人工知能A

Topic 2: 探索による問題解決

Problem solving by search

2

内容

- 問題の定式化
- 状態空間と探索木
- 探索アルゴリズム
  - Uninformed search
    - Brute-force search, exhaustive search
  - Informed search (heuristic search)
    - A*, (Realtime A* etc.)
  - 反復改良アルゴリズム

3

問題の定式化

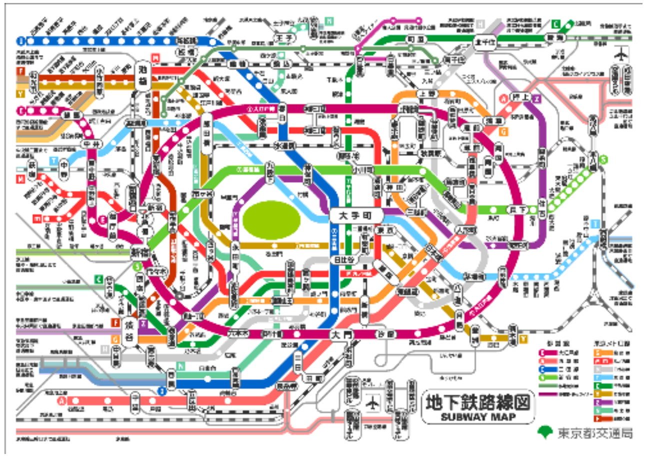
エージェント

- ゴール（目的を）持ち、その状態にたどり着く
  - メモリ内に、ゴール、行為（オペレータともいう）の集合と状態遷移の一覧をもつ。
  - ゴールにたどり着く行為の列を探索で求める
  - その列に従って、実際の行動を進める。

  - ロボットが迷路やパズルを解いたり、効率的に移動するための計画を自ら立てるために必要な機能はなにか？

探索問題：簡単な例から（例１）

- 地下鉄の経路探索



探索問題（例１）

- （地下鉄のみで）新宿から湯島に行きたい。
- もし地図も何もなければ………（状態遷移図のような）
  - 各駅まで行けば観測して乗換や隣の駅はわかるだろう。その先は不明なので、ランダムに進めて運にまかせるだけ。
  - 初期状態、ゴール、行為（ここでは次の駅に移動すること）だけでは問題を解決できない。必要なものは？
- たとえば
  - 地図、路線図：環境のモデル（の一部分）[計算機はモデルを持つ]
  - 隣接駅と目的地との（路線上の）営業距離、または途中の駅数
- （実際に移動するのでなく）地図や営業距離を使って経路を探す過程を探索（search）する。
  - 実行前に行動を計画する「プランニング」の一手法でもある。

探索問題：問題の表現（例１）

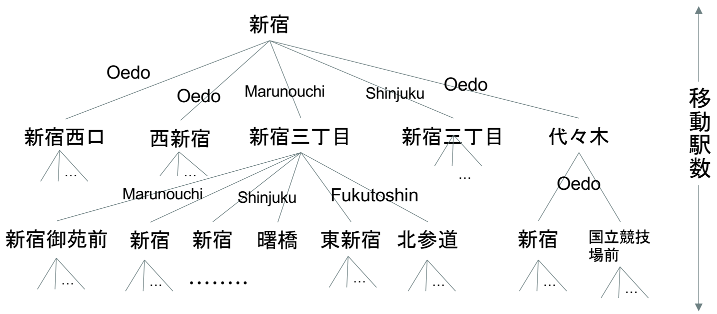
- 問題の表現（探索と捉えた問題解決）
  1. 初期状態 （エージェントがいる「今」の状態）
  2. オペレータ （各状態ごとで可能な行為、action. 例では隣の駅に移動）
  3. 状態空間＝初期状態からオペレータにより到達できる状態全体の集合。なお、オペレータを順に適用すると状態空間を順に移動することになる。これを（状態空間の中の）経路という。
  4. ゴール検査＝ゴールの状態にあるかどうかを判定する。
  5. 経路コスト＝経路のコストを判定する関数。ここでコストは広義の意味。経路長、所要時間、徒歩時間、乗換回数、運賃、これらの融合など …..
- 以上から初期状態からゴールへの最小コストの経路（オペレータの列）を求めることが探索による問題解決である。
  - 実際に行為を行うのではなく、事前に経路を見つける。
  - たとえば、「丸の内線に乗っていて赤坂見附で銀座線が前に止まっていたら乗り換えるが、それ以外はそのまま進む」のように実行してみないと決定できないことは（さしあたり）考えない。

探索問題：問題解決の性能と抽象化

- 問題解決の性能（探索の有効性の判定）
  - 解が必ず見つかるか
  - 良い解か？（経路が最適か、または受け入れられるか？）
  - 解を得るまでに要した時間とメモリ量（探索コストという）
  - 探索による問題解決の総コスト（ただしコストは多様）
    ＝ 経路コスト ＋ 探索コスト
    - コストは多様なので、単純に＋できないこともある。
- （状態と行為の）抽象化---コンピュータで表現するために
  - 状態の抽象化（問題解決の目的を考慮し、不要なものは排除）
    - 例１では、隣の駅の関係が大切で、売店がある、トイレが３つあるなどは関係ない。（問題：乗り換え時間は？ 問題の目的と精度に依存）
  - 行為の抽象化（例１では駅の移動だけが重要）
    - 駅の移動以外。たとえば、ドアが開たら電車に乗るなど。。これも不要。

状態空間の遷移 (遷移のグラフ)

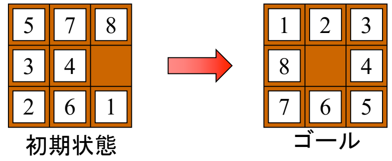
- オペレータにより状態から状態へ遷移する



- 全体として木構造のようになるが、
  - 同じ状態に戻るかもしれないので、正確にはグラフ構造
  - ただし、木構造の方が表現が容易。

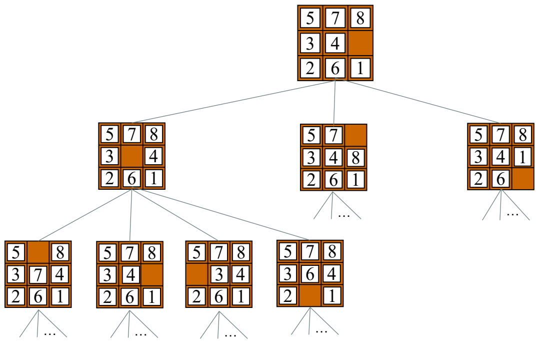
探索問題：例２　8パズル

- 状態
  - 8つのタイルが9つの区画のどこにあるかを示す。
    （次のオペレータの表現のためには、空白も記述することは有効）



- オペレータ
  - 空白を上下左右に動かすと考える（タイルと空白が交換される）
- ゴール検査
  - ゴール状態との一致
- 経路コスト
  - 各オペレータがコスト1として和を求める（ステップ数）
- 同様に「15パズル」も考えられる。

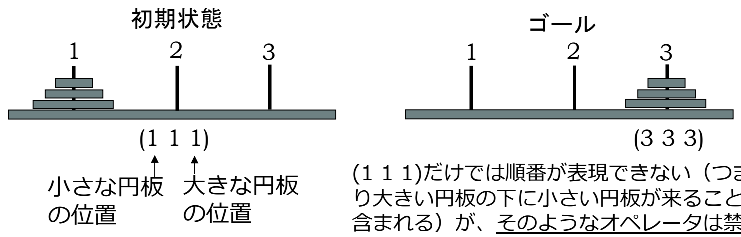
状態遷移のグラフ：例２　8パズル



ただし、同じ状態でもたどり着く経路のコストを考えるなら、何らかの工夫は必要。

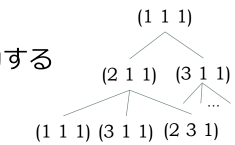
- 状態遷移がグラフ構造になるのは同様。
  - たどった状態を記憶し、毎回確認すれば、元の状態に戻ることは防げるので木構造と考える。他の例題も同様

探索問題：例３　ハノイの塔



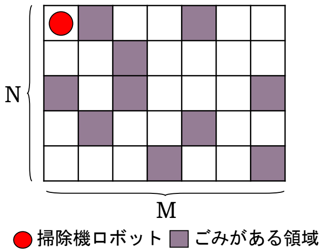
(1 1 1)だけでは順番が表現できない（つまり大きい円板の下に小さい円板が来ることも含まれる）が、そのようなオペレータは禁止されているとする。

- 状態：各円盤の位置を表すリスト。
- オペレータ：各塔の一番上の円板を別の塔に移動する
  - ただし小さい円板の上には乗せない。
- ゴール探査：状態が (3 3 3) か?
- コスト：各オペレータが1として和を求める。



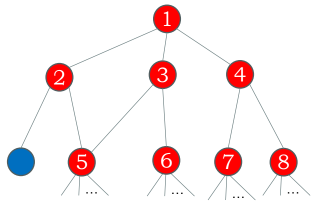
探索問題：例４　Vacuum world

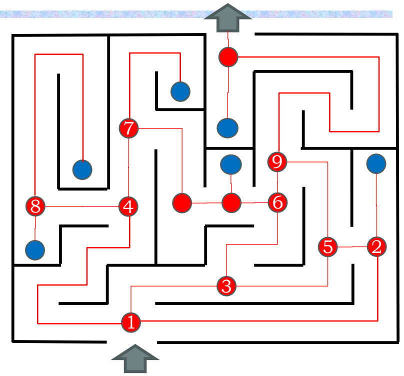
- Vacuum world (掃除機ロボット)
  - 部屋をN x Mのマスに区切る
  - 掃除機ロボットが動き回り、ごみを吸い取り、もとの位置に戻る。
  - ロボットは部屋の形状は分かっているが、ごみの場所はその場所に行ってみないと分からない。



- 問題の定式化
  - 状態：ロボット（エージェント）の位置、それぞれのマスのゴミの有無
  - オペレータ：上下左右に移動、吸い込む、ごみがあるか見る
  - ゴール探査：ゴミのあるすべてのマスを通り、元の位置に戻ったか？
  - 経路コスト：移動したマスの数。
- 注意：たとえば、図の状態で右に移動し吸い込み、また左に移動すると元の位置に戻りますが、状態は異なります（地下鉄の例との差）

探索問題と遷移のグラフ：例５ 迷路





- 問題の定式化
  - 状態：エージェントの位置、スタート、分岐点、ゴール、袋路
  - オペレータ：上下左右に移動、ゴール探査：ゴールの位置か？
  - 経路コスト：移動した距離（最短を見つけたいなら）

