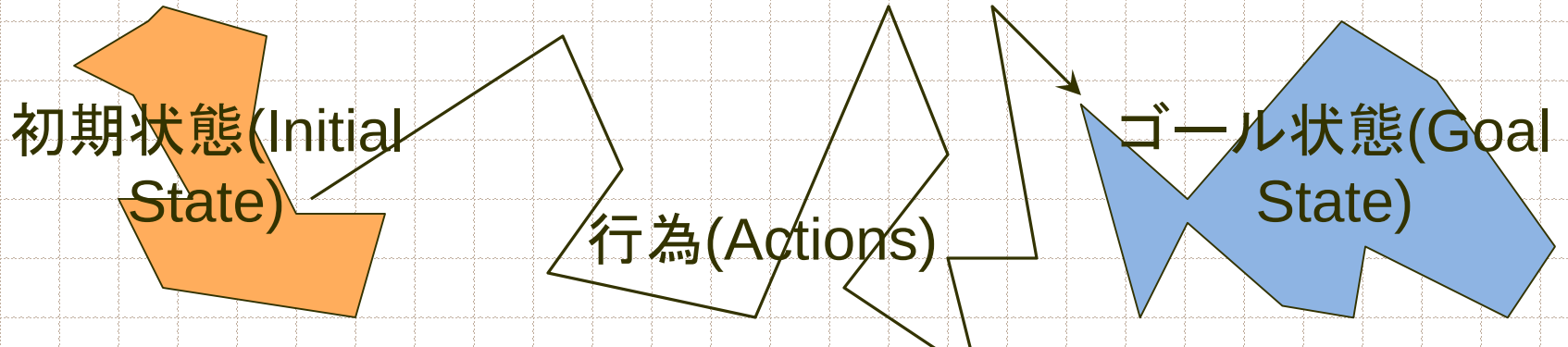
人工知能基礎
第7回

探索による問題解決(1)

ソフトウェア情報学部
David Ramamonjisoa

ゴールを用いるエージェントの構築

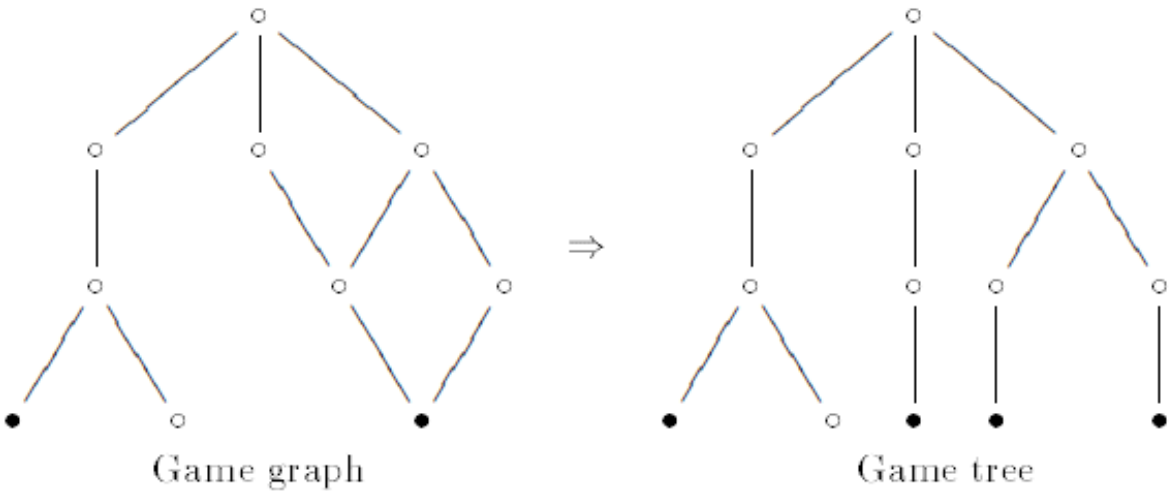
- エージェントの環境を状態で表現する(**state**)?
- エージェントのゴールは何か(**goal** to be achieved?)
- 可能な行為は何かある(What are the **actions**?)
- 問題を解決するためにどのような情報が状態と状態遷移に記述するべきか



状態空間とグラフ

- 与えれた問題の状態空間による表現は、グラフ概念と次のように対応づける

状態空間	グラフ
状態 初期状態、目標状態	節点（ノード） 初期ノード、目標ノード
変換、操作（オペレータ）	枝、エッジ

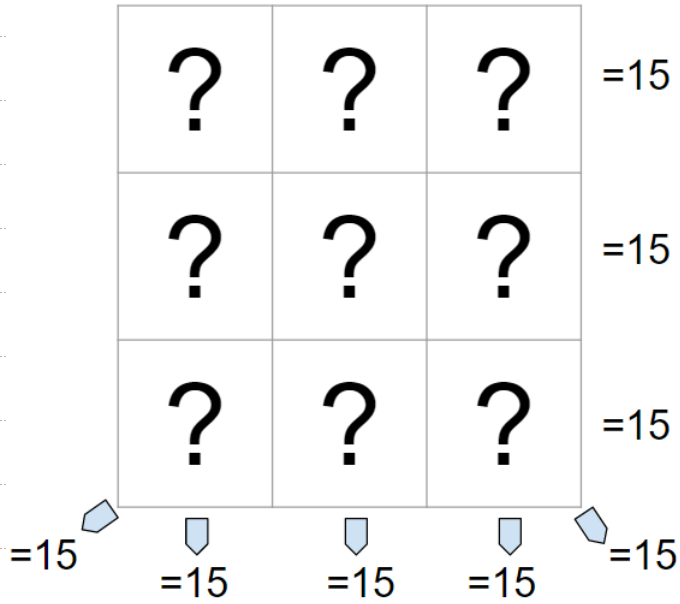


全数探索(exhaustive search)

- 問題： 3x3の表を1から9の9つの数字で理めよ。ただし、各行、各列、対角線上のマスの和は等しくなければならない。

問題:魔方陣

?	?	?
?	?	?
?	?	?



定式化：

- 初期状態：空表
- ゴール状態：すべての数の各行、各列、対角線の和を等しくなる
- 行為：一つの数をマスに置く
- ゴール検査：条件を満たしているかどうか
- コスト：1行為は1コスト
- この問題は9!=362,880通りの状態空間になる
- 全ての配置を生成し、それぞれについてゴール検査し、探索を行う。

探索法

問題の状態空間を有向グラフで表現する。

探索木を生成しながら、目標節点への順路をより効率的に探し求める

1. 初期状態を出発ノード
2. 隣接するノード（候補リスト）が空であれば、終了する
3. 候補リストが空で無い場合、その中から次に調べるノードを選ぶ
4. ノードが目標ノードかどうか
5. そうでないとき、そのノードを展開し、候補リストから除去する
6. 2.へ戻る

回答例:プログラムと出力

```
def print_solutions(solutions):
    for s in solutions:
        print("¥n")
        print('|' + '|' . join ( map ( str , s[: 3 ])) + '|')
        print('|' + '|' . join ( map ( str , s[3: 6 ])) + '|')
        print('|' + '|' . join ( map ( str , s[6:])) + '|')

solutions = [g for g in grids if _all_sums(g)]

print("solutions = ")
print_solutions(solutions)
```

```
magic_num =  15
solutions =


|2|7|6|
|9|5|1|
|4|3|8|


|2|9|4|
|7|5|3|
|6|1|8|


|4|3|8|
|9|5|1|
|2|7|6|


|4|9|2|
|3|5|7|
|8|1|6|
```

水差し問題(Water Jug Problem)

<具体例1>　水差し問題

4ガロンと3ガロンの二つの水差しと，水を入れるためのポンプが与えられています．4ガロンの水差しに正確に2ガロンの水を入れるにはどうしたらよいでしょうか．もちろん，水差しに目盛は付いていません．

状態遷移関数(状態)→オペレータ

水差し1＝4ガロン、水差し2＝3ガロンの例では、以下のルールである。

R1：4ガロンの水差しが満杯でないなら，それを満杯にする．

R2：3ガロンの水差しが満杯でないなら，それを満杯にする．

R3：4ガロンの水差しに少しでも水が入っているならそれを空にする．

R4：3ガロンの水差しに少しでも水が入っているならそれを空にする．

R5：3ガロンの水差しの水を移して4ガロンの水差しを満杯にする．

R6：4ガロンの水差しの水を移して3ガロンの水差しを満杯にする．

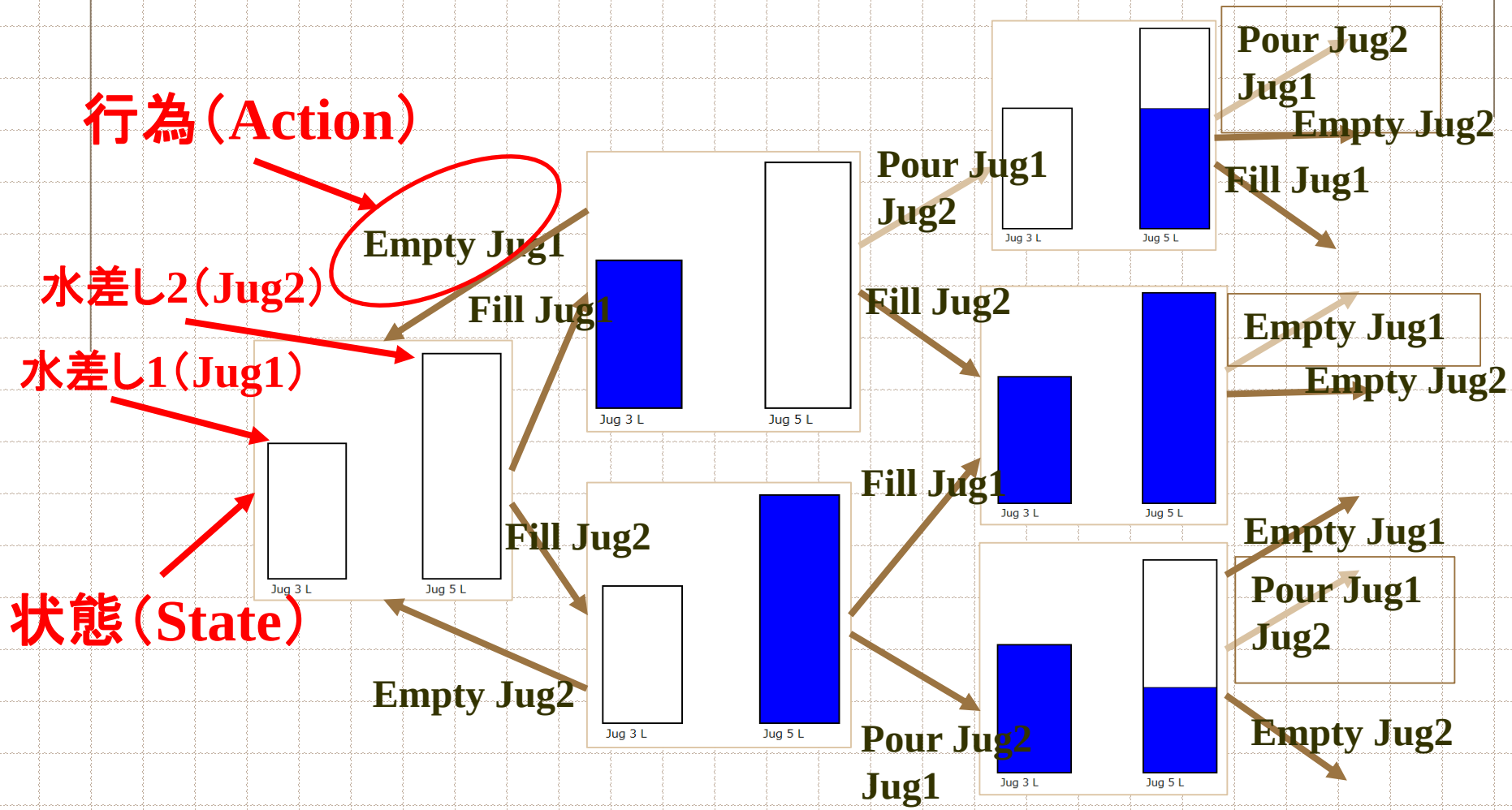
R7：3ガロンの水差しの水を全て4ガロンの水差しに移す．

R8：4ガロンの水差しの水を全て3ガロンの水差しに移す．

オペレータ（行為）

- Fill Jug 1: 水差し1を満杯する
- Fill Jug 2: 水差し2を満杯する
- Empty Jug 1: 水差し1を空にする
- Empty Jug 2: 水差し2を空にする
- Pour Jug1 to Jug2: 水差し1の水を水差し2に満杯にする
- Pour Jug2 to Jug1: 水差し2の水を水差し1に満杯にする

水差し問題(Water Jug Problem)の状態空間(グラフ):



