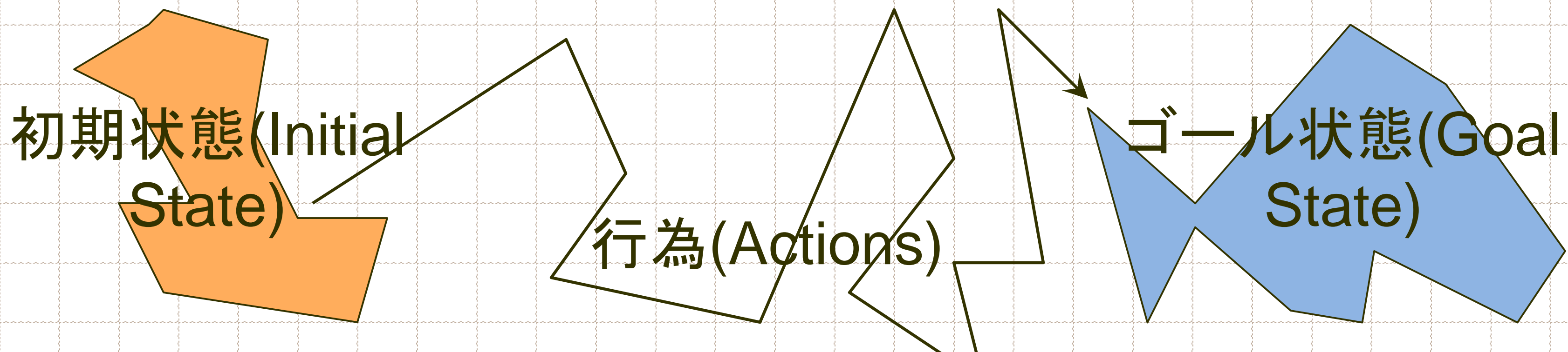
人工知能基礎
第８回

探索による問題解決(2)、実装方法、事例

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



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(uninformed 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．

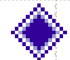
情報のない探索戦略(Uninformed search strategies)

- 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。ゴールに至るまでの情報(知識,経験,学習)がない。

- 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)

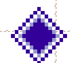
幅優先探索の性質(Properties of breadth-first search) [尺度]

- 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))



幅優先探索の実装

[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)

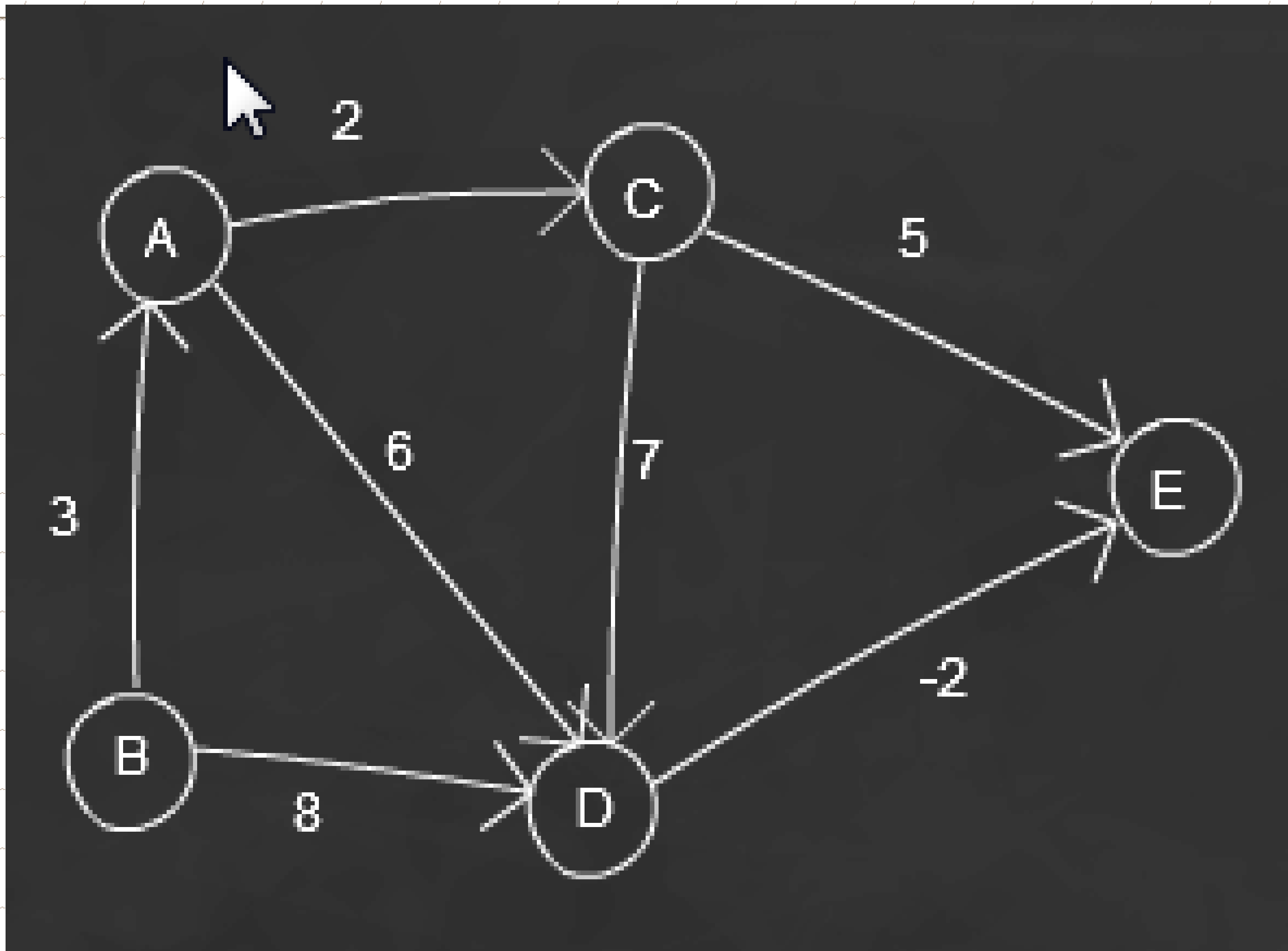
8.　P_A が空でなければ、その節点を適当な順序で OpenList の末尾に追加する。