課題２－１ ：

- ８パズルの表現
  - 実際に状態のデータ構造を（たとえばＣ，Javaなどを想定して）考えなさい.
  - またそのデータへの操作としてのオペレータの動き（動作や変化），ゴール探査のためのデータの条件などを（プログラムあるいは日本語などの言葉で）書いてみよ.
  - １５パズルではどうか？

課題２－２ ：

- ハノイの塔のオペレータ
  - ハノイの塔のオペレータについて、可能なもの許されないものを検討せよ。たとえば、

    $(2, 1, 1) \rightarrow (2, 3, 1)$,　$(2, 3, 1) \rightarrow (3, 3, 1)$

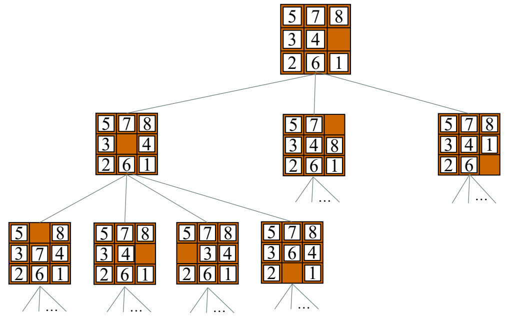
    などは可能ですが、

    $(2, 1, 1) \rightarrow (2, 2, 1)$

    禁止となります。状態を (x_1, x_2, x_3) としたとき、禁止となる条件について検討しなさい。板の枚数が増えてn枚となることも考えること。

17 探索と探索木

探索と探索木



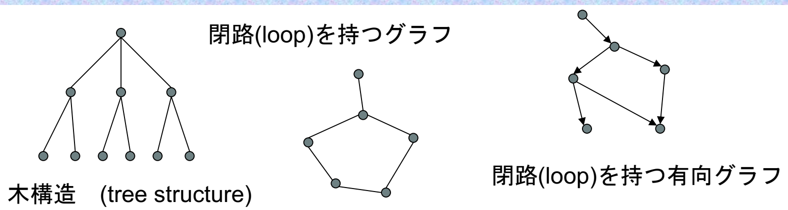
- たとえば、8パズルの状態は、9!ある。
  - 初期状態から到達可能な状態を順に展開してゴールを見つけることを探索 (search)、できる状態遷移の木を探索木 (search tree)と呼ぶ。
  - 状態遷移のグラフをたどるので似ていますが、実質的に同じ状態を２回探索しないようにするので、木構造となる。
  - 過去にたどった同じ状態については、何らかの選択を行い、重複して探索しないようにする。
- ゴール状態になるまで探索
  - ゴールにたどり着く経路は一つとは限らないかも。

ゴール

探索の戦略

- 情報のない戦略 (uninformed search)
  - 力ずく探索 (brute-force) 、しらみつぶし探索(exhaustive search)、盲目的探索 (blind search, あまり良い言葉ではないが一応) ともいう。
  - 特別の情報がなく、したがって、基本的には網羅的に状態を探索し、ゴールを見つける。
- 情報を使った戦略 (informed search)
  - = ヒューリスティック探索 (Heuristic search)ともいう。
  - なんらかの情報や知識 (heuristic) を使い、探索木をある指針にしたがって進みゴールを見つけること。
  - 例：地下鉄の経路探索では「目的地が西にあれば、西に進む路線を優先的に探索する」というもの。必ずしも正しいとは限らない。
  - ヒューリスティックは各分野で意味が変わるが、この分野では「必ず正解となるわけではないが、その代わりに効率的に正解に近い解を得る」、「多くの事例において効率的に正解を得る」こと。

グラフ、木に関する用語の定義



- グラフ（木）はノード (node, 節点)と枝(link, edge)から構成される
- 枝に向きがあるとき、その枝を有向枝 (directed link)といい、有向で構成されるグラフを有向グラフと言う。
- 木には自然に有向と考えられる。
- 木において、有向枝で到着する次のノードを子ノード (child node)、元となるノードは親ノード (parent node)という。
- 木の末端のノードを末端ノード、または葉（leaf, リーフ）ともいう。

探索の評価

- 探索アルゴリズムの評価尺度
  - 完全性：解があるならそれをみつけられるか？
  - 計算量：計算時間量
  - 空間量 （メモリ量）：使用するメモリ量（探索時に記憶する量）
  - 最適性：複数の解があるとき、最適なものを見つけられるか？
- 「情報のない探索」は「情報のある探索」と比べ効率的ではないことが多い（➔次ページ）
  - ただし、一般には探索の指針となる情報が明快に書けない場合があり、重要度は大きい。
  - 探索は、AIに限らず一般システムでも使われる重要な方法
- 良い（適切）な探索アルゴリズムは問題にも依存

情報のない探索 (Exhaustive search)

網羅的探索

情報のない探索 (Exhaustive search)

…[the ant] knew that a certain arrangement had to be made, but it could not figure out how to make it. It was like a man with a tea-cup in one hand and a sandwich in the other, who wants to light a cigarette with a match. But, where the man would invent the idea of putting down the cup and sandwich—before picking up the cigarette and the match—this ant would have put down the sandwich and picked up the match, then it would have been down with the match and up with the cigarette, then down with the cigarette and up with the sandwich, then down with the cup and up with the cigarette, until finally it had put down the sandwich and picked up the match. It was inclined to rely on a series of accidents to achieve its object. It was patient and did not think… Wart watched the arrangements with a surprise which turned into vexation and then into dislike. He felt like asking why it did not think things out in advance…

T.H. White, *The Once and Future King*

探索（の評価）のための用語

- あるノードから一回のオペレータで到達可能な状態の数（つまり、子ノードの数）の平均
  → 分岐数 (branching factor) といい b で表す。
- 頂点から最も深い探索木までの長さを探索木の深さといい、m と表す。mは有限とは限らない。
- 頂点から最も浅い位置にあるゴールまでの階層（深さ）をゴールまでの深さと言いd と表す。
- すべての子ノードの探索が完了したものを閉ノード (closed node)、そのリストをclosed リスト (closed list)とよぶ
- 探索し終えていないノードを開ノード (open node) という。開ノードもしくはそこから出ている探索していない枝のリストを open リスト (open list) とよぶ。

情報のない探索

- 基礎的な探索アルゴリズム
  - ランダム探索
  - 深さ優先探索
  - 幅優先探索
  - コスト均一探索
  - 反復深化探索
  - 双方向探索

ランダム探索 (random search)

- 探索の基本パターン

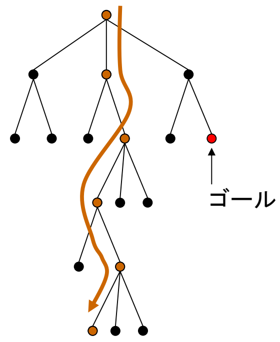
Step1: 出発のノードをnとする。

Step2: ノードn がゴールであれば成功(終了)

Step3: ノードn からたどり着くの次のノードを展開し、子ノードの集合を作る

Step4: 子ノードの集合からノードをランダムに一つ選び、n'とする。

Step5: n'を新しいノードとし、Step2へ。



- 性質（評価）
  - 必ずしもゴールにたどりつくとは限らない。
  - 必要メモリ量は小さい。効率は悪そう（ほとんど見つけられない）、。
  - ループに入る可能性がある（一度たどった状態を覚えていないので、たとえば、ハノイの塔で、(1 1 1) →(2 1 1)→(1 1 1)と繰り返す）

ランダム探索の改良（その１）

- Closedリスト（以下は一つだけ子ノードを探索するがあとで「その2」と合わせる）
  - <u>同じ状態に対応するノードには２回行かないように履歴として保持</u>

Step1: 出発のノードをnとする。

Step2: ノードnがゴールであれば成功として終了する。

Step3: ノードnを展開し、子ノードの集合を作る。 nをClosedリストに追加する。

Step4: 子ノードの集合からClosedリストに含まれないものを一つ選び、n'とする。もしそのようなものがなければ失敗として終了する。

Step5: ノードn'を新しいノードとし、Step2へ。

- 上記のアルゴリズムは、探索木が有限であるなら必ず終了する。
- ただし解が得られないこともある（得られないほうが多い）。探索木の下まで行ってしまうとそれで抜け出せず、終了する。

ランダム探索の改良（その２）

- Openリストの導入
  - <u>探索木をこれ以上展開できない探索木の下に着いたとき、戻れるように途中で展開した状態を保持しておく</u>。

Step1: 出発のノードnに対し(n *null*)をopenリストに追加する。

Step2: Openリストから要素、つまりlink (n p) をランダムに選択し、<u>取り出す</u>。n がゴールであれば成功として終了。Openリストが空集合であれば失敗として終了する。

Step3: ノードnを展開し、子ノードの集合を作る。nをclosedリストに追加。

Step4: 子ノードの集合からclosedリストに含まれないすべてのノードn'に対して、(n' n)をOpenリストに追加する。

Step5: Step2へ。

- 上記のアルゴリズムで、n がゴールであると分かったとき、nへ至る経路が分かる。（➔ 課題。あとで）

ランダム探索のここまでの改良で

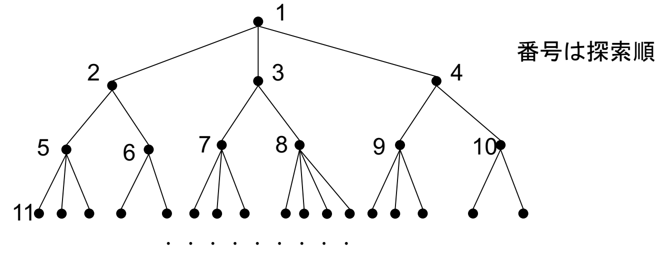
- Openリストを導入したこのアルゴリズムは、探索木が有限であるなら、必ず終了し、解が存在すればそれを見つけられる。

定義：

- Openリストの要素$(n\ m)$で、ノードnは探索中で、まだその先のノードを展開していないものである（これ以上展開できないかも知れないが）。このようなノードを縁 (fringe) と呼ぶ。

幅優先探索 (breadth-first search)

- 探索木を各レベル（深さ）ごとにすべて展開し、解を見つける



- 探索がある深さまで完了したとき、縁の集合はその深さのノード全体となる。

幅優先探索 (breadth-first search)

- アルゴリズム

Step1: 出発のノードnに対し$(n\ null)$をOpenリストに加える。 nがゴールなら終了。

Step2: 直前のStepでOpenリストに加えられた要素$(n\ p)$ をすべて選択し、取り出す。Openリストから選択できなければ失敗として終了する。

Step3: 選択したノードnを順に展開し、子ノードの集合を作る。子ノードにゴールがあれば終了。展開したnをCLOSEDリストに追加する。

Step4: 子ノードの集合からCLOSEDリストに含まれないすべてのノードn'に対し、$(n'\ n)$をOpenリストに追加する。ただし、nはn'の親ノード。

