インテリジェントシステム

#2 探索による問題解決：知識を用いない探索
uninformed search

信州大学工学部電子情報システム工学科
丸山稔

問題解決エージェント（problem solving agent）：

取るべき行動が即時明らかにならない場合⇒ゴールに到達する行動系列を考える必要あり

問題解決のための探索アルゴリズムを扱う

→ 環境は：

エピソード的、単一エージェント、完全可観測、決定的、静的、離散的、既知

アルゴリズム分類：uninformed （知識を用いない探索）, informed（知識に基づく探索）

知識とは？：エージェントがゴールからどれだけ離れているか推定可能

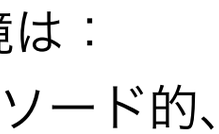
例題：ルーマニア旅行

現在地：Arad → Bucharestへ行きたい

Aradからの３つの道路

→Sibiu行, Zerind行, Timisoara行

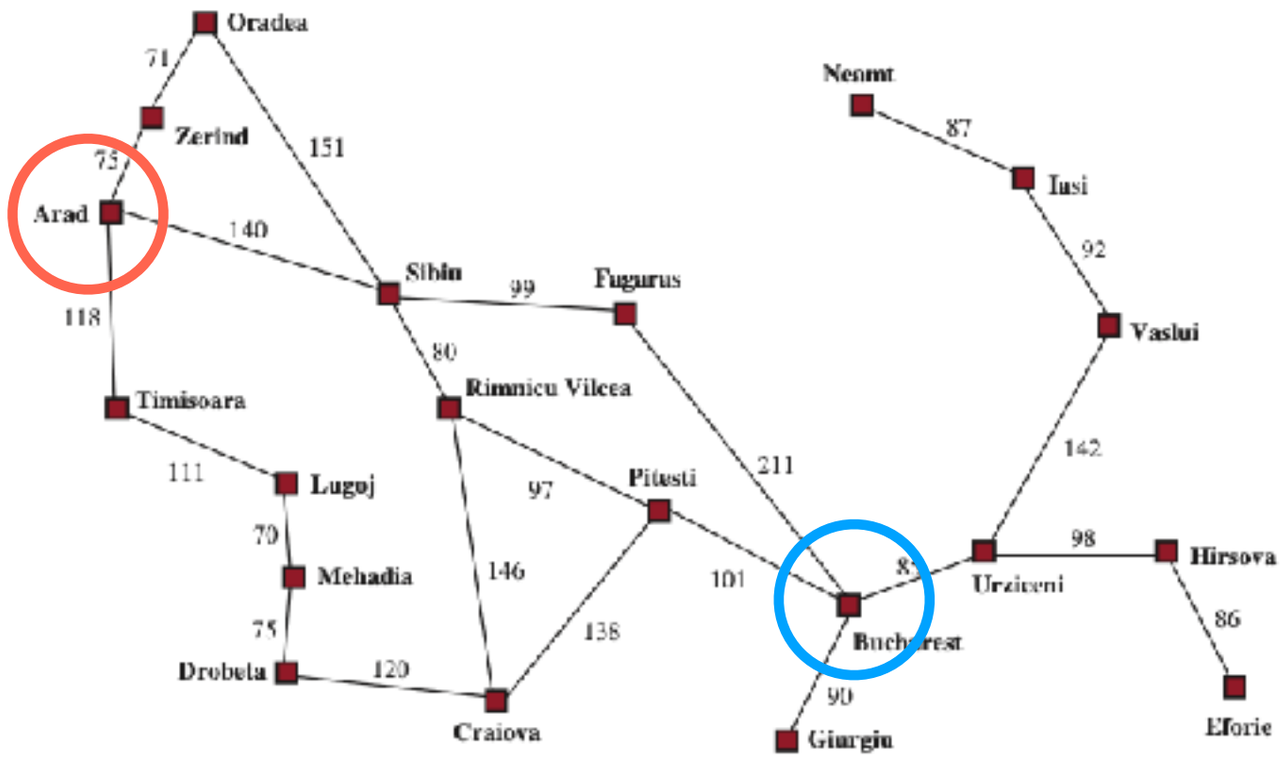
どれもゴールではない



agentは世界に関する情報へアクセス可とする

（e.x. ルーマニア地図）

⇒ これに基づいて問題解決



S.Russell & P.Norvig, Artificial Intelligence 4th Ed., Pearson, 2021

http://aima.cs.berkeley.edu/figures.pdf

問題解決の４段階
①ゴール設定：ex Bucharestへの到着
②問題設定：状態とゴールに到達するために必要な行動を記述
③探索：現実の行動を取る前に行動系列のシミュレーション
　　⇒ゴールに到達する解（行動系列）を見つけるまで
④実行：シミュレーション中の行動を実行

探索問題：形式的に以下のように定義できる
・状態空間（state space）：取り得ることが可能な状態の集合…ex. 都市の集合
・初期状態（initial state）：スタート時の状態…ex. Arad
・ゴール状態（goal states）→ ゴールに到達したかテスト：`IS-GOAL`メソッド
・エージェントが取り得る行動→状態sのとき`ACTIONS(s)`
　… ex. `ACTIONS(Arad) = {ToSibiu, ToTimisoara, ToZerind}`
・遷移モデル（transition model）：エージェントの行動により何が起こるか記述
　→ `RESULT(s,a)` …状態sで行動aを行った結果どんな状態になるかを返す関数
　　ex. `RESULT(Arao, ToZerind) = Zerind`
・行動コスト関数（action cost function）：`ACTION-COST(s,a,s')`
　状態sで行動aを行った結果、状態s'になったときのコスト