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。

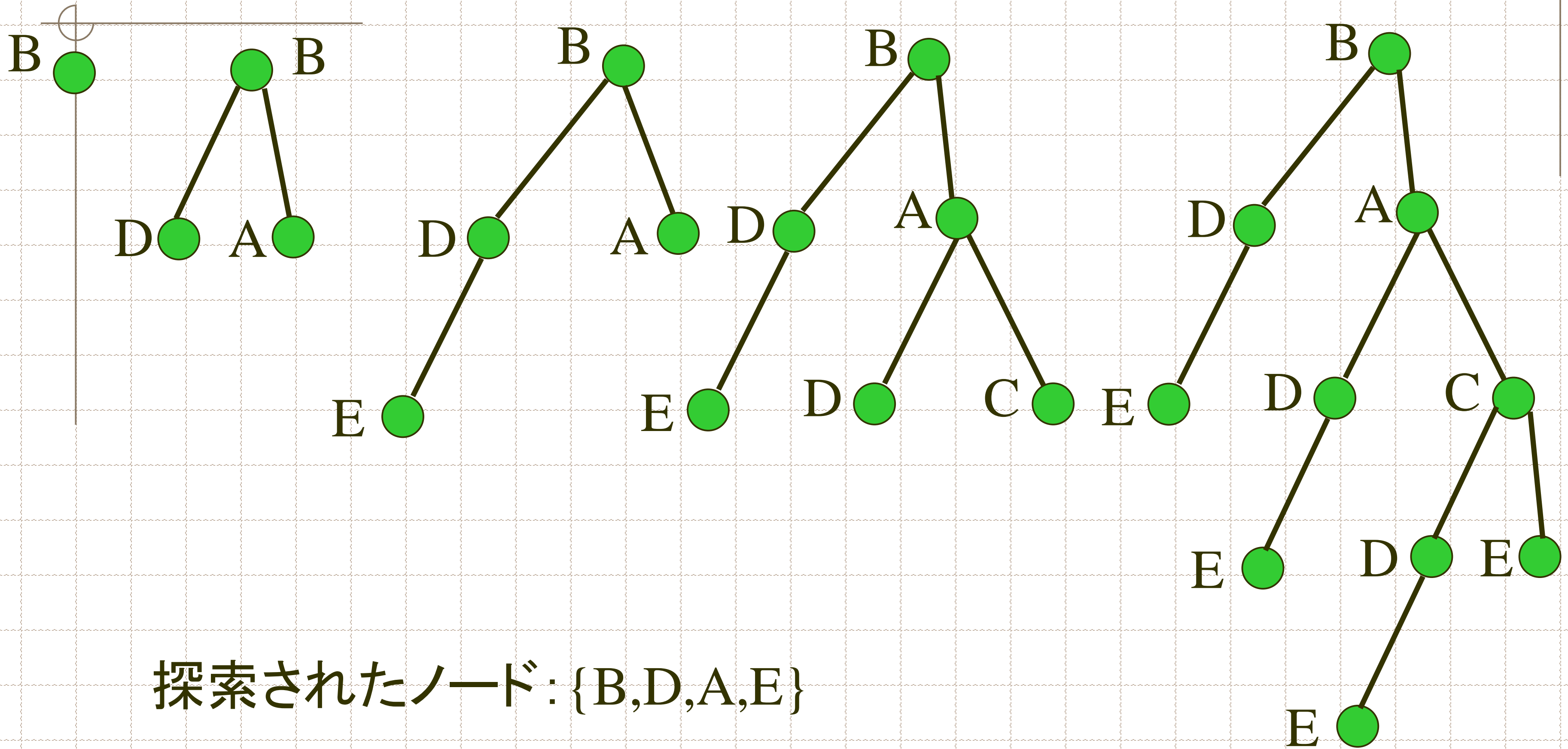
8b.　P_A に OpenList あるいは ClosedList の節点と同じ節点があれば、それを P_A から除いておく。