Step5: Step2へ。

- 良い点：解があれば必ず見つかる。各オペレータのコストが一定であれば、最適な解が得られる。
  - でもあまり使われない。それは……………

幅優先探索 (breadth-first search)

- メモリ量の問題（指数的増）
  - 次のレベルを展開するために、あるレベルのノード（＝状態）をすべて覚えておく必要がある。
  - 解が見つかったときに、そこへの経路を導くために、それより上位のノードを覚えておく必要がある。

  ➔結局、深さdにあるゴールを見つけるまで展開したすべての状態を覚えておく必要がある。
  - 各ノードが展開されてできる子ノードの平均数をbとすると、

  $1+b+b^2+b^3+ \dots\dots +b^d = O(b^d)$
- もちろん計算量も問題

深さd	ノード	時間	メモリ
2	111	0.1秒	11KB
6	10^6	18分	111MB
8	10^8	31時間	11GB
10	10^{10}	128日	1TB
12	10^{12}	35年	111TB

b=10、1ノードあたり10ms, 100Bとして計算
bは迷路などではもう少し小さいが、囲碁や将棋だど莫大となる
速度を仮に100倍速いとしてもd=14ぐらいで年単位となる

均一コスト探索 (uniform-cost search)

- オペレータのコストが異なるとき、幅優先探索は最良の解を見つけられない可能性あり。この欠点を補うために、幅優先探索を一般化し、コストをOpenリストに加える。

Step1: 出発のノード*n*に対し(*n null 0*)をOpenリストに追加する。

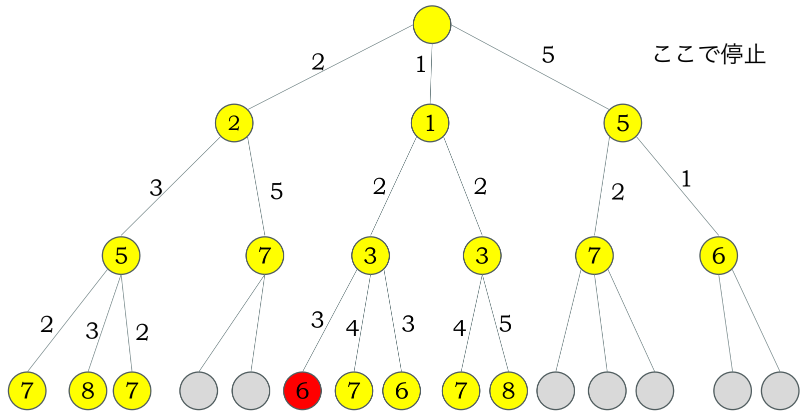
Step2: Openリストから最小のコスト*c*を持つ要素 (*n p c*) を選択する（ここでコスト比較している。 複数あればランダム）。*n*がゴールなら成功として終了。Openリストが空集合なら失敗として終了。

Step3: ノード*n*を展開し子ノードの集合を作る。*n*をCLOSEDリストに追加する。

Step4: 子ノードの集合からCLOSEDリストに含まれないノード*n'*に対して、(*n' n c'*)をOpenリストに追加する。なお、$c' = c + c(n, n')$ [$c(n, n')$ は$n \to n'$のコスト]とする。

Step5: Step2へ。

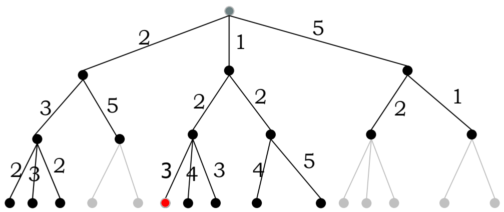
均一コスト探索 (uniform-cost search)



ここで一応解は見つけたがもっとコストが低い解が見つかるかもしれないので、すべてのノードのコストが6以上になるまで継続する。

均一コスト探索 (uniform-cost search)

- 良い点：
  - 解があれば必ず見つかる（完全性）
  - 各オペレータのコストが
  - 分かっていれば、コストの観点から最適な解が得られる。（最適性）

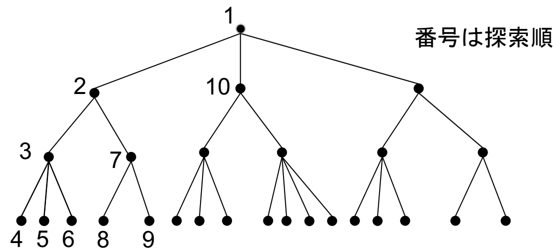


番号はコスト

- 幅優先探索と同じ欠点。
  - 計算量もメモリ使用量も基本的には幅探索と同じ。
  - したがって大きな問題に対しては、あまり使われない。

深さ優先探索 (depth-first search)

- 探索木の最も深いレベルのノードの一つを展開する。探索が行き止ったとき逆戻り（バックトラック）し、近くの次に深いレベルのノードの一つを展開する。
- 縦型探索とも言う。



深さ優先探索 (depth-first search)

- アルゴリズム

Step1: 出発のノード*n*に対し(*n null*)をOpenスタック（リストをスタックに変えた）に加える。*n*がゴールなら終了。

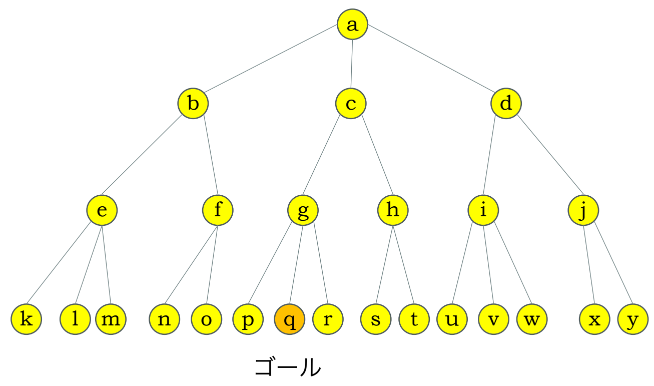
Step2: 直前のStepでOpenスタックの先頭の要素(*n p*) を参照する。選択できなければ失敗として終了する。*n*がすでにCLOSEDリストに<u>含まれていれば</u>それをポップして、再度Step2へ。

Step3: 参照したノード*n*を展開し子ノードの集合を作る（空集合の可能性もある）。子ノードの集合にゴールがあれば、そのゴール（と必要であればそこへ至る経路）を出力して終了。 *n*をCLOSEDリストに追加する。

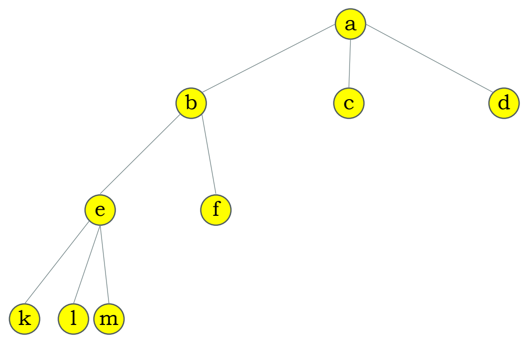
Step4: 子ノードの集合からCLOSEDリストに含まれないすべてのノード*n'*をある順に並べ、その順に従って(*n' n*)をOpenスタックにプッシュする。ただし、*n* は*n'*の親ノード。

Step5: Step2へ。

深さ優先探索 (depth-first search)

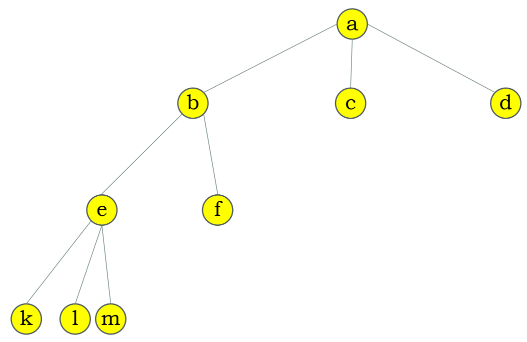


深さ優先探索 (depth-first search)



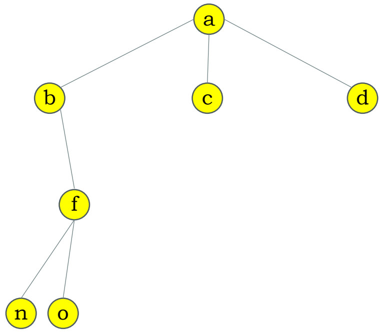
(k e)
(l e)
(m e)
(e b)
(f b)
(b a)
(c a)
(d a)
(a null)

深さ優先探索 (depth-first search)



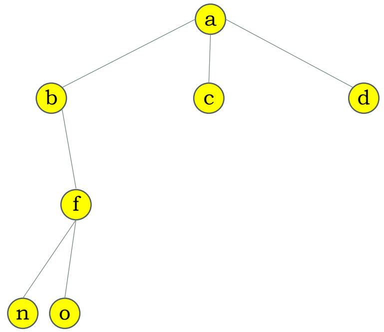
(k e)
(l e)
(m e)
(e b)
(f b)
(b a)
(c a)
(d a)
(a null)

深さ優先探索 (depth-first search)



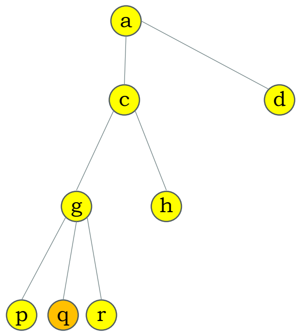
(n f)
(o f)
(f b)
(b a)
(c a)
(d a)
(a null)

深さ優先探索 (depth-first search)



(n f)
(o f)
(f b)
(b a)
(c a)
(d a)
(a null)

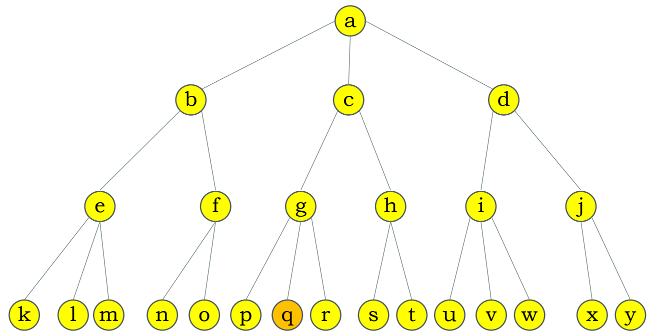
深さ優先探索 (depth-first search)