行動の系列⇒経路（path）を生成

解（solution）：初期状態からゴールまでの経路

最適解（optimal solution）：解の中で経路のコストが最小となるもの

以下では行動コストは正であると仮定

状態空間：グラフで表現できる

状態→頂点（vertices）、行動→状態間の有向辺（directed edges）

（無向）辺：双方向への状態変化を示す

状態空間グラフの例：２セル-掃除機世界

状態　２つのセル・agent（掃除機）はどちらかのセルに居る・ゴミの有無⇒8状態

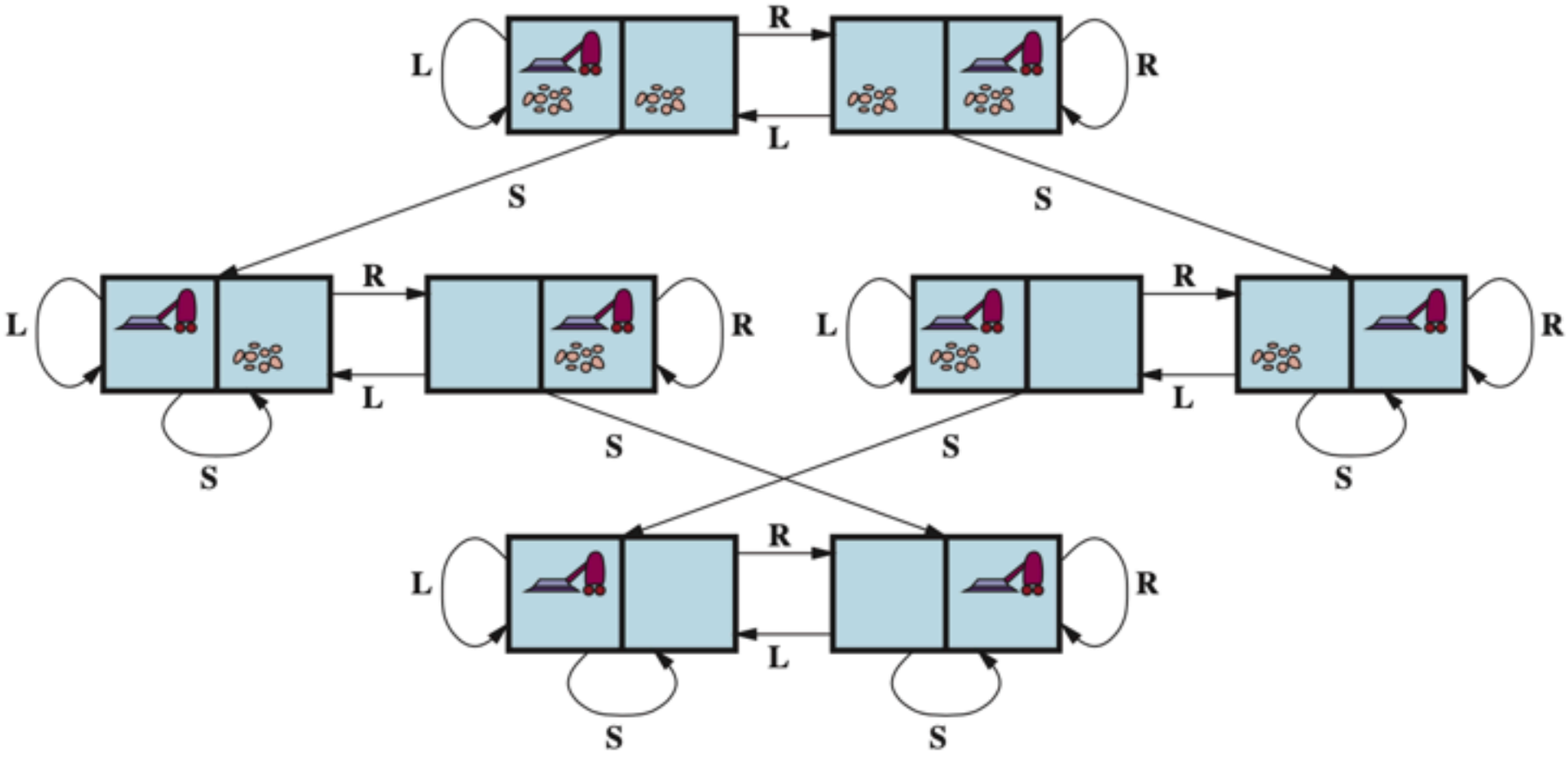
初期状態　どの状態でも可

行動　L : 左へ移動（Left）, R：右へ移動（Right）, S：吸引（Suck）

遷移モデル　Sによりagentが居るセルからゴミは無くなる, L,Rによりagent移動

ゴール　全てのセルにゴミが無くなる

行動のコスト　各行動につきコスト１

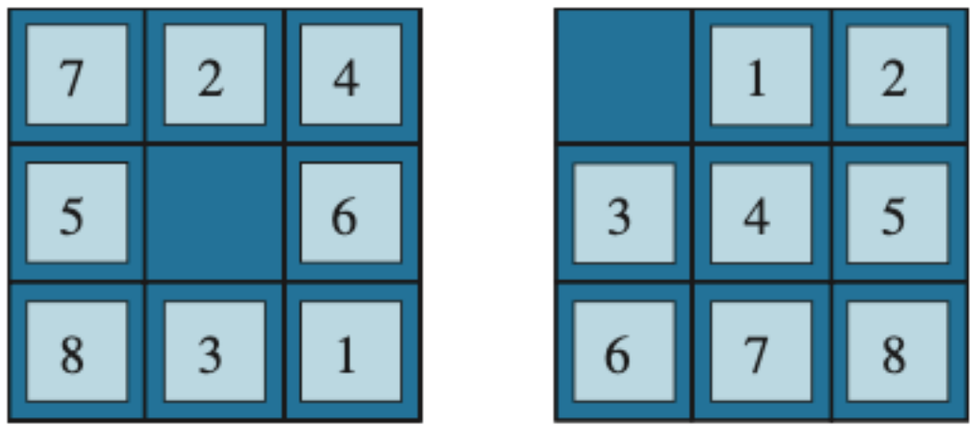


S.Russell & P.Norvig, Artificial Intelligence 4th Ed., Pearson, 2020
http://aima.cs.berkeley.edu/figures.pdf

状態空間グラフの例：8-puzzle

3x3グリッド上に8個のタイル＆１つの空白が配置されている
空白箇所に隣接されたタイルは空白部へスライドできる
⇒指定された配置に並べ直したい

状態　各タイルの位置
初期状態　どの状態でも可
行動　空白部が移動すると考えればよい：Left, Right, Up or Down
遷移モデル　空白部の向かった先のタイルと空白の入替え
ゴール　指定された配置
行動のコスト　各行動につきコスト１

