着いても目標状態に相当する点が見つからないときは, 最も近くの未探索の点を探索します．
- この過程で，目標となる点が見つかれば探索は成功して終了します．
- 最後まで探索して，目標となる点が見つからなければ，探索は失敗して終了します．
- この例では，点 A から，B, D, F, H の順に探索し，H で突き当ったら，H に近い未探索の点，I, G, E, 最後に C の順で探索します．

系統的探索

- 有向グラフにおける
  - 枝の始点 … 親
  - 枝の終点 … 子



- ここで，グラフに関して一つ用語を導入します．
- 有向グラフにおいて，枝で結ばれた点のうち，枝の始点を「親」，枝の終点を「子」と呼びます．
- この図においては，点 B が親で，点 D と点 E が子になります．
- それでは，深さ優先探索のアルゴリズムを見てみましょう．

系統的探索

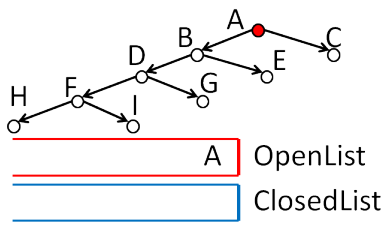
深さ優先探索

(1) 探索を開始する点を OpenList に入れる
　　ClosedList は空にする
(2) OpenList が空 (解がない) なら探索は失敗して終了
(3) OpenList の先頭の点 n を取り除き，ClosedList に入れる
(4) n が目標点であるなら探索は成功して終了
(5) n の子のうち，OpenList や ClosedList に含まれて
　　いない点はすべてを OpenList の**先頭**に入れて (2) へ
　　そのような点がなければ，そのまま (2) へ

- これが深さ優先探索のアルゴリズムです … と言っても，これを見て即座に探索の過程と結びつけるのは難しいかもしれませんが，まずは簡単に説明して，次に具体的にアルゴリズムの動作を見ていきます．
- まず，(1) は初期化で，探索する対象となる点を入れるリストである OpenList と，探索を終えた点を入れるリストである ClosedList を用意します．OpenList には探索を開始する点を入れ，ClosedList は空にします．
- (2) は終了判定で，もしこの時点で OpenList が空なら，解，つまり目標となる点は存在しないことになり，探索は解を見つけられずに失敗して終了します．
- OpenList が空でなければ，(3) へ進みます．
- (3) では，OpenList の先頭の点，これを点 n としますが，OpenList の先頭の点 n を取り除き，ClosedList に入れます．
- (4) は終了判定で，もし点 n が目標となる点であれば，探索は成功して終了します．
- n が目標となる点でなければ，(5) へ進みます．
- (5) では，点 n の子のうち，OpenList や ClosedList に含まれていない点をすべて OpenList の先頭に入れて，(2) へ戻ります．
- もし，そのような点がなければ，そのまま (2) へ戻ります．
- 以降，(2) と (5) の間を終了条件を満たすまで，繰り返します．
- それでは，このアルゴリズムの具体的な動作を例で見てみましょう．

系統的探索

(1) 探索を開始する点を OpenList に入れる
ClosedList は空にする

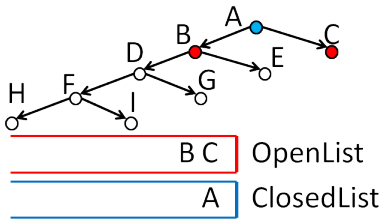


- 点 A から探索を開始し，最後まで探索させるために，目標は点 C としておきましょう．
- 最初は，(1) の初期化です．
- 点 A から探索しますので，OpenList に A を入れます．
- ClosedList は空にします．

系統的探索

(3) OpenList の先頭の点 n を取り除き，ClosedList に入れる

(5) n の子のうち，OpenList や ClosedList に含まれて
いない点はすべてを OpenList の**先頭**に入れて (2) へ
そのような点がなければ，そのまま (2) へ

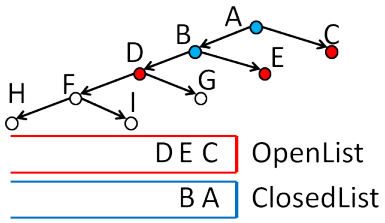


- OpenList は空ではないので，(2) の終了条件は満たさず，(3) へ進みます.
- (3) では，OpenList の先頭の点である点 A を取り除き，ClosedList に移します.
- 点 A は目標ではないので，(4) の終了条件は満たさず，(5) へ進みます.
- (5) では，点 A の子である，点 B と C を OpenList の先頭に入れます.
- ここでは，OpenList が空でしたので，OpenList は点 B, C だけになります.
- そして，(2) へ戻ります.

系統的探索

(3) OpenList の先頭の点 n を取り除き，ClosedList に入れる

(5) n の子のうち，OpenList や ClosedList に含まれて
いない点はすべてを OpenList の**先頭**に入れて (2) へ
そのような点がなければ，そのまま (2) へ

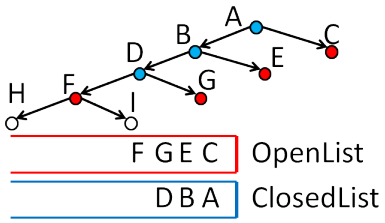


- ここから 2 巡目です.
- OpenList は空でないので，(2) の終了条件は満たさず，(3) へ進みます.
- (3) では，OpenList の先頭である点 B を取り除き，ClosedList に入れます.
- 点 B は目標ではないので，(4) の終了条件は満たさず，(5) へ進みます.
- (5) では，点 B の子である，点 D と E を OpenList の先頭に入れます.
- そして，(2) へ戻ります.

系統的探索

(3) OpenList の先頭の点 n を取り除き，ClosedList に入れる

(5) n の子のうち，OpenList や ClosedList に含まれて
いない点はすべてを OpenList の**先頭**に入れて (2) へ
そのような点がなければ，そのまま (2) へ

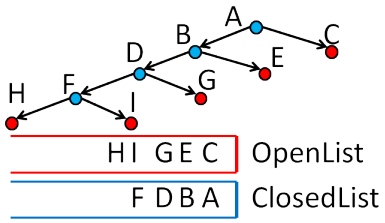


- 3 巡目です.
- OpenList は空でないので，(2) の終了条件は満たさず，(3) へ進みます.
- (3) では，OpenList の先頭である点 D を取り除き，ClosedList に入れます.
- 点 D は目標ではないので，(4) の終了条件は満たさず，(5) へ進みます.
- (5) では，点 D の子である，点 F と G を OpenList の先頭に入れます．そして，(2) へ戻ります.

系統的探索

(3) OpenList の先頭の点 n を取り除き，ClosedList に入れる

(5) n の子のうち，OpenList や ClosedList に含まれて
いない点はすべてを OpenList の**先頭**に入れて (2) へ
そのような点がなければ，そのまま (2) へ

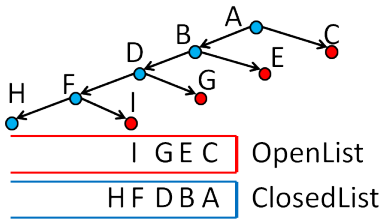


- 4 巡目です.
- OpenList は空でないので，(2) の終了条件は満たしません.
- (3) では，OpenList の先頭である点 F を取り除き，ClosedList に入れます.
- 点 F は目標ではないので，(4) の終了条件は満たさず，(5) へ進みます.
- (5) では，点 F の子である，点 H と I を OpenList の先頭に入れます.
- そして，(2) へ戻ります.

系統的探索

(3) OpenList の先頭の点 n を取り除き，ClosedList に入れる

(5) n の子のうち，OpenList や ClosedList に含まれて
いない点はすべてを OpenList の**先頭**に入れて (2) へ
そのような点がなければ，そのまま (2) へ

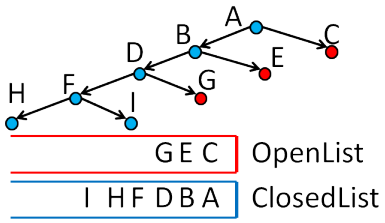


- 5 巡目です．
- OpenList は空でないので，(2) の終了条件は満たしません．
- (3) では，OpenList の先頭である点 H を取り除き，ClosedList に入れます．
- 点 H は目標ではないので，(4) の終了条件は満たさず，(5) へ進みます．
- さて，(5) ですが，点 H には子の点がありません．未探索の子がない場合，そのまま (2) へ戻ります．
- OpenList の先頭ですが，点 H の未探索の子の点はないので，一番近い未探索の点 I が OpenList の先頭，すなわち次の探索対象になっています．

系統的探索

(3) OpenList の先頭の点 n を取り除き，ClosedList に入れる

(5) n の子のうち，OpenList や ClosedList に含まれて
いない点はすべてを OpenList の**先頭**に入れて (2) へ
そのような点がなければ，そのまま (2) へ

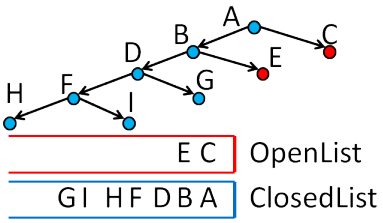


- 6 巡目です.
- OpenList は空でないので，(2) の終了条件は満たしません.
- (3) では，OpenList の先頭である点 I を取り除き，ClosedList に入れます.
- 点 I は目標ではないので，(4) の終了条件は満たさず，(5) へ進みます.
- (5) では，点 I の未探索の子の点がありませんので，そのまま (2) へ戻ります.
- 点 I の未探索の子の点はないので，一番近い未探索の点 G が OpenList の先頭，すなわち次の探索対象になっています.

系統的探索

(3) OpenList の先頭の点 n を取り除き，ClosedList に入れる

(5) n の子のうち，OpenList や ClosedList に含まれて
いない点はすべてを OpenList の**先頭**に入れて (2) へ
そのような点がなければ，そのまま (2) へ

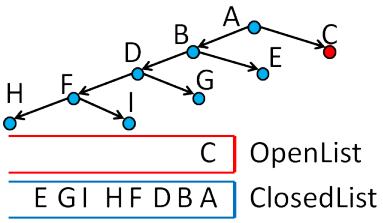


- 7 巡目です.
- OpenList は空でないので，(2) の終了条件は満たしません.
- (3) では，OpenList の先頭である点 G を取り除き，ClosedList に入れます.
- 点 G は目標ではないので，(4) の終了条件は満たさず，(5) へ進みます.
- (5) では，点 G の未探索の子の点がありませんので，そのまま (2) へ戻ります.