LOOP_END：2 に戻る。

グラフ探索の例：Bスタート,Eゴール



幅優先探索



探索されたノード: {B,D,A,E}

最適な解: S={B->D,D->E}, コスト=2(深さのd)
枝のリストはアクションのリストと同じである。

OpenListとClosedList(径路の解)

- OpenList

1. B
2. [D,A]
3. [A,E]
4. [E,D,C]
5. [D,C]

- ClosedList (パス)

1. {}
2. {B}
3. {B->D}
4. {B->D,B->A}
5. {B->D,B->A,B->D->E}

Solution={B->D->E}

プログラムの実行例(一つの方法)

```
現在の BFSパス: B
経路キュー= [['B', 'D']]
経路キュー= [['B', 'D'], ['B', 'A']]
現在の BFSパス: B->D
経路キュー= [['B', 'A'], ['B', 'D', 'E']]
現在の BFSパス: B->A
経路キュー= [['B', 'D', 'E'], ['B', 'A', 'D']]
経路キュー= [['B', 'D', 'E'], ['B', 'A', 'D'], ['B', 'A', 'C']]
現在の BFSパス: B->D->E
最短経路 (BFS): B->D->E
```

深さ優先探索の実装

[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。

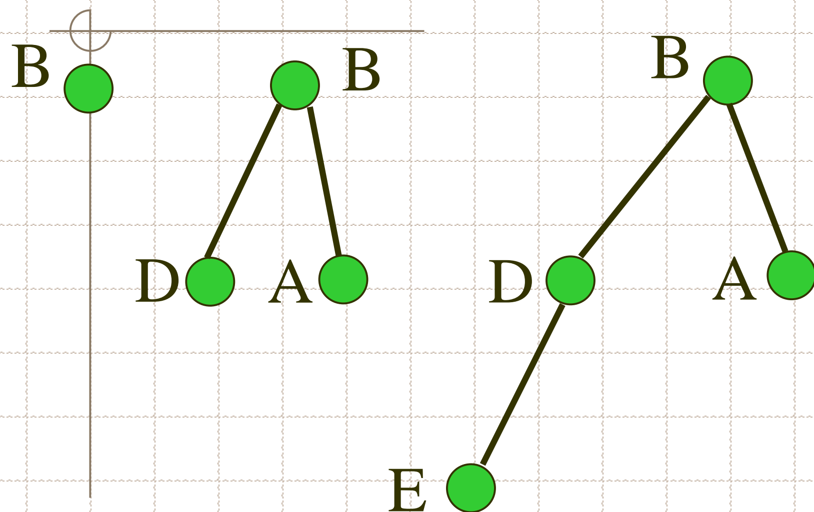
8a. このとき、追加した節点には [A] を添字として付けておく。

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。

深さ優先探索



探索されたノード：{B,D,E}

最適な解：S={B->D,D->E}, コスト=2(深さのd)

