探索による問題の解決

Chapter 3

概要

- 問題解決エージェント
- 問題の種類
- 問題の形式化
- 例題
- 基本的な探索アルゴリズム

問題解決エージェント

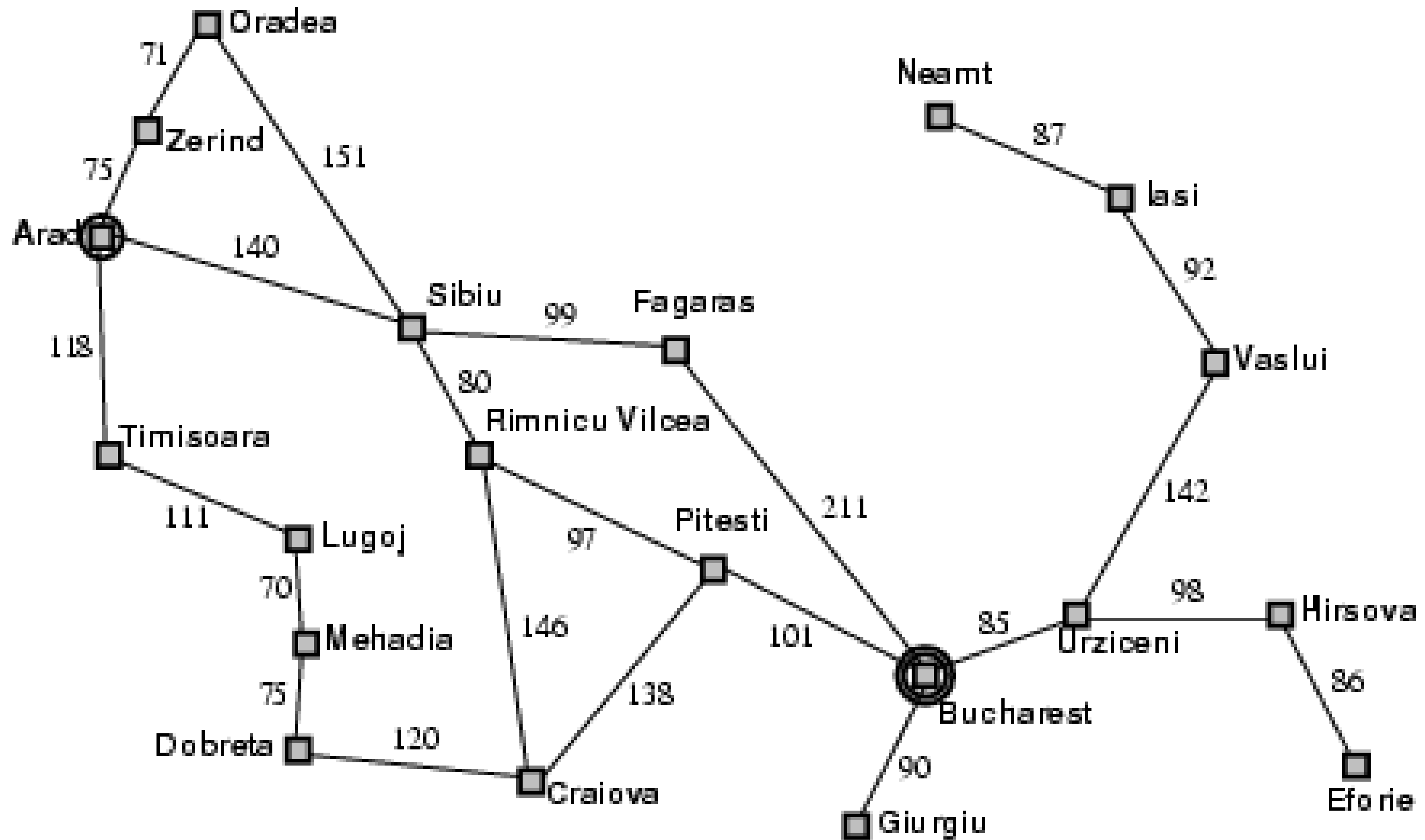
```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  static: seq, an action sequence, initially empty
          state, some description of the current world state
          goal, a goal, initially null
          problem, a problem formulation

  state ← UPDATE-STATE(state, percept)
  if seq is empty then do
     goal ← FORMULATE-GOAL(state)
     problem ← FORMULATE-PROBLEM(state, goal)
     seq ← SEARCH(problem)
  action ← FIRST(seq)
  seq ← REST(seq)
  return action
```

例: ルーマニア

- ルーマニアでの休日; 現在アラドにいる
- 明日ブカレストから飛行機で出発する
- ゴールを形式化:
  - ブカレストにいる
- 問題を形式化:
  - 状態: たくさんの市
  - 動作: 市の間のドライブ
- 解の発見:
  - 市の列, e.g., Arad, Sibiu, Fagaras, Bucharest

Romania
International boundary
County (judeţ) boundary
National capital
County (judeţ) center
Railroad
Expressway
Road
Bucuresti is a municipality (municipiu).
0 25 50 75 100 Kilometers
0 25 50 75 100 Miles
Lambert Conformal Conic Projection, SP 40N/56N
Serbia and Montenegro have asserted the formation of a joint independent state, but this entity has not been formally recognized as a state by the United States.
UKRAINE
MOLDOVA
HUNGARY
SLOVAKIA
Serbia
BULGARIA
THE FORMER YUGOSLAV REPUBLIC OF MACEDONIA
Vojvodina (autonomous province)
Kosovo (autonomous province)
Black Sea
Bucharest
Chisinău
Belgrade
Sofia
Skopje
SATU MARE
MARAMUREŞ
SUCEAVA
BOTOŞANI
BISTRIŢA-NĂSĂUD
IAŞI
NEAMŢ
SĂLAJ
CLUJ
BIHOR
MUREŞ
HARGHITA
BACĂU
VASLUI
ARAD
ALBA
COVASNA
SIBIU
BRAŞOV
VRANCEA
GALAŢI
TIMIŞ
HUNEDOARA
VÎLCEA
BUZĂU
CARAŞ-SEVERIN
GORJ
ARGEŞ
PRAHOVA
BRĂILA
TULCEA
DÎMBOVIŢA
BUCUREŞTI
IALOMIŢA
MEHEDINŢI
DOLJ
OLT
CĂLĂRAŞI
GIURGIU
TELEORMAN
CONSTANŢA
Danube
Prut
Dnistr
Nistru
Tisa
Olt
Velika Morava
Južna Morava
Morava
Maritsa
Base 802205 (R01144) 5-96

例: ルーマニア

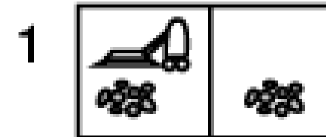


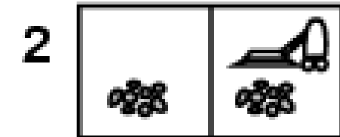
問題のタイプ

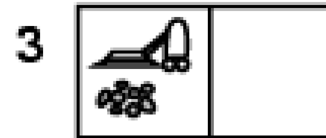
- 決定的で、完全に観察できる→ 単一状態問題
  - エージェントはどこの市にいるかを知っている; 解は列である
- 非観察的 → センサーのない問題 (整合問題)
  - エージェントはどこにいるかがわからない;解は列である
- 非決定的で部分的に観察可能→ 偶発性の問題
  - 近くは現在の状態について新しい情報を与える
  - しばしば介入の探索、実行
- 未知の状態空間 → 探索問題

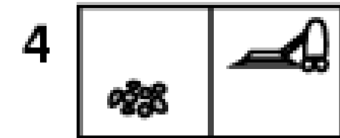
例：掃除機の世界

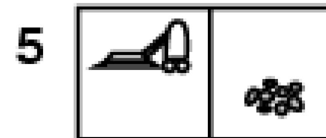
- 単一状態, #5で開始.
  Solution?

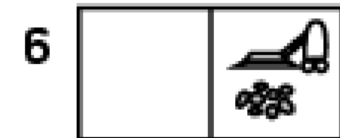
1 

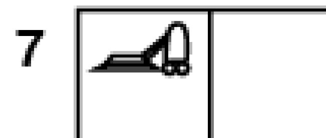
2 

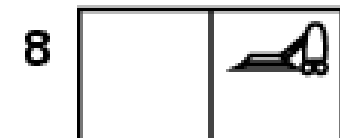
3 

4 

5 

6 

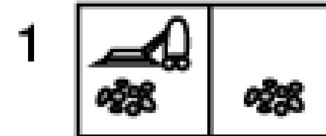
7 

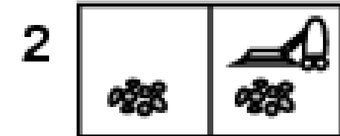
8 

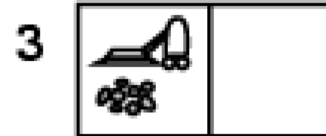
例：掃除機の世界

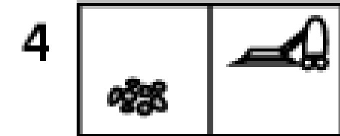
- 単一状態, #5で開始.
  Solution? *[Right, Suck]*

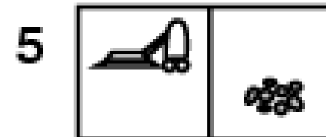
- 知覚なし, {*1,2,3,4,5,6,7,8*}
  のどれかで開始
  右に行くと {*2,4,6,8*}
  Solution?