ゴールあったのでここで終了

(p g)	
(q g)	⇐
(r g)	
(g c)	⇐
(h c)	
(c a)	⇐
(d a)	
(a null)	⇐

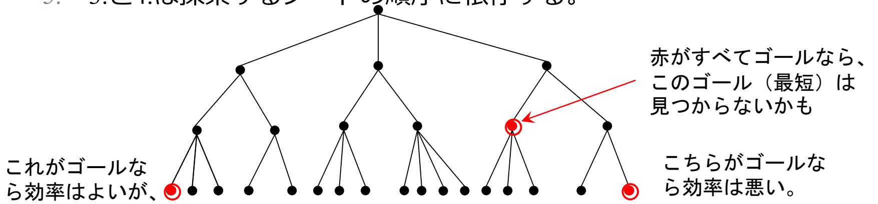
深さ優先探索 (depth-first search)



ゴール

深さ優先探索の性質

1. 木（グラフ）が有限の階層でなければ停止しない可能性がある。
2. メモリ量は幅優先探索や均一コスト探索に比べ少ない（探索した部分から忘れることができる）------ $O(bm)$
3. 幅優先探索より効率が良いことが多い（幅優先探索では、長さdの経路を調べるには、その前に長さ$d-1$の経路を全探索しなくてはならないから）。計算量は最悪$O(b^m)$である。一般には$m \geqq d$。
4. 最適の解が求まるわけではない。
5. 3.と4.は探索するノードの順序に依存する。

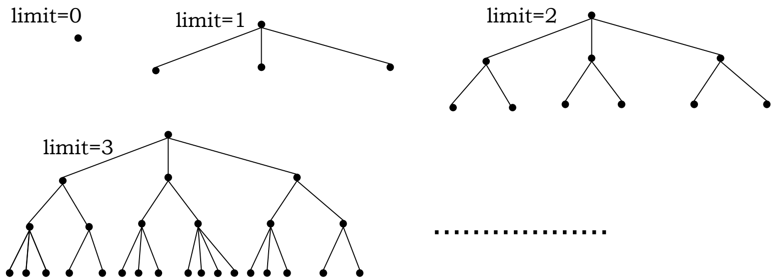


深さ制限探索 (depth-limited search)

- 深さ優先探索と同じだが、予め探索する深さを制限 (*limit*) し、その深さまで達したらさらに深くは進まずにバックトラックする。
- 深さが有限でない探索木で停止しない状態を防ぐ。
- 無駄に深く進まず探索の効率が上がる。メモリ量も減る
- 問題点：
  - あらかじめゴールへ達するゴールまでの深さdは分からない。適切な深さを設定しないと解を求められなかったり、適切な解を求められない（仮に、深さdと$d-1$にゴールがある場合、$limit \geqq d$とすると$d-1$のゴールの方が適切だが、順序によっては深さdのゴールを先に見つけてしまう。$limit = d-1$なら最適な解が見つかる。$limit < d-1$だと解は見つからない）。

反復深化探索 (Iterative deepening search)

- アイデア：深さ制限探索を繰り返し、その欠点を補う。
- Limitを0から順に大きくしながら深さ優先探索を行う。



反復深化探索 (Iterative deepening search)

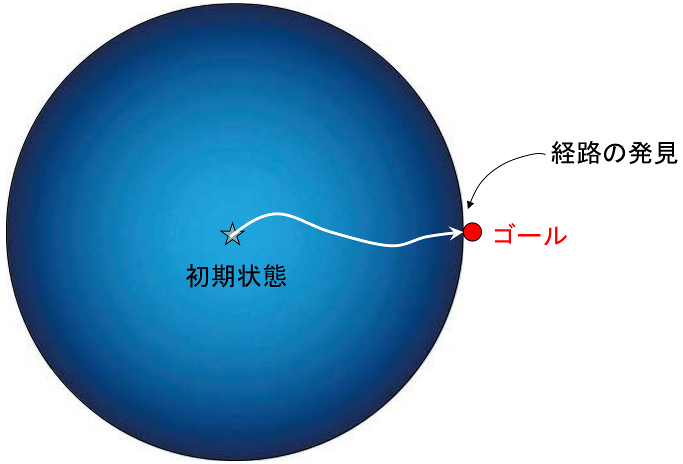
- 性質
  - 最適の解を求めることができる（完全性と最適性）
  - メモリ量は深さ優先と同じ（メモリ量小）
    - $Max(O(b), O(b), \cdots, O(b)) = O(bd)$
  - 計算量について
    - $O(b) + O(b^2) + \cdots + O(b^d) = O(b^d)$
    - オーダーとしては深さ優先よりよい$(b^d \leq b^m)$
- 大きな探索空間、ゴールへの経路長が分からないときによく使われる。
  - 一見、無駄に見えるが、完全性、最適性、メモリ効率化を引き出すためめの代償コスト。計算量のオーダとしては繰り返し分は低い（たとえば無駄と思って状態を記憶するとメモリ量が爆発する）

付録

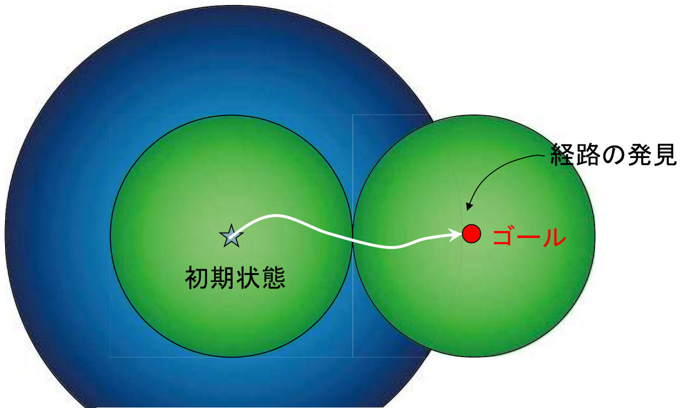
双方向探索 (bi-directional search)

- これまでのように、探索木を如何にたどるかではない。
- ゴールは分かっていて、初期状態との経路（オペレータの列を知りたいとき）
  - 初期状態からゴールに向かった探索（前向き探索）と、それとは逆にゴールから逆向きに初期状態に向かって進む逆向き探索を同時に行う。
  - 同時に行うと効率が良くなると思われる。そのイメージは、次のスライド。

双方向探索 (bi-directional search)



双方向探索 (bi-directional search)



双方向で進めることで、探索の範囲を狭くできる。
２次元では面積は半分。３次元では1/4となる。

双方向探索 (bi-directional search)

- これまでのように、探索木を如何にたどるかではない。
- ゴールは分かっていて、初期状態との経路（オペレータの列を知りたいとき）
  - 初期状態からゴールに向かった探索（前向き探索）と、それとは逆にゴールから逆向きに初期状態に向かって進む逆向き探索を同時に行う。
  - 同時に行うと効率が良くなると思われる。そのイメージは、次のスライド。
  - つまり

$$O(b^{2d}) \gg 2O(b^d)$$

  - で、探索に要するコスト（計算量、メモリ量）が減る。（通常は幅優先、均一コスト優先探索を使う）

双方向探索 (bi-directional search)

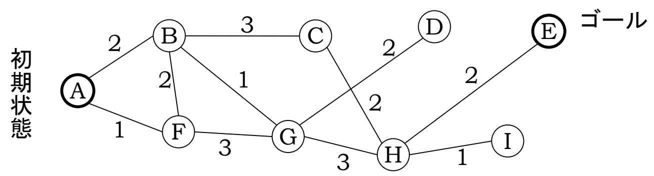
- 実現の前に考えること
  - 後向きに探索するとは？ゴールノードから出発し、これを子ノードとする親ノードを展開し、初期状態に向かう。双方向探索とは、双方向から進め、共通のノードが存在するときそこへ至る経路を合成
  - ある親ノードで、あるオペレータを適用して目的とする子ノードになるとき、オペレータが一つでも、対象となる親ノードは複数あるかも知れない。それら全て求める必要がある。ある問題領域では、これが簡単ではないこともある。
  - ゴールがあまりにもたくさんあるときには使えない。たとえば、チェスで、逆向き探索とは何か?チェックメイトからの逆向き探索は考えられない。
  - それぞれの向きで探索して得られた新しいノードに共通部分があることを効率的に調べられなくてはならない。
  - それぞれの向きからの探索の方法は、よく考慮すべき。

情報のない探索のまとめ

	幅優先	均一コスト	深さ優先	深さ制限	反復深化	双方向（可能な場合）
時間	b^d	b^d	b^m	b^l	b^d	$b^{d/2}$
空間（メモリ）	b^d	b^d	bm	bl	bd	$b^{d/2}$
最適性	Yes	Yes	No	No	Yes	Yes
完全性	Yes	Yes	No	Yes $l \geqq d$ なら	Yes	Yes

課題２－３

1. Openリストを導入したランダム探索のアルゴリズムで、Step2であるノードnがゴールと分かったとき、そこへ至る経路は如何にして得られるか考えてみよ。
2. 下記の状態空間グラフをそれぞれの探索方法で探索したとき、探索する状態空間の順序を書け（たとえば、A→B→C…のように）。双方向探索は、一ステップづつ交互に行うものとする。なお、数値はコストを表す。



課題２－４

- 反復深化が深さ優先探索より効率がかなり悪くなるような探索木は、どのような場合か？（逆に、どのような条件で、深さ優先探索は有利となるか？）
- 上記の探索木を、反復深化と幅優先探索でゴールを求める場合の計算量とメモリ量について言及せよ。
- 課題2-3 2) の各ノードでOpenリストとClosedリストはそれぞれどうなるか。特定の探索戦略を決め、探索にあわせてその変化を調べよ。

57 情報を利用した探索（ヒューリスティック探索）

情報を利用した探索

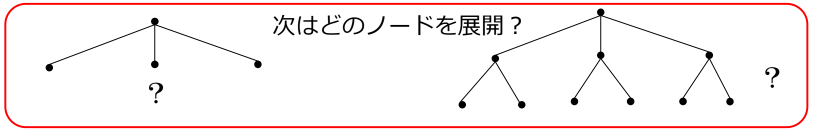
- ヒューリスティック探索とも言う
  1. 基礎的なヒューリスティック探索手法
     - 最良優先探索 （アルゴリズムの基本形—template—となる）
     - 欲張り探索 (greedy search, greedy best-first search)
     - A*探索 (A* search)
  2. ヒューリスティック関数の生成
  3. より効率的なヒューリスティック探索
     - 反復深化A*探索 (IDA*)
     - 単純メモリ限定A*探索 (SMA*)
     - Realtime A* (RTA*)
  4. 反復改良アルゴリズム
     - 山登り法、焼き鈍し法

ヒューリスティック(Heuristic)とは

- Heuristic --- 「発見する」、「見つける」という意味。
  1. 「与えられた問題の多くは解けるかもしれないが、必ず解けるという保障はない」 --- アルゴリズムと対比した概念。
  2. 経験則 --- 実用的な問題解決システムが膨大な探索空間をすべて調べることなしに、人間が経験的に良いと考えられる順に調べることを実現する、問題や状況に依存した評価関数。
  3. 現代的な解釈：問題解決の平均的な問題は効率的になるが、最悪の場合の性能は必ずしもよくならない。（実際に多く直面する問題を効率的にしよう！）このための解のコストを見積もる関数のこと。

ヒューリスティック(Heuristic)とは

- 探索木で知識を適用し効率を上げる場所は、子ノードに展開後に、次にたどるノードを選択するところ。



- この展開の順序を決定する知識は、評価関数として与えられる。好ましさに応じた順序づけを行い、どれが（どちらが）最適なゴールに近いかをガイドする。この評価関数をヒューリスティック関数という。
- 評価関数の値（評価値）が最大のノードを最初に展開する戦略を最良優先探索(best-first search)という。

最良優先探索 (best-first search)

- 基本的なアルゴリズム（基本パターン）

function Best-First-Search(*search-tree*, eval-fn)

returns a solution sequence of operation.