OpenListとClosedList(径路の解)

- OpenList

1. B
2. [D,A]
3. [E,A]
4. [A]

- ClosedList (パス)

1. {}
2. {B}
3. {B->D}
4. {B->D->E}

Solution={B->D->E}

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．

Romania

International boundary
County (*județ*) boundary
National capital
County (*județ*) center
Railroad
Expressway
Road

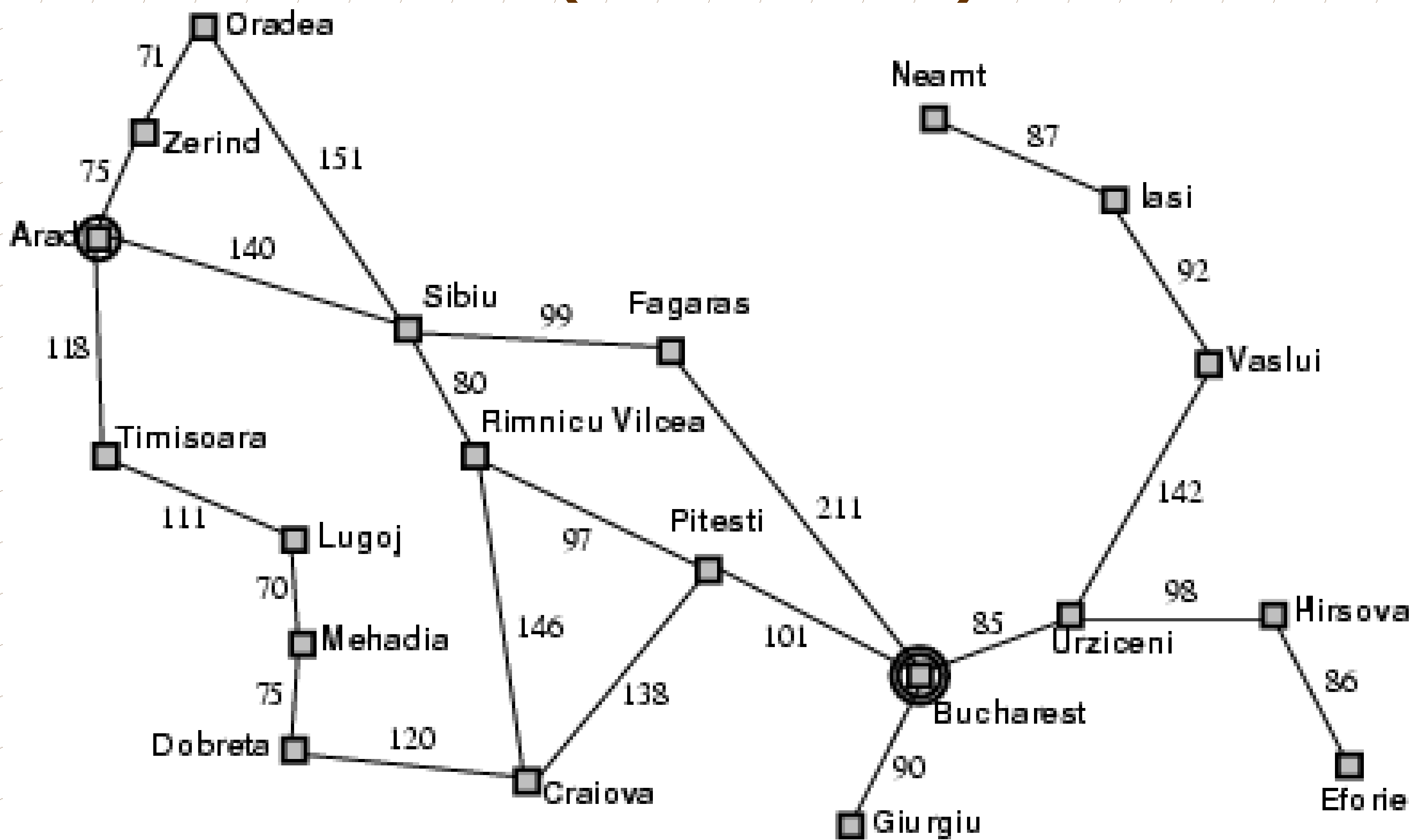
UKRAINE
SLOVAKIA
HUNGARY
MOLDOVA
UKRAINE
Serbia
BULGARIA
Black Sea
Vojvodina (autonomous province)
Kosovo (autonomous province)