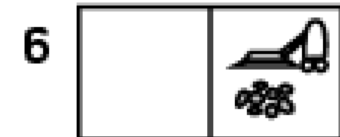
1 

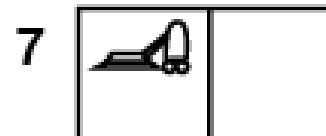
2 

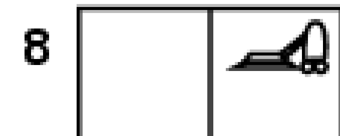
3 

4 

5 

6 

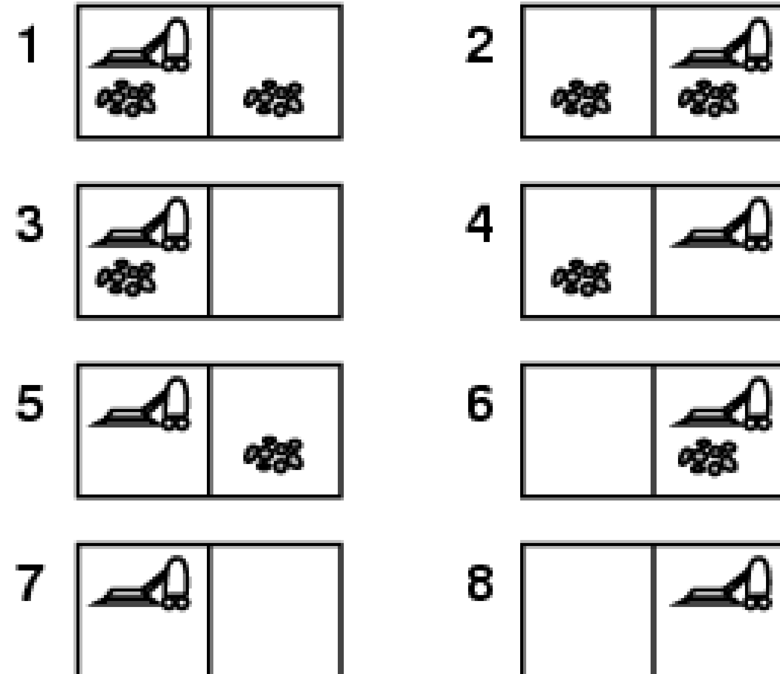
7 

8 

例：掃除機の世界

- 知覚なし, *{1,2,3,4,5,6,7,8}*
  のどれかで開始
  *右に*行くと *{2,4,6,8}*
  Solution?
  [Right,Suck,Left,Suck]

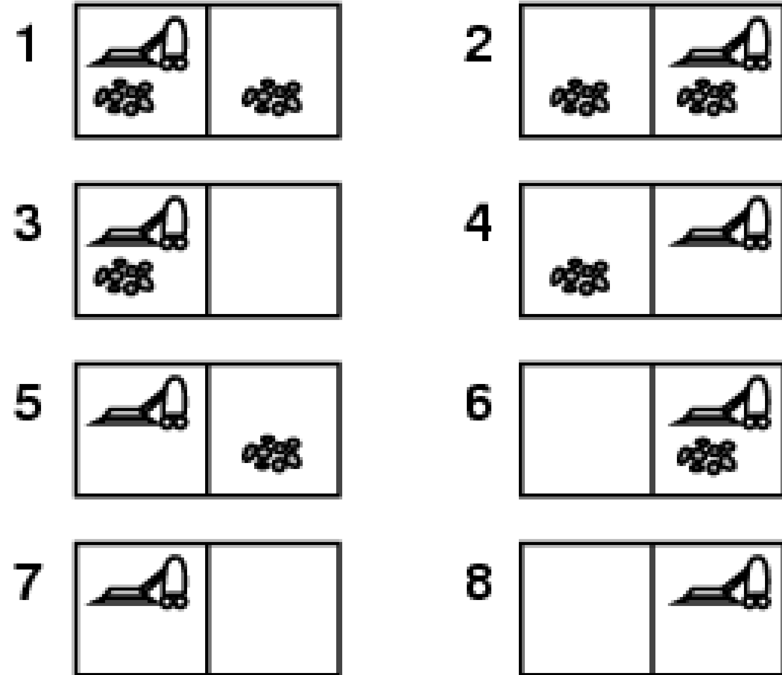
- 偶発的
  - 非決定的: *吸込み*は
    きれいな所を汚す可能性も
  - 部分的に観察可能: 場所, 汚れ
  - 知覚: *[L, Clean]*。即ち、#5 or #7で開始
    Solution?



例：掃除機の世界

- 知覚なし, *{1,2,3,4,5,6,7,8}*
  のどれかで開始
  右に行くと {2,4,6,8}
  Solution?
  [Right,Suck,Left,Suck]

- 偶発的
  - 非決定的: *吸込みは*
    きれいな所を汚す可能性も
  - 部分的に観察可能: 場所, 汚れ
  - 知覚: *[L, Clean]*。即ち、#5 or #7で開始
    Solution? *[Right, **if** dirt **then** Suck]*



単一状態問題の定式化

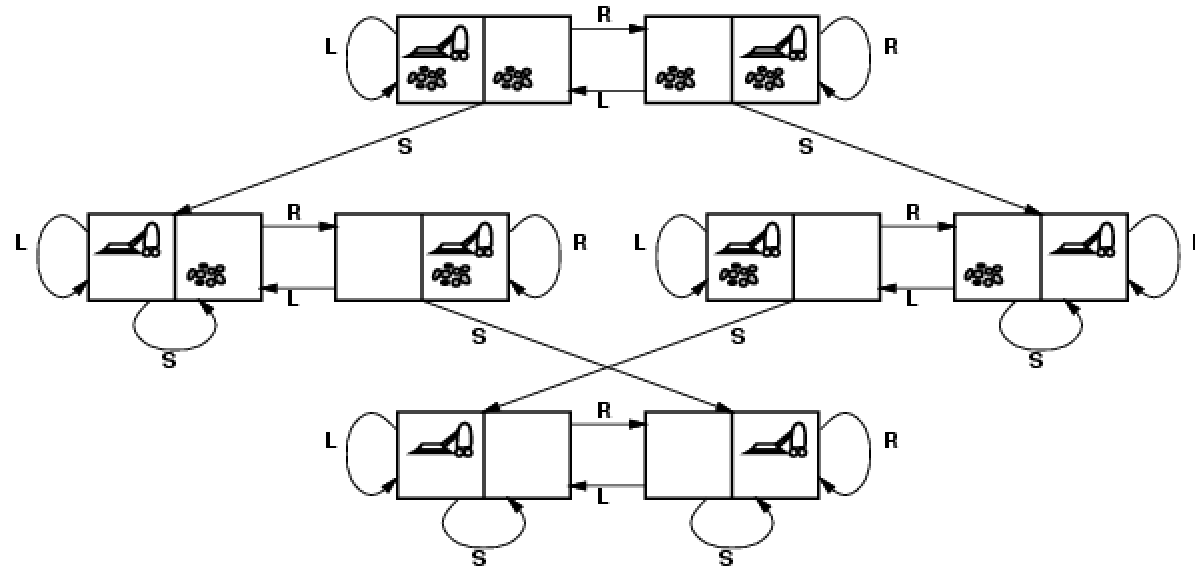
問題は4つのアイテムで定義される:

1. 初期状態 例えば "at Arad"
2. 動作 又は 後継関数 *S(x)* = 動作と状態の対の集合
   - 即ち、*S(Arad)* = {<*Arad* → *Zerind, Zerind*>, ... }
3. ゴールテスト
   - 明示的、例えば *x* = "at Bucharest"
   - 暗示的、例えば *Checkmate(x)*
4. 経路のコスト (additive)
   - 距離の総和、実行された動作の数
   - *c(x,a,y)*はステップコスト、0よりは大きいとする
   - 解 は初期状態からゴール状態に導く動作の列

状態空間の選択

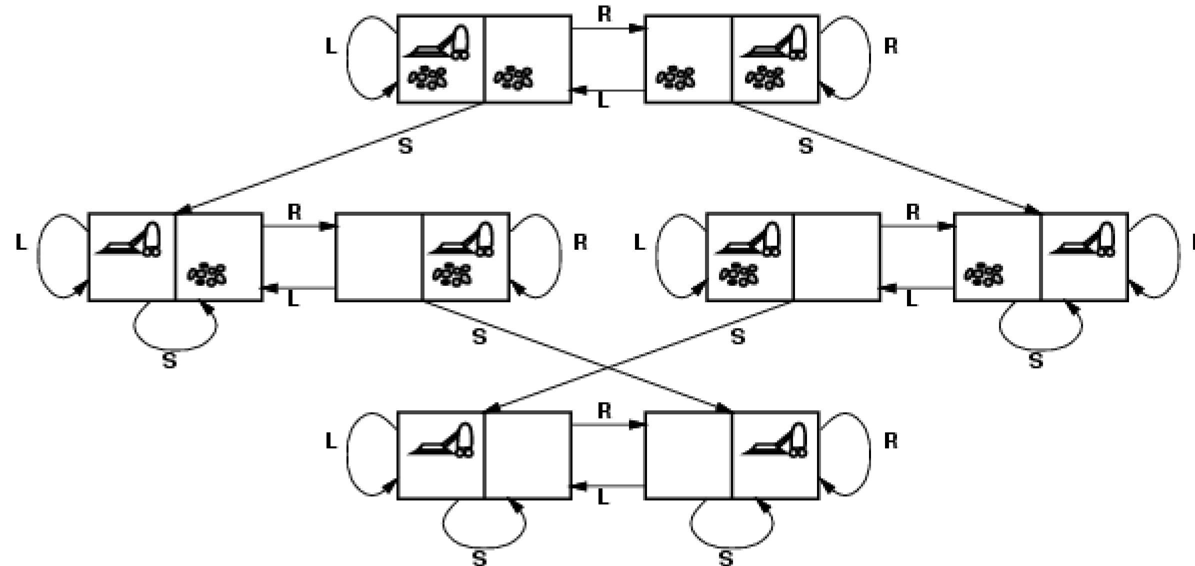
- 現実世界は途方もなく複雑
  → 状態空間は問題を解くために抽象化する
- (抽象) 状態 = 現実の状態の集合
- (抽象) 動作 = 実際の動作の複雑な組合せ
  – 例: “Arad → Zerind” は可能なルート、迂回路、休憩などの複雑な集合
- 実現性を確保するために、“in Arad“ のいかなる現実の状態は”in Zerind”のある現実の場所に到達しなければならない
- (抽象) 解 =
  – 現実の世界での解である実際の経路の集合
- 抽象化された動作は最初の問題よりもやさしくなっている

掃除機の世界の状態空間グラフ



- states?
- actions?
- goal test?
- path cost?