- Openリストにある緑ノードを評価関数eval-fnで並び替えて、その先頭の $(n\ p)$ についてノード n の子ノードを展開する。これを繰り返すして、ゴールに達したときにそこに至るlinkの列（つまりオペレータの列）を返す。
- eval-fnがよいと判断したものを優先的に探索

最良優先探索 (best-first search)

- 最良優先探索というが、評価関数が最良のノードを選択できるとは限らない（選択できるなら探索は不要）
  - 「最良と推定される、あるいは多くの場合は（ほぼ）最良のノード」を選択するというイメージ。
- （理想的な）評価関数は、ある状態から最も近いゴールへの経路を何らかの尺度で評価すべき。それを使い
  - ゴールに近づくため、ゴールに最も近いと判断したノードを展開する
  - 最終的に経路のコストが最小となる（とよいが）。
- 注意：均一コスト探索 (最良優先探索の一例だが．．)
  - コストはそれまで使用したオペレータのコストであり、必ずしもゴールに（つまり未来の方向へ）向いてはいない。

欲張り探索（Greedy search）

- ゴールに達するのに予想されるコストを最小とするノードを最初に選択したい。
  - 各ノードからゴールまでのコストを見積もるヒューリスティック関数hを導入する。

    $h(n)$ = ノードnからゴール状態までの最短の経路の見積り（nがゴールなら$h(n)=0$）
- hを使った最良優先探索をgreedy search（欲張り探索）という。

function Greedy-Search(*problem*) **returns** a solution or failure

return Best-First-Search(*problem*, h)

欲張り探索（Greedy search）

- アルゴリズム

Step1: 出発のノードnに対し(n *null* c)をOpenリストに追加する。 ただし、$c = h(n)$である。

Step2: Openリストから<u>cが最小である要素$(n\ p\ c)$ を一つ取り出す</u>（複数あればランダムに）。n がゴールであれば成功として終了。Openリストが空集合であれば失敗として終了する。

Step3: ノードnを展開し、子ノードの集合を作る。nをCLOSEDリストに追加する。

Step4: 子ノードの集合からCLOSEDリストに含まれないノードn'に対して、$(n'\ n\ c')$をOpenリストに追加する。ただし、$c' = h(n')$である。

Step5: Step2へ。

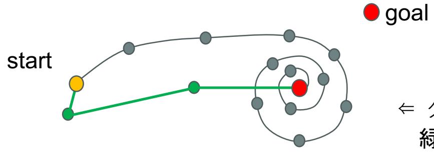
考え方：ゴールに近いと考えられるノードから展開する

欲張り探索（Greedy search）

- 最適な解は必ずしも見つからない
  - 例：地下鉄探索の場合
    - 平面地図での各駅の位置の座標が与えられているとする。
    - $h(n)$ = 駅：nと目的地の地図上の直線距離とする。
    - 新宿から明治神宮前へ行く（地下鉄だけで。次のスライド）
- 実は、hはnがゴールのとき$h(n)$=0となる正の(有界)関数ならなんでもよい。効率的ではないが、探索木が有限なら、いつかはゴールに達する。
- ただし、効率的にするには何らかの情報を使った知識をヒューリスティック関数として実現するのが望ましい。この設計が「ポイント」となる。

欲張り探索（Greedy search）

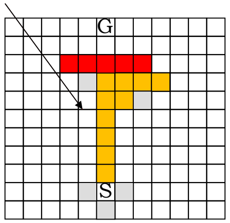
- 例：路線図の位置関係は正確ではないが
  - 新宿の隣接駅（新宿三丁目、初台、西新宿、都庁前、代々木）で、代々木からの直線距離が一番近そう
  - 代々木の次は国立競技場。緑（新宿三丁目、初台、西新宿、都庁前、国立競技場）を比べると、国立競技場が近そう。
  - 以下同様に「青山一丁目」「表参道」と進み、明治神宮前につく。



⇐ グレーが先。
緑の経路は直ぐには見つからない

欲張り探索（Greedy search）

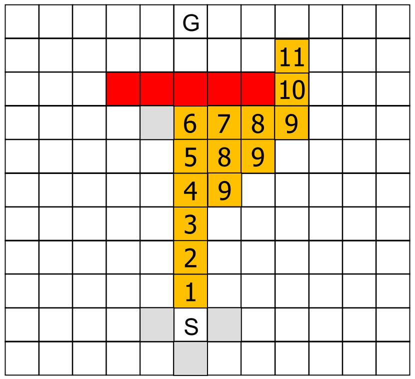
本当はこちら側も同じように交互に探すが省略



- うまくいく例 (効率は良くないが)
  - SからGへの経路
  - 上下左右のみに移動できる
  - 赤のマスは障害物で通れない。
  - 障害物はそこに行って（ぶつかって）はじめて分かる
  - Gの座標と自分の場所のマスの座標は分かる。
  - ヒューリスティック関数hの定義は、マスnとGとのマンハッタン距離とする。（マンハッタン距離＝縦と横の距離の和）

欲張り探索（Greedy search）

- 参考



課題２－５：
右の例を実装してみよう。
（数字はあまり気にしない）

欲張り探索：まとめ

- ヒューリスティック関数hを使った最良優先探索
- 最適な解を求められるとは限らない。求めるためには全探索となるかもしれない。
  - ヒューリスティック関数の定義に依存
- もし行き止まりになったら、hの値についてこれまでたどった次によいノードを展開する。
- これはスタート近くで間違った選択をすると、それを解消するには、それ以下の探索木をすべて検証し、失敗して元にもどらなくてはならない。探索初期の選択は特に重要である。（階層的探索。探索木を会社組織と考える。偉い人が間違えると、それを覆すためには、その部下たちがすべてを尽くし、正しい解はないことを示さない限り、偉い人は意見を変えないことに似ている）
- hを如何に定義するか。これがミソ。いくつか例をあとで示す。

A* 探索　(A* search)

- Greedy searchは$h(n)$で探索の優先順位を与え、探索を効率化できるが、最適でも完全でもない。
- 経路のコストを最小化できる均一コストは完全で最適であるが、効率的ではない。➔ では合わせせよう
- $g(n)$を出発点からノードnまでの経路コストとする。
  - これは探索しながら計算すればよい。
- $h(n)$はnからゴールまでの最短経路の見積もりコスト

$$f(n)=h(n)+g(n)$$

$f(n)$は状態nを経由する最短経路の見積もりコスト

- ヒューリスティック関数として$f(n)$を使う
- 実は$h(n)$がある条件を満たすとき、$f(n)$を使った最良優先探索は完全で最適となる。

A* 探索 (A* search)

- $h(n)$ の条件
  - hの値は、実際のコストを超えない。
  - hは許容的ヒューリスティック(admissible heuristic)という。
  - 実際のコストより小さめに見積もるので楽観的とも言われる。
- $f(n)$もこの性質（許容性）を受け継ぐ。
  - $f(n)$の値は、nを経由する経路のコストを越えることはない。
  - このような$f(n)$を使ったとき、A*探索 (A* search)という。
  - [許容的アルゴリズム（最短経路を発見できること）A1, A2, …探索があり、その中で効率が良かった探索がA*探索]

function A-Search(*problem*) **returns** a solution or failure
　　return Best-First-Search(*problem*, *g*+*h*)

- 実は、A*は完全で最適である。（証明はあとで）

A* 探索　(A* search)

- アルゴリズム

Step1: 出発のノードnに対し (n *null* c) をOpenリストに追加する。 ただし、$c = h(n) = f(n)$である。$g(n) = 0$に注意。

Step2: Openリストからcが最小の要素 (n p c) を一つ選択。nがゴールであれば成功として終了。Openリストが空集合なら失敗として終了。

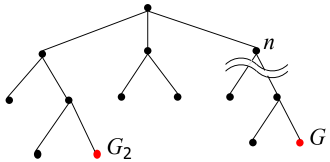
Step3: ノードnを展開し、子ノードの集合を作る。nをCLOSEDリストに追加する。

Step4: 子ノードの集合からCLOSEDリストに含まれないノードn'に対して、(n' n c')をOpenリストに追加する (なお、$c' = f(n') = h(n') + g(n) + c(n\ n')$, $c(n\ n')$ はnからn'へのコストである) である。<u>このときすでに (n' m c'')がOpenリストに入っており、$c'<c''$なら新しいものに置き換え、そうでなければOpenリストには追加しない。さらに、CLOSEDリストに含まれるn'に対して、そのときに探索した (n' m c'') について、$c' < c''$ならn'をCLOSEDリストから削除し、(n' n c') をオープンリストに加える。</u>

Step5: Step2へ。

A* 探索の最適性の証明

- Gを最適ゴールとし、Gへの実際の経路コストをf^*とする。
- G_2を最適でないゴールとすると、$g(G_2) > f^*$である。A* 探索がG_2を選択したと仮定する。　➔ 矛盾を導く
- ということは、Gへの最適経路については、まだ探索されていない。Gへ向かっているがまだ展開されていない縁となるノードがある。それをnとする（右の図）。
- h が許容的であるから、$f^* \geqq f(n)$
- n は展開されなかったから $f(n) \geqq f(G_2)$
- G_2はゴールだから $h(G_2) = 0$
- これら３式から $f^* \geqq g(G_2)$
- G_2が最適でないことに反する。



A* 探索の性質

- 完全性:
  - ノード数が有限であれば。（直感的には、探索を進めるにつれて $f(n)$ の値が大きいものが選ばれる。いつかは $f(G)$ の値となり、Gに達する）。
- 計算量：
  - h の値が実際の値に近ければ指数オーダーにはならない。
  - が、h をうまく定義できるかが問題となる。