水差し問題の探索グラフ

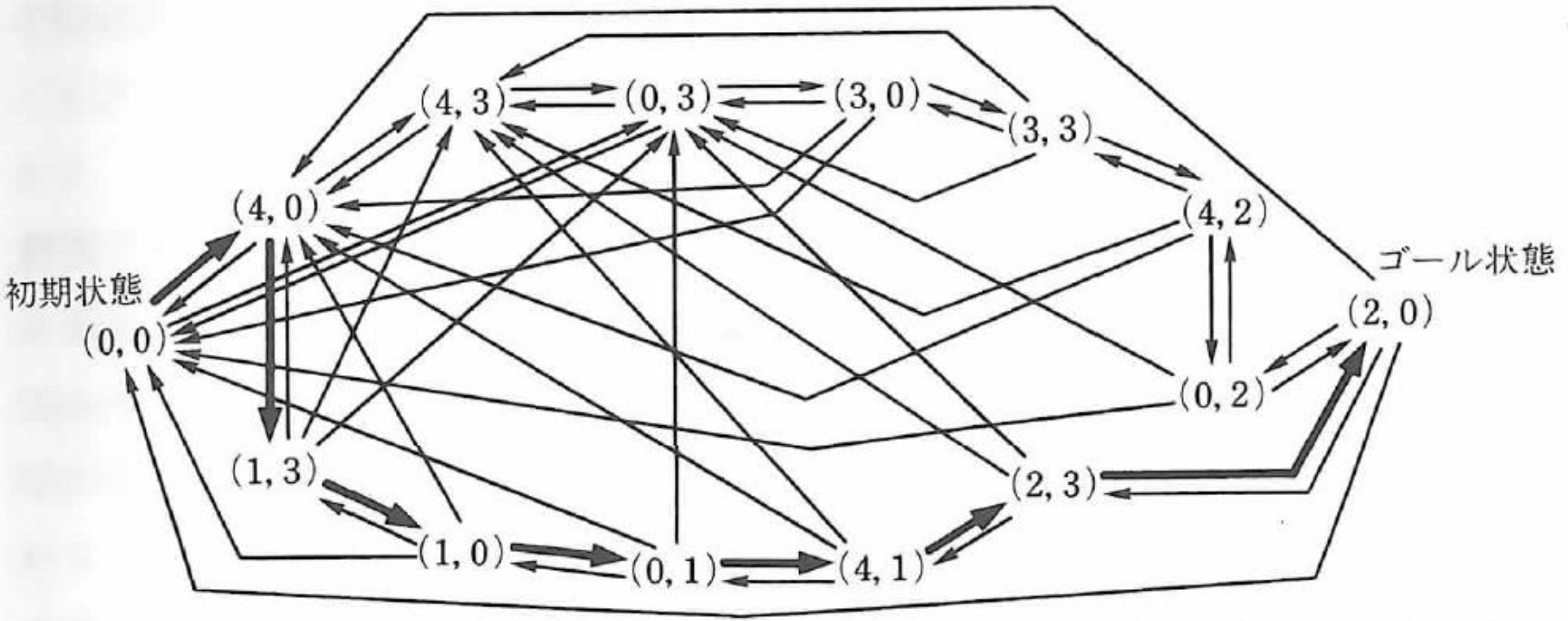


図 2.3　水差し問題の探索グラフ

水差し問題の探索木

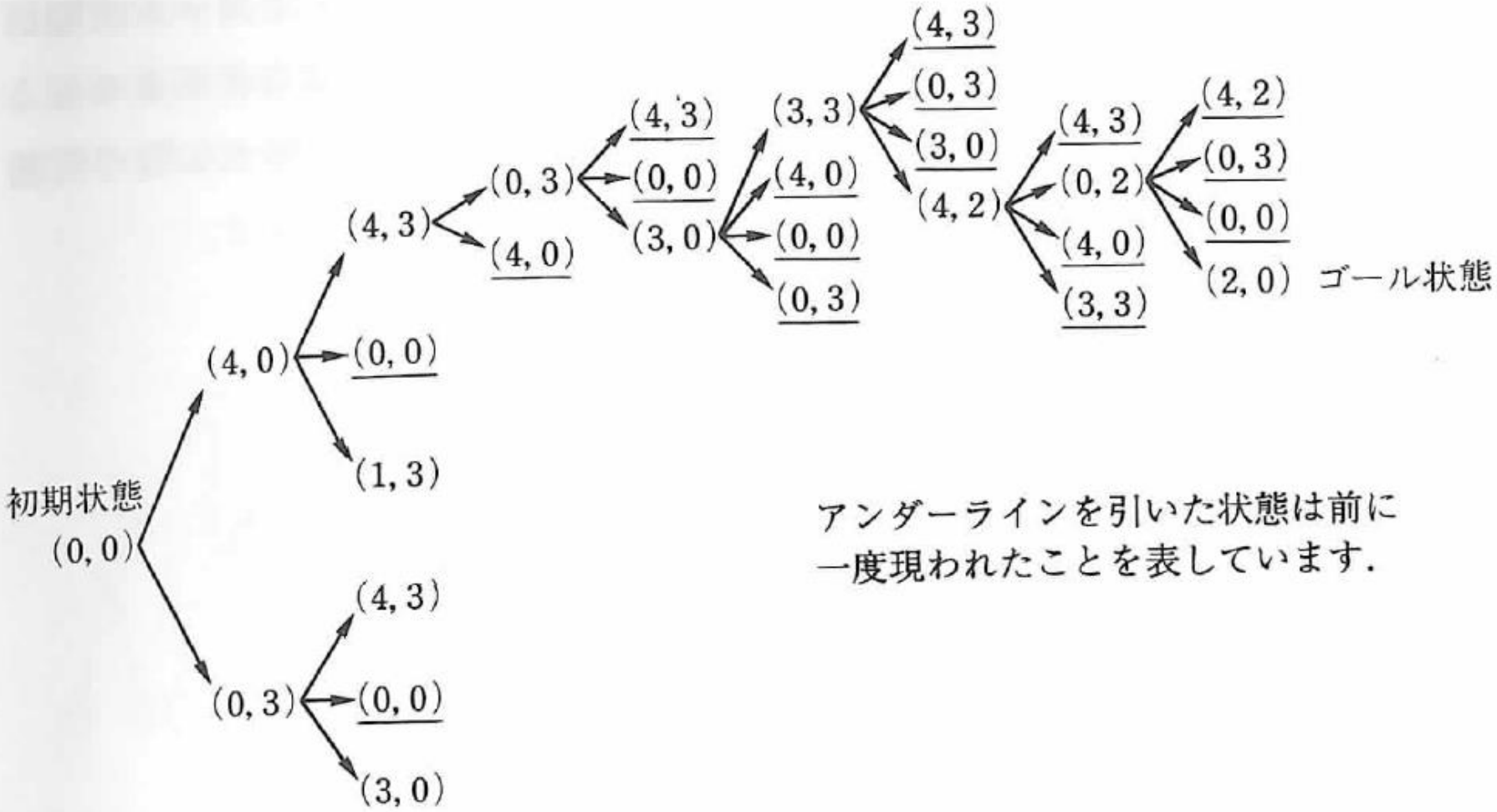


図 2.2　水差し問題の探索木（一部分）

水差し問題の解

- オペレータのリスト
  - 初期状態からゴール状態までの操作
- 経路またはパス(枝のリスト)
- (4L,3L)の解例
- [Fill Jug 2, Pour Jug2 to Jug1, Fill Jug 2, Pour Jug2 to Jug1, Empty Jug 1, Pour Jug2 to Jug1]
- [(0,0)->(0,3), (0,3)->(3,0), (3,0)->(3,3), (3,3)->(4,2), (4,2)->(0,2), (0,2)->(2,0)]

探索戦略

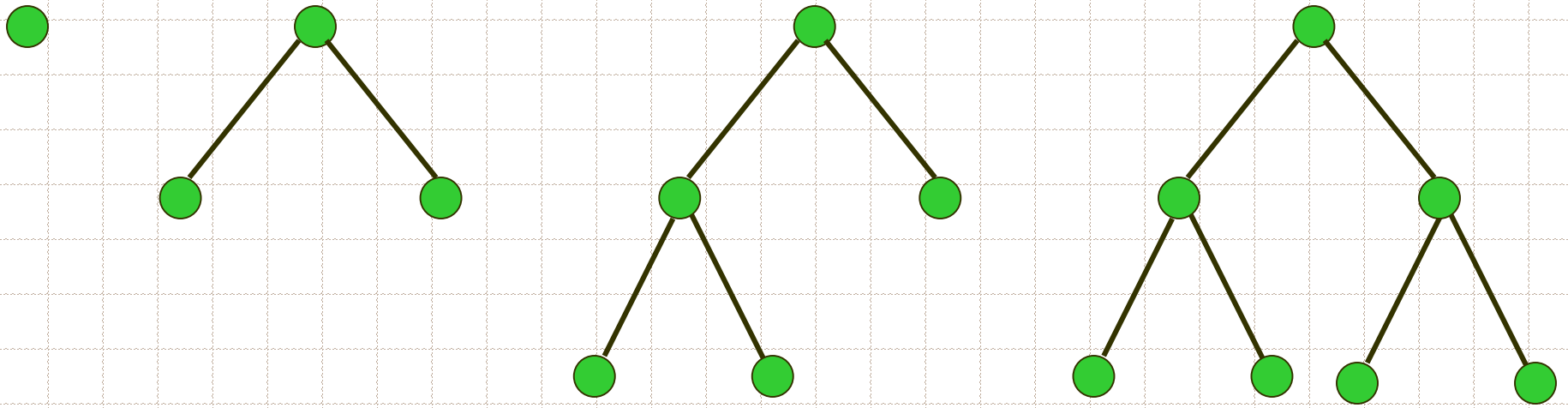
- 探索戦略選択のための四つの基準
  - 完全性： 解が存在するとき，それを見つけることが保証されているか？
  - 最適性： いくつか異なる解があるとき，戦略は最も良い解を見つけるか？
  - 時間計算量： 解を見つけるまでにどれくらい時間が掛かるか？
  - 空間計算量： 探索を行うためにどのくらいメモリを必要とするか？
- 情報のない探索(uniformed search)，ある探索
  - 現在の状態からゴールに至るステップ数や経路コストに関する情報を持っていないか，持っているかによって，区別する．ここでは，情報のない探索（**盲目的探索**ともいう）を取り上げる．

情報のない探索戦略(Uninformed search strategies)

- 情報のない探索戦略は問題定式化の使用可能情報のみ利用する。
- 幅優先探索(Breadth-first search)
- 均一コスト探索(Uniform-cost search)
- 深さ優先探索(Depth-first search)
- 深さ制限探索(Depth-limited search)
- 反復深化探索(Iterative deepening search)

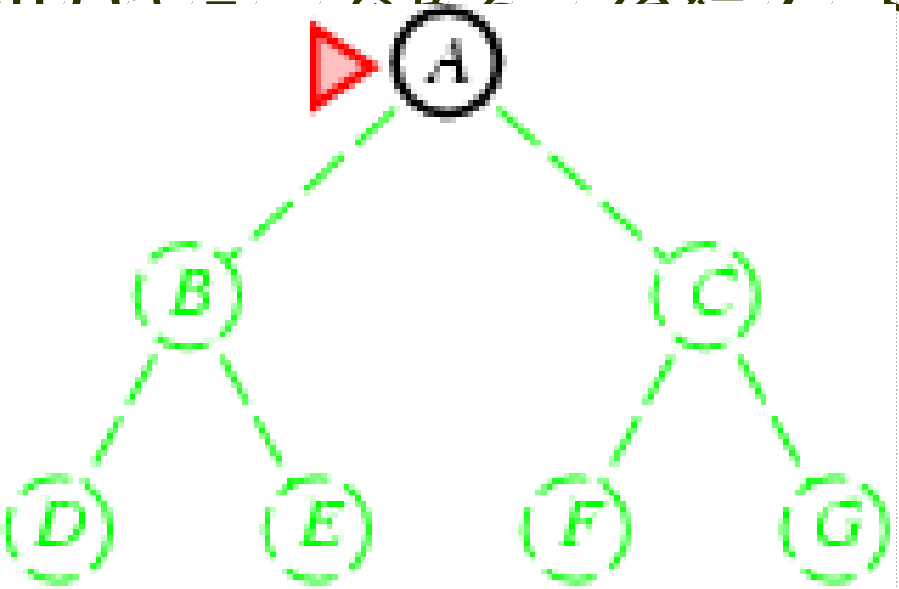
幅優先探索

- ルートノードがまず展開され，つぎにルートノードによって生成されたすべてのノードが展開され，そしてそれらの後続ノードが展開される，というように続く.
- 一般に，探索木における深さdのすべてのノードが，深さd+1のノードの前に展開される.



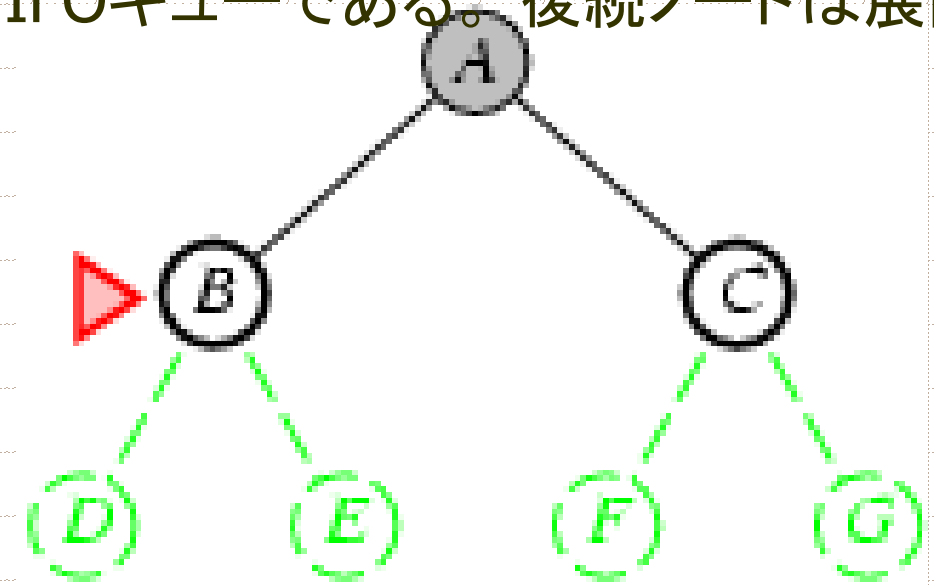
幅優先探索(Breadth-first search)

- 展開されてない浅いノードを展開する
-
- Implementation:
  - 縁はFIFOキューである。後続ノードは展開される
  -



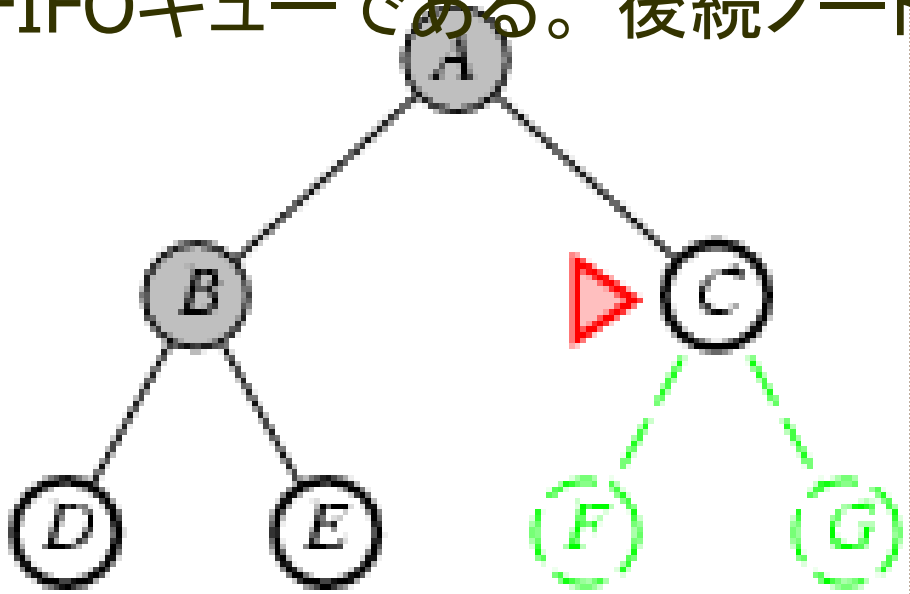
幅優先探索(Breadth-first search)