掃除機の世界の状態空間グラフ



- states? 埃と掃除機の場所
- actions? *左、右、吸取り*
- goal test? 全ての場所で埃がない
- path cost? 動作あたり1

例: 8パズル

7	2	4
5		6
8	3	1

Start State

	1	2
3	4	5
6	7	8

Goal State

- states?
- actions?
- goal test?
- path cost?

例: 8パズル

7	2	4
5		6
8	3	1

Start State

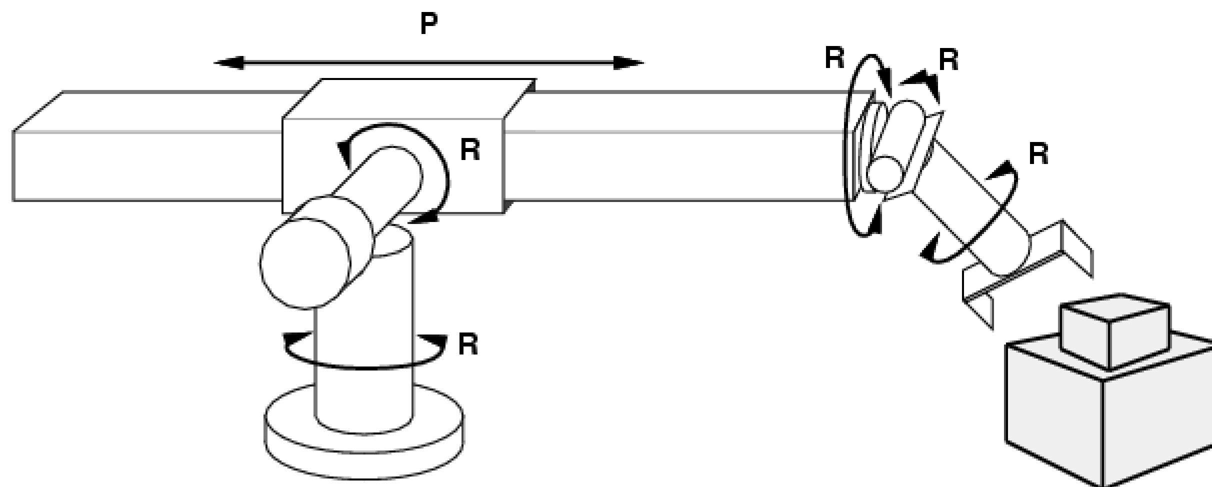
	1	2
3	4	5
6	7	8

Goal State

- states? タイルの場所
- actions? ブランクを左右上下に移動
- goal test? = ゴールの状態 (与えられている)
- path cost? 動作あたり1

[Note: optimal solution of *n*-Puzzle family is NP-hard]

例: ロボットのアセンブリ



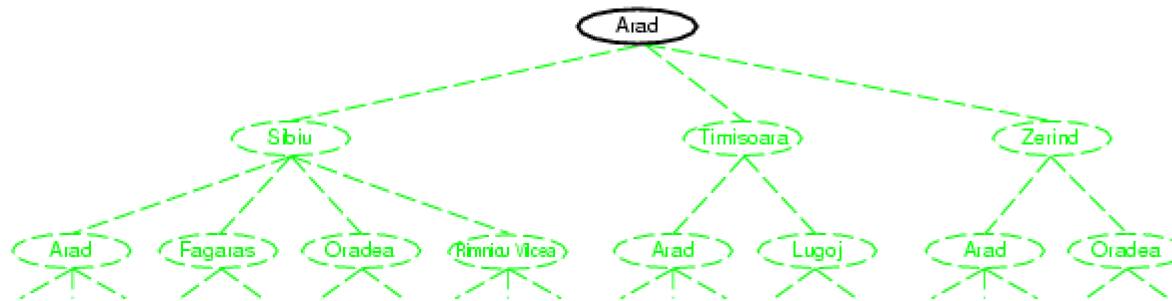
- states?: ロボットの関節部分の角度
- actions?: ロボットの関節の連続的な動き
- goal test?: 完全な組立て
- path cost?: 実行時間

木探索アルゴリズム

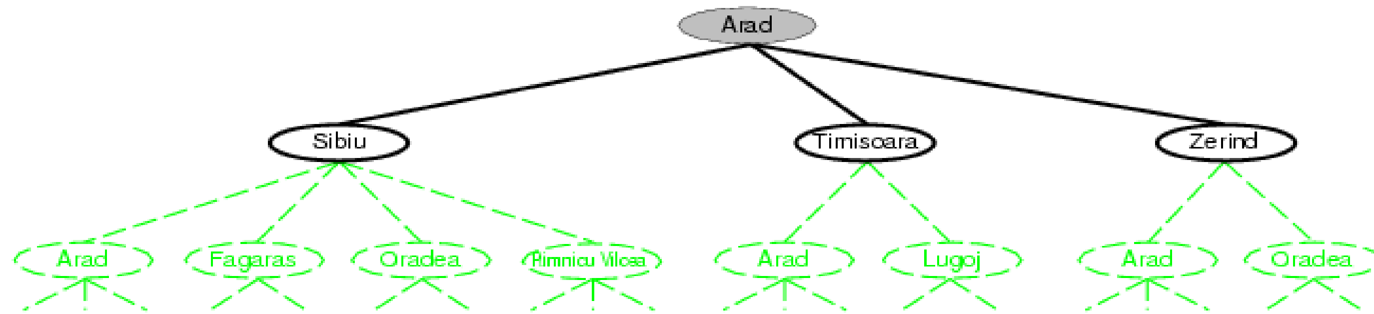
- 基本的な考え方:
  - 既に探索された状態に続く状態を生成することで状態空間をオフラインで模擬的に探索する

```
function TREE-SEARCH( problem, strategy) returns a solution, or failure
   initialize the search tree using the initial state of problem
   loop do
      if there are no candidates for expansion then return failure
      choose a leaf node for expansion according to strategy
      if the node contains a goal state then return the corresponding solution
      else expand the node and add the resulting nodes to the search tree
```

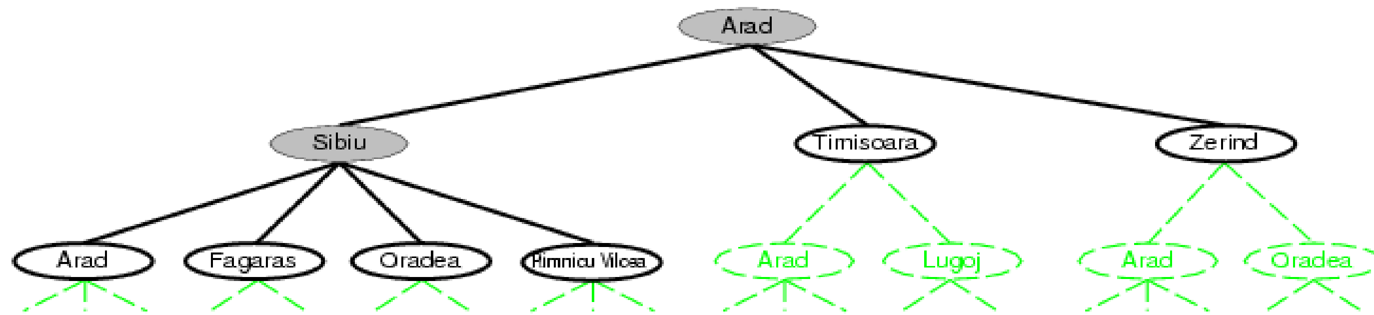
木探索アルゴリズム



木探索アルゴリズム



木探索アルゴリズム



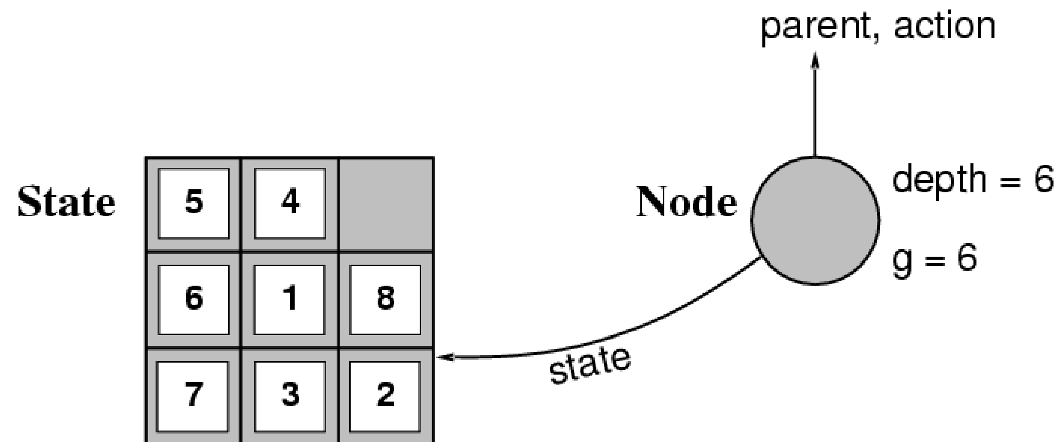
実現: 一般的な木探索

```
function TREE-SEARCH(problem, fringe) returns a solution, or failure
   fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
   loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
        fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