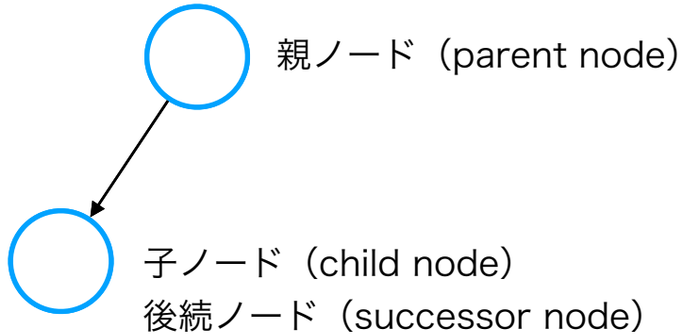


S.Russell & P.Norvig, Artificial Intelligence 4th Ed., Pearson, 2021
http://aima.cs.berkeley.edu/figures.pdf

探索アルゴリズム（Search Algorithms）：

探索問題が与えられたとき→解、または”失敗”（failure…ゴールに到達できない）を返す

探索木（search tree）を考える
初期状態をrootとし、初期状態からの種々の経路を表す→ゴールに至る経路を探す
木の各ノード（node）：状態に対応
木の辺（edge）：行動に対応→行動による状態遷移を表す

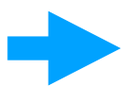


探索木と状態空間グラフの違い
探索木：ゴールに向かう状態間の経路を示す
…任意の状態についてそこへ至る複数の経路が存在することもある
→ 状態に関して木のノードがユニークに決まるとは限らない
木の各ノードはroot へのユニークな経路を持つ