- 展開されてない浅いノードを展開する
-
- Implementation:
  - *縁* はFIFOキューである。後続ノードは展開される
  -



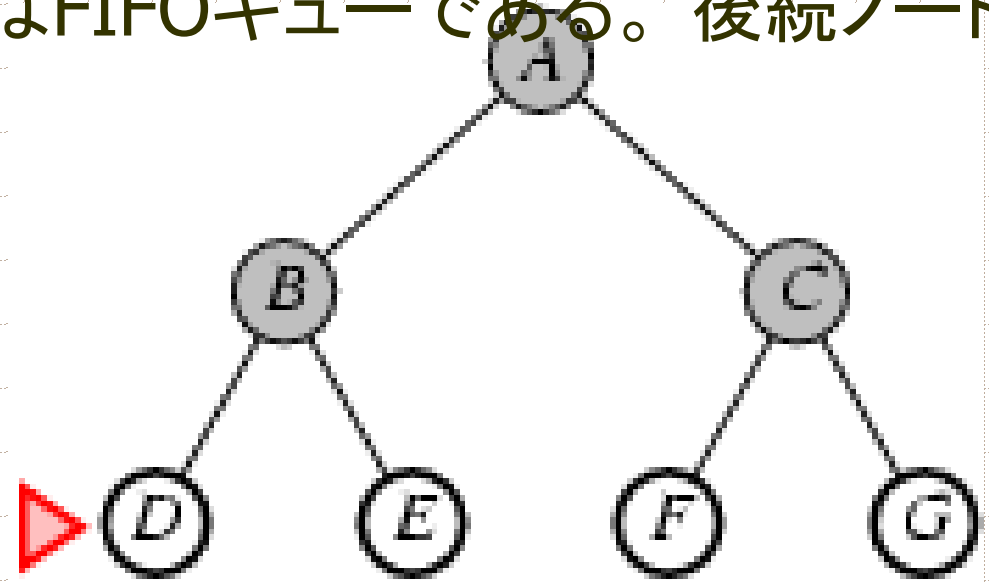
幅優先探索(Breadth-first search)

- 展開されてない浅いノードを展開する
-
- Implementation:
  - *縁* はFIFOキューである。後続ノードは展開される
  -



幅優先探索(Breadth-first search)

- 展開されてない浅いノードを展開する
-
- Implementation:
  - *縁* はFIFOキューである。後続ノードは展開される
  -



幅優先探索の性質(Properties of breadth-first search)

- 幅優先探索は完全で，経路コストがノードの深さとともに減少しなければ，最適である．
- どの状態もb個の新しい状態に展開されるものとする（分岐度＝b）．この問題に対する解が，経路長dを持つと仮定する．
- 最悪の場合，レベルdの最後のノード以外をすべて展開する（最後のノードは解であり，それは展開されない）．
- 展開されるノード数は，　$1+b+b^2+b^3+\cdots+(b^d-1)=O(b^d)$

- b＝10，
  1000ノード/秒
  100バイト/ノード
  とする．

深さ	ノード	時間		メモリ	
0	1	1	ミリ秒	100	バイト
2	111	0.1	秒	11	キロバイト
4	11,111	11	秒	1	メガバイト
6	10^6	18	分	111	メガバイト
8	10^8	31	時間	11	ギガバイト
10	10^10	128	日	1	テラバイト
12	10^12	35	年	111	テラバイト
14	10^14	3500	年	11,111	テラバイト

幅優先探索の性質(Properties of breadth-first search) [尺度]

- 完全性? Yes (if *b* is finite)
- 時間複雑さ? $1+b+b^2+b^3+\ldots+b^d + b(b^d-1) = O(b^{d+1})$
- 空間複雑さ? $O(b^{d+1})$ (keeps every node in memory)
- 最適? Yes (if cost = 1 per step)
- 計算量よりメモリ量が大問題である。(Space is the bigger problem (more than time))

[横型探索のアルゴリズム]

INITIALIZE :

0. OpenList, ClosedList とも空とする(O(), C())。
1. 開始節点 S を OpenList に入れる(O(S))。

LOOP :

2. OpenList が空ならば、S からゴールに至る径路は存在しない。終了。
3. OpenList の先頭を取りだし、OpenList から除く。それを A とする。
4. A がゴールならば、S からゴールに至る節点の系列を出力して終了。
5. A がゴールでなければ、A を展開し到達可能な節点の集合 P_A を求める。
6. A を ClosedList の最後に入れる。
7. P_A が空ならば 2 へ戻る。(A が葉節点のとき)
8. P_A が空でなければ、その節点を適当な順序で OpenList の末尾に追加する。
   8a. このとき、追加した節点には [A] を添字として付けておく。
   8b. P_A に OpenList あるいは ClosedList の節点と同じ節点があれば、それを P_A から除いておく。

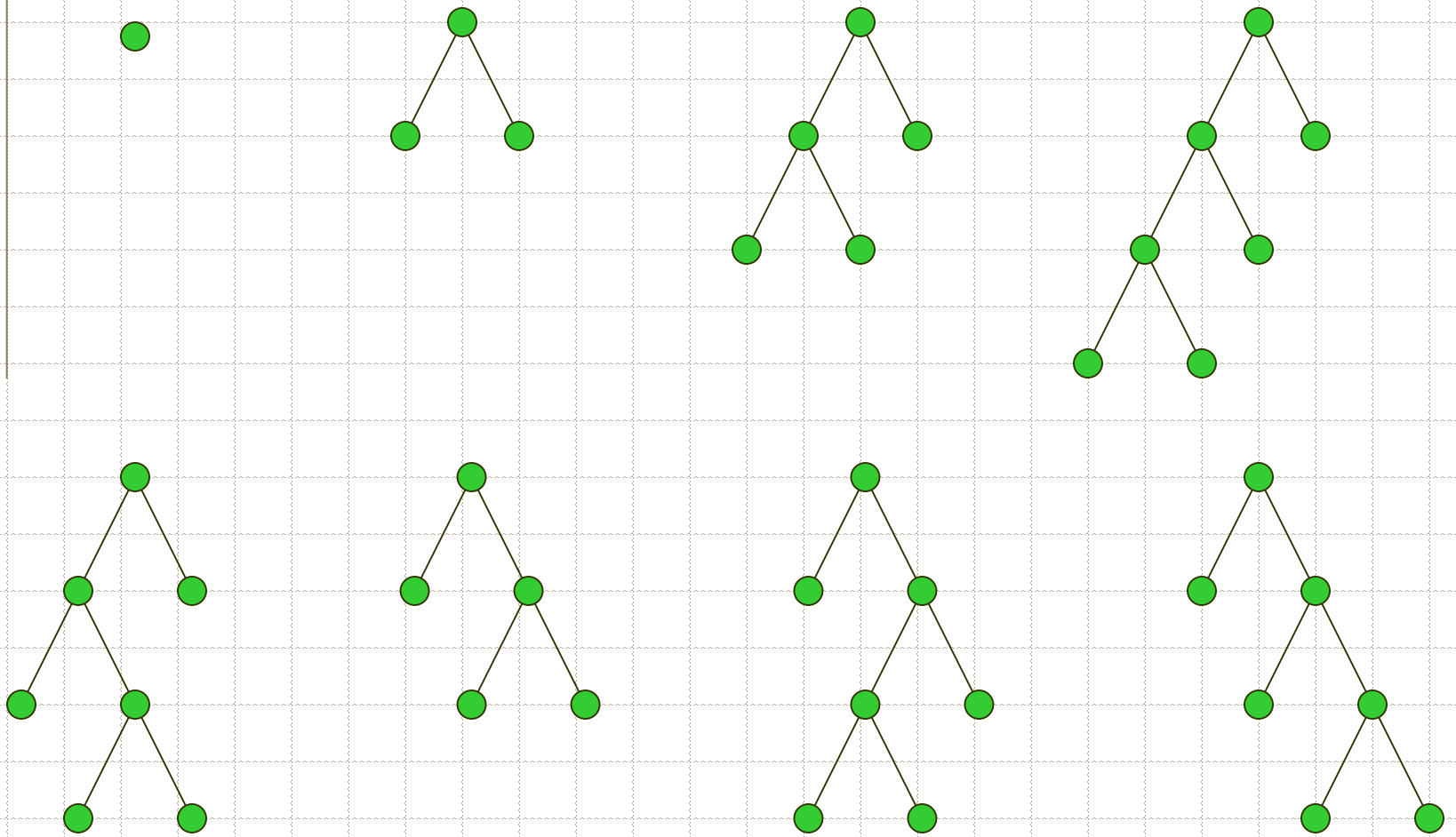
LOOP_END : 2 に戻る。

水差し問題の幅優先探索(BFS)プログラム実行例

```
>>>
WJ((4, 3),(0, 0),(2, 0))
BFS:
(0, 0)
<Node (0, 0)>
<utils1.FIFOQueue instance at 0x01AAF530>
('Found goal:', <Node (2, 0)>)
('Cost of the search:', 6)
('Path:', [<Node (0, 0)>, <Node (0, 3)>, <Node (3, 0)>, <Node (3, 3)>
, <Node (4, 2)>, <Node (0, 2)>, <Node (2, 0)>])
```

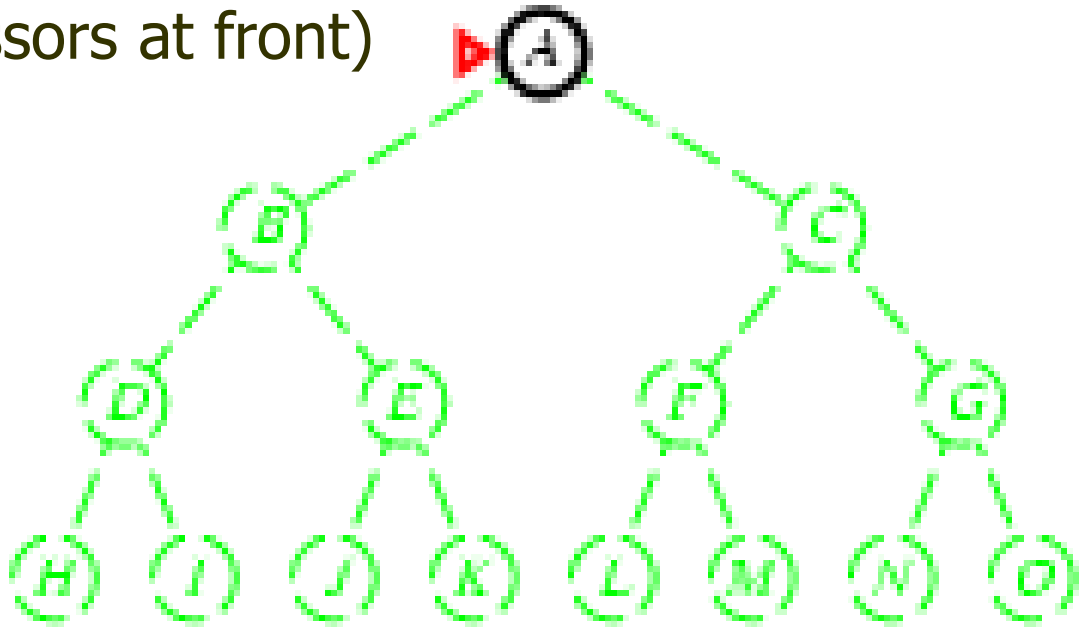
深さ優先探索(Depth first search)

- 深さ優先探索のアルゴリズム
  - 木の最も深いレベルのノードの一つをいつも展開する.
  - 探索が行き止まりになるときに限り，探索は後戻りし，より浅いレベルのノードを展開する.
- 深さ優先探索の性質
  - 分岐度**b**で最大の深さがmの状態空間に対して，bm個のノードだけを格納すればよい．たとえば，深さd=12において，111テラバイトの代わりに，12キロバイトしか必要としない.
  - 時間計算量は，幅優先探索と同様，$O(b^m)$ である.
  - 深さ優先探索は，場合によっては，無限にループして，解が求まらないかもしれない.
  - 深さ優先探索は，完全でも最適でもない.