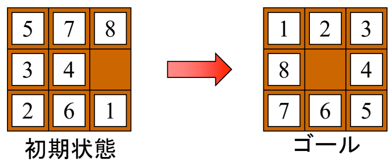
- 課題２－６：先の地下鉄の例でも最短経路が見つかることを考えよ。
- 課題２－７：$h(n)$=0 も許容的である。このときのA*探索を考えよ。

ヒューリスティック関数と作成

- ヒューリスティック関数の正確さと探索の性能の関係を調べる。
  - そのための例として、8-パズルとそのヒューリスティック関数を２つ導入して、効率を比較する。
- ヒューリスティック関数の作り方のヒント

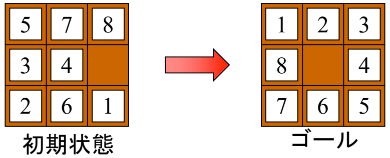
探索問題：例２　８パズル（再掲）

- 状態
  - ８つのタイルが９つの区画のどこにあるかを示す。
    （次のオペレータの表現のためには、空白も記述することは有効）



- オペレータ
  - 空白を上下左右に動かすと考える（タイルと空白が交換される）
- ゴール検査
  - ゴール状態との一致
- 経路コスト
  - 各オペレータがコスト１として和を求める（ステップ数）
- 同様に「１５パズル」も考えられる。

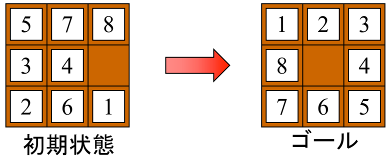
探索問題：例２　８パズル（再掲）



- 分岐数は平均すると約3である（空白が中央なら4、角だと2、そのほかは3である）。もし20ステップ先まで考えるしらみつぶし探索を行うと、$3^{20} \cong 3.5\mathrm{x}10^{9}$である。状態数は9!=362880ある。

８パズルのヒューリスティック関数

- 許容的なヒューリスティックの例を２つ
  - h_1 = ゴールの位置にないタイルの数
  - h_2 = ゴール状態からのタイルの距離の和。マンハッタン距離(Manhattan distance)とも言う。

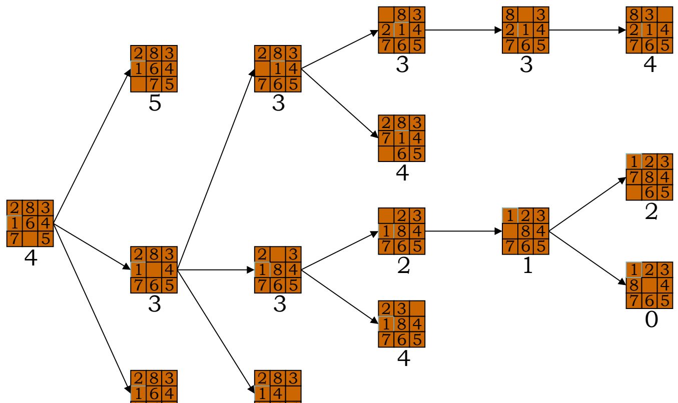


- 上記の例では
  - $h_1 = 7$　(ブロック6以外は別の位置にある)
  - $h_2 = 4+3+3+1+4+0+3+3=21$　(たとえばブロック1を正しい位置に移動するには4ステップ必要)

８パズルのヒューリスティック関数

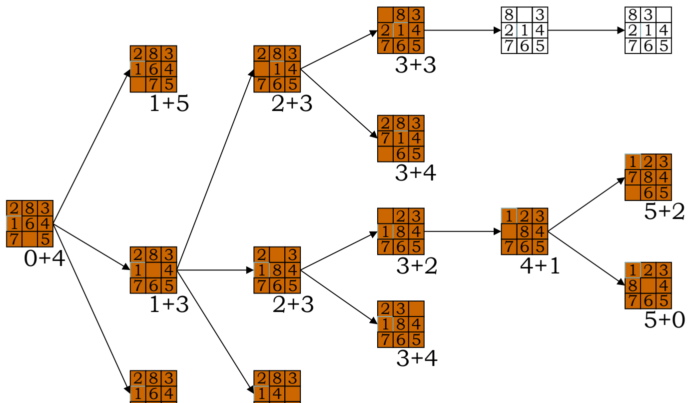
- 許容的か？
  - h_1 …… 一致していないタイルをゴールの位置にあわせるには、それぞれ１ステップは必要。しかも一回にタイルは一つしか動かせない。
  - h_2 …… いかなる動作も、一回でゴールに１だけ近づけること以上はできない。また、一回にタイルは一つしか動かせない。
- どちらが必要なステップ数（探索木の経路数）に近いか（実際の経路数は関数h^*で与えられるとする）。
  - $h_1 \leq h_2 \leq h^*$は明らか。つまり、h_2のほうが正解に近い。
  - このとき、h_2はh_1より優位(dominate)であるという。
- 重要：ヒューリスティック関数の計算量は十分小さい。

8パズル - Greedy Best-First Search



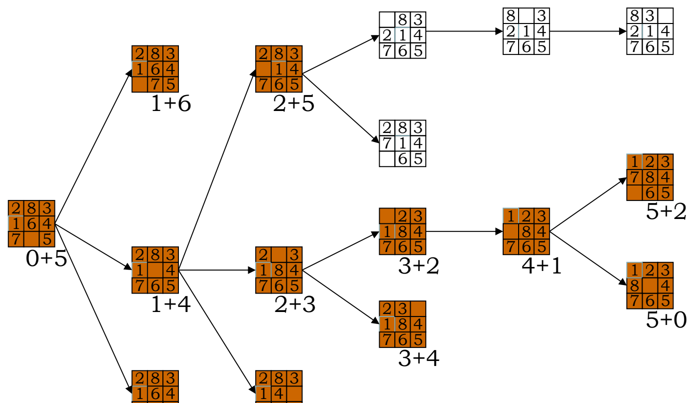
$h(n)=h_1(n)$を使用

8パズル- A*探索　(h_1)



$h(n)=h_1(n)$を使用

8パズル- A*探索　(h_2)



$h(n)=h_2(n)$を使用

課題

- 課題２−８：
  これまで８パズルのヒューリスティック関数を$h_0(n) = 0$とすると前スライドの探索はどうなるか考えよ。
- 課題２−９：
  8パズルを実装し、IDS、 A*(h_0)、 A*(h_1)、A*(h_2)の探索効率の違いを調べよ。初期配置はランダム。プログラム言語は何でもよい。
- 課題２−１０：
  ヒューリスティック関数はないが、二つの状態が与えられたときにどちらが良さそうかを判断する関数$h(n, n')$があるとする。これでも最良優先探索は可能であることを説明せよ。

課題２−１１

- 課題２−９の内容を15パズルでもためしてみよ。
  - ゴールは以下の通り。
  - h_0, h_1, h_2を使う。他にアイデアがあればそれを利用してもよい、

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

ヒューリスティック関数の探索性能

- 有効分岐数 (effective branching factor)
  - A*で探索されたノードの総数をN、探索した解の深さをdとする。ゴールまでの深さdの均一の木（子ノードの数が均一）がN+1のノードを持つ分岐数を有効分岐数といい、b*と表す。

$$N+1 = 1+b^*+(b^*)^2+ \ldots\ldots + (b^*)^d = (1-b^{*d+1})/(1-b^*)$$

- B*の意味
  - たとえば、A*探索によりゴールまでの深さ5にある解を52個のノードを探索し、探し当てたとする。このときだいたい$b^* = 1.9$である。この値が小さければ、平均1.9の分岐数の探索木で、ゴールの深さ5にある解を見つけるのと同じ程度の効率となる。b*の値が1に近いほど効率的な探索ができたことになる。（$b^* = 1$なら、解まで探索なしに一直線で求まることを意味する）。

ヒューリスティック関数の探索性能

８パズルの場合

	（平均）探索コスト			（平均）有効分岐数		
d	IDS	A*(h_1)	A*(h_2)	IDS	A*(h_1)	A*(h_2)
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	364404	227	73	2.78	1.42	1.24
14	3473941	539	113	2.83	1.44	1.23
16		1301	211		1.45	1.25
18		3056	363		1.46	1.26
20		7276	676		1.47	1.27
22		18094	1219		1.48	1.28

ヒューリスティック関数の作り方

- 良いヒューリスティック関数が探索の効率化のポイントだが、その作成は難しいことも多い。そのヒント。
- 弱条件問題 (relaxed problem) の利用が多い：
  - 問題の条件を緩くし、解への見積もりを容易にする。
  - 例：h_1は位置が合っていないタイルをはずし、正しい場所へはめ込み直すパズルの解の見積り。
  - 例: h_2はタイルがあっても、その上を移動できるようにしたパズルの解の見積もり。
  - 地下鉄の例は、障害物もレールもなく、直線で移動できると仮定した弱条件問題。
  - 複数のの弱問題を作り、その見積もり関数を作ったときには、

$$h(x) = \max(h_1(n), h_2(n), \ldots, h_m(n))$$

とする。

89 A*探索の拡張

A*探索の拡張

- 高速で、実時間性に耐えうる入門的探索を紹介する。
  - 反復深化A*(IDA*)
  - 単純化メモリ限定A*（SMA*）
  - 実時間A* (Realtime A*)
- これまでの探索の問題点
  - 計算量もあるが、それ以上に空間量（メモリ使用量）が問題
  - 高速性をあげる目的で限定された量のメモリを使用

  IDA*　: 反復深化探索にヒューリスティック関数を導入

  SMA* : A*でメモリの使用量を制限したもの
  - ゴールまで到達する経路を求める時間がない。ゴールが移動？

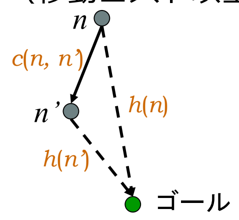
  RTA* : 一定時間内である程度進む。そこで再計算。

準備

- 無矛盾なヒューリスティック関数
  - 許容的なヒューリスティック関数hが無矛盾(consistent) もしくは単調 (monotonic) であるとは、任意のノードnと、nの任意の子ノードn'について、

    $h(n) \leq c(n, n') + h(n')$　[三角不等式]

    が成立。（移動コスト以上に$h(n')$は小さくならない）



反復深化A*探索 －　IDA*

- 反復深化は完全で最適であると同時に、メモリ使用量が少なかった。
- 反復深化探索を行うが、展開する順序にA*で使ったヒューリスティック関数を使用する。
  - $f(n) = g(n) + h(n)$ 許容的で無矛盾とする
  - アイデア：
    各探索は深さ優先探索だが、反復深化のように探索木の深さで探索をカットするのではなく、関数fの値によって探索をカットする。

反復深化A*探索 －　IDA*

- アルゴリズム

Step1: $limit = c$ (c=$h(n)$, nは出発のノード。c以上の適切な値でもよい)