探索木の生成

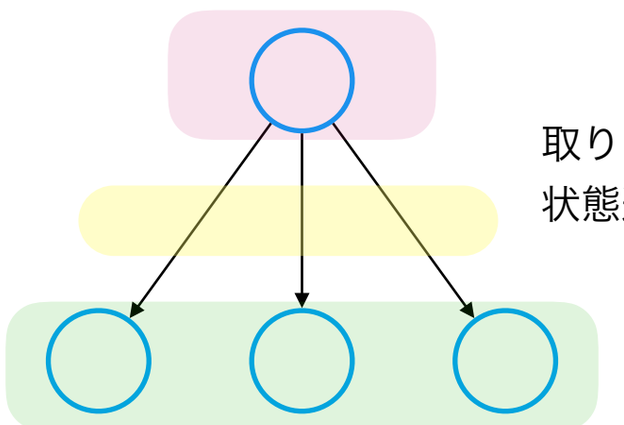
初期状態（root）

ノードの**展開**（**expand**）



・この状態における可能な
行動を考える
・各行動についてどのような
状態が得られるか調べる
→ 新規ノード生成

展開済のノード（**expanded**）

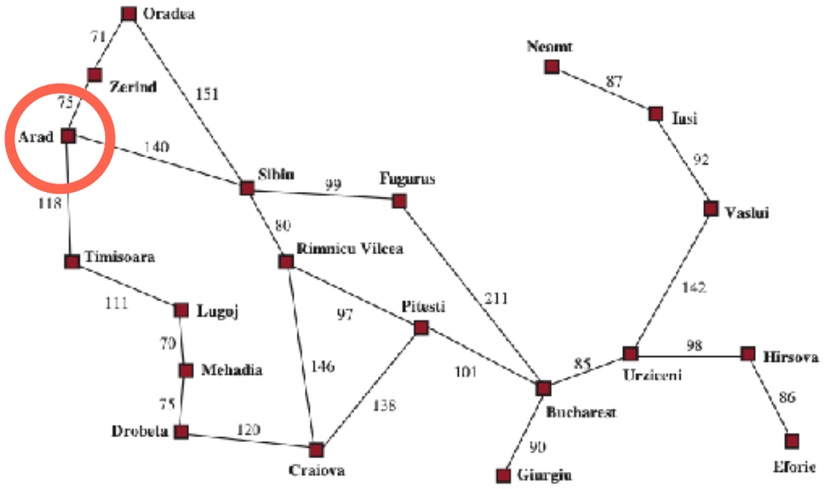


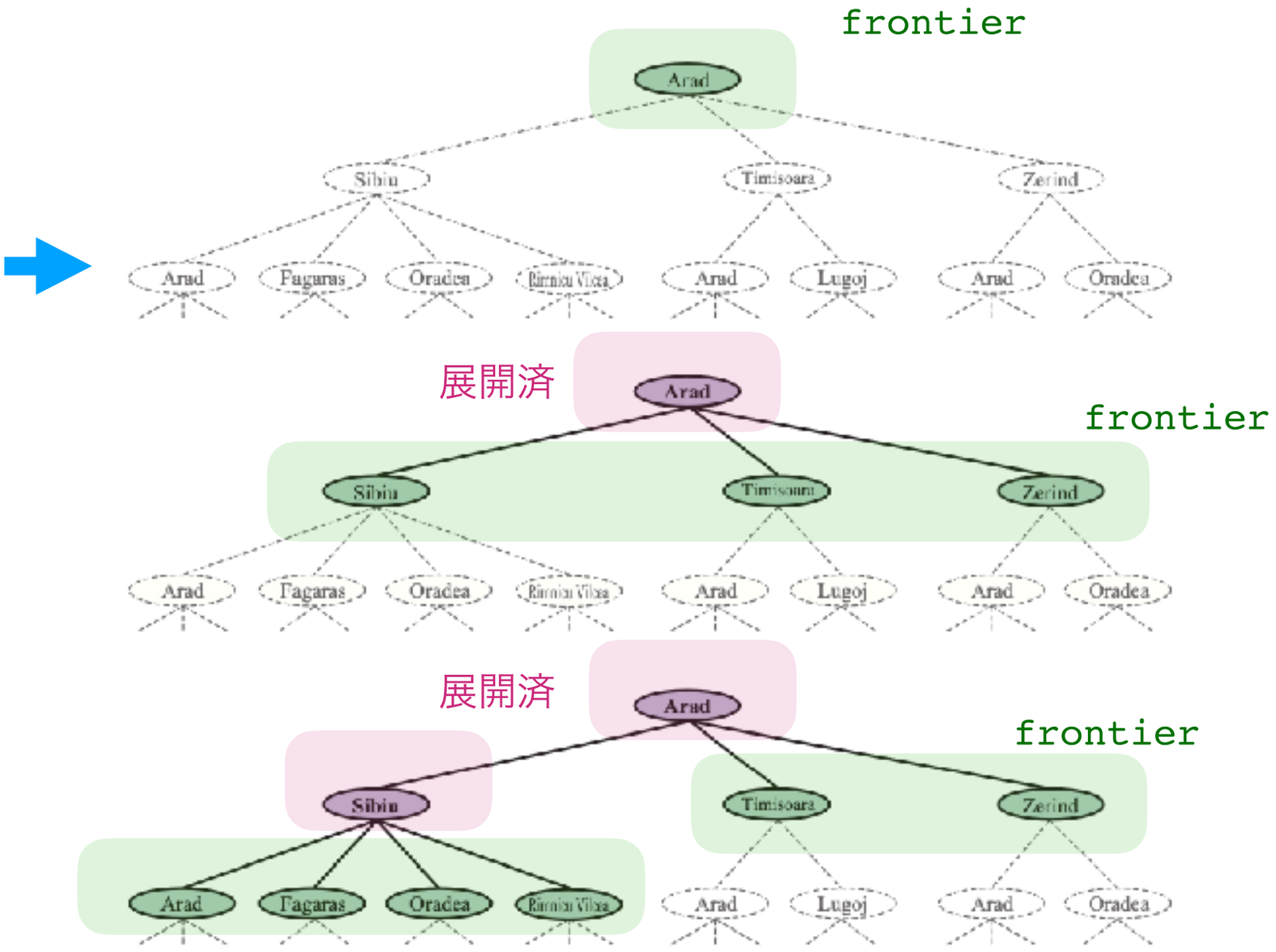
取りうる行動による
状態遷移を表す

未展開のノード集合→**frontier**と呼ぶ

$$\text{reached} = \text{expanded} \cup \text{frontier}$$

探索木の例（途中経過）：ルーマニア旅行の場合





S.Russell & P.Norvig, Artificial Intelligence 4th Ed., Pearson, 2020
http://aima.cs.berkeley.edu/figures.pdf

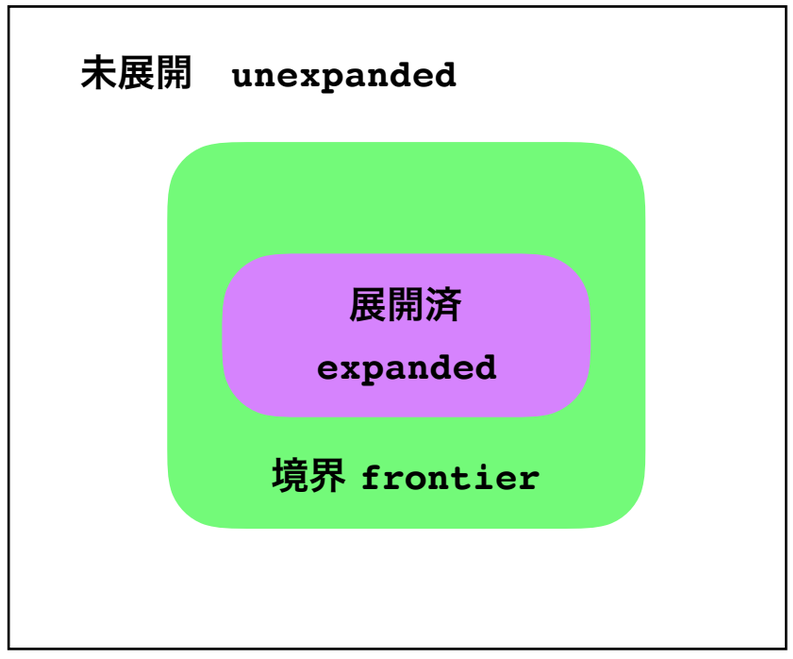
最良優先探索（best-first search）

初期状態（root）からノードを展開していく→次に考慮の対象にするノードはどれにする？

frontier中のどのノードから展開していくか

一般的アプローチ⇒**best-first search**

- ノードに対する評価関数f(n)を考え、frontier中f(n)-値が最良のものを選択
- 選択したノードnがゴールだった場合、それを返す
- ゴールでなければnにEXPANDを適用して展開→子ノード生成
- 子ノードが未到達（reachedに含まれない）ときfrontierに追加
  到達済でも以前の経路より良好なコストで到達している場合は再追加



reached ＝ expanded ∪ frontier

探索アルゴリズム：探索木の経過を保持するデータ構造が必要

nodeデータ構造

node.STATE：nodeに対応する状態
node.PARENT：木におけるnodeの親ノード（nodeを生成したノード）
node.ACTION：nodeが生成された際に親ノードに適用されたaction
node.PATH-COST：初期状態からnodeまでの総コスト

各nodeからPARENTをたどることにより、そのnodeまでの状態と行動の系列を復元できる

`frontier`データ構造：

なんらかのqueue（動的集合のためのデータ構造）を用いる

以下の演算を実行

- `IS-EMPTY(frontier)`：`frontier`にノードが１つも含まれていないときにのみtrueを返す
- `POP(frontier)`：`frontier`からtop nodeを削除し、そのノードを返す
- `TOP(frontier)`：`frontier`中のtop nodeを返す（削除はしない）
- `ADD(node, frontier)`：queue中の適切な位置に`node`を挿入