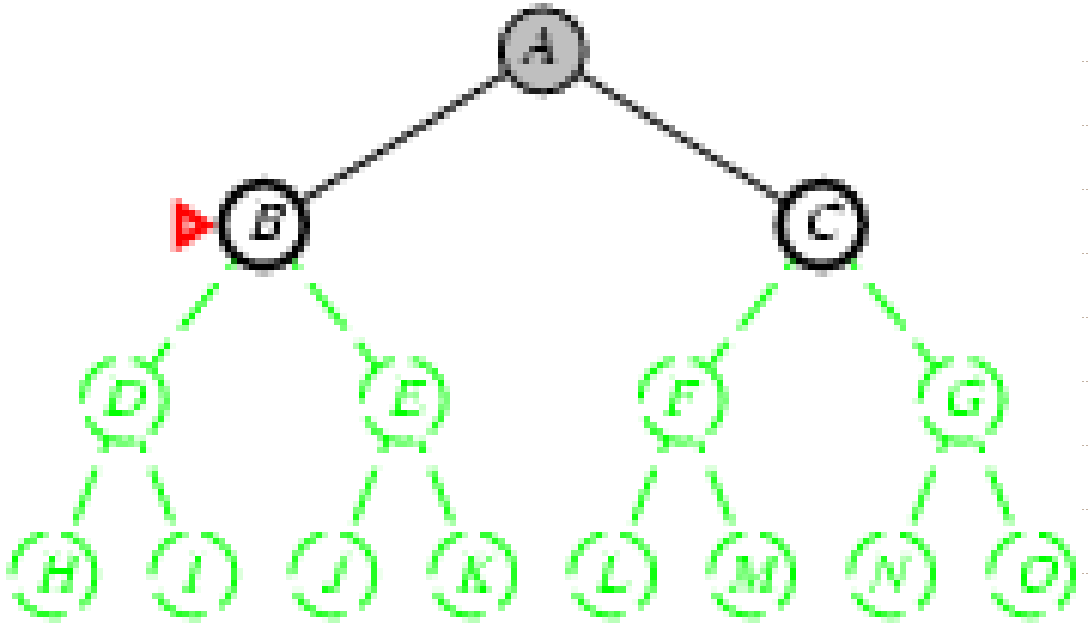
深さ優先探索(Depth first search)

- 木の最も深いレベルのノードの一つをいつも展開する
- 実装(Implementation):
  - *縁(fringe)* = LIFOキュー(LIFO queue, i.e., put successors at front)
  -



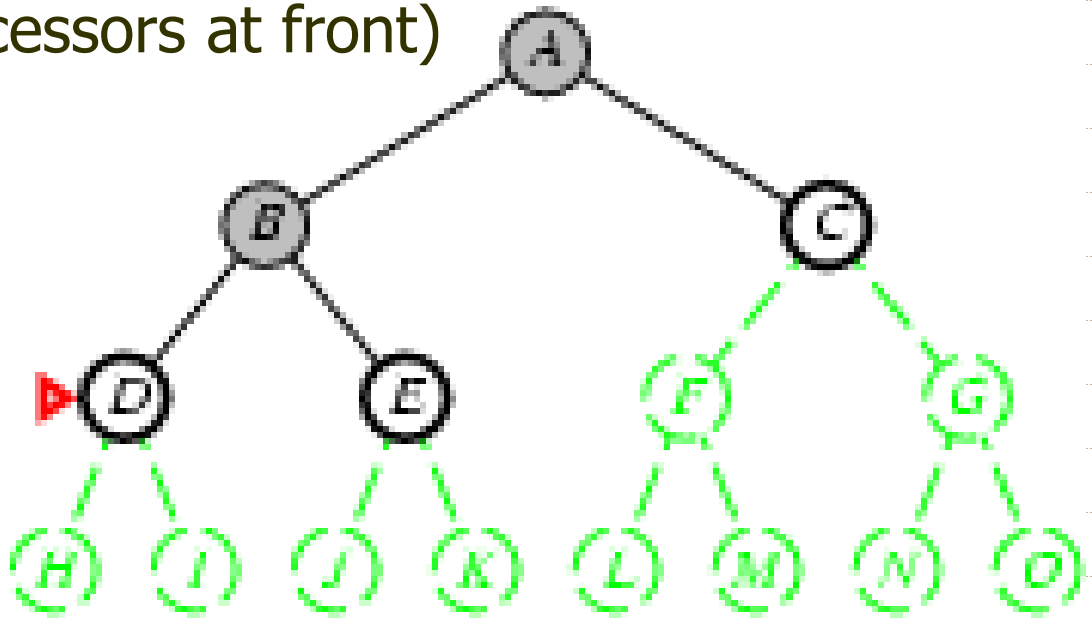
深さ優先探索(Depth first search)

- 木の最も深いレベルのノードの一つをいつも展開する
- 実装(Implementation):
  - *縁(fringe)* = LIFOキュー(LIFO queue, i.e., put successors at front)
  -



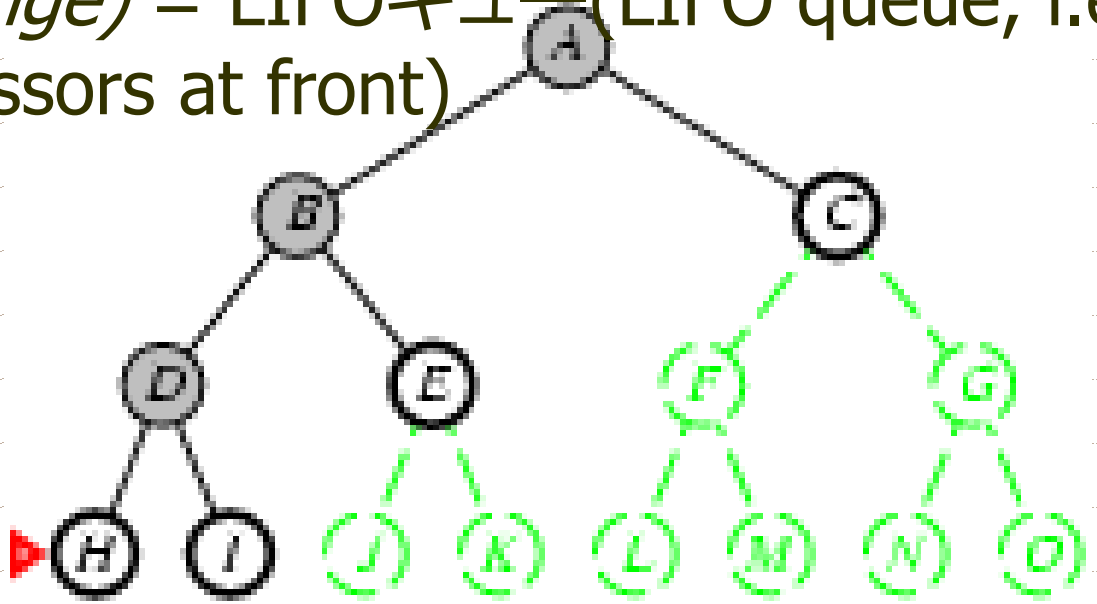
深さ優先探索(Depth first search)

- 木の最も深いレベルのノードの一つをいつも展開する
- 実装(Implementation):
  - *縁(fringe)* = LIFOキュー(LIFO queue, i.e., put successors at front)
  -



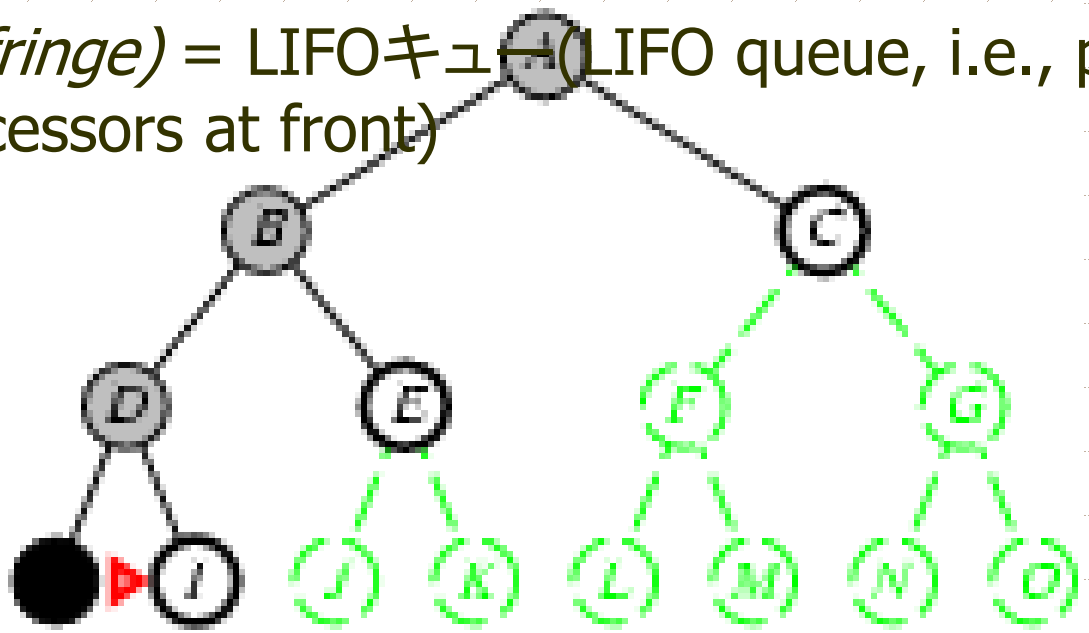
深さ優先探索(Depth first search)

- 木の最も深いレベルのノードの一つをいつも展開する
- 実装(Implementation):
  - *縁(fringe)* = LIFOキュー(LIFO queue, i.e., put successors at front)
  -



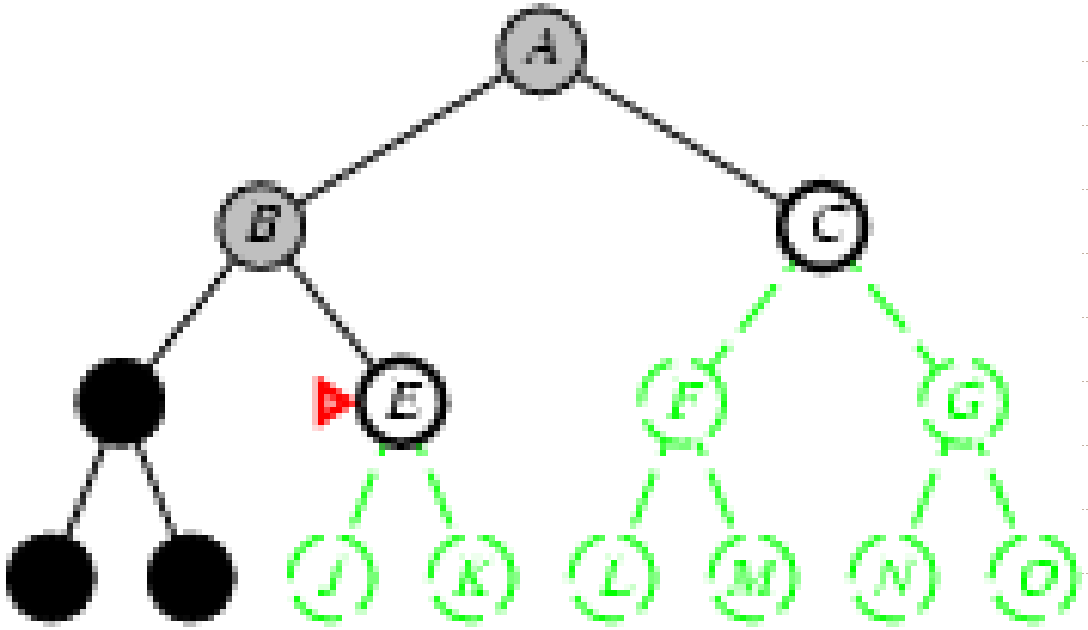
深さ優先探索(Depth first search)

- 木の最も深いレベルのノードの一つをいつも展開する
- 実装(Implementation):
  - *縁(fringe)* = LIFOキュー(LIFO queue, i.e., put successors at front)
  -



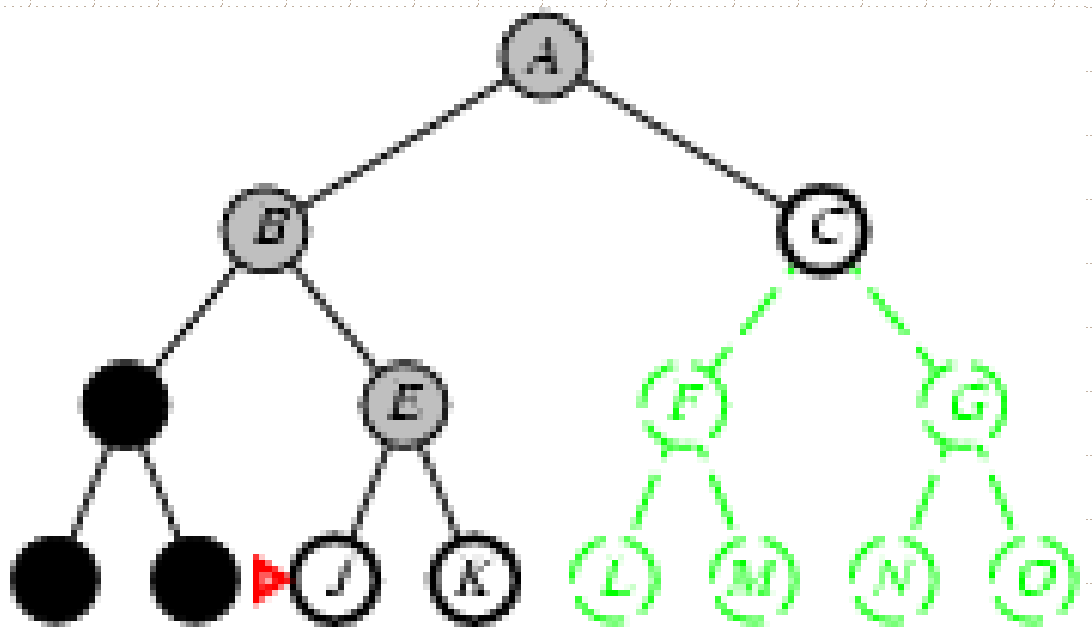
深さ優先探索(Depth first search)

- 木の最も深いレベルのノードの一つをいつも展開する
- 実装(Implementation):
  - *縁(fringe)* = LIFOキュー(LIFO queue, i.e., put successors at front)
  -