系統的探索

(3) OpenList の先頭の点 n を取り除き，ClosedList に入れる

(5) n の子のうち，OpenList や ClosedList に含まれて
いない点はすべてを OpenList の**先頭**に入れて (2) へ
そのような点がなければ，そのまま (2) へ

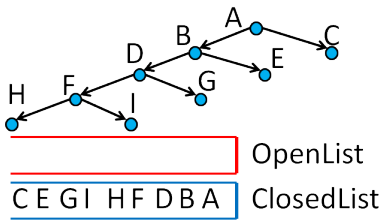


- 8 巡目です.
- OpenList は空でないので，(2) の終了条件は満たしません.
- (3) では，OpenList の先頭である点 E を取り除き，ClosedList に入れます.
- 点 E は目標ではないので，(4) の終了条件は満たさず，(5) へ進みます.
- (5) では，点 E の未探索の子の点がありませんので，そのまま (2) へ戻ります.

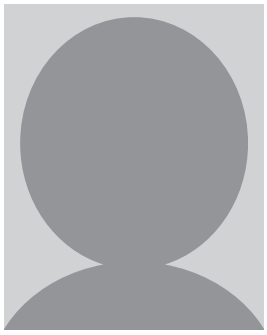
系統的探索

(3) OpenList の先頭の点 n を取り除き，ClosedList に入れる

(4) n が目標点であるなら探索は成功して終了



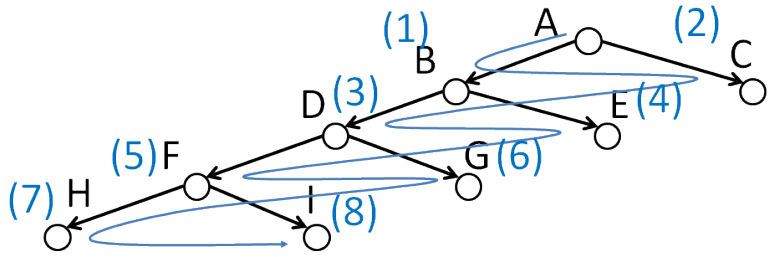
- 9 巡目です．
- OpenList は空でないので，(2) の終了条件は満たしません．
- (3) では，OpenList の先頭である点 C を取り除き，ClosedList に入れます．
- 点 C は目標ですので，(4) の終了条件を満たし，探索は成功して終了します．
- 仮に，目標となる点がない場合，次の 10 巡目で OpenList が空になり，(2) の終了条件が満たされ，探索は失敗して終了します．



- 以上，深さ優先探索の過程を見てきました．
- 今の例では，目標である点は，探索を開始した点から 1 ステップのところにありながら，大きく遠回りをしてしまいました．
- また，無限に深く伸びていくようなグラフでは，目標に到達しないこともあります．
- （ポーズ）
- それでは，次に幅優先探索についてお話します．
-

系統的探索

幅優先探索（幅型探索）



- この図において，点 A を初期状態として，点 A から探索を行うとします．
- 幅優先探索は，横型探索とも呼ばれますが，浅い点から順に探索していきます．
- この例では，点 A から，一番浅い点 B, 点 C, 次に 2 番目に浅い点 D，点 E，次に 3 番目に浅い点 F，点 G，最後に点 H と点 I の順で探索します．

系統的探索

幅優先探索

(1) 探索を開始する点を OpenList に入れる
ClosedList は空にする

(2) OpenList が空 (解がない) なら探索は失敗して終了

(3) OpenList の先頭の点 n を取り除き，ClosedList に入れる

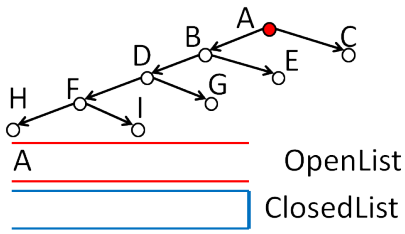
(4) n が目標点であるなら探索は成功して終了

(5) n の子のうち，OpenList や ClosedList に含まれて
いない点はすべてを OpenList の**末尾**に入れて (2) へ
そのような点がなければ，そのまま (2) へ

- これが幅優先探索のアルゴリズムです.
- 深さ優先探索のアルゴリズムと似ている…と思われるかもしれませんが，実際その通りで，深さ優先探索のアルゴリズムと異なるのは，(5) において，OpenList から取り除いた点 n の子のうち，OpenList や ClosedList に含まれていない点のすべてを，OpenList の末尾に入れるというところです.
- 深さ優先探索では，n の子を OpenList の先頭に入れるのに対して，幅優先探索では，n の子を OpenList の末尾に入れます.
- これだけの違いで，まったく異なる探索になります.
- それでは，このアルゴリズムの具体的な動作を例で見てみましょう.
-

系統的探索

(1) 探索を開始する点を OpenList に入れる
ClosedList は空にする

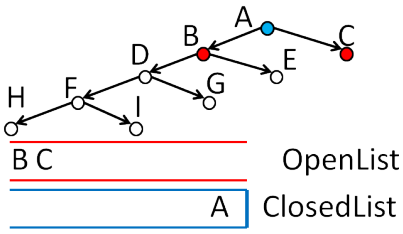


- 点 A から探索を開始し，最後まで探索させるために，目標は点 I としておきましょう．
- 最初は，(1) の初期化です．
- 点 A から探索しますので，OpenList に点 A を入れます．
- ClosedList は空にします．

系統的探索

(3) OpenList の先頭の点 n を取り除き，ClosedList に入れる

(5) n の子のうち，OpenList や ClosedList に含まれて
いない点はすべてを OpenList の**末尾**に入れて (2) へ
そのような点がなければ，そのまま (2) へ

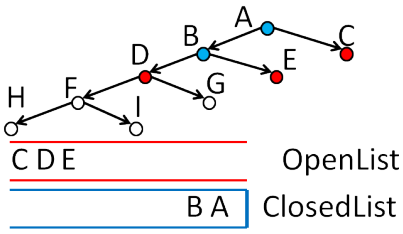


- OpenList は空でないので，(2) の終了条件条件は満たさず，(3) へ進みます．
- (3) では，OpenList の先頭の点である A を取り除き，ClosedList に移します．
- 点 A は目標ではないので，(4) の終了条件は満たさず，(5) へ進みます．
- (5) では，点 A の子である，点 B と C を OpenList の末尾に入れます．ここでは，OpenList が空でしたので，OpenList は点 B, C だけになります．そして，(2) へ戻ります．

系統的探索

(3) OpenList の先頭の点 n を取り除き，ClosedList に入れる

(5) n の子のうち，OpenList や ClosedList に含まれて
いない点はすべてを OpenList の末尾に入れて (2) へ
そのような点がなければ，そのまま (2) へ

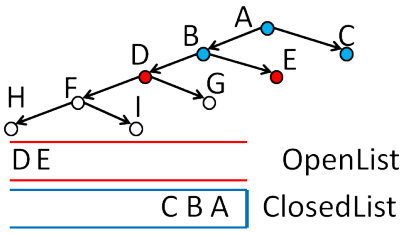


- ここから 2 巡目です.
- 終了条件 (2), (4) は両方とも満たされません.
- (3) では，OpenList の先頭である点 B を取り除き，ClosedList に入れます.
- (5) では，点 B の子である，点 D と E を OpenList の末尾に入れます.
- そして，(2) へ戻ります.

系統的探索

(3) OpenList の先頭の点 n を取り除き，ClosedList に入れる

(5) n の子のうち，OpenList や ClosedList に含まれて
いない点はすべてを OpenList の**末尾**に入れて (2) へ
そのような点がなければ，そのまま (2) へ

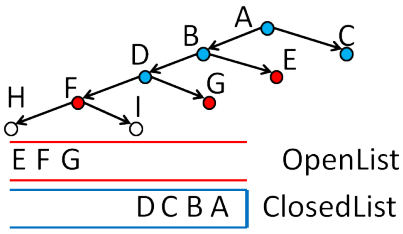


- 3 巡目です.
- 終了条件 (2), (4) は両方とも満たされません.
- (3) では，OpenList の先頭である点 C を取り除き，ClosedList に入れます.
- 点 C には子の点がありませんので，(5) ではそのまま (2) へ戻ります.
- これで一番浅い，点 B と点 C が探索済みになりました.

系統的探索

(3) OpenList の先頭の点 n を取り除き，ClosedList に入れる

(5) n の子のうち，OpenList や ClosedList に含まれて
いない点はすべてを OpenList の**末尾**に入れて (2) へ
そのような点がなければ，そのまま (2) へ

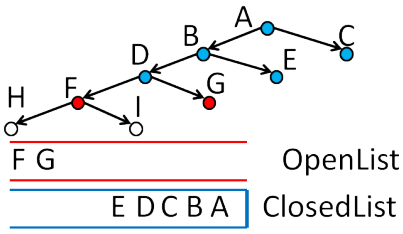


- 4 巡目です.
- 終了条件 (2), (4) は両方とも満たされていません.
- (3) では，OpenList の先頭である点 D を取り除き，ClosedList に入れます.
- (5) では，点 D の子である，点 F と G を OpenList の末尾に入れます.
- そして，(2) へ戻ります.

系統的探索

(3) OpenList の先頭の点 n を取り除き，ClosedList に入れる

(5) n の子のうち，OpenList や ClosedList に含まれて
いない点はすべてを OpenList の末尾に入れて (2) へ
そのような点がなければ，そのまま (2) へ

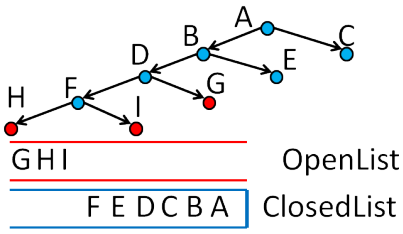


- 5 巡目です.
- 終了条件 (2), (4) は両方とも満たされていません.
- (3) では，OpenList の先頭である点 E を取り除き，ClosedList に入れます.
- (5) では，点 E の子の点はありませんので，そのまま (2) へ戻ります.
- これで二番目に浅い，点 D と点 E が探索済みになりました.

系統的探索

(3) OpenList の先頭の点 n を取り除き，ClosedList に入れる

(5) n の子のうち，OpenList や ClosedList に含まれて
いない点はすべてを OpenList の**末尾**に入れて (2) へ
そのような点がなければ，そのまま (2) へ

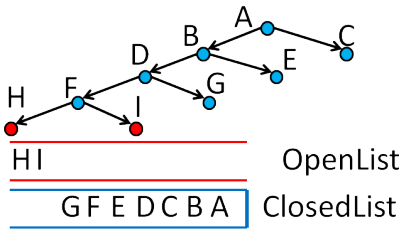


- 6 巡目です.
- 終了条件 (2), (4) は両方とも満たされていません.
- (3) では，OpenList の先頭である点 F を取り除き，ClosedList に入れます.
- (5) では，点 F の子である，点 H と I を OpenList の末尾に入れます.
- そして，(2) へ戻ります.