使用されるqueueの種類：

- **プライオリティキュー（priority queue）**

  評価関数fに基づいて、最小コストのノードを最初にpopする

  ⇒最良優先探索（best-first search）で用いられる
- **FIFO-queue**（いわゆるキュー）

  first-in-first-out 最初に挿入されたノードを最初にpopする

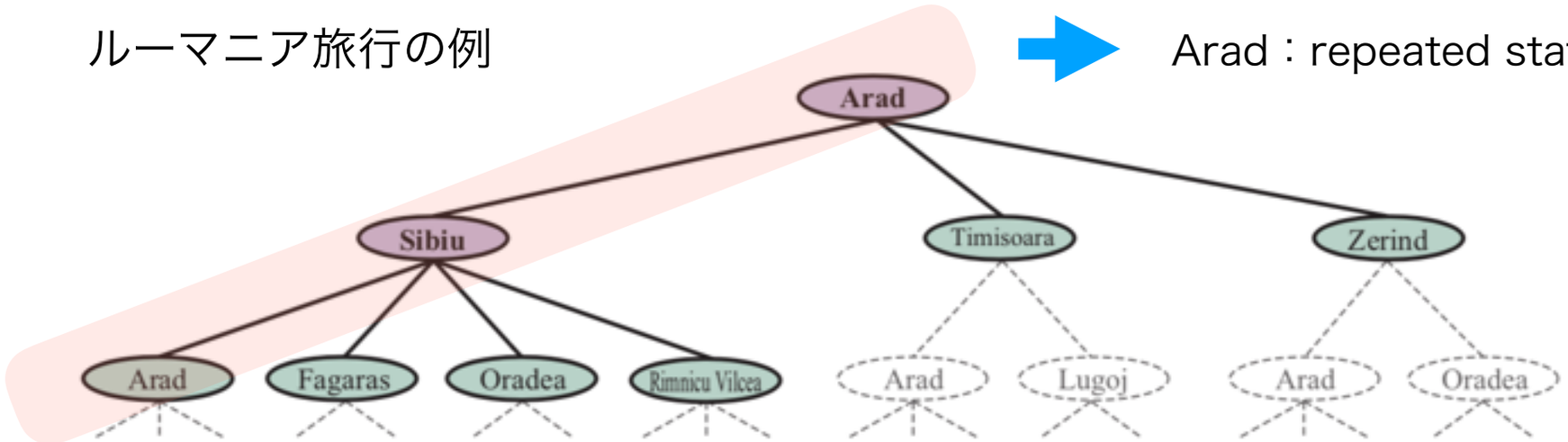
  ⇒ 幅優先探索（BFS, breadth-first search）で用いられる
- **LIFO-queue**（stack）

  last-in-first-out いちばん最後（最近）に挿入されたノードを最初にpopする

  ⇒ 深さ優先探索（DFS, depth-first search）で用いられる

冗長な経路の処理

ルーマニア旅行の例



Arad：repeated state … 閉路（cycle）を成す

cycle: 冗長な経路の一種

…他にも… Sibiuへ行くには… Arad→Sibiu（コスト=走行距離：140）

Arad→Zerind→Orada→Sibiu（297）← 冗長な経路

対処方法

・**graph search** → 冗長な経路をチェック

・**tree-like search** → 冗長な経路のチェックを行わない

…必要なメモリは削減できるが計算時間は増大する可能性

S.Russell & P.Norvig, Artificial Intelligence 4th Ed., Pearson, 2020

http://aima.cs.berkeley.edu/figures.pdf

Best-First Search…graph search版

```
function BEST-FIRST-SEARCH(problem, f) returns solution nodeまたはfailure
    node ← NODE(STATE=problem.INITIAL)
    frontier ← nodeを要素としfに基づいたpriority queue
    reached ← keyはproblem.INITIAL, valueはnodeであるような1要素だけのlook-up table
    while not IS-EMPTY(frontier) do
        node ← POP(frontier)
        if problem.IS-GOAL(node.STATE) then return node
        for each child in EXPAND(problem, node) do
            s ← child.STATE
            if sがreachedに含まれない  or child.PATH-COST < reached[s].PATH-COST then
                reached[s] ← child //sが既にreached中に存在していてもコストに基づき上書きされる
                childをfrontierに追加
    return failure

function EXPAND(problem, node) yields nodes
    s ← node.STATE
    for each action in problem.ACTIONS(s) do
        s' ← problem.RESULT(section)
        cost ← node.PATH-COST + problem.ACTION-COST(s, action, s')
        yield NODE(STATE=s', PARENT=node, ACTION=action, PATH-COST=cost)
```

この部分を除くと
tree-like searchとなる

generator：値の列を生成する

探索アルゴリズムの評価基準

様々な探索アルゴリズムが有りうる→どうやって比較・評価する？

評価基準

- **完全性（completeness）**
  解が存在するときはそれを見つけ、解が存在しないときはfailureと返せるか？

- **コスト最適性（cost optimality）**
  経路コスト最小の解を見つけるか？

- **計算複雑性・計算時間（time complexity）**
  解を発見するまでの所要時間は？

- **必要メモリ量（space complexity）**
  探索を実施するのに必要なメモリ量は？

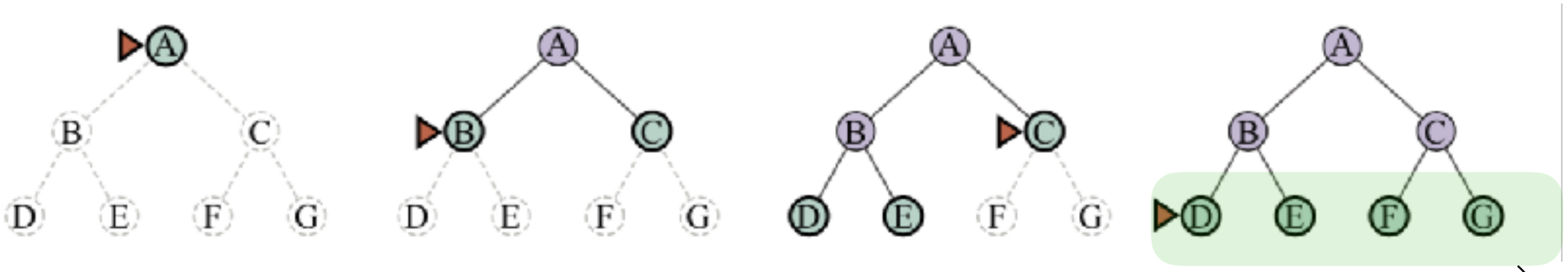
知識を用いない探索アルゴリズム（uninformed search）

uninformed とは？...各状態がゴールからどれだけ近いかの情報が与えられない場合
ルーマニア旅行の場合：AradからSibiu/Zerind/Timisoaraのどちらに行くのが良いか分からない場合