Step2: 出発の節点nに対し(n $null$ c)をスタックに積む。 c=$h(n)$=$f(n)$

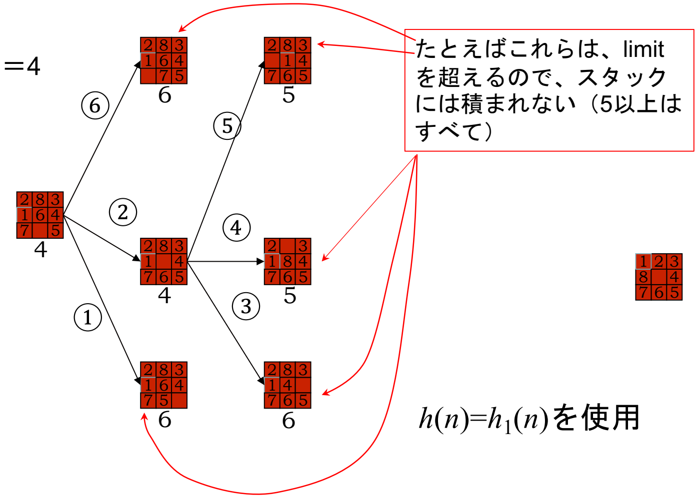
Step3: スタックが空であれば、$limit$=$limit$+1 (1でなく適当な正数としてもより）とし、Step2から再スタートする。スタックの先頭の節点nからまだチェックしていない節点をひとつ展開しそれをn'とする($f(n)$の値が小さいもの から選んでもよい。効果は不定)。もしnの子節点をすべて展開し終えていたら、スタックの先頭を取り出し、Step3を繰り返す。

Step4 : n'がゴールであれば終了。$f(n')$≤$limit$であれば、(n' n $f(n')$)をスタックに積む。Step3へ。

- コスト均一探索は、あるノードを一度に展開するが、IDA*では一つづつ展開し、コスト（ヒューリスティック関数の値）が$limit$を超えたものは忘れ去る。

8パズル- IDA*探索 (h_1)

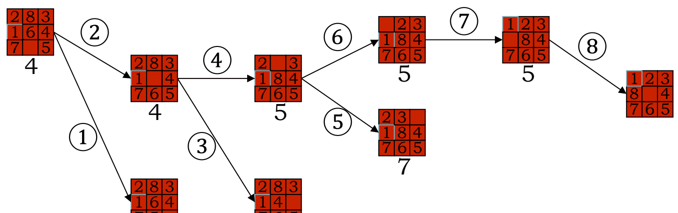
境界値＝4



$h(n)=h_1(n)$を使用

8パズル- IDA*探索 (h_1)

境界値＝5



$h(n)=h_1(n)$を使用

反復深化A*探索の性質

- メモリ使用量は、まだ展開し終えていない、指定されたコスト以下の節点だけをスタック状に積むだけ。他の方法と比べメモリ使用量は極端に小さい。
- 計算量はA*と理論的には同程度（無駄な繰り返しはあるが、オーダーとしては同じ。もったいないと思うと…）。
- メモリ使用量が小さく、実際の計算は効率的。
- 完全性と最適性は明らか。
- 欠点：
  - 問題は無駄な繰り返しが含まれる。つまり、*limit*の値を変えるとき、すべてを忘れ（スタックには何もない）るので、 *limit*の値以外覚えていない。IDSと同じ欠点だがメモリ使用量は少ない。
  - しかし、複雑な領域ではこの繰り返しが影響し、効率を下げる。

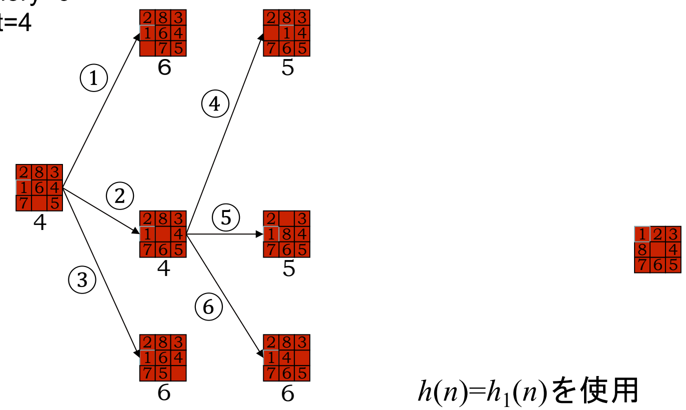
単純メモリ限定A*探索

- Simplified Memory-Bounded A* search (SMA*)
- IDA*の欠点：
  - 問題は無駄な繰り返しが含まれる。
- では、再探索を減らすためにある量 (memory) だけノードを記憶できるようにし、代わりに無駄な繰り返しを減らそう。
  - ノードを繰り返し展開し、そのヒューリスティック関数を計算するよりは、記憶しておいた方が効率的になるだろう。でも全部というわけにはいかない。ある数のノードだけ記憶しておく。

    ➔ SMA*
  - 例による紹介。

8パズル- SMA*探索(h_1)

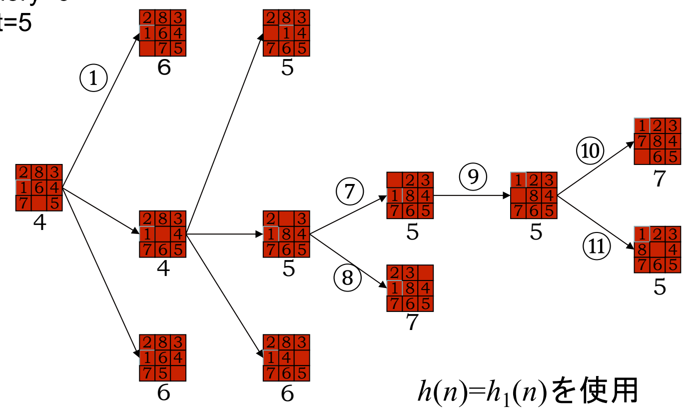
Memory=6
Limit=4



$h(n)=h_1(n)$を使用

8パズル- SMA*探索(h_1)

Memory=6
Limit=5



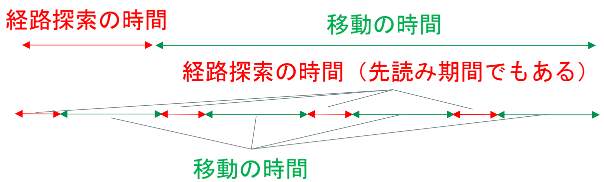
$h(n)=h_1(n)$を使用

単純メモリ限定A*探索

- Memoryが十分大きければA*やIDA*でも解くのに時間がかかる問題も効率的に解へる。
  - ただし、メモリが有限のため、同じ状態を生成したり消去するという振動現象がありうる。
- Memoryの値は最低限、ゴールまでの経路を記録できなくてはならない（前の例で、ゴールまでの経路は6状態あるので、これより小さければ経路を覚えられない）。
  - 特に、経路が長くてもコストが低い解もあるかも知れない。この最適の経路が表現できるだけのmemoryは必要である。
- 欠点は、消去節点の選択・メモリ管理など実装がやや複雑になること（プログラマの能力しだい）。

実時間A*探索 (Realtime A*, RTA*)

- 実時間で経路探索にあまり時間をかけない
  - 見かけでは停止せずに常に移動しているようにみせたい
  - センサーの視野範囲が限られていて、その先の状況は分からない
- ゴールが移動し、最適経路が変わる場合（やや上級）
  - 相手も移動体、あるいは環境の変化（障害物の変化）がある場合
    （少し進んではゴールを確認。変化の敏感度と効率の関係は単純ではない）



実時間A*探索 (Realtime A*, RTA*)

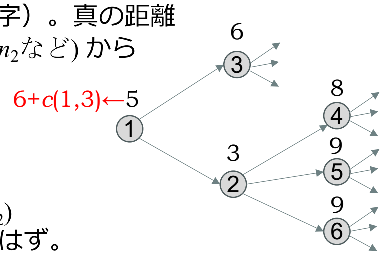
- 確認：　$f(n) = h(n) + g(n)$
- 1ステップ先読みする場合（多くの場合これを使う）
  - 現在の状態をn。fの値から上位2つの次のノードn_1, n_2を選択する。$f(n_1) \le f(n_2)$。 $h(n) = f(n_2)$と置き換える。探索を一時止めn_1に進む。
    - 一番良いノードを探索するが、後に後戻りすることがあるかもしれない。そのときn_1からスタートし、n へ戻りn_2に行くことになるが、 $h(n) = f(n_2)$ としたので、 $h(n)$ はその先の移動コストも考慮している（進んでしまえば最適経路も変わるということ）
    - 同時に探索の無限ループを防ぐ。後戻りしてもその方が良い場合もある。
    - バックトラックしたときには、あらためて上位２つのノードを選択することに注意。

実時間A*探索 (Realtime A*, RTA*)

- 2ステップ先読みする場合
  - 現在の状態をn。fの値から上位2つの次のノードn_1, n_2を選択する。$f(n_1) \le f(n_2)$。
  - n_1を展開して、同じく上位二つのノードn_3, n_4を選択
  - (必要に応じてn_2を展開し、同じく上位二つのノードn_5, n_6を選択)
  - n_3に進むときは、$h(n) = f(n_2)$、 $h(n_1) = f(n_4)$と置き換える。n_5に進むときは、 $h(n) = f(n_1)$、 $h(n_2) = f(n_6)$と置き換える。
- アルゴリズムは例（１ステップ先読み）で
  - ８パズルではオペレータのコストが常に１なので、別の問題で説明。

実時間A*探索 (Realtime A*, RTA*)

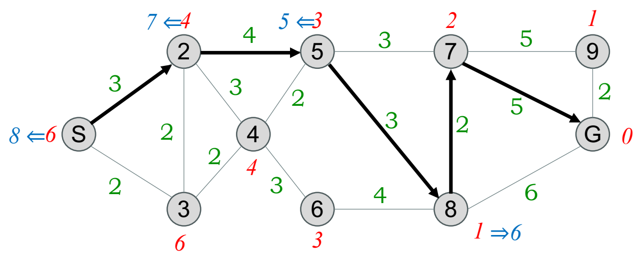
- hの値を書き換えたが、それでも許容的であるか？
  - もしバックトラックするなら、そのhの値は許容的
  - 最初の h の値は許容的（上の数字）。真の距離をh^*とする。もしノード2 (以下n_2など) からバックトラックして n_1に戻ったとすれば、hは許容的なので実際のh^*はさらに大きいはず。