```
function EXPAND(node, problem) returns a set of nodes
   successors ← the empty set
   for each action, result in SUCCESSOR-FN[problem](STATE[node]) do
        s ← a new NODE
        PARENT-NODE[s] ← node;  ACTION[s] ← action;  STATE[s] ← result
        PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)
        DEPTH[s] ← DEPTH[node] + 1
        add s to successors
   return successors
```

実現: 状態vsノード

- 状態 は物理的な構成(の表現)
- ノードは探索木の部分を構成しているデータ構造。ノードは状態、親ノード、動作、経路のコスト*g(x)*、深さなどを含む



- `Expand`関数は、たくさんのフィールドを埋めることでさらに対応する状態を生成するために`Successor`を使って、新しいノードを構成する

探索戦略

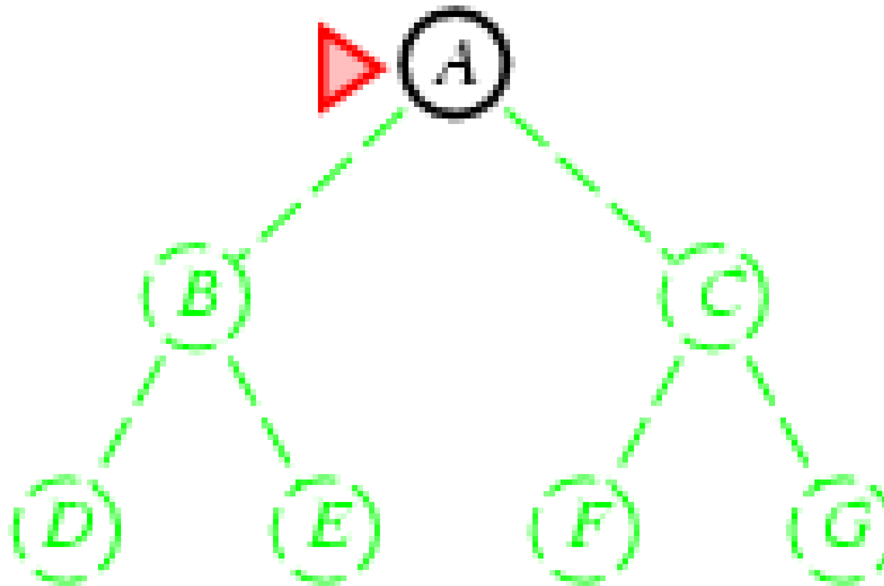
- 探索戦略はノードの拡張の順番を選択することにより定義される
- 戦略は次の特徴から評価される:
  - 完全性: 解が存在する場合には常に見つけられるか
  - 時間の複雑性: 生成されるノードの数
  - 空間の複雑性: メモリ内での最大のノード数
  - 最適性: 最小のコストの解をいつも発見できるか
- 時間と空間の複雑性は次の項で計られる
  - *b:* 探索木での最大の分岐数
  - *d:* 最小コストの解に対する深さ
  - *m*: 状態空間の最大の深さ (∞のことも)

一様探索戦略

- 一様探索戦略は問題を定義したときに得られる情報だけを用いる
- 幅優先探索
- 均一コスト探索
- 深さ優先探索
- 深さ制限探索
- 反復深化探索

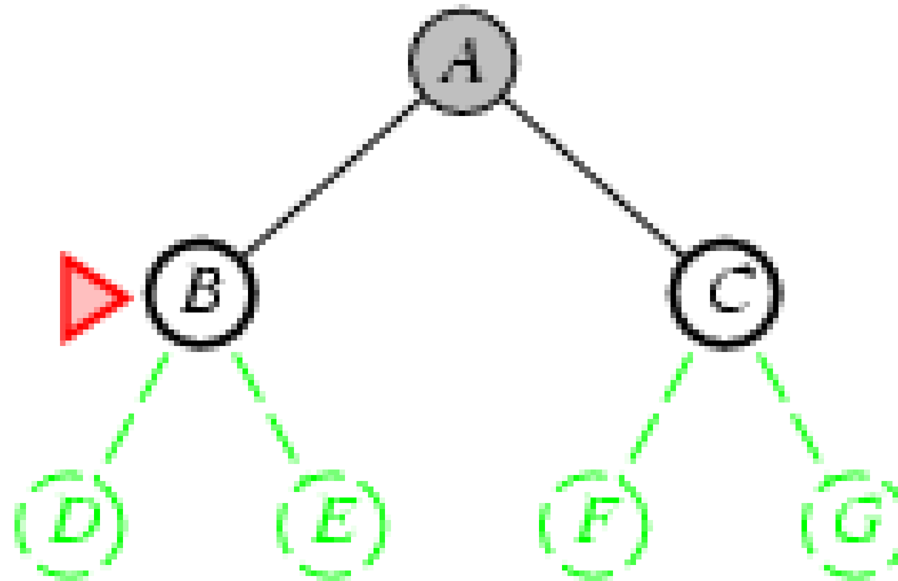
幅優先探索