系統的探索

(3) OpenList の先頭の点 n を取り除き，ClosedList に入れる

(5) n の子のうち，OpenList や ClosedList に含まれて
いない点はすべてを OpenList の末尾に入れて (2) へ
そのような点がなければ，そのまま (2) へ

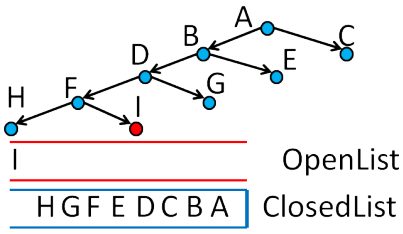


- 7 巡目です.
- 終了条件 (2), (4) は両方とも満たされていません.
- (3) では，OpenList の先頭である点 G を取り除き，ClosedList に入れます.
- (5) では，点 G の子の点はありませんので，そのまま (2) へ戻ります.
- これで三番目に浅い，点 F と点 G が探索済みになりました.

系統的探索

(3) OpenList の先頭の点 n を取り除き，ClosedList に入れる

(5) n の子のうち，OpenList や ClosedList に含まれて
いない点はすべてを OpenList の末尾に入れて (2) へ
そのような点がなければ，そのまま (2) へ



I　OpenList

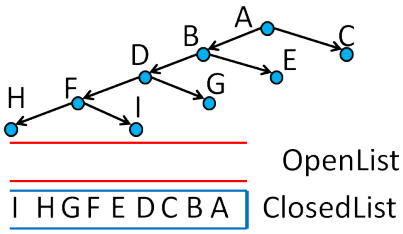
H G F E D C B A　ClosedList

- 8 巡目です.
- 終了条件 (2), (4) は両方とも満たされていません.
- (3) では，OpenList の先頭である点 H を取り除き，ClosedList に入れます.
- (5) では，点 H の子の点はありませんので，そのまま (2) へ戻ります.

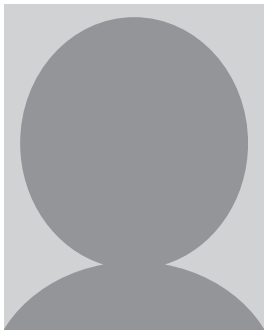
系統的探索

(3) OpenList の先頭の点 n を取り除き，ClosedList に入れる

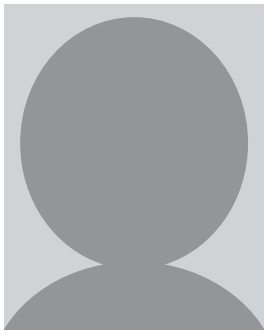
(4) n が目標点であるなら探索は成功して終了



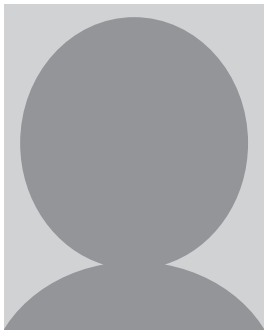
- 9 巡目です.
- OpenList は空でないので，終了条件 (2) は満たされていません.
- (3) では，OpenList の先頭である点 I を取り除き，ClosedList に入れます.
- 点 I は目標ですから，終了条件 (4) が満たされ，探索は成功して終了します.



- 以上，幅優先探索の過程を見てきました．
- 幅優先探索は，浅い点から探索していきますので，目標に至る最短経路を発見することができます．
- また，無限に深く伸びていくようなグラフでも，遠回りすることなく目標を見つけることができます．
- （ポーズ）
- こう言いますと，深さ優先探索に比べ，幅優先探索が全面的に優れているように思えますが，幅優先探索にも問題があります．



- 幅優先探索では，見つかった点を全て記録する必要がありますので，メモリーを大きく消費します．
- 特に，枝別れの数が多く，つまり親の点が多数の子の点を持つ場合，記憶する点の数は膨大になり，規模の大きな探索は現実的に不可能になります．
- 一方，先ほど一つも褒めなかった深さ優先探索ですが，メモリー消費は幅優先探索よりはるかに小さいという利点があります．
- 深さ優先探索と幅優先探索にはそれぞれ長所短所がありますが，これらの長所を合わせた方法が，次にお話する反復深化深さ優先探索です．



- 反復深化深さ優先探索では，深さ優先探索の，探索の深さに制限を設け，探索の深さが制限に達したら，それ以上深い方向へは探索しないようにします．
- 深さの制限は，最初は 1 として，解を発見せず探索が終了するたびに，深さの制限を 1 ずつ大きくしていきます．
- したがって，深さ優先探索でありながら，浅い点から探索していきますので，幅優先探索的な性質を持ちます．

系統的探索

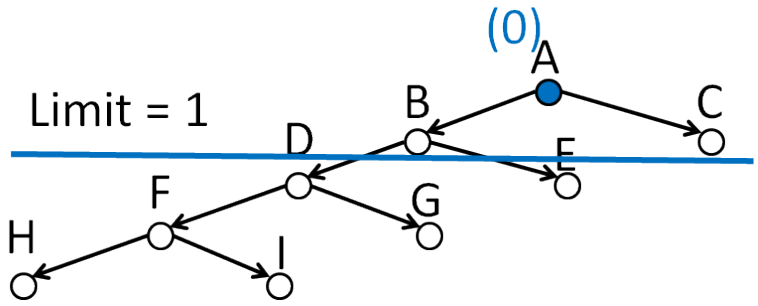
◆反復深化深さ優先探索

(0) 探索する深さの制限 **Limit** を **1** とする
(1) 探索を開始する点を **OpenList** に入れ，**ClosedList** は空にする
(2) **OpenList** が空なら **Limit** を **1** 大きくし，**(1)** へ
(3) **OpenList** の先頭の点 n を取り除き，**ClosedList** に入れる
(4) n が目標点であるなら探索は成功して終了
(5) n の深さが **Limit** より小さい場合，n の子のうち，**OpenList** や **ClosedList** に含まれていない点はすべてを **OpenList** の先頭に
入れて **(2)** へ．それ以外の場合は，そのまま **(2)** へ

- これが反復深化深さ優先探索のアルゴリズムです.
- ほとんど深さ優先探索のアルゴリズムと同じですが，最初に探索の深さの制限である Limit を 1 に設定し，深さの制限内で解が見つからなかった場合には，Limit を 1 つ増やして，深さ優先探索をやり直しています.
- それでは，このアルゴリズムの具体的な動作を例で見てみましょう.

系統的探索

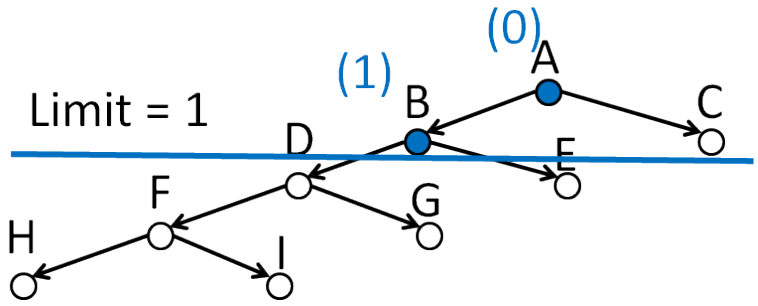
反復深化深さ優先探索



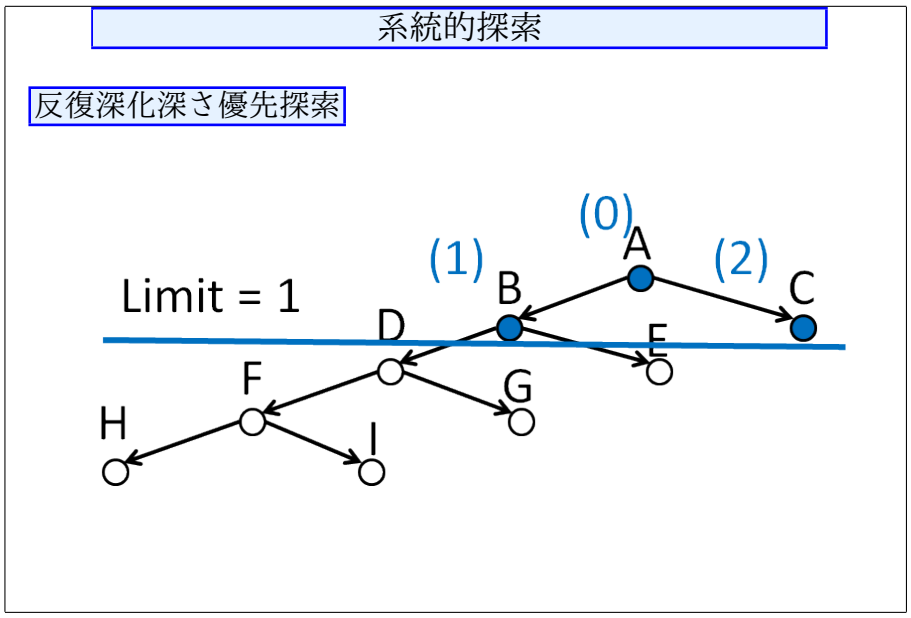
- まず，深さの制限 Limit が 1 の時です.
- 探索の開始は点 A です.

系統的探索

反復深化深さ優先探索



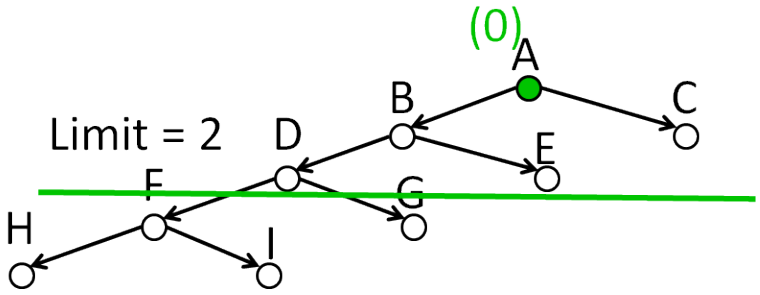
- 点 A の子は点 B と点 C です.
- まず，点 B を探索します.