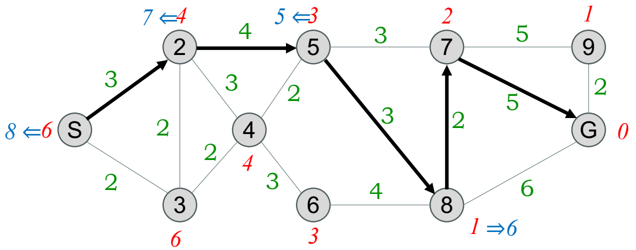
  - したがって、例では、
    $$6 + c(n_1, n_3) + c(n_1, n_2) \le h^*(n_2)$$
    n_2を経由する経路はさらに長いはず。
    以上から、 $6 + c(n_1, n_3) \le h^*(n_1)$
  - 直感的には、 hの値に惑わされてn_2ゴールに近いと判断しn_2に向かうが、その先ではゴールから遠いと分かった（戻った方が近い）。n_3側に別の経路があるかもしれず、バックトラックした。

RTA* 例題

緑：経路長
赤：hの値
青：修正後のh



RTA* 例題



緑：経路長
赤：hの値
青：修正後のh

課題２－１２：この例題の最短経路を確認せよ。
課題２－１３：RTA*を実装し、上記の例題を動かしてみよ。
課題２－１４：RTA*探索でも最短経路を得られるときのhの性質について検討せよ。

RTA*の性質

- 最適性は保証されない
- 実時間性、つまり迅速な行動選択が可能
  - 最適ではないが、まあ受け入れられる (acceptable) 経路をとる。
- 環境適応性：経路の一部だけが見えていればよい。
  - ただし見えている範囲のhは分かる。
  - 同時に、マップの一部を作成していることになる。
- メモリ量：移動回数に対して線形
  - 古いものは忘れられる。見通しのないものは忘れられる。
- 時間計算量：移動回数に対して線形
  - 探索の深さが定数、一回の移動のための探索も定数
- 完全性：解を見つけることはできる。

RTA*の拡張

- RTA*のhの値を書き換えることを応用し、hを与えなくてもhを学習しながら探索するLearning Real time A* (LRTA*)
- ゴールが移動することを想定したMoving target search algorithm (MTS) などがある
  - これらは入門の範囲を超えるので割愛（大学院レベル）。
  - これらはプランニングの研究などと合わせて理解すると良い。

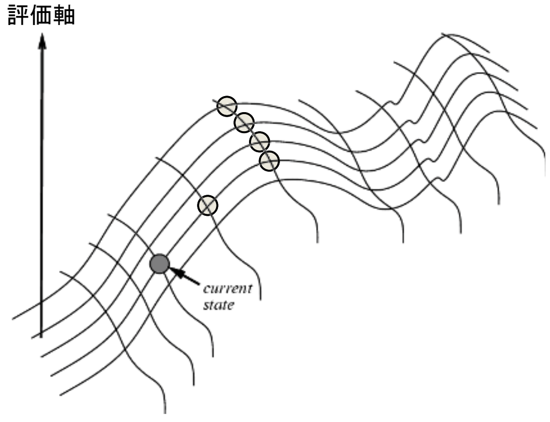
反復改良アルゴリズム

反復改良アルゴリズムとは

- 問題によっては解を得る経路（オペレーションの列）はどうでもよく、解のみが分かればよいことがある。
  - たとえばLSIの設計などでは時間はかかってもよいから質の高いものを得たい。
- 反復改良アルゴリズム (ニューラルネットワークでも利用)
  - アイデア：任意の状態からはじめて、その質を少しでも向上させるということを繰り返して、望むゴールを得ること。
  - 山登り法 (hill-climbing)
    [または勾配降下法 (gradient descent)とも言う]
  - Simulated annealing （焼きなまし法）

山登り法　(Hill-climbing)

- アイデア：
  現在の状態に可能なオペレーションを施し、その中で最適な状態へ移動する。
  「良さ」 (value) を高さにたとえると…..
- 「良さ」を高さにたとえる代わりに、コストで表現すると低いほうがよい、つまり谷を見つける
  ➔ 勾配降下法ともいう。



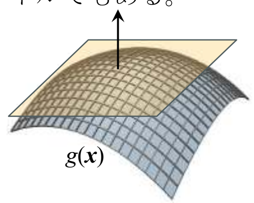
山登り法　(Hill-climbing)

function Hill-Climbing (*problem*) **returns** a solution state
  inputs: *problem* // a problem
  local variables: *current*, // a node
          next // a node
  current ← Make-Node (Initial-State[*problem*])
  loop do
    next ← a highest-values successor of *current*
    if Value[*next*] < Value[*current*] **then return** *current*
    current ← *next*
  end

- オペレーションが離散的なら、successors（次の状態）のValueを比較する。連続量（たとえばハンドルの角度、照度など）の場合は、微分して勾配の高い方向に進む。

次の状態 (successors) の決め方

- まずは数学の復習から：
  - 平面$a_1x_1 + \cdots + a_nx_n = 0$の法線ベクトルは $(a_1, \ldots, a_n)$である（内積を考えれば良い。平面が原点を通ることに注意）。したがってこれを平行移動した$a_1x_1 + \cdots + a_nx_n = C$の法線ベクトルでもある。
  - 一般に（２階連続微分可能な）関数$g(\boldsymbol{x})$に対し
    $$\left(\frac{\partial g(\boldsymbol{x})}{\partial x_1}, \cdots, \frac{\partial g(\boldsymbol{x})}{\partial x_n}\right)$$
    を勾配と言い、$\nabla g(x)$または$grad\ g(x)$と表す。
  - 図のような曲面に接する面は、
    $$\frac{\partial g(\boldsymbol{x})}{\partial x_1} x_1 + \cdots + \frac{\partial g(\boldsymbol{x})}{\partial x_n} x_n = C$$
    と書け、勾配$\nabla g(x)$は曲面の法線ベクトルとなる。



次の状態 (successors) の決め方

- 最急降下法
  - よって評価関数の曲面に接する面、
$$\frac{\partial g(\boldsymbol{x})}{\partial x_1}x_1 + \cdots + \frac{\partial g(\boldsymbol{x})}{\partial x_n}x_n = C$$
を求め、評価関数が大きく（小さく）なる方向にある移動量 (α) だけ動けばよい。これを最急降下法 (steepest descent method) とよぶ。
- 他にもいくつかの方法がある。概要のみ
  - モーメンタム (Momentum) ：$x(t)$ から$x(t+1)$へ移動するとき、$x(t-1)$からx(t)での向上した量を反映させよう。
  - Adagrad: 方向や実績に応じてαの値を変える。
  - これらは大学院で必要に応じて講義されると思います。

山登り法 (Hill-climbing) の特徴

- 単純な方法であり、メモリ使用量も小さい。
- 有名な欠点：
  - 局所最大（極大）：　局所的に極大の位置に達すると、そこから抜け出せない。谷を越えた先のもっと高い山にはいけない。
    - 勾配降下なら極小（local minimum）という。この分野では、いずれにしろこの欠点をlocal minimumと呼ぶ習慣がある。
  - 高原：平たい高い位置。平らなのでランダムに動き回る。
  - 峰（尾根）：尾根には容易に達するが、山頂まで緩やかで（起伏がある場合も）質 (value) がなかなか向上しない。局所的極大になりやすい。
（できれば沢のぼりをしたい。これは直接、頂上を目指すこと）

山登り法の欠点への対応

- ランダム再スタート (random-restart hill-climbing)
  - 探索がとまったり、進行しなくなったら、任意の状態を選択し、再スタートする。Local maximum (minimum)に入ったときに抜け出し、もっとより状態を探索できる。
  - 何度も再スタートをすればいつかは最適な解を見つけられる。ただ問題が複雑（たとえばNP）であれば、やはり再スタートも指数乗回必要。実際の問題では許される回数で、「最適ではないが受理可能」な解を見つけることが多い。
- Simulated annealing
  - Local maximumに入ったとしても、ある確率で周囲を動き回る。その結果、局所的なピークを抜け出せるかも知れない。

Simulated Annealing （焼きなまし法）

【これはあくまで基本形】

function Simulated-annealing (*problem, schedule*) **returns** a solution state
  inputs: *problem*, // a problem
    schedule, // a mapping from time to temperature
  local variables: *current*, // a node
    next, // a node
    T, // temperature controlling the probability of downward
  current ← Make-Node (Initial-State[*problem*])
  for t ← 1 **to** ∞ **do**
    T ← *schedule*[*t*]
    if *T* = 0, then return *current*
    next ← a randomly selected successor of *current*
    ΔE ← Value[*next*] – Value[*current*]
    if *ΔE* > 0 **then** *current* ← *next*
    else *current* ← *next* only with the probability $e^{\Delta E/T}$ ($\Delta E \leq 0$ に注意)

Simulated Annealing （焼きなまし法）

- T の値（温度）がランダムに動き回る確率を決める。
  - *schedule*(t)は時間とともに温度Tを徐々に下げる関数。
  - ランダムに選択した新しい状態が、今より「良い」なら無条件でそこに進むが、良くない場合にもΔE とT に依存する確率でその状態に進む。
  - $T = 0$のときランダムに進むことはなくなる（「良い方向のみ」したがって、極大点に達すると止まる）。Tの値が大きくなれば、動き回る確率が大きくなる。このことから、Tを温度という。
  - 焼きなましとは、液体の温度を徐々に下げて凍らしていく方法のこと。分子の動きを徐々に下げて、きれいな固体を生成する。

反復改良アルゴリズム

- 局所最大（最小）や局所最適の問題はあるが、比較的低コストで解を求められることがある。
- ただし本アルゴリズムでは、問題の結果が連続的に変化することを仮定している。
  - これは問題のパラメータに微小の変化を加えると解も微小の変化にとどまることを前提としている（連続微分可能性）。
  - たとえば、囲碁（将棋）などのように一マス石の位置が変わっただけで場面の評価が大きく変わったり、最善手が変わることはないと仮定している。
    - 連続的変化でないことが分かっていても、実際にはこれを仮定してまずは適用し、あとで学習により変更・改善するという戦略もある。よいと保証はされないが、それを気にせず利用している。実用上問題はないが、失敗が許されない領域（たとえば自動運転）では注意が必要。

ヒューリスティック探索まとめ

- ヒューリスティックを利用した探索
  - 最良優先探索、Greedy search
  - A*探索（完全かつ最適、ただしメモリ使用量がやや大）
  - IDA*、SMA*メモリ使用量を減らすように改良したもの
  - RTA* 実時間アプリケーション向け（時間をかければ完全。最適ではない）
  - 山登り法、焼きなまし法
- 大切なこと
  - 良いヒューリスティック関数を作ることが探索を効率化する
  - ただし最適性を求めるには、それなりの探索コストがかかる。
  - 各探索方法に、利点・欠点があるので、問題の性質を考慮して選択しなくてはならない。