幅優先探索（BFS, Breadth-First Search）

rootノードを展開→全ての子ノードを展開→ついでそれらの子ノードを全て展開→…
f(n) をノードnの木における深さ（＝nに到達するまでのアクション数＝辺の数）
としたときのbest-first-search

BFSによる探索例：



仮定：どの状態においても子ノード数=b, 解は深さdに存在する⇒ノード生成総数は？

$$1 + b + b^2 + \cdots + b^d = \frac{b^{d+1} - 1}{b - 1} = O(b^d)$$

`frontier`のサイズも： $O(b^d)$

S.Russell & P.Norvig, Artificial Intelligence 4th Ed., Pearson, 2020
http://aima.cs.berkeley.edu/figures.pdf

best-first searchからの変更点（効率化）

・プライオリティキューではなくFIFO-queue（キュー）を使用：高速かつ正しい順序を与える

・`reached` ：状態からノードへのマッピングではなくて可⇒状態の集合でよい

一度ある状態に到達したら、それより良いコストの経路は存在しない

・早期のゴールテストが可能：ノードが生成された時点でゴールテストをしてもよい

(∵ その後でそれより良い経路が見つかることはない)

```
function BREADTH-FIRST-SEARCH(problem, f) returns a solution nodeまたはfailure
    node ← NODE(STATE=problem.INITIAL)
    if problem.IS-GOAL(node.STATE) then return node
    frontier ← nodeを要素としたFIFO queue
    reached ← {problem.INITIAL}
    while not IS-EMPTY(frontier) do
        node ← POP(frontier)                          ← FIFO
        for each child in EXPAND(problem, node) do
            s ← child.STATE
            if problem.IS-GOAL(s) then return child   ← 早期のゴールチェック
            if sがreachedに含まれない then
                sをreachedに追加
                childをfrontierに追加
    return failure
```

Dijkstraのアルゴリズム / 均一コスト探索（uniform-cost search）

各行動が異なるコストを持つ場合

⇒ best-first searchで評価関数を`root`から現在のノードまでの経路コストとする

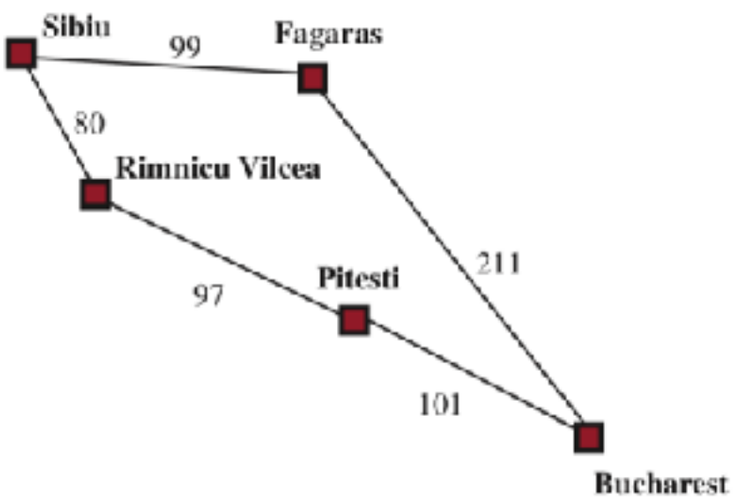
Dijkstraのアルゴリズム　または　均一コスト探索（UCS, uniform-cost search）

評価関数として`PATH-COST`を用いて、`BEST-FIRST-SEARCH`を呼ぶ

ゴールテストはノードが展開されたとき

```
function UNIFORM-COST-SEARCH(problem) returns a solution nodeまたはfailure
    return BEST-FIRST-SEARCH(problem, PATH-COST)
```

UCSによるSibiu→Bucharestへの経路探索例



frontier

Sibiu 0

Rimnicu Vilcea 80　Faragas 99

Sibiu 0

Rimnicu Vilcea 80　Faragas 99

Pitesti
80+97=177

Sibiu 0

Rimnicu Vilcea 80　Faragas 99

Pitesti
80+97=177　Bucharest
99+211=310

Sibiu 0

Rimnicu Vilcea 80　Faragas 99

Pitesti
80+97=177

Bucharest
99+211=310

Bucharest
177+101=278

reachedの書き換え

UCSの性質

C^* ：最適解のコスト

ε ：各actionのコストの下界>0

b ：木の枝分かれ数（子ノード数）

→ 最悪時コストとメモリ量　$O(b^{1+\lfloor \frac{C^*}{\varepsilon} \rfloor})$

$O(b^{1+\lfloor \frac{C^*}{\varepsilon} \rfloor})$ ：⇒ b^d よりかなり大きくなる可能性…低コストの経路を大量に探索する可能性

全てのactionのコストが同じ値⇒　$b^{1+\lfloor \frac{C^*}{\varepsilon} \rfloor} = b^{d+1}$　：BFSと同様

UCS：完全＆コスト最適性を持つ

∵ 最初に見つけた解は`frontier`内のどのノードよりもコストが低い（かまたは同じ）

⇒ その後見つかる解はそれらよりもコストが低くなることは有り得ない

深さ優先探索（DFS, depth-first search）

深さ優先探索：`frontier`中で深さが最大であるノードから展開する

f(n) = - depth(n) … とした場合の`BEST-FIRST-SEARCH`で実現可

↓

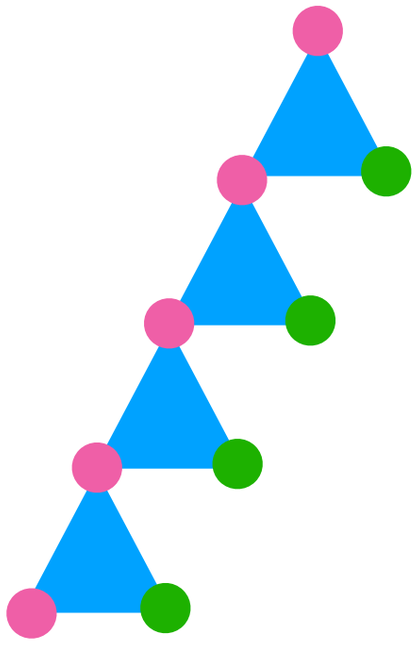
但し、通常はgraph searchではなく
tree-like searchとして実装される（冗長経路のチェック無し）
reached statesの保持は行わない