- もっとも浅い拡張されていないノードを拡張する
- 実現:
  - フリンジはFIFOのキュー：新しい後継は最後に



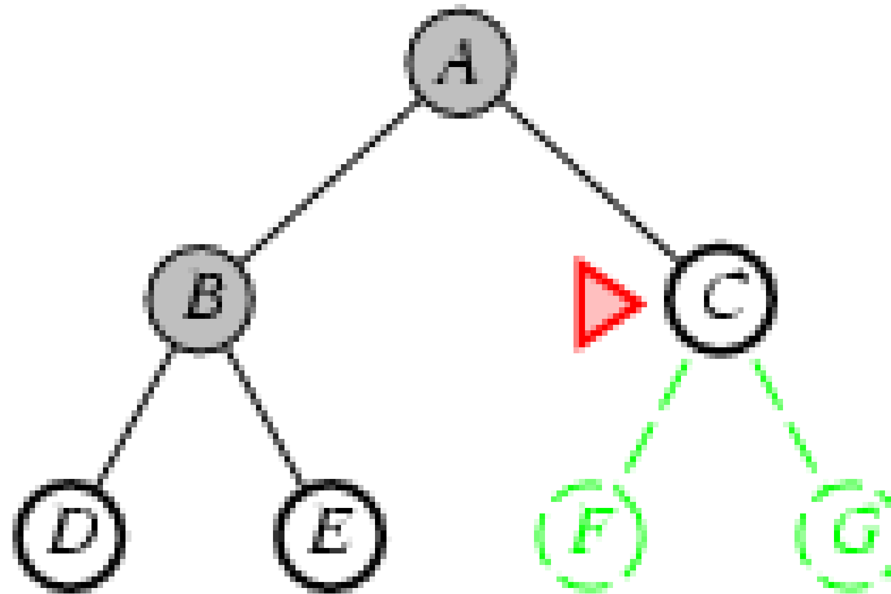
幅優先探索

- もっとも浅い拡張されていないノードを拡張する
- 実現:
  - フリンジはFIFOのキュー :新しい後継は最後に



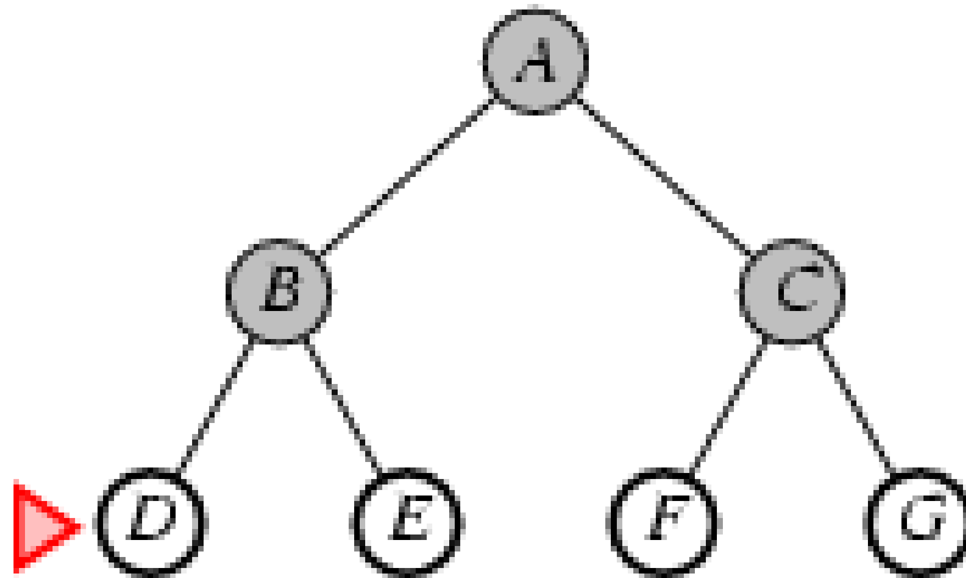
幅優先探索

- もっとも浅い拡張されていないノードを拡張する
- 実現:
  - フリンジはFIFOのキュー :新しい後継は最後に



幅優先探索

- もっとも浅い拡張されていないノードを拡張する
- 実現:
  - フリンジはFIFOのキュー :新しい後継は最後に



幅優先探索の性質

- Complete? Yes (bが有限なら)
- Time? $1+b+b^2+b^3+\ldots+b^d + b(b^d-1) = O(b^{d+1})$
- Space? $O(b^{d+1})$ (全てのノードをメモリに)
- Optimal? Yes (ステップあたりcost = 1 なら)

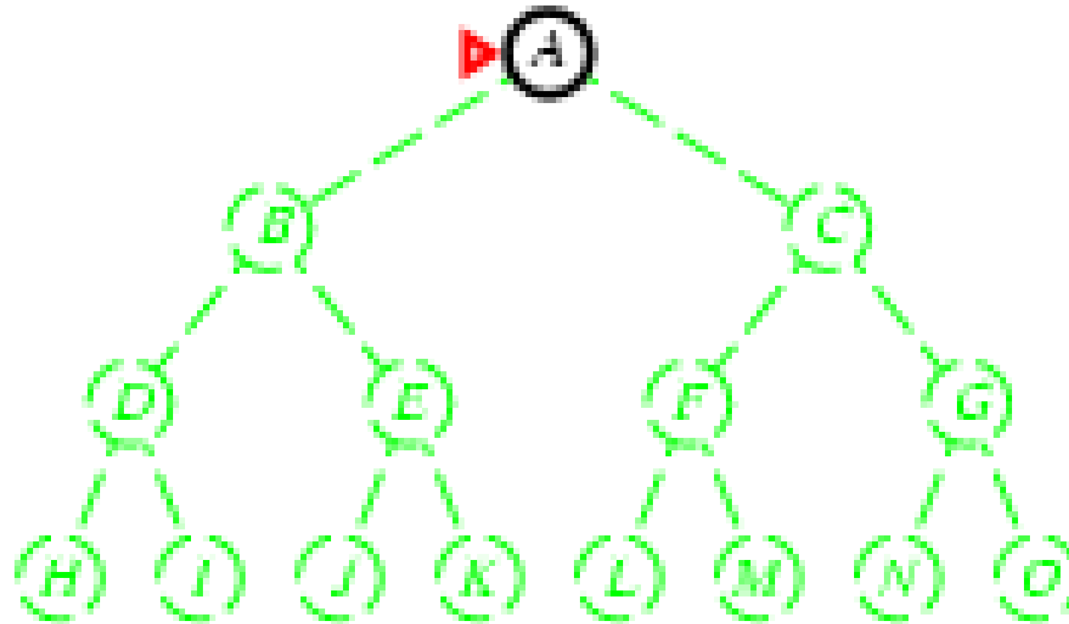
- 空間は大きな問題 (時間よりも)

均一コスト探索

- 最小コストの拡張されていないノードを拡張
- 実現:
  - フリンジ= 経路のコストで並べられたLIFOキュー
- ステップコストが全て同じであるときは幅優先探索と同じ
- Complete? Yes, ステップコスト ≥ εであるなら
- Time? そこに至るまでのコストが 最適解のコストよりも小さいノードの総数
  $O(b^{ceiling(C^*/\varepsilon)})$ ここで C^* は最適解のコスト
- Space?そこに至るまでのコストが 最適解のコストよりも小さいノードの総数
  $O(b^{ceiling(C^*/\varepsilon)})$
- Optimal? Yes –ノードは$g(n)$を増大させながら拡張される

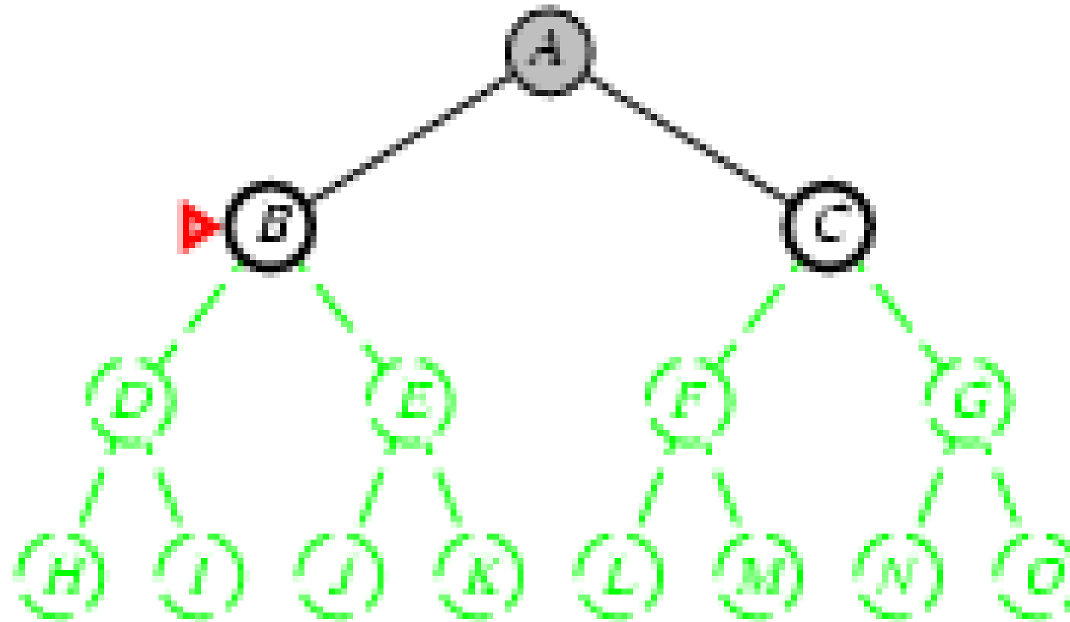
深さ優先探索

- 最も深い拡張されていないノードを拡張
- 実現:
  - フリンジ = LIFOキュー、後継は先頭に



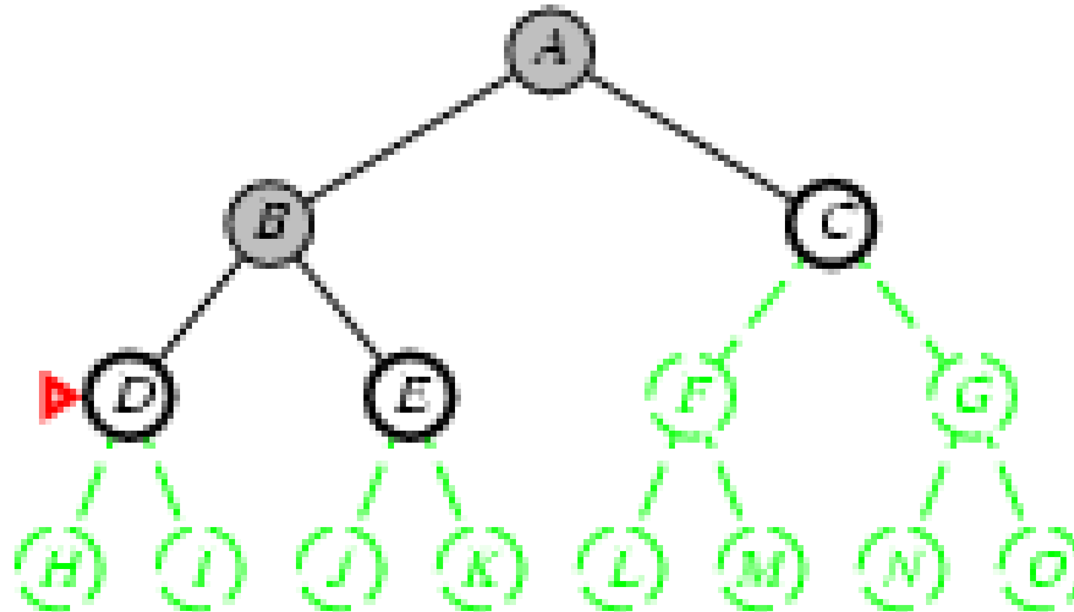
深さ優先探索

- 最も深い拡張されていないノードを拡張
- 実現:
  - フリンジ = LIFOキュー、後継は先頭に



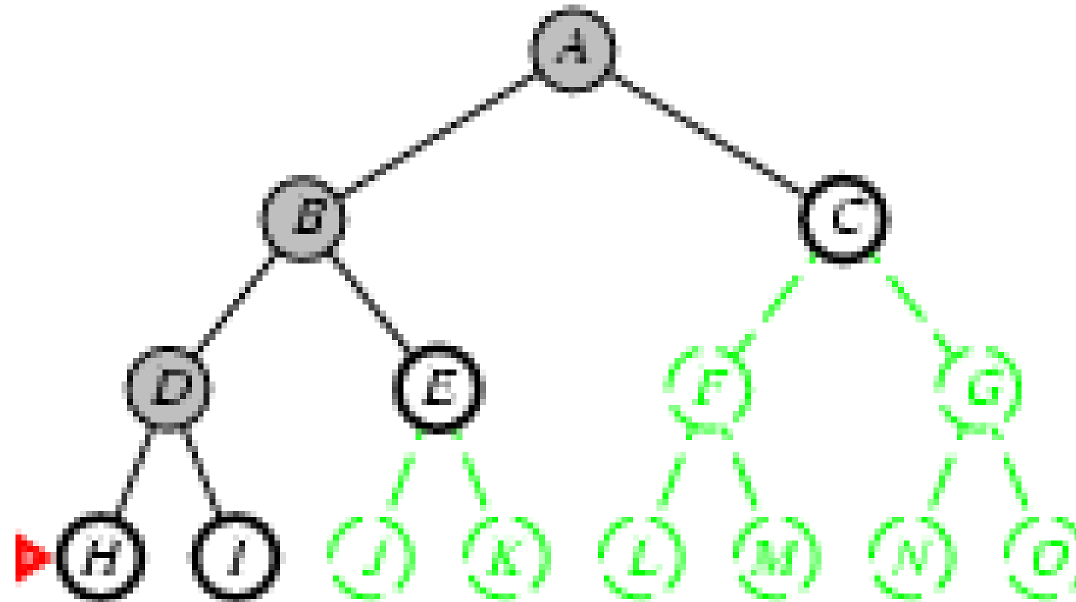
深さ優先探索

- 最も深い拡張されていないノードを拡張
- 実現:
  - フリンジ = LIFOキュー、後継は先頭に



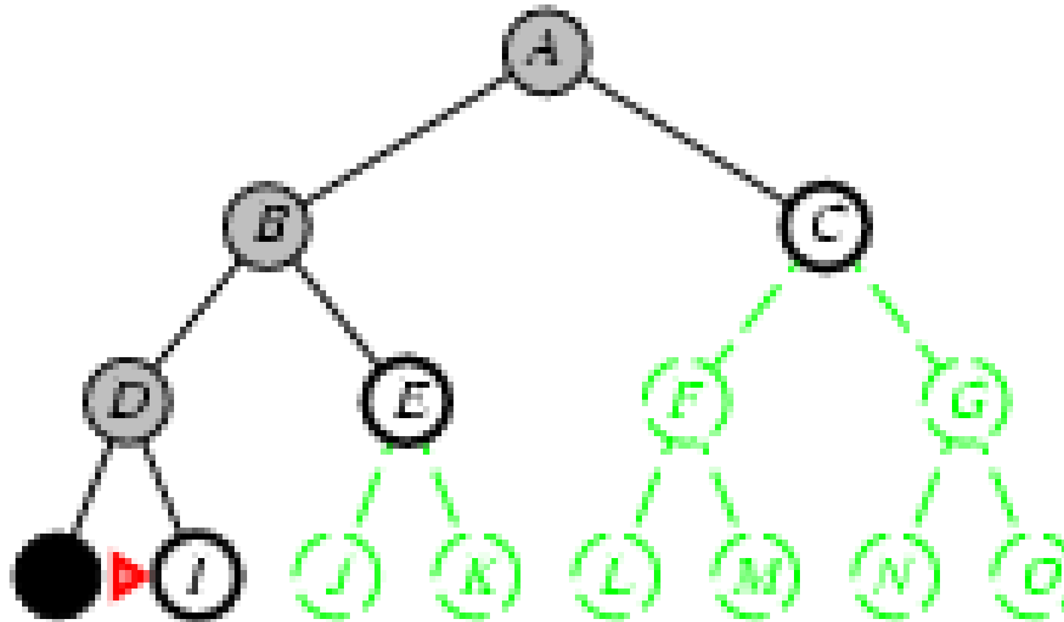
深さ優先探索

- 最も深い拡張されていないノードを拡張
- 実現:
  - フリンジ = LIFOキュー、後継は先頭に



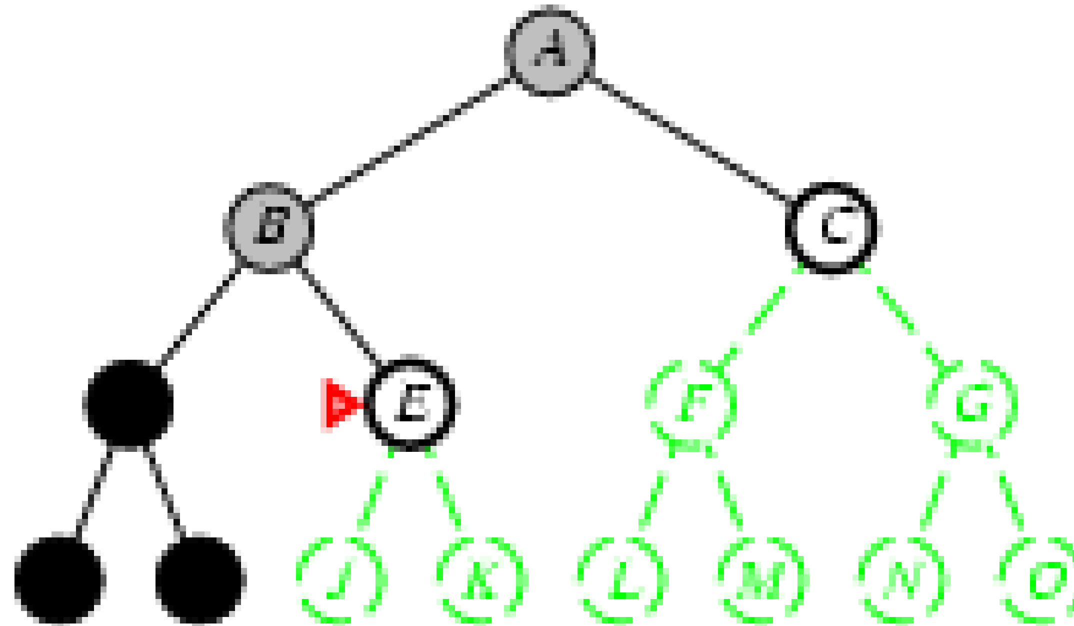
深さ優先探索

- 最も深い拡張されていないノードを拡張
- 実現:
  - フリンジ = LIFOキュー、後継は先頭に



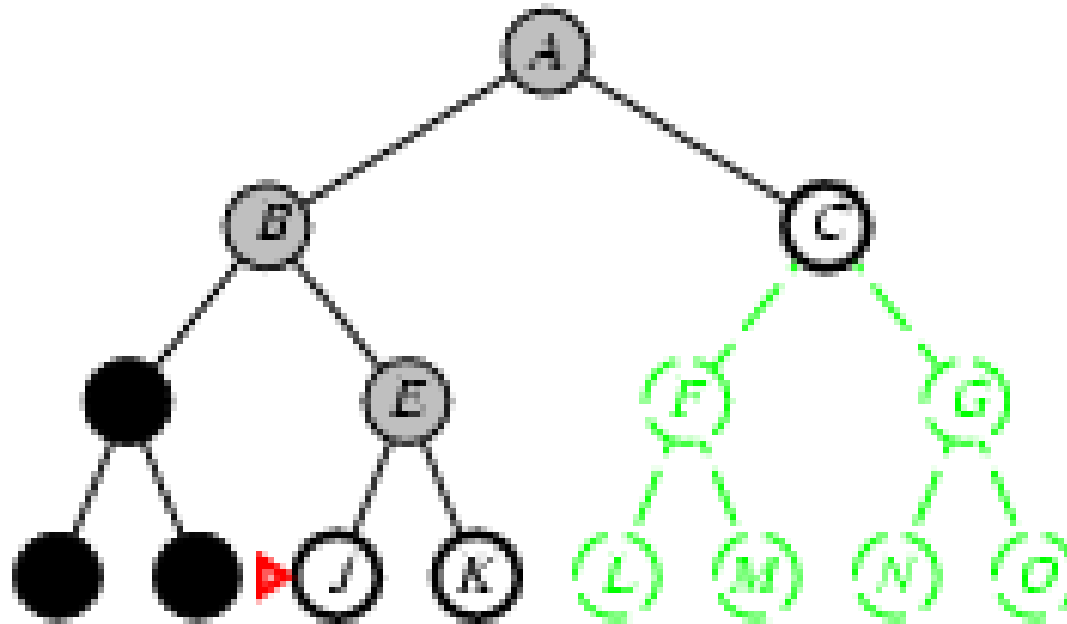
深さ優先探索

- 最も深い拡張されていないノードを拡張
- 実現:
  - フリンジ = LIFOキュー、後継は先頭に



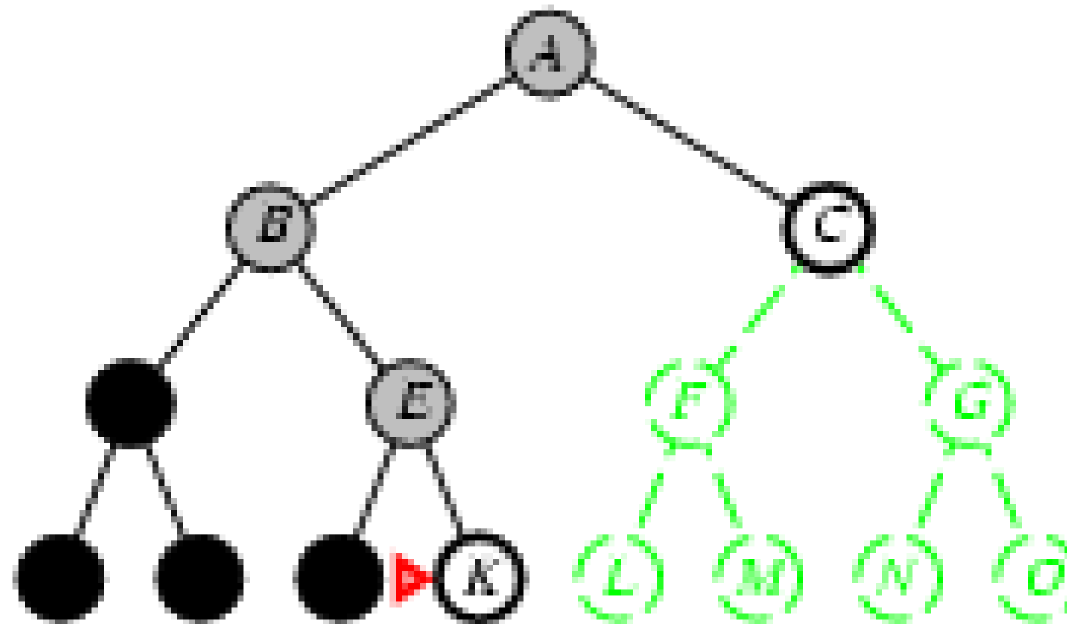
深さ優先探索

- 最も深い拡張されていないノードを拡張
- 実現:
  - フリンジ = LIFOキュー、後継は先頭に



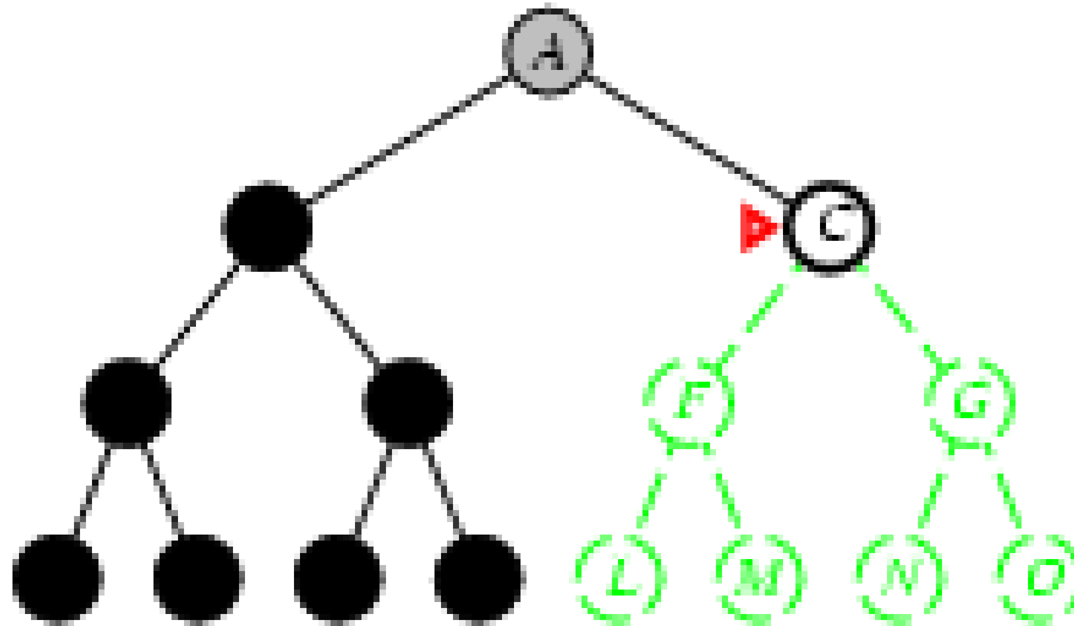
深さ優先探索

- 最も深い拡張されていないノードを拡張
- 実現:
  - フリンジ = LIFOキュー、後継は先頭に



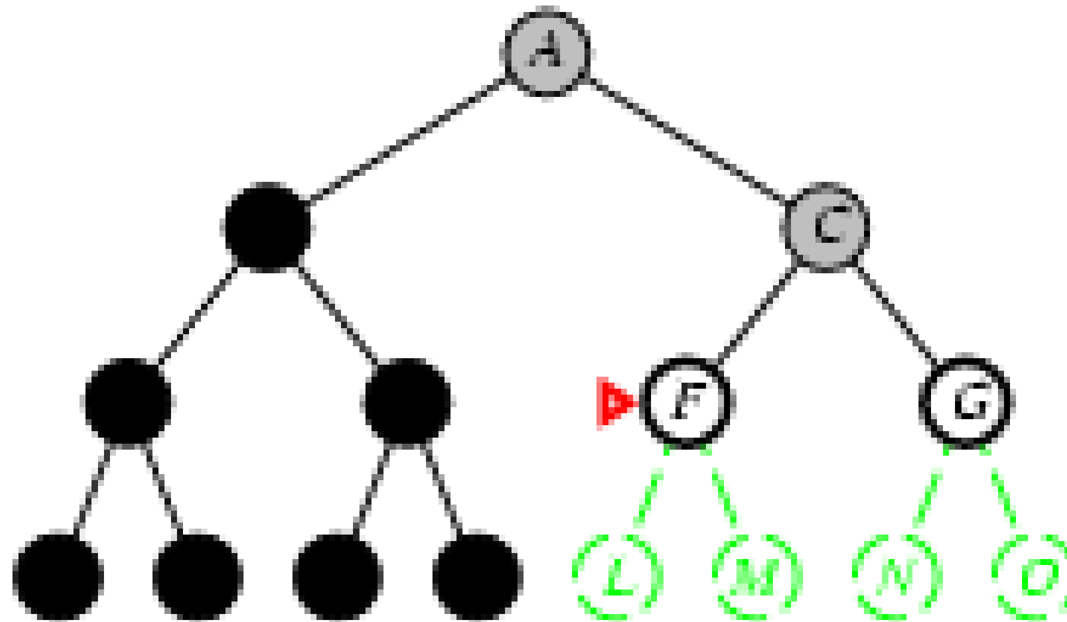
深さ優先探索

- 最も深い拡張されていないノードを拡張
- 実現:
  - フリンジ = LIFOキュー、後継は先頭に



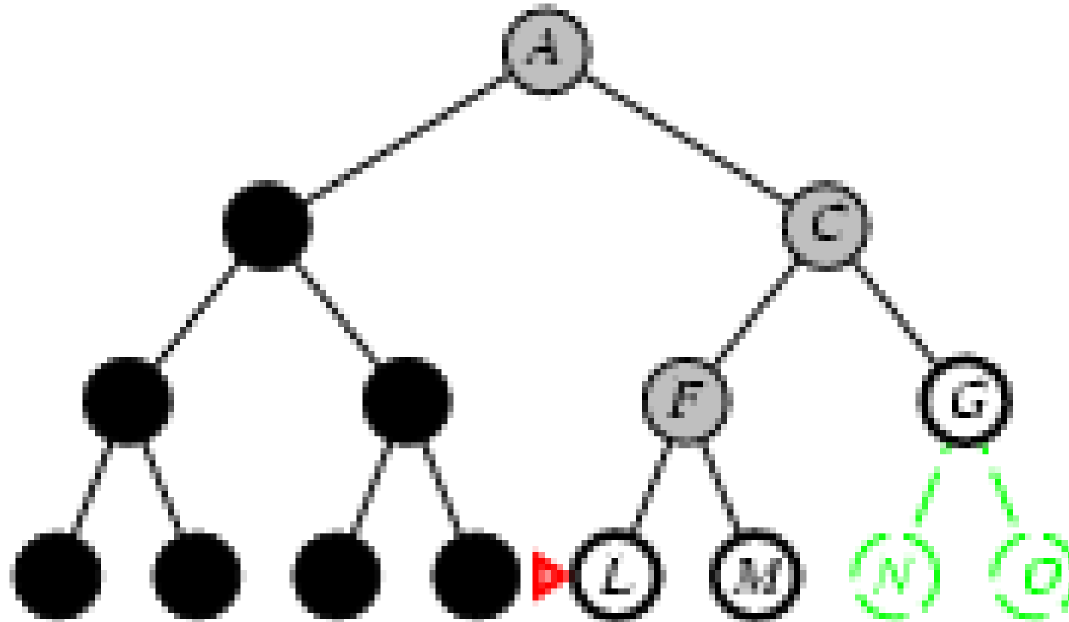
深さ優先探索

- 最も深い拡張されていないノードを拡張
- 実現:
  - フリンジ = LIFOキュー、後継は先頭に



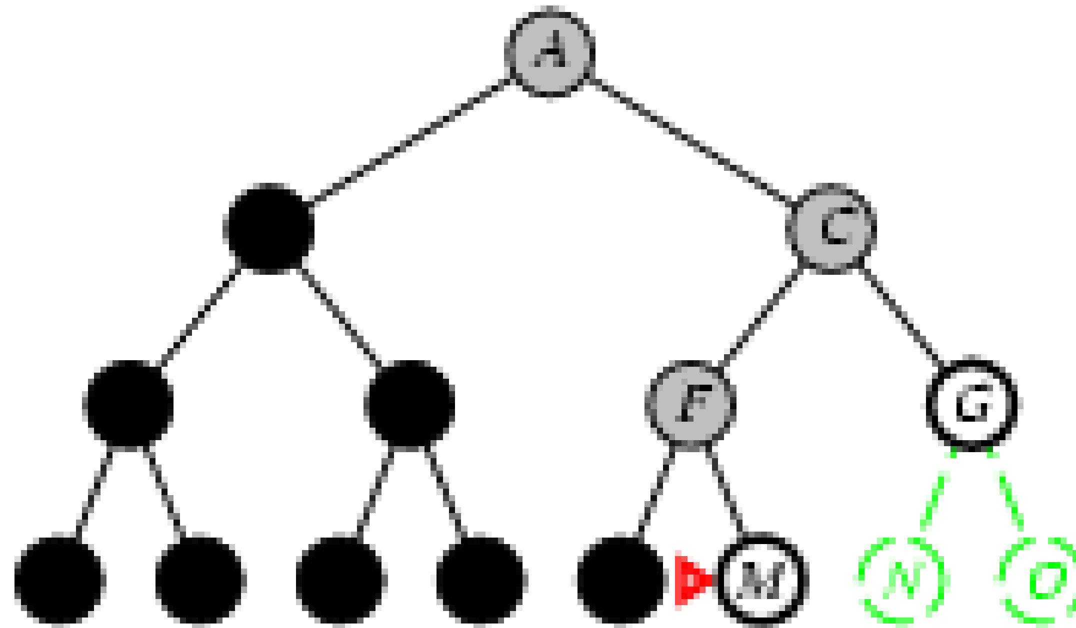