Košice
Ivano-Frankivs'k
Uzhhorod
Mukachëve
Kolomyya
Kam''yanets'-Podil's'kyy
Kel'mentsi
Chernivtsi
Mohyliv-Podil's'kyy
Uman'
Miskolc
Nyíregyháza
Debrecen
Szolnok
Békéscsaba
Szeged
Bălți
Ribnița
Broshtyany
Ungheni
Chisinău
Tighina
Cahul
Artsyz
Reni
Izmayil
Kiliya
Belgrade
Smederevo
Kragujevac
Kraljevo
Kruševac
Niš
Leskovac
Priština
Uroševac
Vidin
Montana
Vratsa
Pleven
Pernik
Sofia
Ruse
Silistra
Dobrich

BOTOȘANI
Botoșani
SUCEAVA
Suceava
SATU MARE
Satu Mare
Carei
Baia Mare
MARAMUREȘ
BISTRIȚA-NĂSĂUD
Bistrița
IAȘI
Iași
NEAMȚ
Roman
Piatra-Neamț
Oradea
BIHOR
SĂLAJ
Zalău
Dej
CLUJ
Cluj-Napoca
Turda
Tîrgu Mureș
MUREȘ
HARGHITA
Miercurea-Ciuc
BACĂU
Bacău
VASLUI
Vaslui
Bîrlad
ARAD
Arad
ALBA
Alba Iulia
Mediaș
COVASNA
Sfîntu Gheorghe
VRANCEA
Focșani
GALAȚI
Galați
Timișoara
TIMIȘ
Deva
Hunedoara
HUNEDOARA
SIBIU
Sibiu
BRAȘOV
Brașov
BUZĂU
Buzău
Brăila
BRĂILA
Tulcea
TULCEA
Reșița
CARAȘ-SEVERIN
Petroșani
VÎLCEA
Rîmnicu Vîlcea
Tîrgu-Jiu
GORJ
ARGEȘ
Pitești
PRAHOVA
Ploiești
Tîrgoviște
DÎMBOVIȚA
IALOMIȚA
Slobozia
Drobeta-Turnu Severin
MEHEDINȚI
BUCUREȘTI
Bucharest
CĂLĂRAȘI
Călărași
CONSTANȚA
Constanța
Craiova
DOLJ
Slatina
OLT
GIURGIU
Giurgiu
TELEORMAN
Alexandria
Calafat

Danube
Prut
Dnistr
Tisa
Nistru
Olt
Velika Morava
Južna Morava

例: ロマニア(Romania)