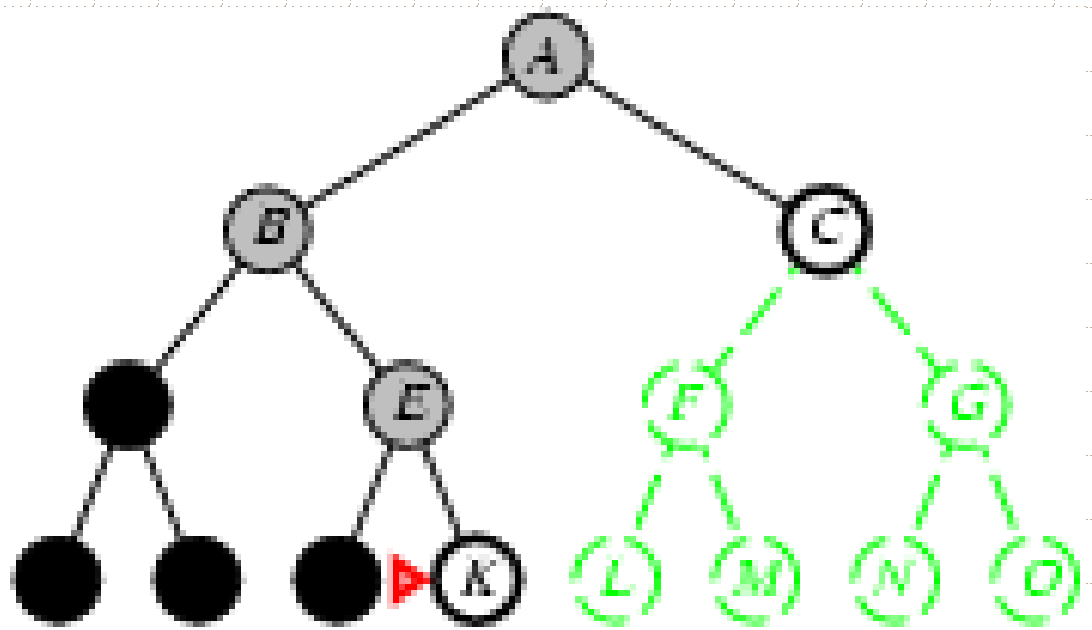
深さ優先探索(Depth first search)

- 木の最も深いレベルのノードの一つをいつも展開する
- 実装(Implementation):
  - *縁(fringe)* = LIFOキュー(LIFO queue, i.e., put successors at front)
  -



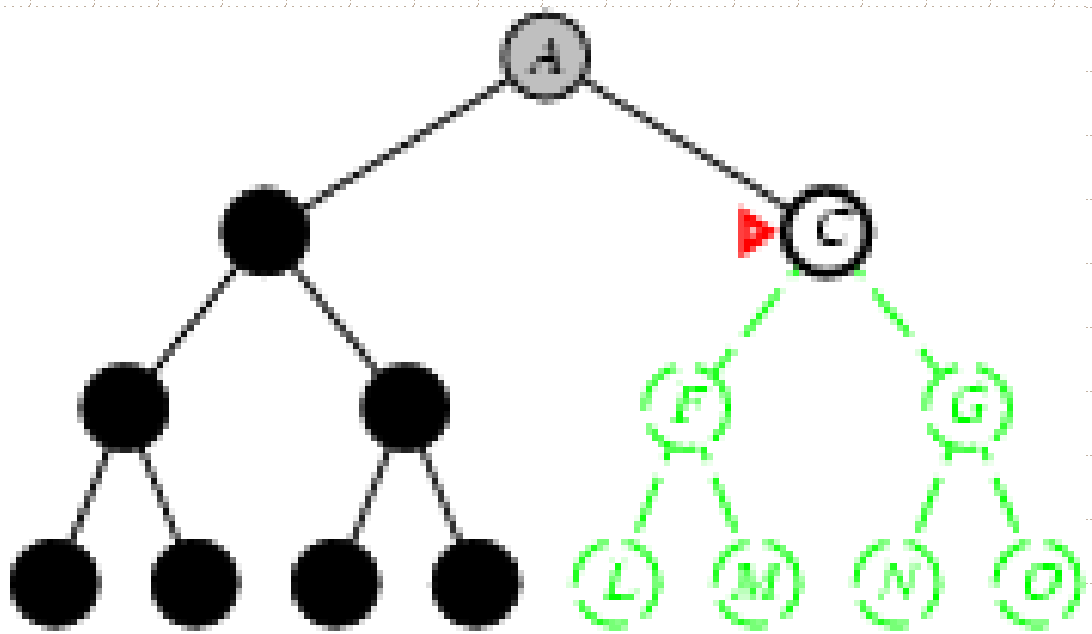
深さ優先探索(Depth first search)

- 木の最も深いレベルのノードの一つをいつも展開する
- 実装(Implementation):
  - *縁(fringe)* = LIFOキュー(LIFO queue, i.e., put successors at front)
  -



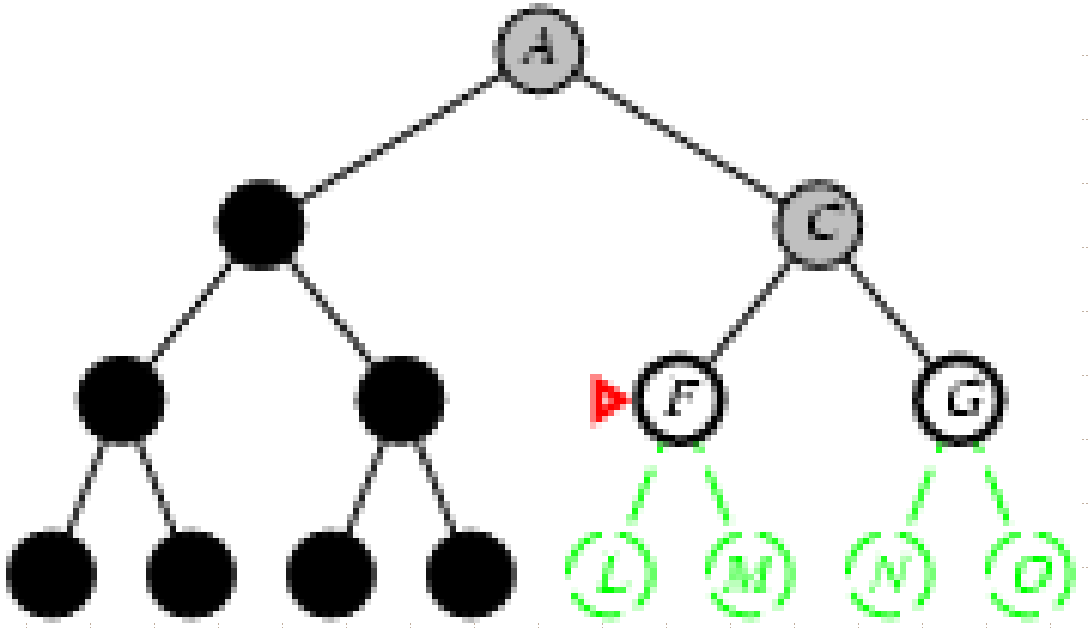
深さ優先探索(Depth first search)

- 木の最も深いレベルのノードの一つをいつも展開する
- 実装(Implementation):
  - *縁(fringe)* = LIFOキュー(LIFO queue, i.e., put successors at front)
  -



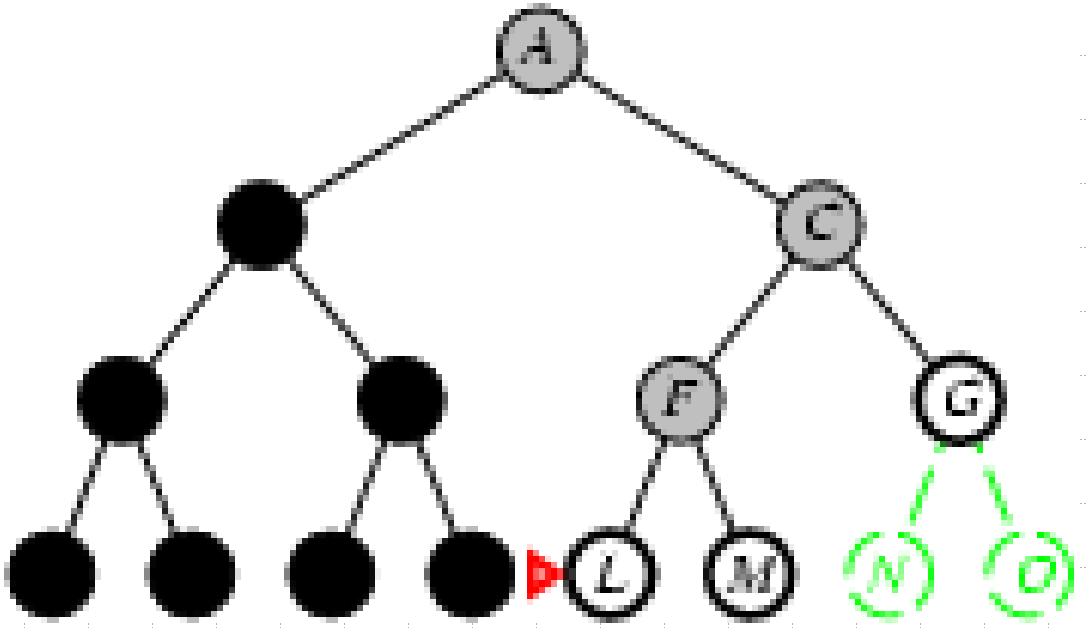
深さ優先探索(Depth first search)

- 木の最も深いレベルのノードの一つをいつも展開する
- 実装(Implementation):
  - *縁(fringe)* = LIFOキュー(LIFO queue, i.e., put successors at front)
  -



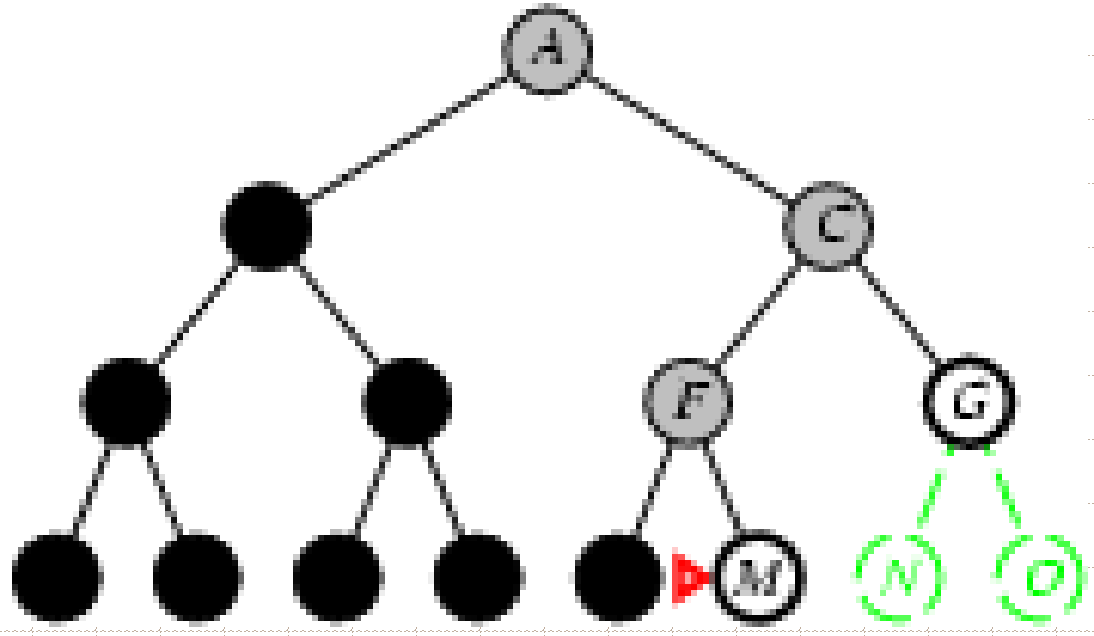
深さ優先探索(Depth first search)

- 木の最も深いレベルのノードの一つをいつも展開する
- 実装(Implementation):
  - *縁(fringe)* = LIFOキュー(LIFO queue, i.e., put successors at front)
  -



深さ優先探索(Depth first search)

- 木の最も深いレベルのノードの一つをいつも展開する
- 実装(Implementation):
  - *縁(fringe)* = LIFOキュー(LIFO queue, i.e., put successors at front)
  -



深さ優先探索の尺度(Properties of depth-first search)

- 完全性? No: fails in infinite-depth spaces, spaces with loops
  - Modify to avoid repeated states along path
    → complete in finite spaces
- 時間? $O(b^m)$: terrible if m is much larger than d
  - but if solutions are dense, may be much faster than breadth-first
- 空間? $O(bm)$, i.e., linear space!
- 最適? No

[縦型探索のアルゴリズム]