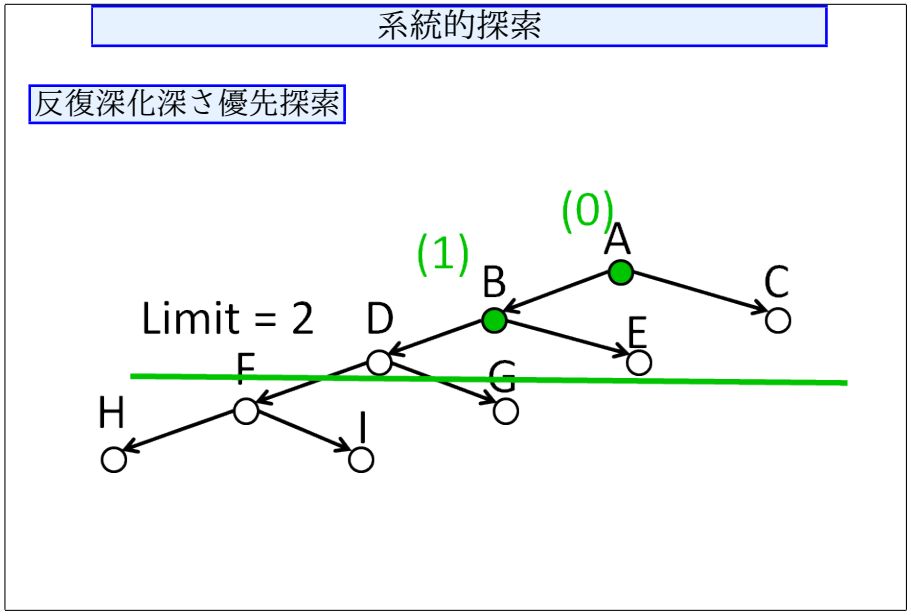
- 点 B には子の点，D と E がありますが，これらは深さの制限 1 に引っかかりますので，ここでは探索せず，残りの未探索の点 C を探索します．
- 深さが 1 の点は B と C ですので，Limit が 1 の探索は終了し，Limit を 2 に増やして，探索をやり直します．

系統的探索

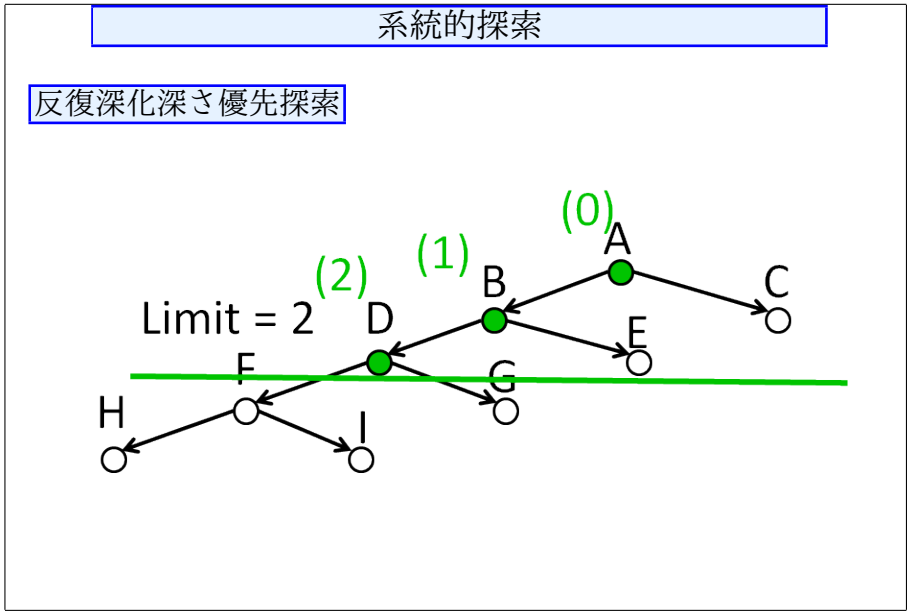
反復深化深さ優先探索



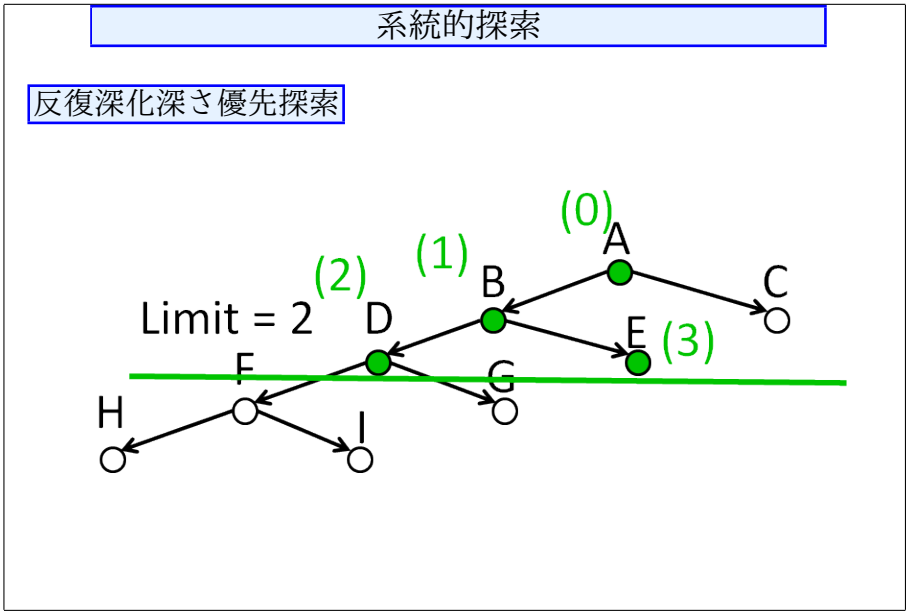
- Limit を 2 にして，最初から探索をやり直します．



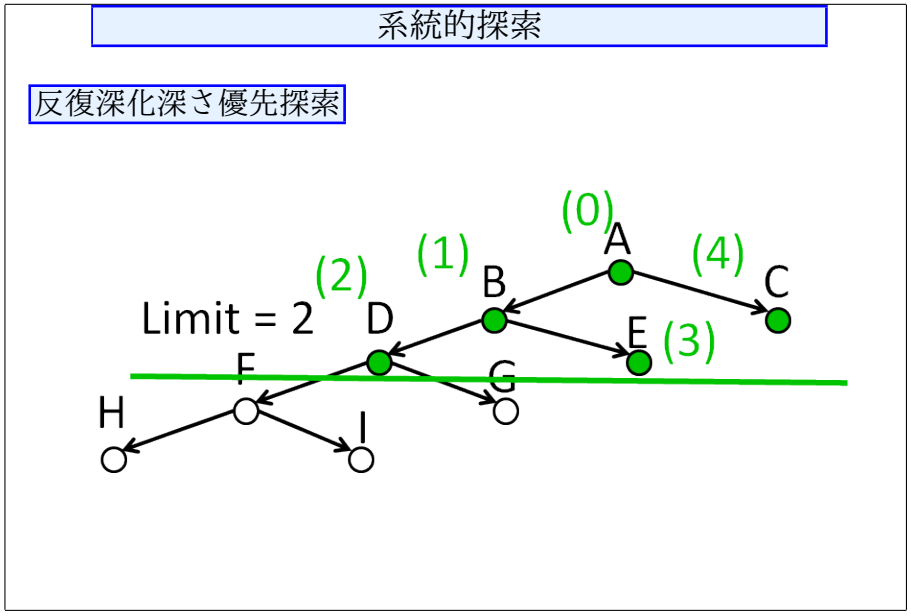
- 点 A の子，点 B と C が OpenList の先頭に入り，まず点 B を探索します.



- 深さ優先探索ですので，
- 点 B の子，点 D と E が OpenList の先頭に入り，まず点 D を探索します．



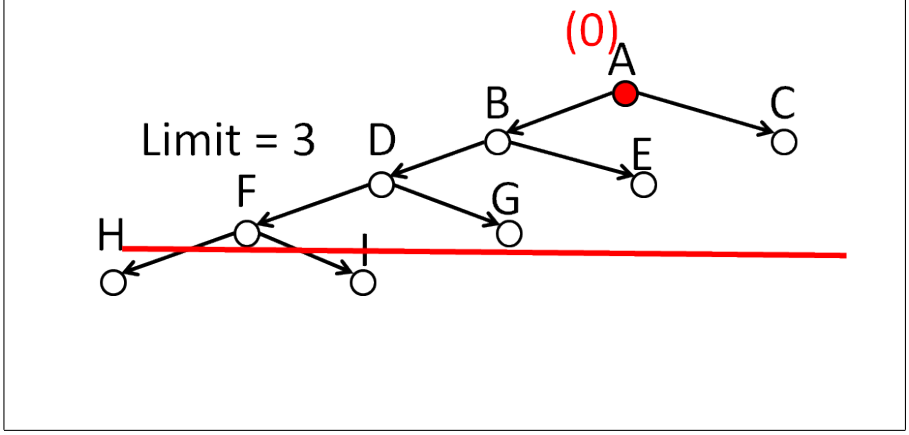
- 点 D には子の点，F と G がありますが，これらは深さの制限 2 に引っかかりますので，ここでは探索せず，残りの未探索の点 E を探索します．