深さ優先探索

- 最も深い拡張されていないノードを拡張
- 実現:
  - フリンジ = LIFOキュー、後継は先頭に



深さ優先探索

- 最も深い拡張されていないノードを拡張
- 実現:
  - フリンジ = LIFOキュー、後継は先頭に



深さ優先探索の性質

- Complete? No: 無限の深さの空間、ループのある空間では失敗
  - 同じ状態を繰り返さないようにする
    - → 有限の空間では完全
- Time? $O(b^m)$: mがdより大きい場合は悲劇的
  - 解が密集しているなら幅優先探索よりずっと早い可能性がある
- Space? $O(bm)$、線形
- Optimal? No

深さ制限探索

= 深さ制限lを有する深さ優先探索,
即ち、深さlのノードは後継を持たない

- 再帰での実現:

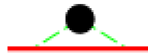
```
function DEPTH-LIMITED-SEARCH( problem, limit) returns soln/fail/cutoff
   RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns soln/fail/cutoff
   cutoff-occurred? ← false
   if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
   else if DEPTH[node] = limit then return cutoff
   else for each successor in EXPAND(node, problem) do
       result ← RECURSIVE-DLS(successor, problem, limit)
       if result = cutoff then cutoff-occurred? ← true
       else if result ≠ failure then return result
   if cutoff-occurred? then return cutoff else return failure
```

反復深化探索