INITIALIZE：

0. OpenList，ClosedList とも空とする(O()，C())。

1. 開始節点 S を OpenList に入れる(O(S))。

LOOP：

2. OpenList が空ならば、S からゴールに至る径路は存在しない。終了。

3. OpenList の先頭を取りだし、OpenList から除く。それを A とする。

4. A がゴールならば、S からゴールに至る節点の系列を出力して終了。

5. A がゴールでなければ、A を展開し到達可能な節点の集合 P_A を求める。

6. A を ClosedList の最後に入れる。

7. P_A が空ならば 2 へ戻る。(A が葉節点のとき)

8. P_A が空でなければ、その節点を適当な順序で OpenList の先頭に追加する。

8a. このとき、追加した節点には［A］を添字として付けておく。

8b. P_A に ClosedList の節点と同じ節点があれば、それを P_A から除いてお

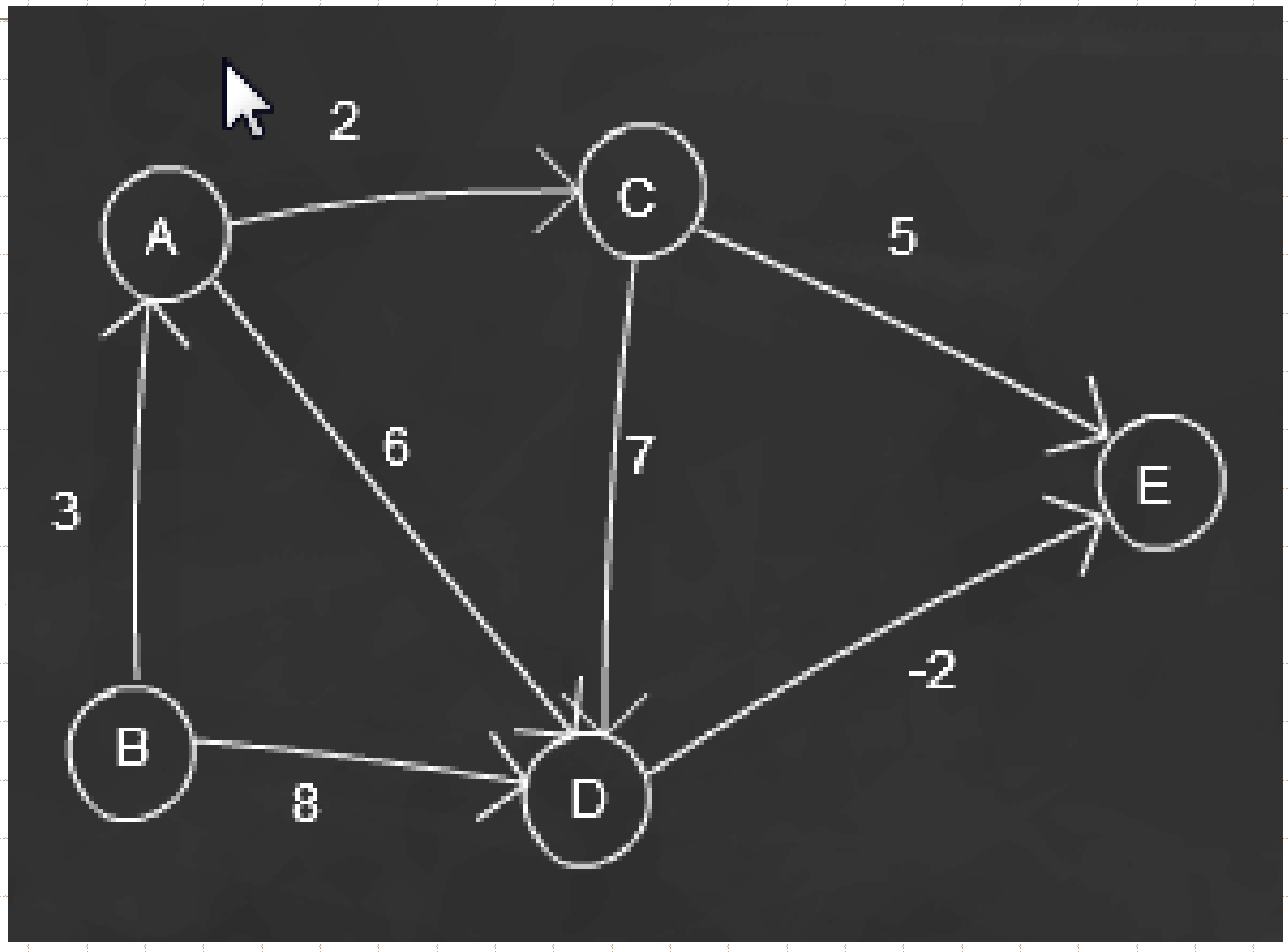
8c. また、OpenList に P_A の節点と同じ節点があれば、OpenList から除

LOOP_END：2 に戻る。

水差し問題の深さ優先探索(DFS)プログラム実行例

```
>>>
WJ((4, 3),(0, 0),(2, 0))
DFS:
(0, 0)
<Node (0, 0)>
[<Node (0, 0)>]
('Found goal:', <Node (2, 0)>)
('Cost of the search:', 11)
('Path:', [<Node (0, 0)>, <Node (0, 3)>, <Node (3, 0)>, <Node (3, 3)>
, <Node (4, 2)>, <Node (4, 0)>, <Node (1, 3)>, <Node (1, 0)>, <Node (
0, 1)>, <Node (4, 1)>, <Node (2, 3)>, <Node (2, 0)>])
```

グラフ探索の例：Bスタート,Eゴール

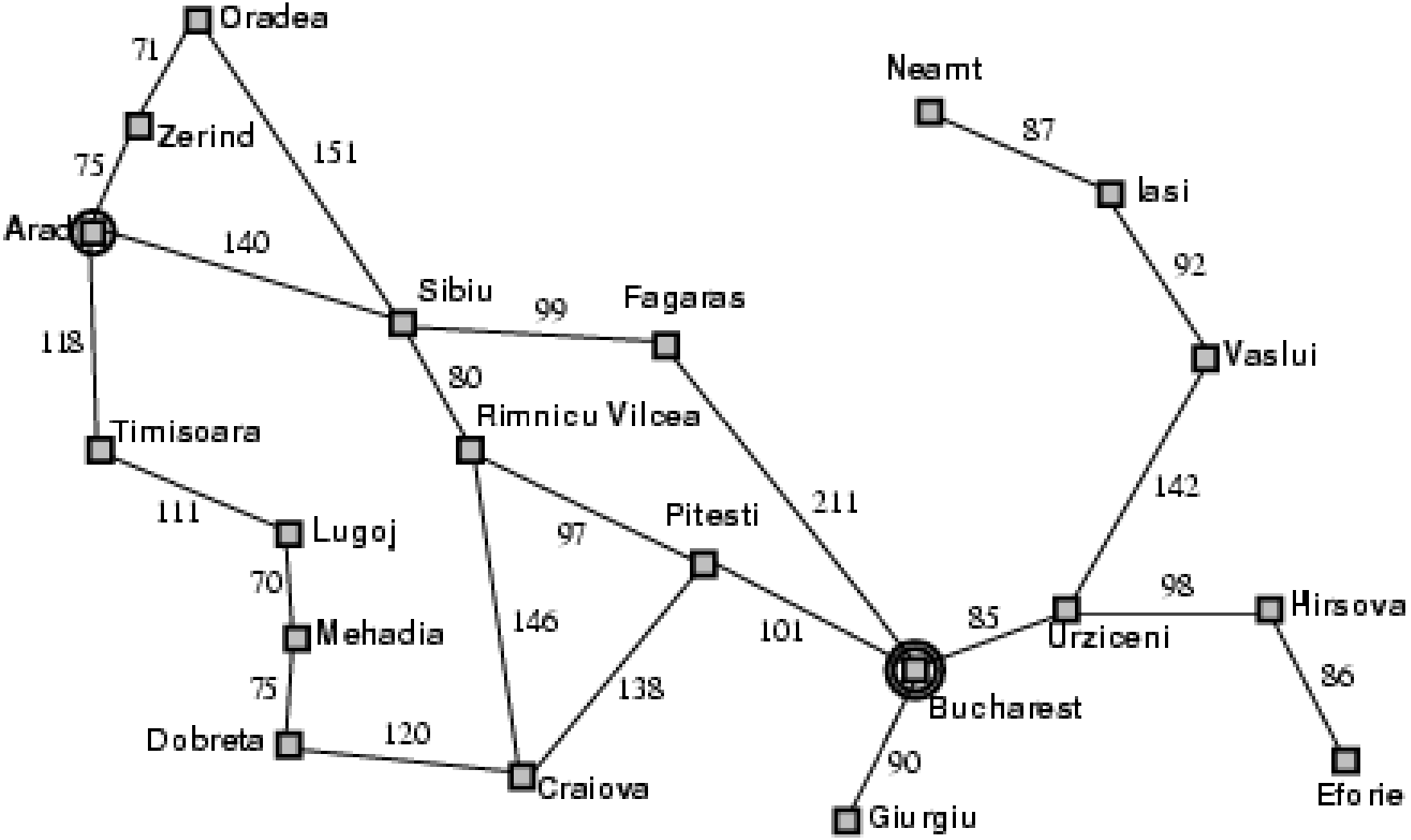


問題解決エージェントの例

- 旅行者（エージェント）が，ルーマニアの都市Aradに滞在している．エージェントは，次の日Bucharestから飛び立つチケットを持っている．どうすればよいか．
  - **ゴールの定式化**：
    - 「ドライブしてBucharestに行く」を，ゴールとして設定すべきである．
  - **問題の定式化**：
    - 「左足を前方に18インチ動かす」あるいは「ハンドルを左へ6度回す」などは，行為としては不適切である．
    - 都市間をドライブするというレベルの行為を考える．
  - **解の探索**：
    - 都市間の隣接情報から，AradからBucharestに行く可能なコースを探索し，最適なプランを立てる．
  - **解の実行**：
    - 実際に解に従って，AradからBucharestにドライブする．


SLOVAKIA
Košice
Uzhhorod
Mukachëve
Ivano-Frankivs'k
UKRAINE
Kam''yanets'-Podil's'kyy
Kolomyya
Prut
Dnistr
Kel'mentsi
Mohyliv-Podil's'kyy
Uman
Chernivtsi
Miskolc
Nyíregyháza
Tisza
BOTOŞANI
Botoşani
Bălți
Ribnița
Broshtyany
Satu Mare
SATU MARE
Carei
Baia Mare
MARAMUREŞ
Suceava
SUCEAVA
HUNGARY
Debrecen
Szolnok
BISTRIȚA-NĂSĂUD
IAŞI
MOLDOVA
Ungheni
Oradea
SĂLAJ
Zalău
Dej
Bistrița
NEAMȚ
Iași
Chisinău
BIHOR
CLUJ
Roman
Békéscsaba
Cluj-Napoca
Piatra-Neamț
Tighina
Vaslui
Turda
Târgu Mureș
MUREŞ
HARGHITA
Szeged
ARAD
Miercurea-Ciuc
BACĂU
Bacău
VASLUI
ALBA
Alba Iulia
Arad
Birlad
Mediaș
Artsyz
COVASNA
Timişoara
Deva
SIBIU
Sibiu
BRAŞOV
Sfîntu Gheorghe
VRANCEA
GALAȚI
Cahul
UKRAINE
Hunedoara
TIMIŞ
HUNEDOARA
Focșani
Vojvodina
(autonomous province)
Braşov
Galați
Reni
Kiliya
Izmayil
Danube
VÎLCEA
Petroşani
Reşiţa
BUZĂU
Brăila
Tulcea
Târgu-Jiu
GORJ
Rîmnicu Vîlcea
ARGEŞ
PRAHOVA
Buzău
BRĂILA
TULCEA
CARAŞ-SEVERIN
Drobeta Turnu Severin
Belgrade
Danube
Pitești
Târgoviște
Ploiești
DÎMBOVIȚA
IALOMIȚA
Smederevo
BUCUREŞTI
Slobozia
MEHEDINȚI
Slatina
Bucharest
Craiova
OLT
CĂLĂRAȘI
Călărași
Kragujevac
DOLJ
GIURGIU
CONSTANȚA
Constanța
Vidin
TELEORMAN
Silistra
Kraljevo
Calafat
Alexandria
Giurgiu
Ruse
Black Sea
Kruševac
Dobrich
Serbia
Niš
Montana
Pleven
Vratsa
Leskovac
BULGARIA
Kosovo
(autonomous province)
Priština
Pernik
Sofia
Uroševac
Romania
International boundary
County (județ) boundary
National capital
County (județ) center
Railroad
Expressway
Road

例: ロマニア(Romania)



問題を定式化すること

- 問題の構成要素
  - **初期状態**：エージェントが認識している最初の状態
  - **オペレータ**：エージェントが利用できる可能な行為
  - **ゴール検査**：エージェントが到達した状態がゴール状態であるかを決定する検査.
  - **経路コスト**：経路をたどるのに必要なコスト．経路コストは，経路に沿った各々の行為のコストの総和.
- 問題の環境
  - 状態空間：初期状態から行為の任意の系列によって到達可能なすべての状態の集合
  - 経路：1つの状態を他の状態に導く行為列

解の探索(1) 行為列の生成

- AradからBucharestへのルート発見問題を解くためには，初期状態Aradから開始する.
- 最初のステップは，それがゴール状態であるかのテスト.
- それがゴール状態でないので，他の状態を考える．そのために，現在の状態にオペレータを適用し，それによって新しい状態の集合を**生成**する(状態の**展開**).
- ここでは，Sibiu，Timosoara，Zerindという三つの新しい状態が得られる.
- つぎに，この三つから一つの選択肢を選んで，探索を続ける．最初の選択が解に至らない場合，他の選択肢を選んで，探索を続ける．最初にどの状態を展開させるかの選択は，**探索戦略**によって決定される.
- 状態空間上に**探索木**を構成して，探索過程を表現する.

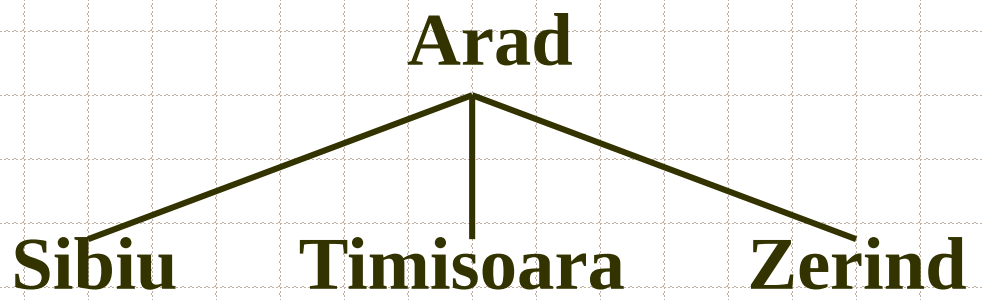
解の探索(2)　部分探索木

- 探索木のルートは，初期状態を表す**探索ノード**.
- 木の葉ノードは，まだ展開していないか，あるいは展開したけれども空集合を生成したために後続ノードを持たない状態に対応する.
- 探索アルゴリズムは，展開させる葉ノードを一つ選ぶ.