- 最後に残った未探索点である点 C を探索し，深さ制限 2 の探索は終了します．

系統的探索

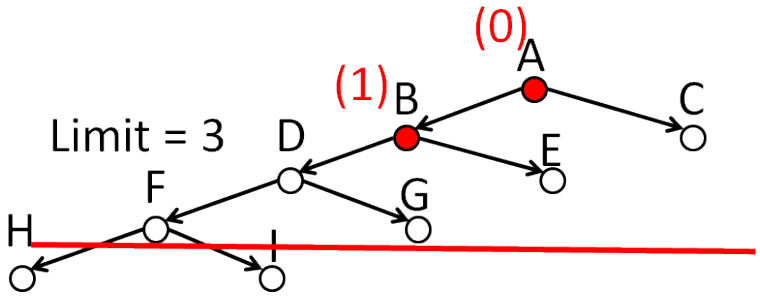
反復深化深さ優先探索



- Limit を 3 にして，最初から探索をやり直します．

系統的探索

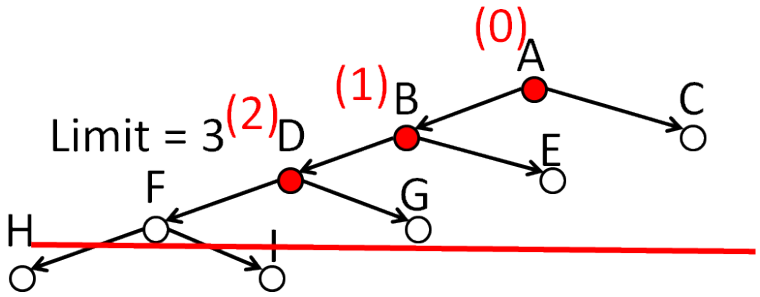
反復深化深さ優先探索



- 深さ優先探索ですから，まず点 B

系統的探索

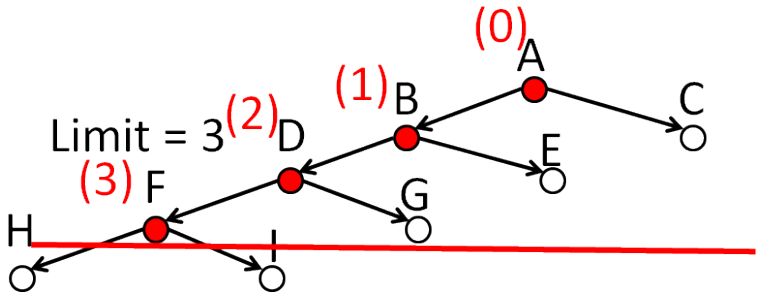
反復深化深さ優先探索



- 次に点 D

系統的探索

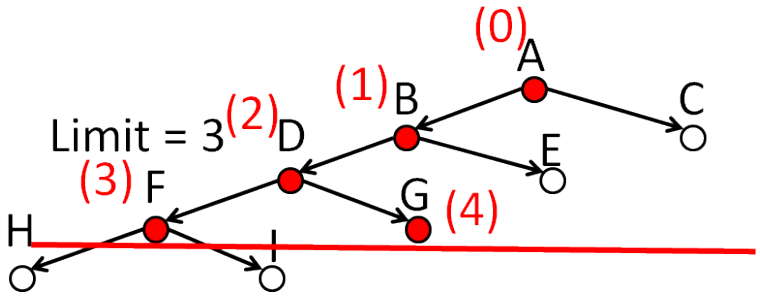
反復深化深さ優先探索



- 次に点 F
- これより先は深さ制限 3 に引っかかります.

系統的探索

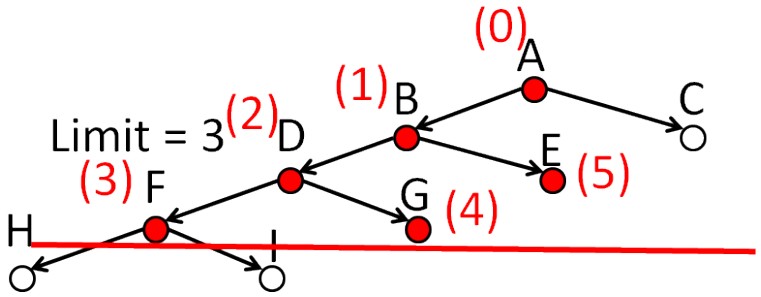
反復深化深さ優先探索



- したがって，次は点 G

系統的探索

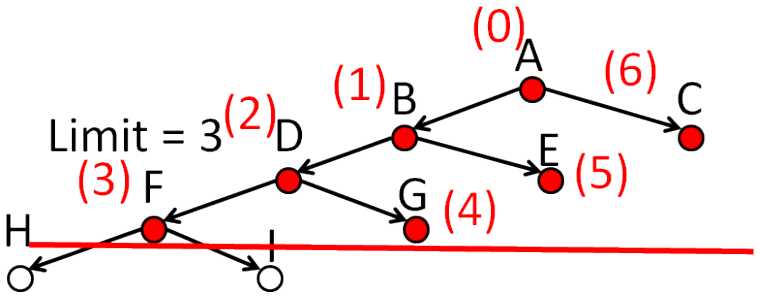
反復深化深さ優先探索



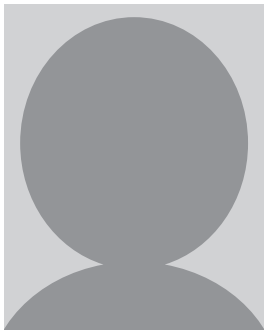
- そして点 E

系統的探索

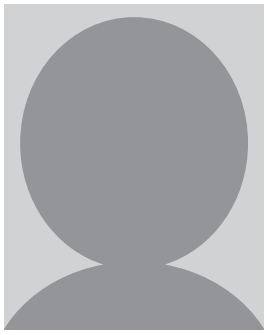
反復深化深さ優先探索



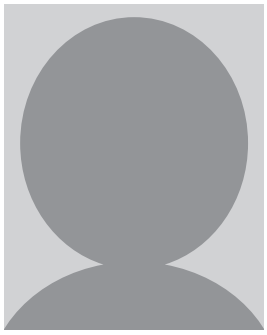
- 最後に点 C を探索して，深さ制限 3 での探索が終わります．
- 深さ制限 Limit を 4 にすると，一番深い点まで達するので，通常の深さ優先探索と同じになります．



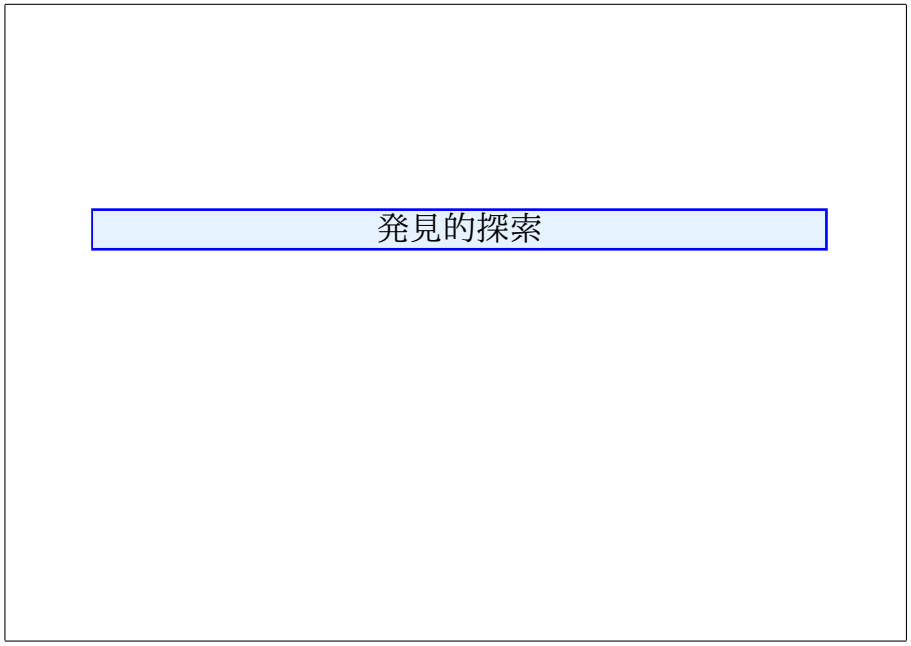
- 以上，反復深化深さ優先探索の動作を見てきました．
- 反復深化深さ優先探索は，深さ優先探索と同様に，メモリーの消費が小さく，幅優先探索と同様に，浅い点から探索していきますので，無限に深く伸びていくようなグラフでも，目標が存在するなら，見つけることができます．

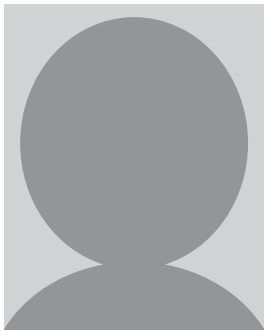


- 反復深化深さ優先探索の動作を見て気づかれた方もいるかと思いますが，反復深化深さ優先探索は，深さの制限を更新するたびに，最初から探索をやり直していて，既に探索された目標でない点を何度も探索していました．
- これは余計な探索ではないかと思われたんではないでしょうか？

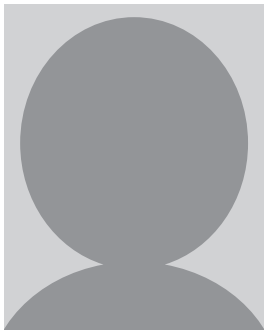


- 確かに無駄は無駄ではあるのですが，探索の重複は探索全体のサイズからすると，それほど大きくないことが多く，反復深化深さ優先探索は実際によく利用されます.


発見的探索



- 3 種類の系統的探索法についてお話しました.
- いずれの方法も，問題に関する知識を用いないので，問題の記述さえ与えておけば，原理的にはどんな問題にも対応できるのですが，現実には，著しい非効率，あるいは現実的な時間では解けないという問題にぶち当たります.
- そこで，問題に関する知識を利用して，探索効率の向上を図ろうというのは，ごく自然な発想です.



- ここでは，問題に関する知識を利用して，探索効率の向上を図る，発見的探索の方法を幾つか紹介します.

発見的探索

発見的（ヒューリスティック）探索

- 問題に関する知識を利用
- 系統的探索では現実的に解を発見できない場合
- 問題に関する知識は経験的知識を含む
  → 必ずしも最適解に導くとは限らない
  → 一般には最適解が得られる保証はない

- 発見的探索は，ヒューリスティック探索とも言いますが，問題に関する知識を利用し，系統的探索より効率的に問題を解こうというものです．
- 一口に問題に関する知識と言いましても，理論的に正しい知識だけでなく，経験的知識を含むこともあります．
- したがって，一般には最適解が得られる保証はありませんが，適切に利用すれば有効な手段になります．
- それでは，最初に最も単純な発見的探索法である山登り法について説明します．

発見的探索

山登り法

- 現在の状態から，より評価値の高い状態へ探索を進める
- アルゴリズムは単純
- 開始点から目標までの経路の各点において，子の点のうち，経路上の点の評価値が最も良くなければ，最適解には達しない

- 山登り法は，現在の状態から移れる状態のうち，最も評価値の高い状態に移るという単純な探索法です．
- 山登り法のアルゴリズムは次の通りです．

発見的探索

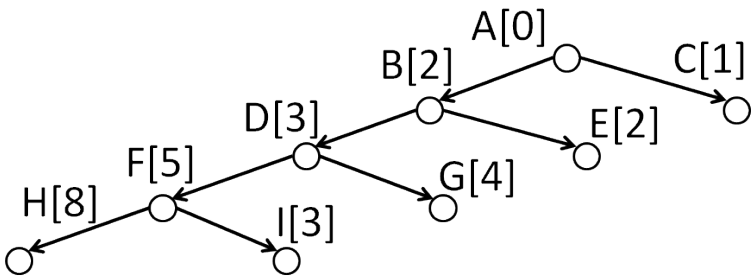
山登り法

(0) $f(n)$: 点 n の評価関数（ヒューリスティック関数）
(1) 開始点を探索対象の点 n とする
(2) もし，点 n に適用可能な作用素がなければ，
　　点 n を解として出力して，探索を終了
(3) 点 n の子の点のうち，評価値が $f(n)$ より良い点が
　　存在しなければ，点 n を解として出力して，探索を終了．評価値が $f(n)$ より良い点が存在すれば，
　　評価値が最も良い点を n として，(2) へ