```
function ITERATIVE-DEEPENING-SEARCH( problem) returns a solution, or failure
    inputs: problem, a problem

    for depth ← 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH( problem, depth)
        if result ≠ cutoff then return result
```

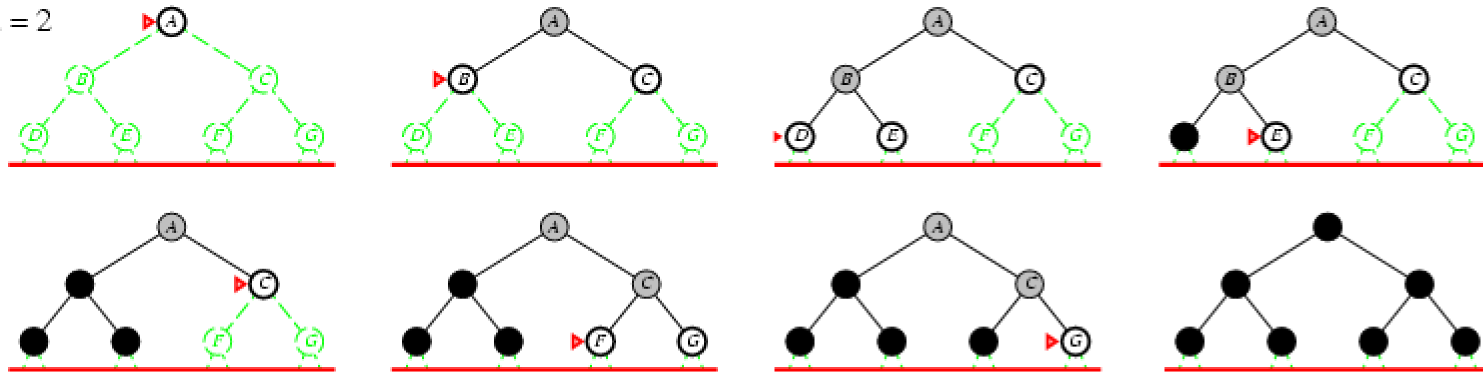
反復深化探索/ =0

Limit = 0

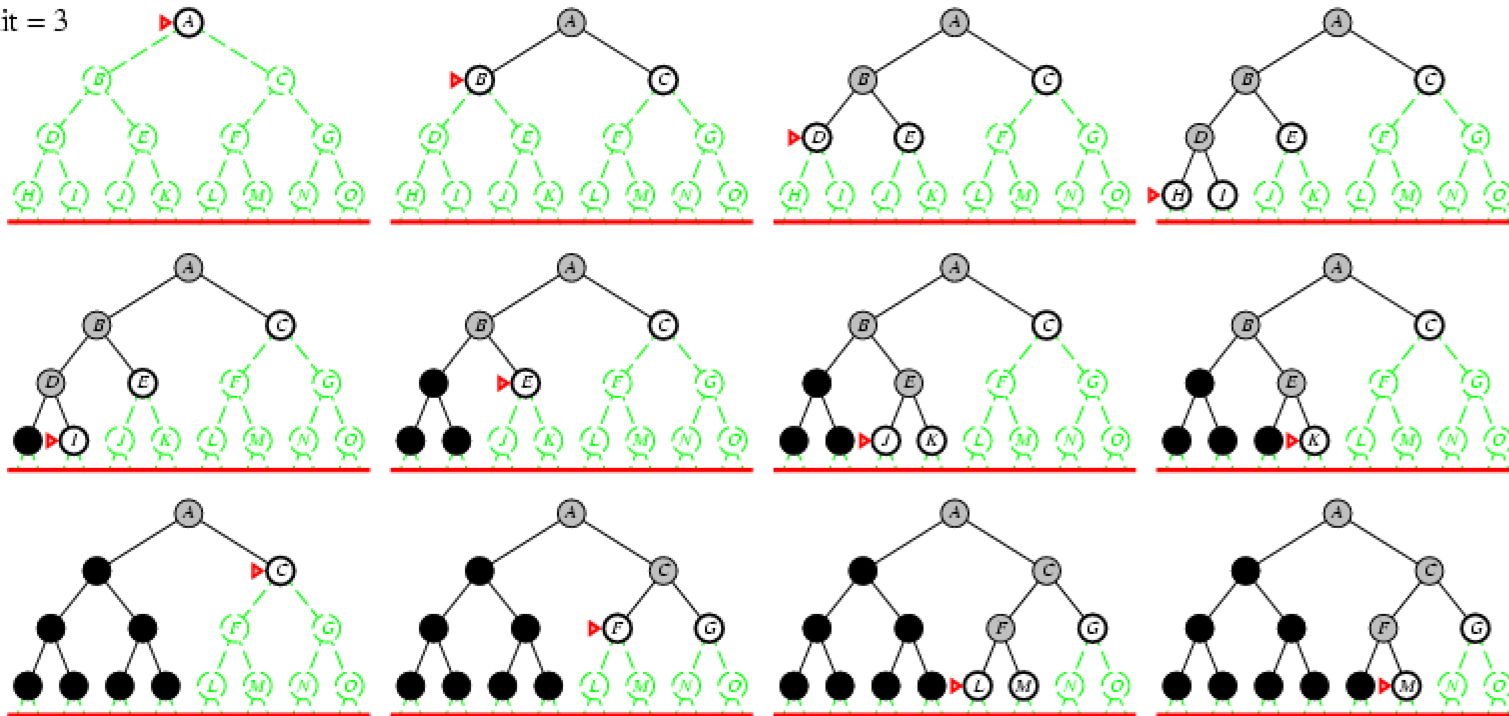


反復深化探索/ =1

反復深化探索/ =2



反復深化探索/ =3



反復深化探索

- 分岐数bでの深さdへの深さ制限探索で生成されるノードの数:

$$N_{DLS} = b^0 + b^1 + b^2 + ... + b^{d-2} + b^{d-1} + b^d$$

- 分岐数bでの深さdへの反復深化探索で生成されるノードの数:

$$N_{IDS} = (d+1)b^0 + d\ b^{\wedge 1} + (d-1)b^{\wedge 2} + ... + 3b^{d-2} + 2b^{d-1} + 1b^d$$

- For $b = 10$, $d = 5$,
  - N_{DLS} = 1 + 10 + 100 + 1,000 + 10,000 + 100,000 = 111,111
  - N_{IDS} = 6 + 50 + 400 + 3,000 + 20,000 + 100,000 = 123,456
- Overhead = (123,456 - 111,111)/111,111 = 11%

反復深化探索の性質

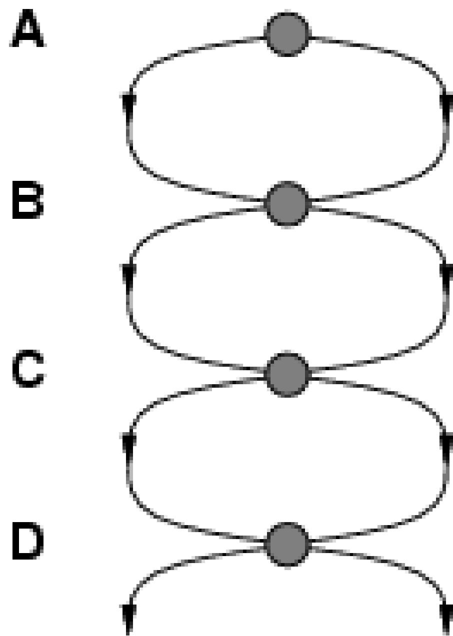
- Complete? Yes
- Time? $(d+1)b^0 + d\,b^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
- Space? $O(bd)$
- Optimal? Yes, ステップコストが1なら

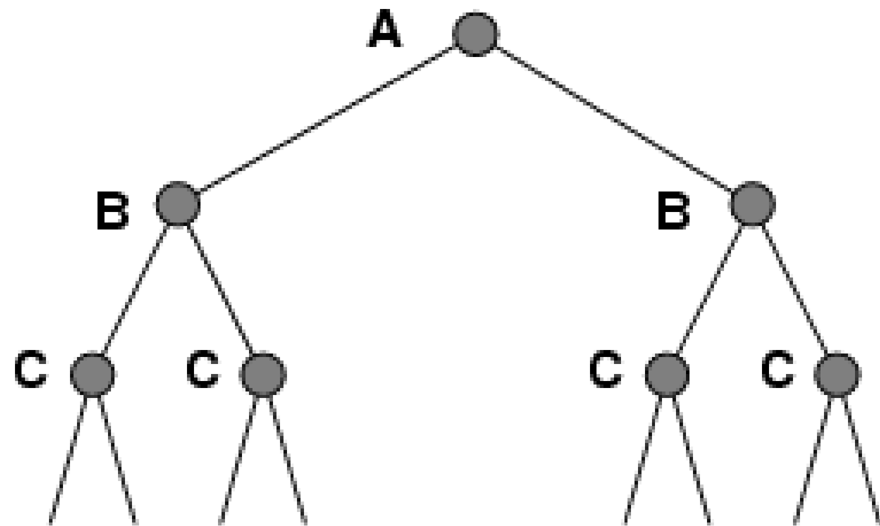
アルゴリズムのまとめ

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes	Yes	No	No	Yes
Time	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(b^m)$	$O(b^l)$	$O(b^d)$
Space	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(bm)$	$O(bl)$	$O(bd)$
Optimal?	Yes	Yes	No	No	Yes

状態の繰り返し

- 状態の繰り返しに失敗すると線形の問題が指数的な問題となる





グラフ探索

```
function GRAPH-SEARCH( problem, fringe) returns a solution, or failure
    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
        if STATE[node] is not in closed then
            add STATE[node] to closed
            fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

まとめ

- 問題の定式化に当たっては、状態空間を都合よく探索できるようにするため、現実世界での詳細は抽象化される
- 多様な一様探索戦略がある
- 反復深化探索は線形空間のときのみ使われ、他の一様でないアルゴリズムよりは時間がかからない