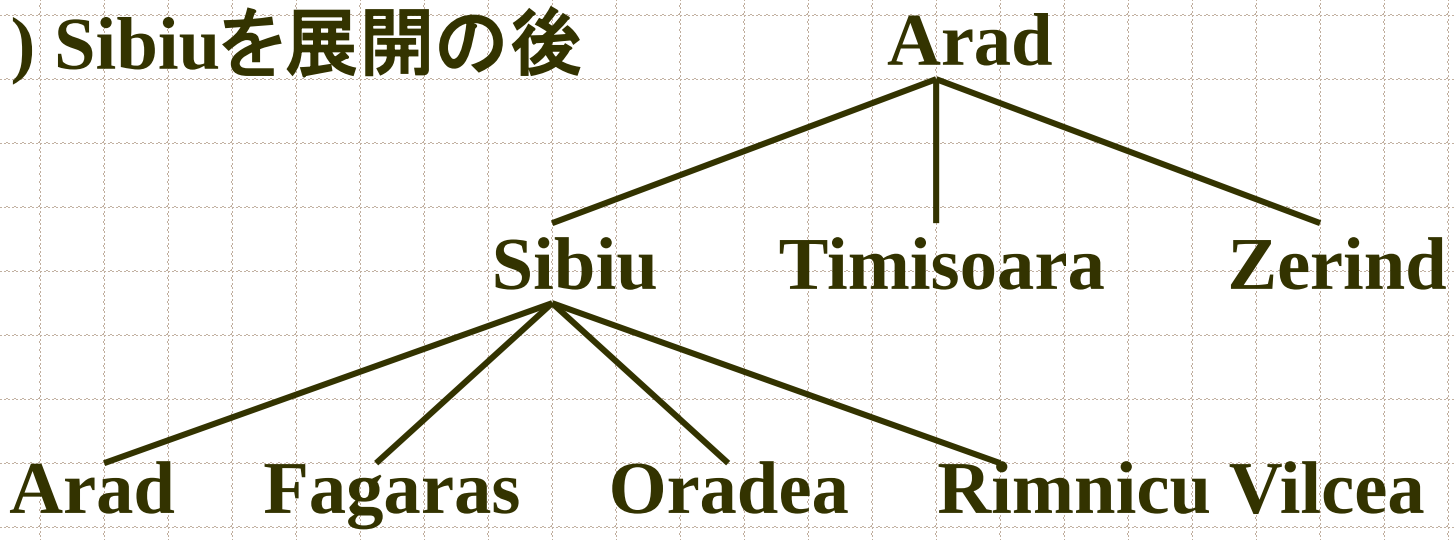
解の探索(3) 部分探索木の例

(a) 初期状態

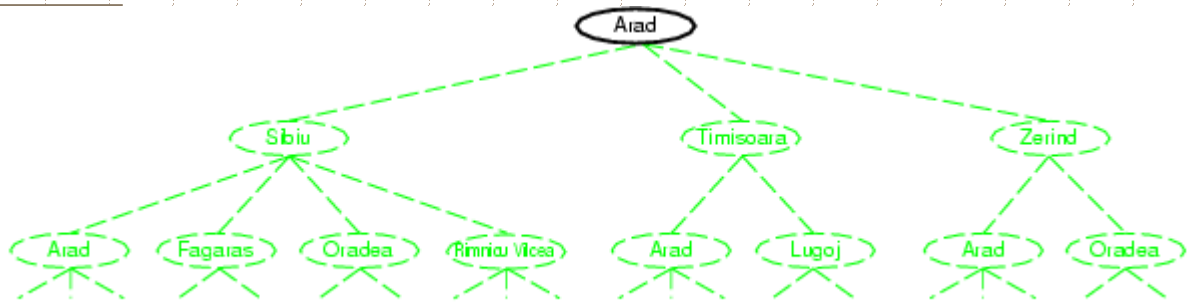
(b) 展開の後



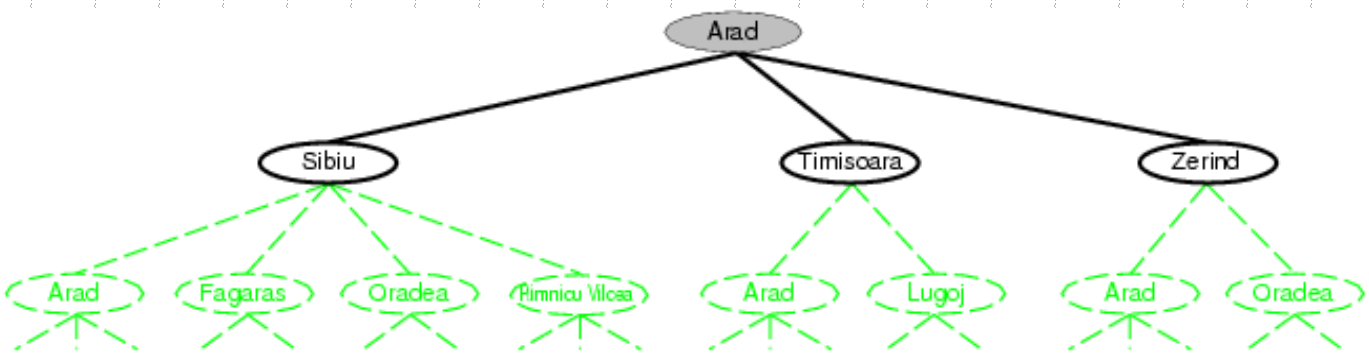
(c) Sibiuを展開の後



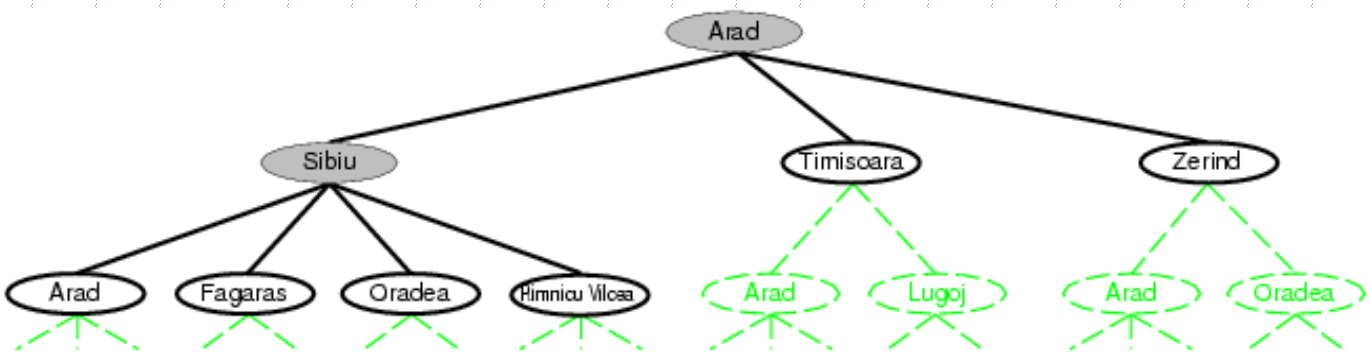
探索木の例(Tree search example)



探索木の例(Tree search example)



探索木の例(Tree search example)



解の探索(4)
探索木のためのデータ構造

- ノードの構成要素
  - 状態空間においてノードが対応している状態
  - 探索木においてこのノードを生成したノード(**親ノード**)
  - ノードを生成するために適用されたオペレータ
  - ルートからこのノードへの経路上のノードの数(ノードの**深さ**)
  - 初期状態からこのノードまでの経路の経路コスト
- ノードのデータタイプ

datatype Node

components: State, Parent-Node, Operator, Depth, Path-Cost

解の探索(4)
一般的探索アルゴリズム

function General-Search(*問題*, *戦略*) returns 一つの解 or 失敗

*問題*の初期状態で，探索木を初期設定する.

loop do

if 展開すべき候補ノードがない **then return** 失敗

*戦略*によって展開すべき葉ノードを選ぶ

if そのノードがゴール状態 **then return** 対応する解

else そのノードを展開して結果のノード群を探索木に加える

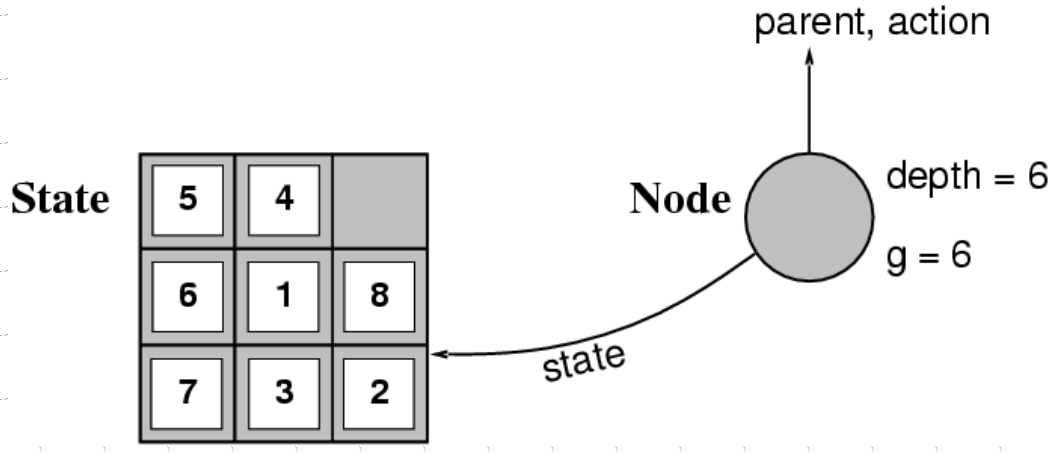
end

実装：一般探索木(Implementation: general tree search)

```
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
```

実装：状態対ノード(Implementation: states vs. nodes)

- 状態は物理的な表現(A state is a (representation of) a physical configuration)
- ノードはデータ構造であり、木の一部の構成になる。(A node is a data structure constituting part of a search tree includes state, parent node, action, path cost *g(x)*, depth)



- Expand関数はSuccessor-Fn関数を用いてノードを生成し、問題の状態を生成する。

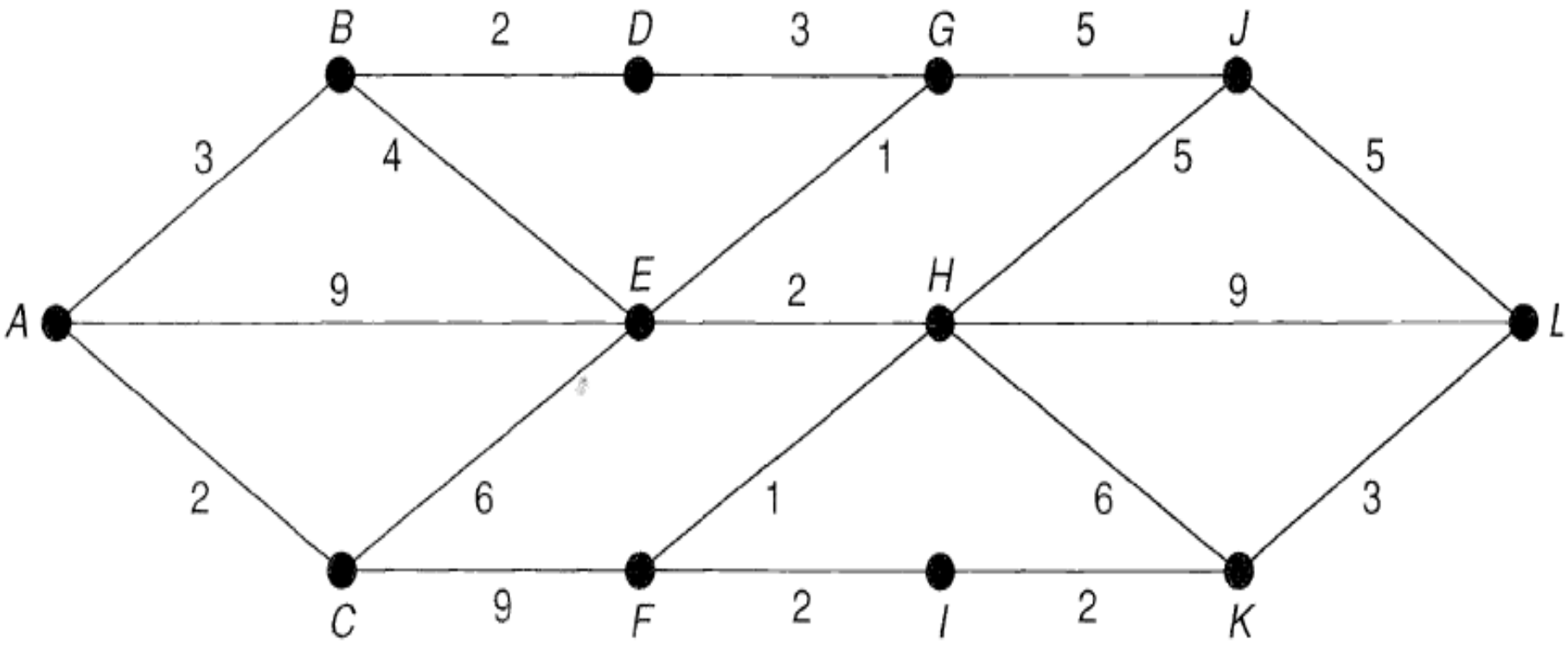
グラフ探索アルゴリズム(Graph search)