DFSの性質：コスト最適（cost-optimal）ではない…最初に見つけた解を返す

有限状態空間⇒完全

閉路を持たない状態空間⇒同じ状態を（異なる経路で）何度も展開する可能性あり

無限状態空間⇒無限経路に陥る可能性あり… DFSは完全ではない



完全ではない…それでもBFSやbest-firstではなくDFSが使われるのは？

・tree-like searchが適用可能な問題についてDFSの必要メモリ数少

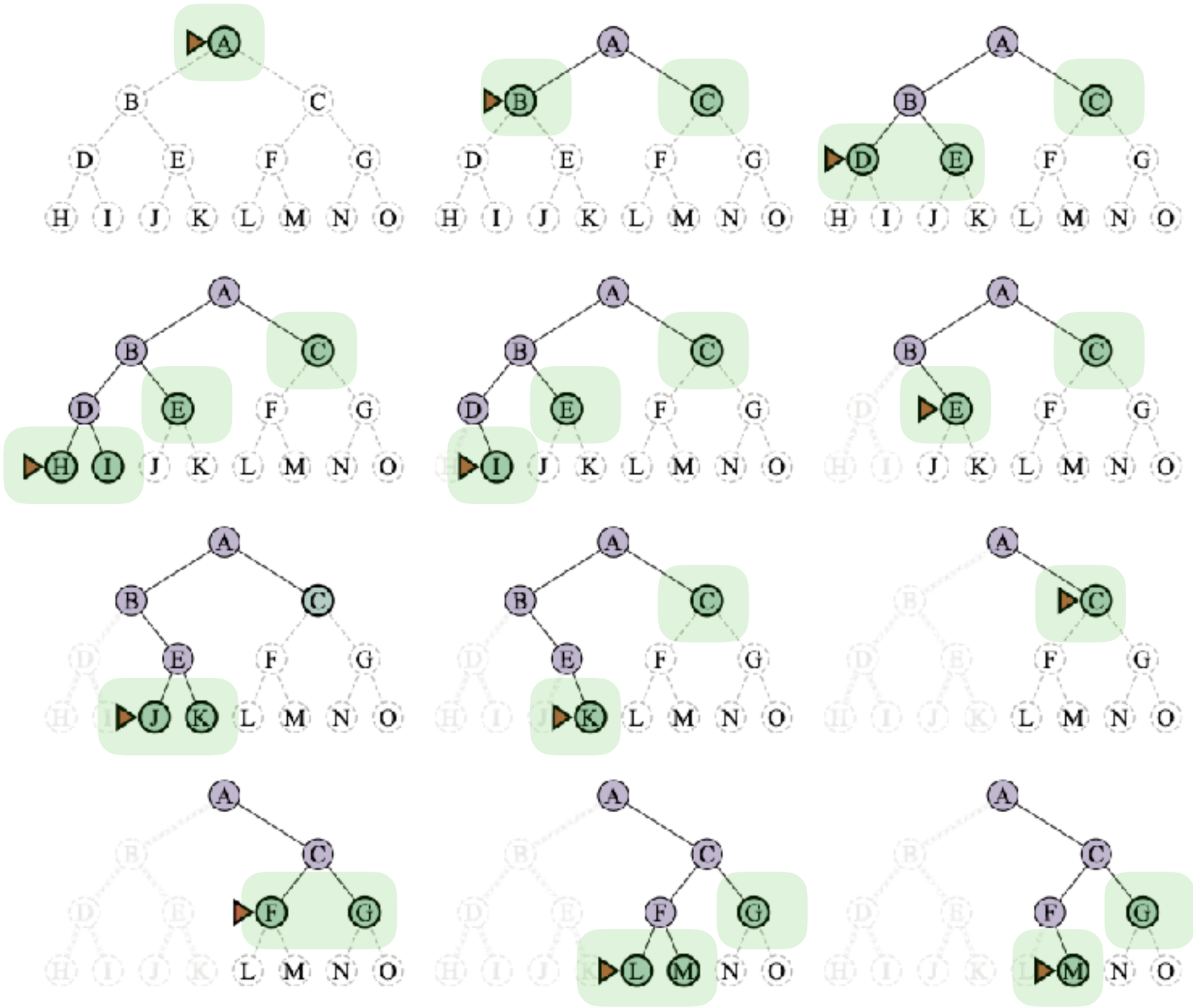
・reached （lookup table）不要

・`frontier`も小さい

DFS必要メモリ量（深さ最大値m、枝分かれ数bとすると）

→ $O(bm)$

DFSの挙動：状態Aから開始→ゴールMを探索



S.Russell & P.Norvig, Artificial Intelligence 4th Ed., Pearson, 2020
http://aima.cs.berkeley.edu/figures.pdf