- 点 n の評価値を与える評価関数を $f(n)$ とします．
- まず，開始点を探索対象の点 n とします．
- もし，点 n に適用可能な作用素がなければ，すなわち子の点がなければ，点 n を最適解として出力して，探索は終了します．
- 点 n に子の点があっても，評価値が $f(n)$ より良い点が存在しなければ，点 n を最適解として出力して，探索は終了します．
- 評価値が $f(n)$ よりも良い子の点が存在すれば，評価値が最も良い点を点 n として，探索を繰り返します．
- それでは，具体例を見てみましょう．

発見的探索

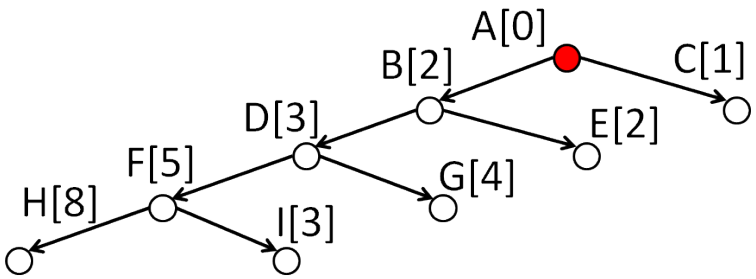
山登り法



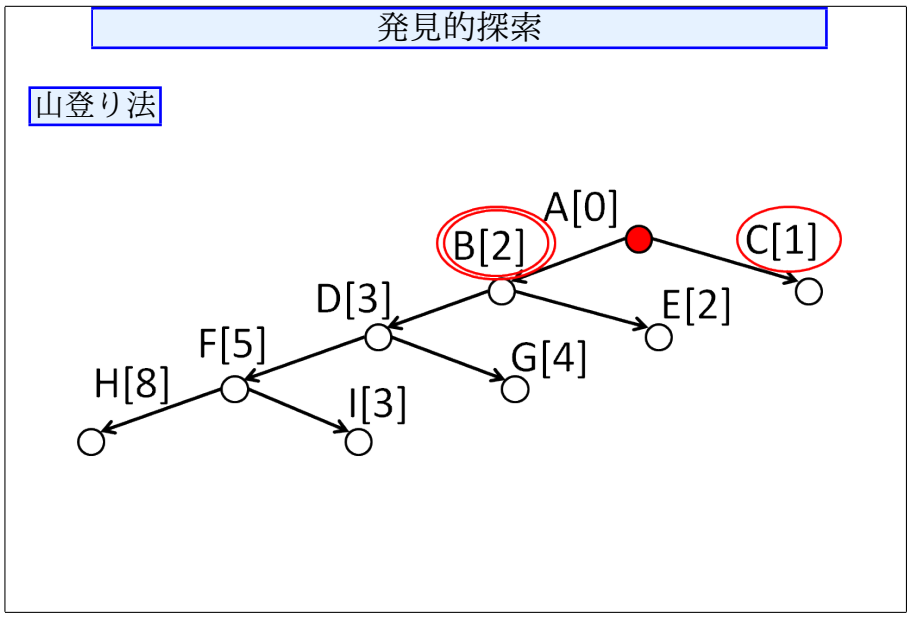
- 図に示すようなグラフを探索します.
- 点の名前のアルファベットの脇に書かれている括弧内の数値はその点の評価値を表します.
- 評価値は大きいほど良いことにします.

発見的探索

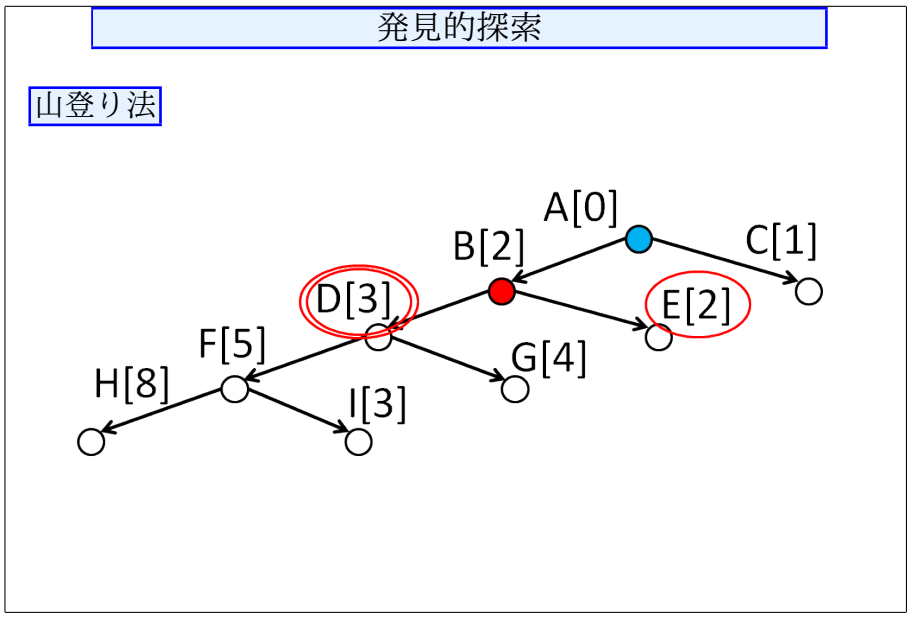
山登り法



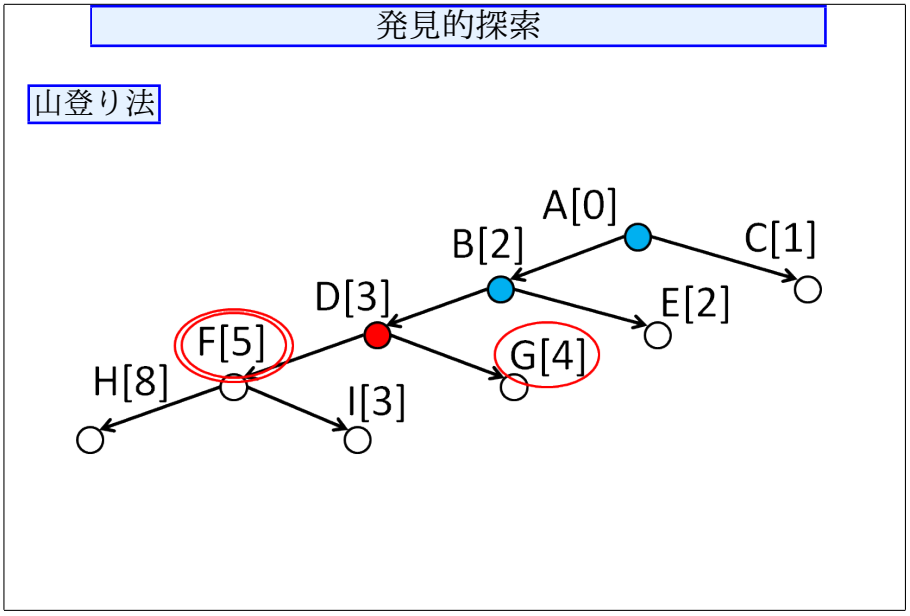
- 点Ａから探索を開始し，評価が最高の点を目指します.



- 点Ａの子の点は，点Ｂと点Ｃの二つです．
- 点Ｂの評価値は 2，点Ｃの評価値は 1，点Ａの評価値は 0 ですから，最も評価値の高い点Ｂに進みます．



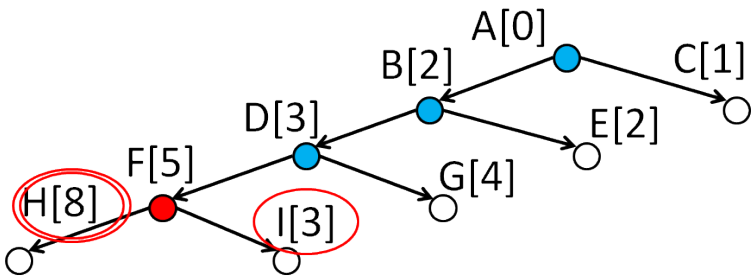
- 点Ｂの子は，点Ｄと点Ｅです．
- 点Ｄの評価値は３，点Ｅの評価値は２，点Ｂの評価値は２ですから，最も評価値の高い点Ｄに進みます．



- 点Ｄの子は，点Ｆと点Ｇです．
- 点Ｆの評価値は５，点Ｇの評価値は４，点Ｄの評価値は３ですから，最も評価値の高い点Ｆに進みます．

発見的探索

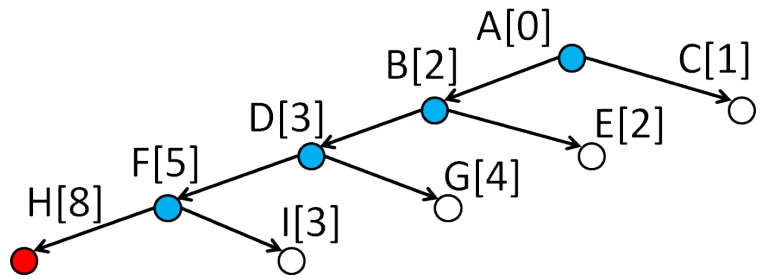
山登り法



- 点Ｆの子は，点Ｈと点Ｉです．
- 点Ｈの評価値は８，点Ｉの評価値は３，点Ｆの評価値は５ですから，最も評価値の高い点Ｈに進みます．

発見的探索

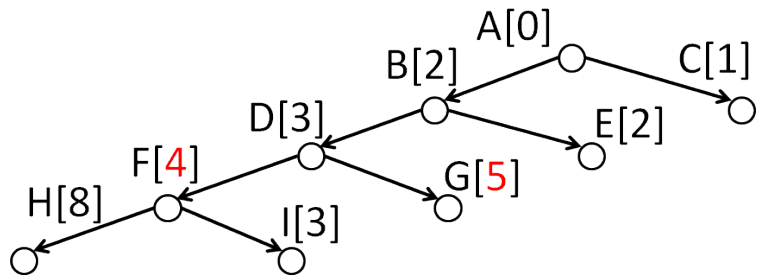
山登り法



- 山登り法による探索の結果，めでたく最も評価の高い点Hに辿りつくことができました．
- （ポーズ）
- しかし，いつもこのようにうまくいく訳ではありません．

発見的探索

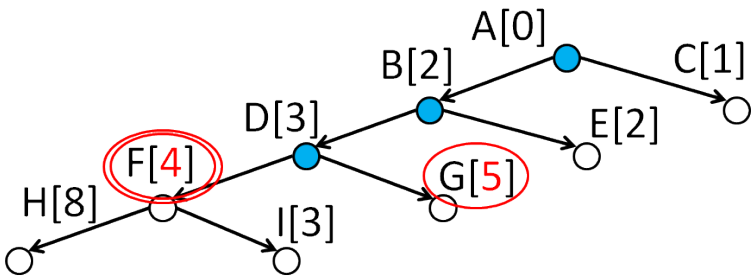
山登り法



- 点の評価値を次のように変更してみます.
- （ポーズ）
- 点ＦとＧの評価値を入れ替えました.