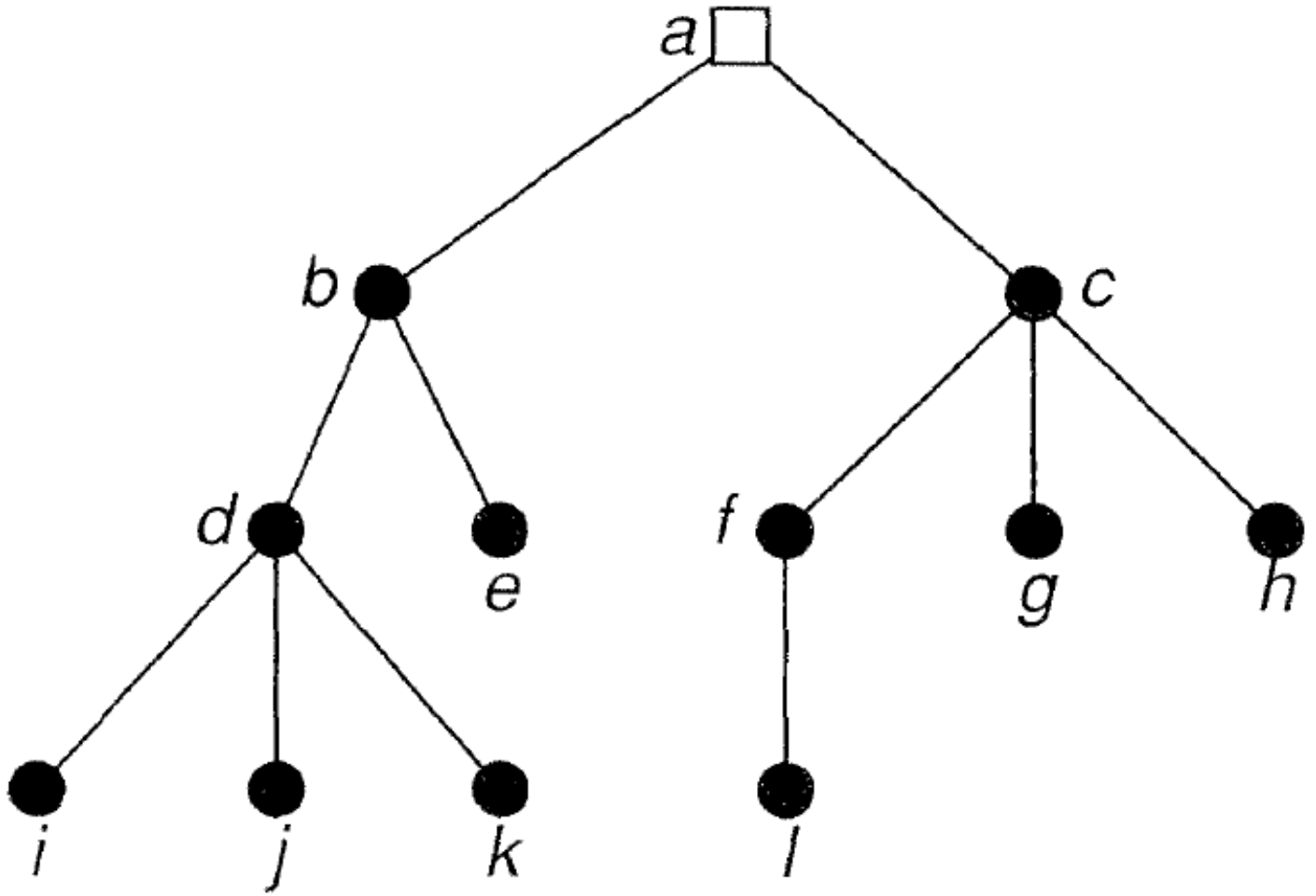
```
function GRAPH-SEARCH( problem, fringe) returns a solution, or failure

    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)
```

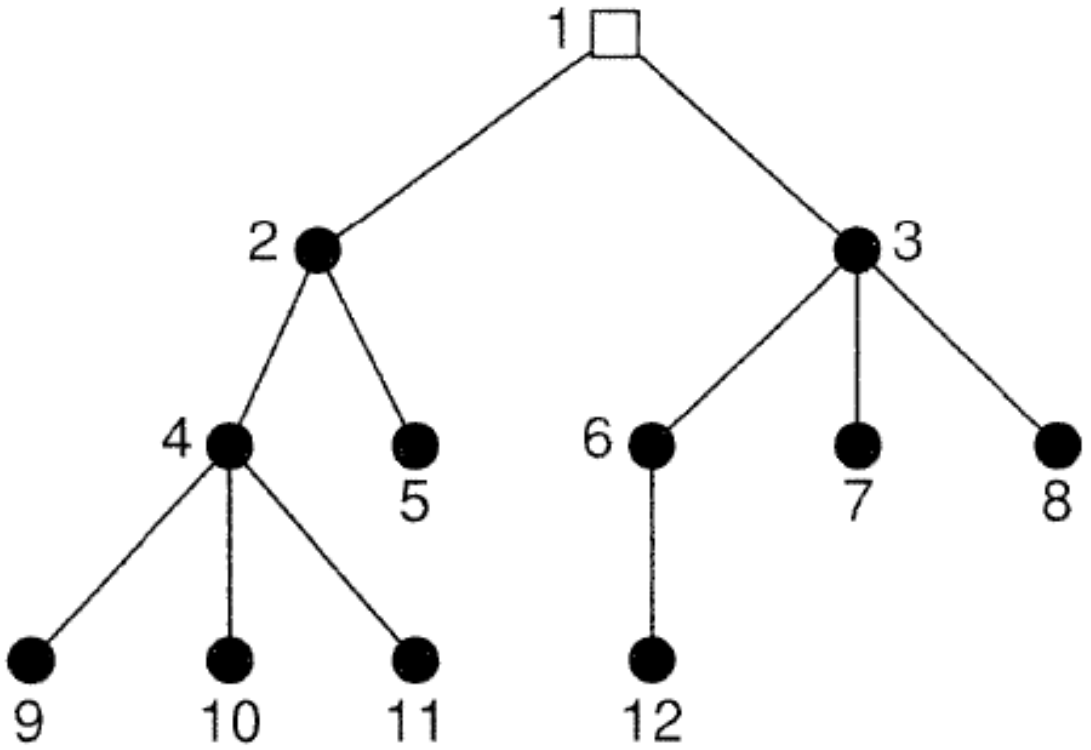
まとめ

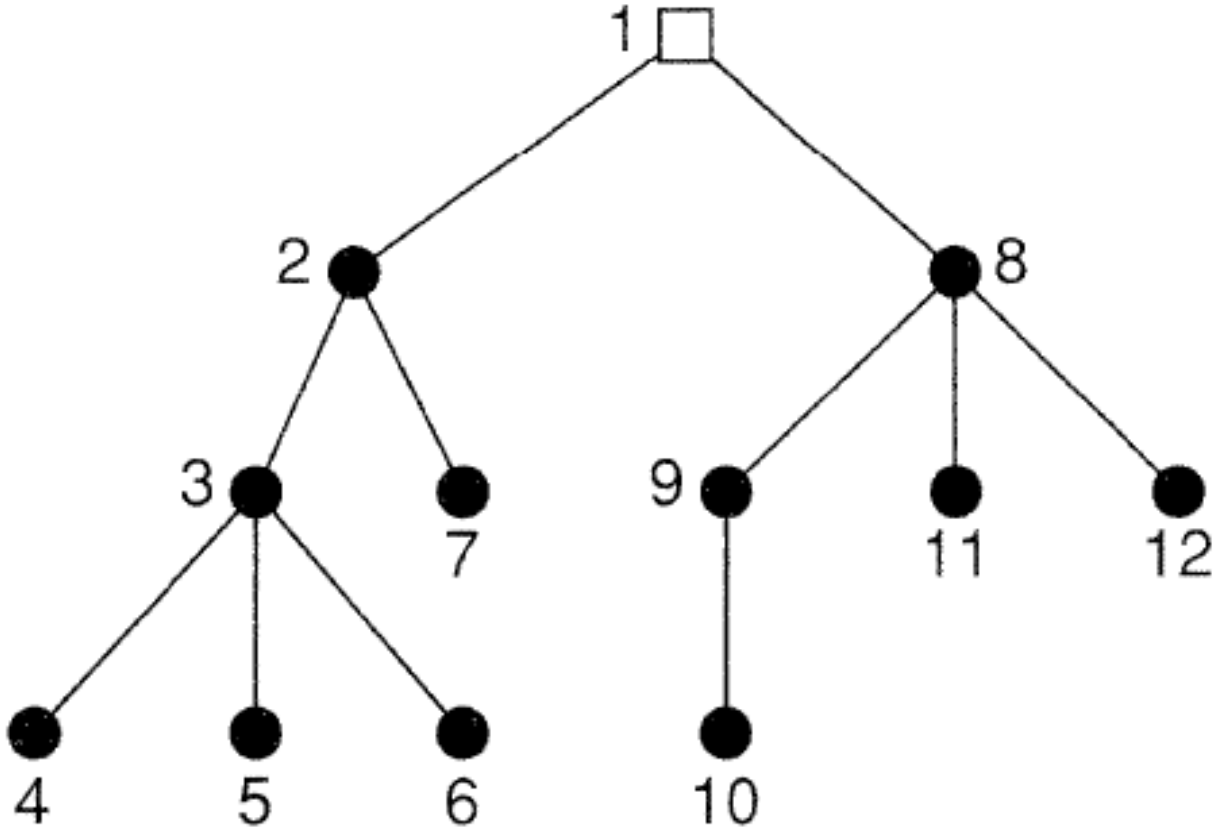
- 問題解決エージェントは，ゴールが与えられて，そのゴールに至る解を探索する．
- はじめに，**ゴール**を定式化し，つぎに，**問題**の定式化を行う．
- 問題は，**初期状態，オペレータ，ゴール検査，経路コスト**から成り立つ．
- 探索アルゴリズムは，**完全性，最適性，時間計算量**および**空間計算量**の基準で判断される．計算量は，状態空間の分岐度bと最も浅い解の深さdに依存する．

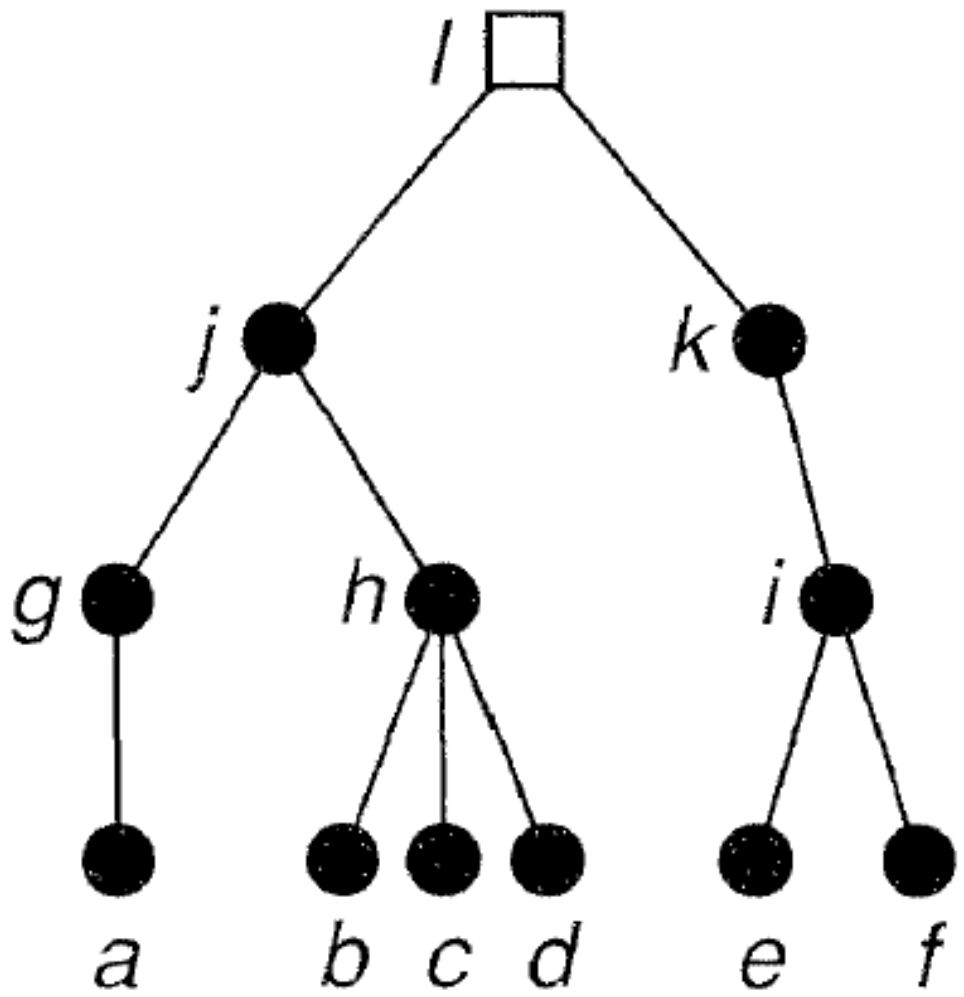
例

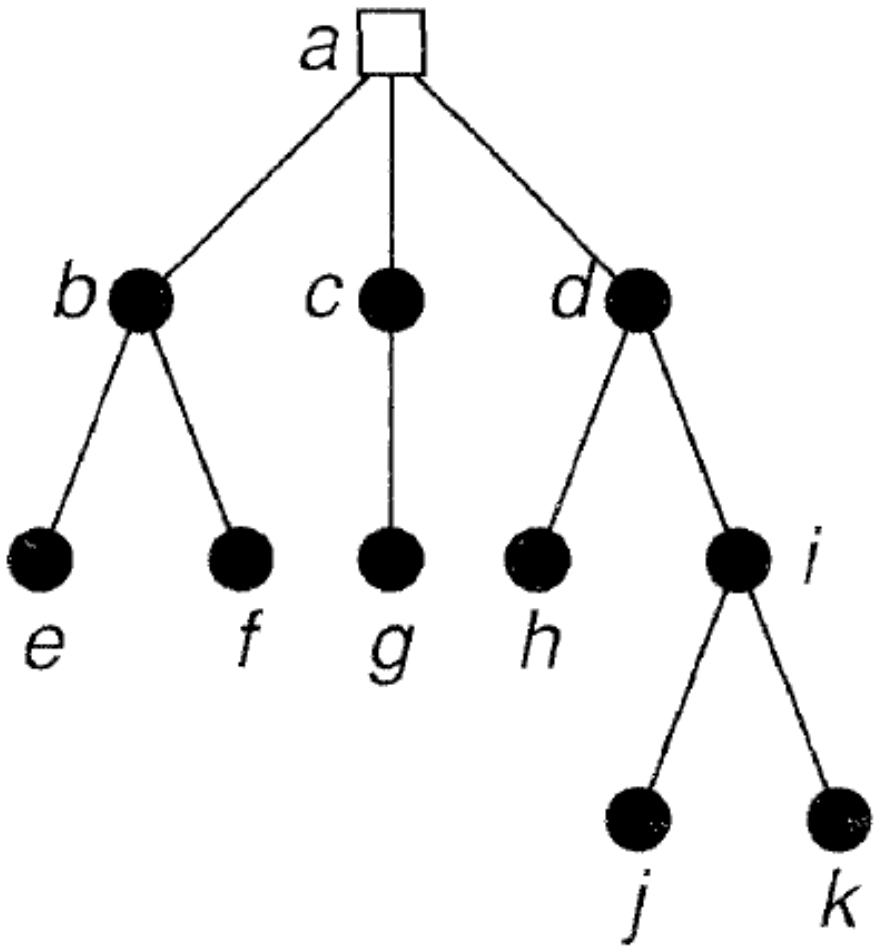



a
b
c
d
e
f
g
h
i
j
k
l


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


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


l
j
k
g
h
i
a
b
c
d
e
f


a
b
c
d
e
f
g
h
i
j
k