深さ制限探索（depth-limited search）
反復深化（深さ優先）探索（IDS, iterative deepening search）

DFS：無限経路に陥る可能性　←　深さに制限を儲ける…深さ制限探索

制限値 ℓ を設定：深さ ℓ のノードの子ノードは考えない

計算時間：$O(b^\ell)$　　メモリ量：$O(b\ell)$

深さ制限値 ℓ をどうやって見つける？⇒全ての値を試してみる…反復深化探索（IDS）

IDS：ℓ =0→depth-limited search→ℓ =1→depth-limited search→ℓ =2 …

IDSの特徴

・DFSと同様にメモリ節約できる：解が存在⇒ $O(bd)$ ，有限状態空間＆解が存在しない⇒ $O(bm)$

（d:解の深さ、m:最大深さ）

・BFSと同様に、全ての行動のコストが等しい場合最適

状態空間が有限で閉路を持たない場合（または閉路のチェックを行う場合）⇒完全

N(IDS)：IDSにより生成されるノード数…漸近的にBFSと同様

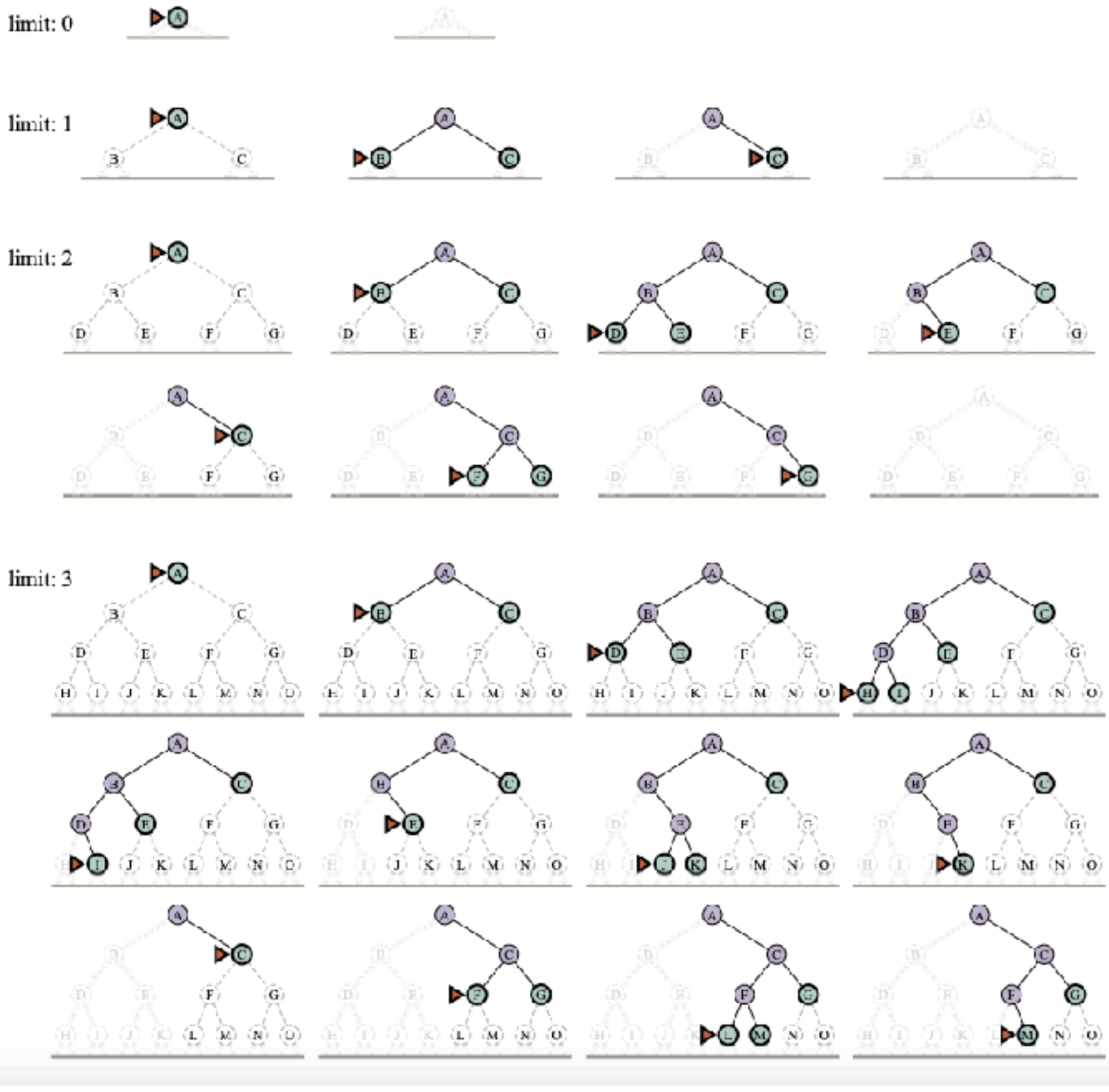
$$
\begin{aligned}
N(IDS) &= db^1 + (d-1)b^2 + (d-2)b^3 + \cdots + (d-i)b^{i+1} + \cdots b^d \\
&= d\sum_{i=1}^{d} b^i - (b^2 + 2b^3 + \cdots + ib^{i+1} + \cdots + (d-1)b^{d-2}) \\
&= d\frac{b^{d+1}-1}{b-1} - b^2\sum_{i=0}^{d-1} ib^{i-1} \\
&= O(b^d)
\end{aligned}
$$

$$
f(x) = \sum_{i=0}^{k} x_i = \frac{x^{k+1}-1}{x-1} \Rightarrow f'(x) = \sum_{i=0}^{k} ix^{i-1} = \frac{kx^{k+1} - (k+1)x^k + 1}{(x-1)^2}
$$

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution nodeまたはfailure
    for depth = 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH(problem, depth)
        if result != cutoff then return failure

function DEPTH-LIMITED-SEARCH(problem, ℓ) returns a nodeまたはfailureまたはcutoff
    frontier ← NODE(problem.INITIAL)を要素とするLIFO queue（stack）
    result ← failure
    while not IS-EMPTY(frontier) do
        node ← POP(frontier)
        if problem.IS-GOAL(node.STATE) then return node
        if DEPTH(node) > ℓ then
            result ← cutoff
        else if not IS-CYCLE(node) do
            for each child in EXPAND(problem, node) do
                childをfrontierに追加
    return result
```

S.Russell & P.Norvig, Artificial Intelligence 4th Ed., Pearson, 2020
http://aima.cs.berkeley.edu/figures.pdf



S.Russell & P.Norvig, Artificial Intelligence 4th Ed., Pearson, 2020
http://aima.cs.berkeley.edu/figures.pdf

知識なしの探索アルゴリズム性能比較

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes[1]	Yes[1,2]	No	No	Yes[1]
Optimal cost?	Yes[3]	Yes	No	No	Yes[3]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$

b：枝分かれ数（子ノードの数）
m：探索木の深さ最大値
d：最も「浅い（shallow）解」の深さ
ℓ ：深さ制限値
1：bが有限のとき、状態空間が解を持つか又は有限のとき完全
2：全ての行動コストが0< ε <= コスト
3：行動コストが全て同じ値のときコスト最適（cost-optimal）

S.Russell & P.Norvig, Artificial Intelligence 4th Ed., Pearson, 2020
http://aima.cs.berkeley.edu/figures.pdf