発見的探索

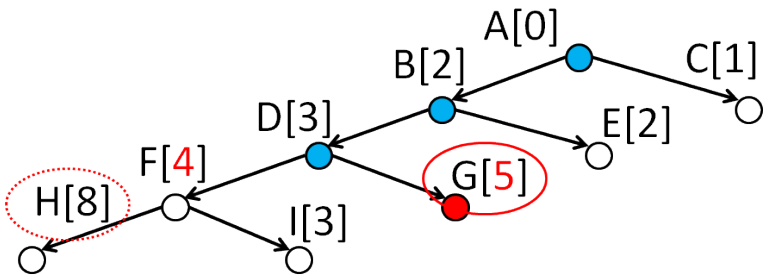
山登り法



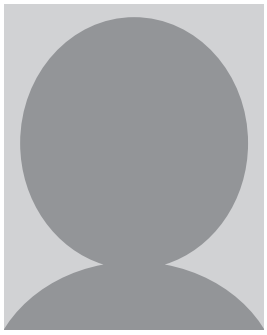
- 点Ｄまでは，先ほどと同様です．
- 点Ｄの子は，点Ｆと点Ｇです．
- 点Ｆの評価値は４，点Ｇの評価値は５，点Ｄの評価値は３ですから，最も評価値の高い点Ｇに進みます．

発見的探索

山登り法

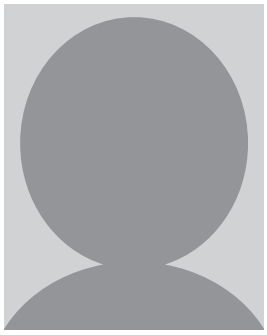


- 点Ｇに辿りついて探索は終了します.
- しかし，点Ｇより評価の高い点Ｈが存在します.
- つまり，点Ａから点Ｈまでの経路上の点が常に移動できる点の中で最も評価が高くなくては，最高評価の点にはたどり着けません.
- 山登りで言えば，山頂に到着するまでに峰に到着すると，そこを山頂と見なして，山登りを終了してしまうことになります.



- 山登り法の他に，発見的探索法としては，最適探索，最良優先探索，A* アルゴリズム等が古くから知られています．
- 興味のある方は参考文献などをご覧ください．

ゲームの木の探索



- 最後にゲームの木の探索についてお話します.
- チェスや将棋など 2 人のプレイヤーが交互に手を指し勝敗を競うゲームも人工知能の初期からの研究テーマです.
- 将棋や囲碁のトップレベルのプロにコンピュータプログラムが勝ったことが一般にも報道されたので，ご存知の方もいるかと思います.

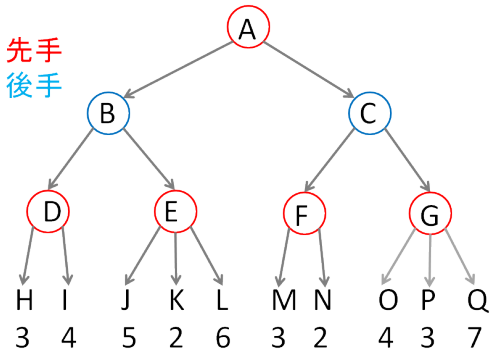
ゲームの木の探索

- チェス，将棋，三目並べ…
- 展開型ゲーム
  - ゲームの木で表現可能
  - 先読み推論で原理的には解ける
- 二人零和ゲーム … ミニマックス法で探索

- こういったゲームは，ゲーム理論でいう展開型ゲームとして扱います.
- 大雑把な言い方ですが，勝ち負けを競うゲームなので，二人零和ゲームとして扱います.
- 展開型ゲームはゲームの木で表現することができ，先読み推論により解くことができます.
- 二人零和ゲームですから，マクシミン戦略で最適解を得ることができます.
- 二人零和ゲームのゲームの木の探索はミニマックス法で行うことができます.
- それでは，ミニマックス法による探索の概要を例により説明します.

ゲームの木の探索

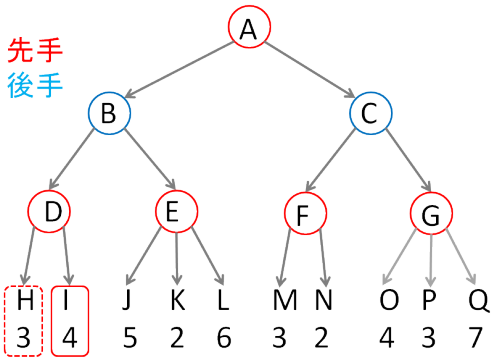
ミニマックス法



- 図に示すように先手，後手，先手と 3 手読むようなゲームを考えます．
- 赤い点が先手の意思決定点，青い点が後手の意思決定点です．
- 一番下の数字は，その状況における先手の評価値を表しています．
- 評価値が大きいほど，先手にとって良い状況となります．

ゲームの木の探索

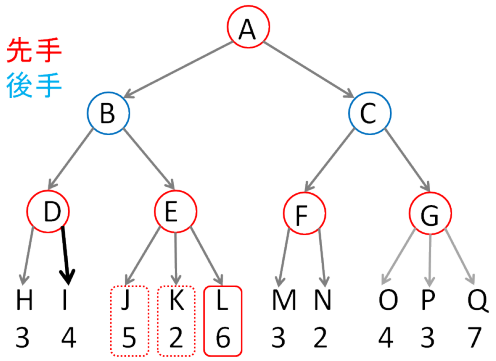
ミニマックス法



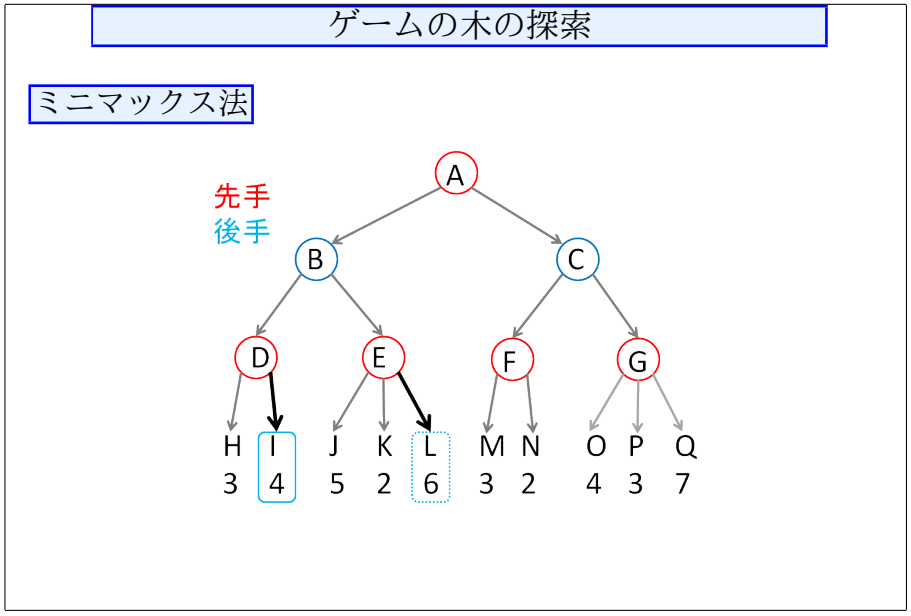
- 先手の立場から３手先を読んで，第１手を決定します．
- 点Ａから，例えば深さ優先探索で，第３手の候補である点Ｄに着きます．
- 点Ｄは先手の意思決定点です．
- 点Ｄにおいて，先手がＨを選択すれば，評価値は３，Ｉを選択すれば評価値は４ですから，評価値の高いＩを選択します．

ゲームの木の探索

ミニマックス法



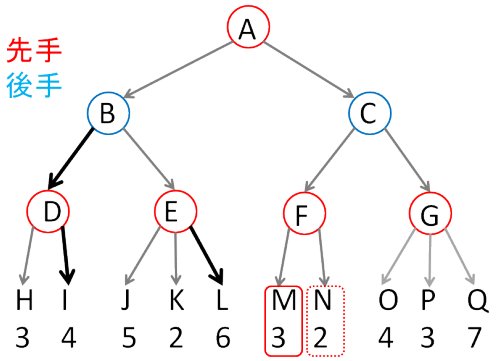
- 次に，第３手の候補である点Ｅに着きます.
- 点Ｅにおいて，Ｊ，Ｋ，Ｌの中から評価値が６で最も大きいＬを選択します.



- 次に，点Ｂに着きます．
- 点Ｂは，第２手の候補で，後手の意思決定点です．
- 点Ｂにおいて，後手がＤを選択すると，先手がＩを選択し，先手の評価値は４になります．
- 点Ｂにおいて，後手がＥを選択すると，先手がＬを選択し，先手の評価値は６になります．
- 後手は，先手の評価値を小さくしたいので，この場合先手の評価値が小さいＤを選択し，その結果，先手の評価値は４になります．

ゲームの木の探索

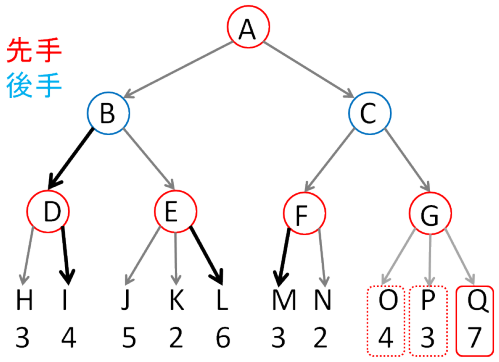
ミニマックス法



- 次に，右側に移り，第３手の候補である点Ｆに着きます．
- 点Ｆは先手の意思決定点です．
- 点Ｆにおいて，Ｍ，Ｎの中から評価値が３で大きいＭを選択します．

ゲームの木の探索

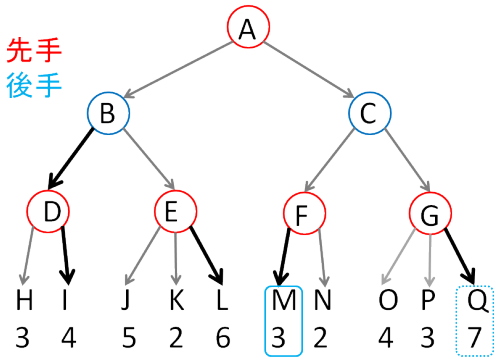
ミニマックス法



- 次に，第３手の候補である点Ｇに着きます．
- 点Ｇも先手の意思決定点です．
- 点Ｇにおいて，Ｏ，Ｐ，Ｑの中から評価値が７で最も大きいＱを選択します．

ゲームの木の探索

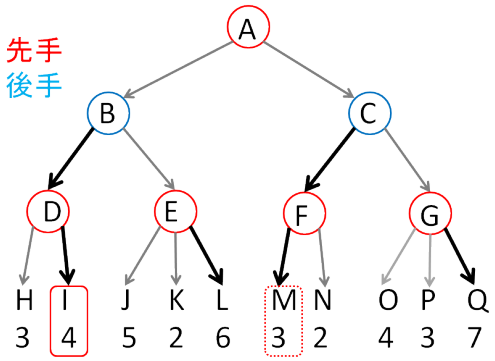
ミニマックス法



- 次に，点Ｃに着きます．
- 点Ｃは，第２手の候補で，後手の意思決定点です．
- 点Ｃにおいて，後手がＦを選択すると，先手がＭを選択し，先手の評価値は３になります．
- 点Ｃにおいて，後手がＧを選択すると，先手がＱを選択し，先手の評価値は７になります．
- 後手は，先手の評価値を小さくしたいので，この場合先手の評価値が小さいＦを選択し，その結果，先手の評価値は３になります．

ゲームの木の探索

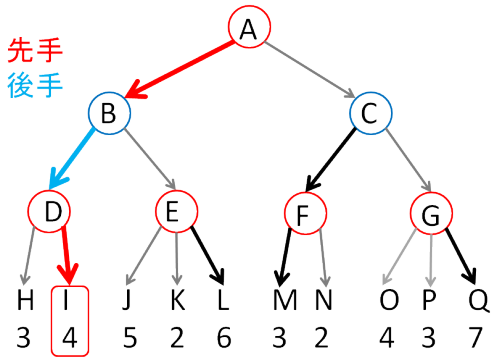
ミニマックス法



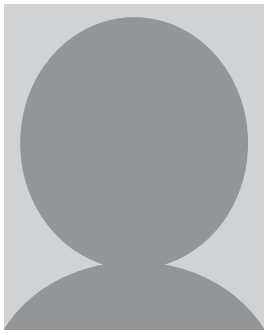
- 最後に点Ａに着きます.
- 点Ａは先手の第１手の意思決定点です.
- 点Ａにおいて，先手がＢを選択すると，後手はＤを選択し，そうすると先手はＩを選択して，その結果，先手の評価値は４となります.
- 点Ａにおいて，先手がＣを選択すると，後手はＦを選択し，そうすると先手はＭを選択して，その結果，先手の評価値は３となります.

ゲームの木の探索

ミニマックス法



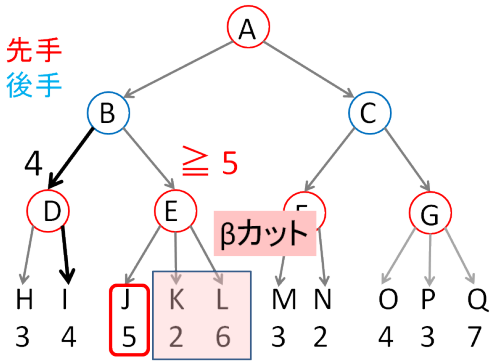
- まとめますと，点Ａにおいて，先手がＢを選択すると，先手の評価値は４となります．
- 点Ａにおいて，先手がＣを選択すると，先手の評価値は３となります．
- したがって，先手が自分の評価値を大きくするには，点ＡにおいてＢを選択するべきとなります．



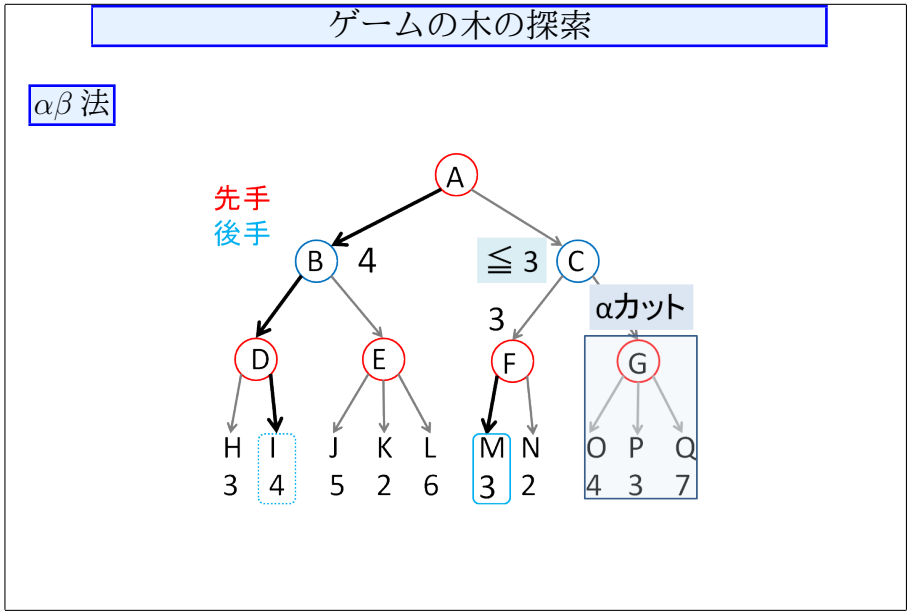
- 以上，ミニマックス法で先手後手のある二人零和ゲームの手を決定しました．
- 原理的には，ここで取り上げたようなゲームはミニマックス法で手を決定することができますが，意思決定点において選択可能な手の数が多い場合，計算機を使ってもそれほど深く先読みはできません．
- ミニマックス法を効率化した探索法としては$\alpha\ \beta$法が知られています．
- $\alpha\ \beta$法は，決定に影響しない部分の探索を省略する枝刈りを行い，探索量の削減を図ります．

ゲームの木の探索

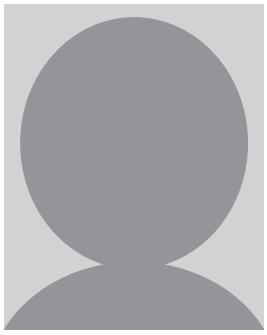
$\alpha\beta$ 法



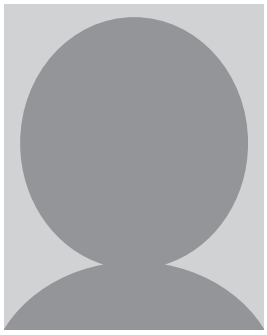
- 先ほどの例で，点Ｄにおいて先手はＩを選択し，評価値が４であるところまで探索したとします．
- 次に，点Ｅにおいて，Ｊを探索すると評価値は５であることがわかります．
- この時点で，点Ｂにおいて，後手は，評価値が５以上のＥではなく，評価値が４のＤを選択するべきであることが決まります．
- したがって，ＫとＬは探索を省くことができます．
- これを β カットと呼びます．



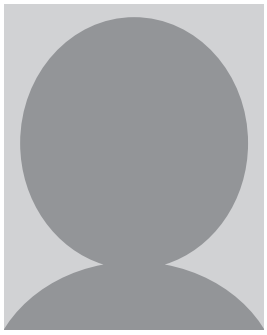
- 今度は，点Ｆで，先手がＭを選択して評価値が３になることまで分かっているとしましょう．
- すると，点Ｃにおいて，後手は評価値を３以下にできることが決まります．
- この時点で，先手は，評価値が３以下のＣではなく，評価値が４のＢを選択するべきであることが決まります．
- したがって，Ｏ，Ｐ，Ｑは探索を省くことができます．
- これを α カットと呼びます．



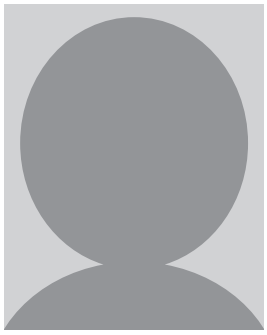
- $\alpha\,\beta$ 法はミニマックス法と同じ結果になることが保証されていて，探索量を大幅に減らすことができますので，先手後手のある二人零和ゲームにおける手の決定には有効な方法です．
- 探索の効率を上げるために，枝刈りを行うのは，ゲームの木の探索に限らず，広く用いられるテクニックです．



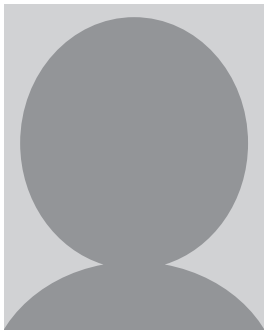
- 今回の講義では，状態空間モデルによる問題の定式化，状態空間の探索としての問題解決ついてお話しました．
- さらに，基本的な探索技法を紹介しました．
- 探索法のアルゴリズムの詳細は，プログラミング言語による実装例と合わせて，インターネットで公開されていますし，書籍も出版されていますので，興味がある方は参照して下さい．
- 今回はこれで終わります．



- （ポーズ）
- 今回の講義では，人工知能における問題解決の一つの方法である，状態空間モデルによる問題の定式化，状態空間の探索としての問題解決ついてお話しました．
- 状態空間モデルによる問題の定式化は，この講義で主に扱っている，決定問題の定式化とは，趣がかなり異なっていますが，目標と制約条件を漏れなく明確に記述するということは共通しています．



- こういったところに，アプローチは違えども，システマティックで，フォーマルな問題解決の背後にある一貫したロジックを見て取れるのではないかと思います.
-
- (時間が余れば)　他の回についても，共通性を意識することにより，さらに理解が深まると思いますので，これを念頭に置いて，復習されると良いでしょう.



- (時間が余れば)　また，探索の手法は，汎用的な方法で，幅広く応用されていて，この科目の他の回においても，特に明示していませんが，今回取り上げたような手法も使われています．
- 探索法のアルゴリズムの詳細は，色々なプログラミング言語による実装例を載せた本が多数出版されていますので，興味がある方は参照して下さい．
- 今回はこれで終わります．