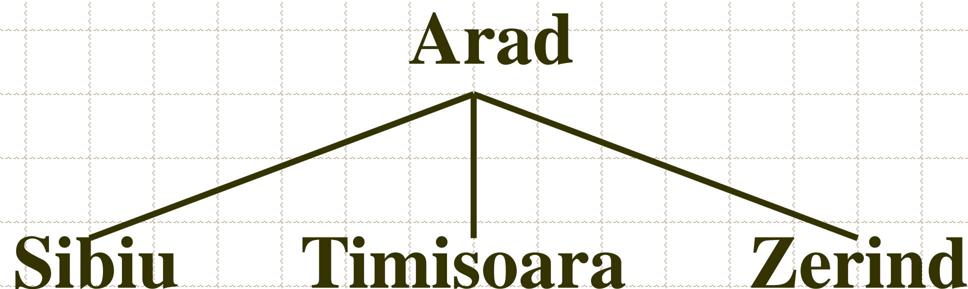


解の探索(3)　部分探索木の例

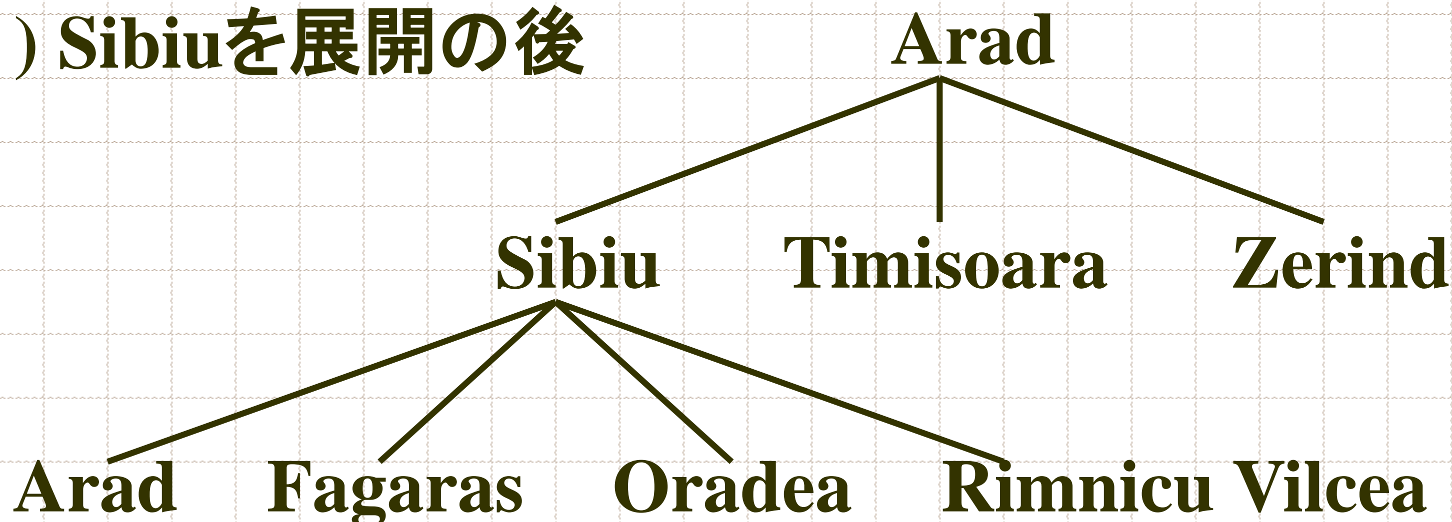
(a) 初期状態

Arad

(b) 展開の後



(c) Sibiuを展開の後



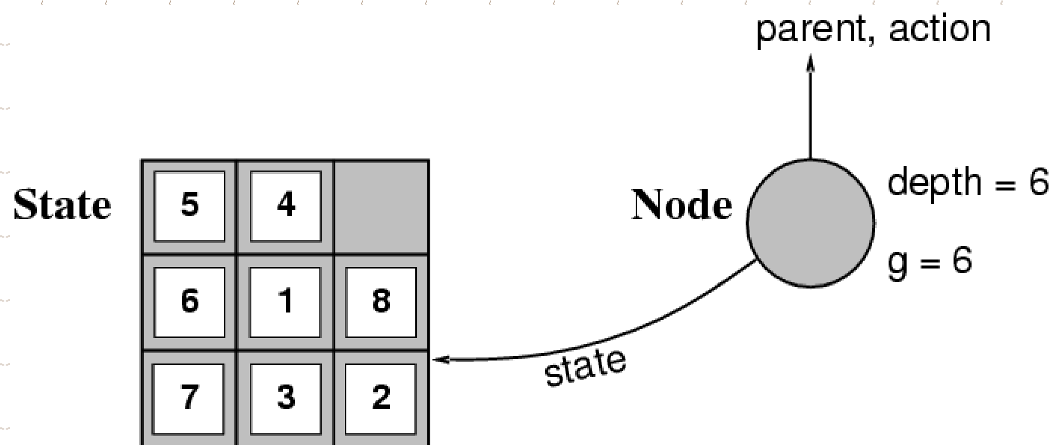
解の探索(4)　探索木のためのデータ構造

- 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

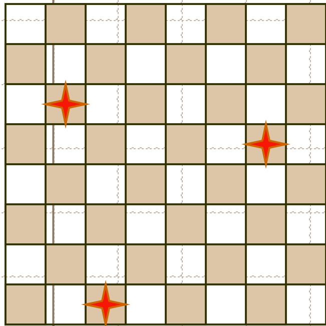
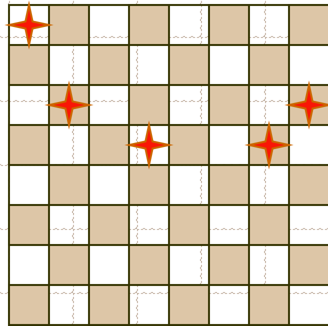
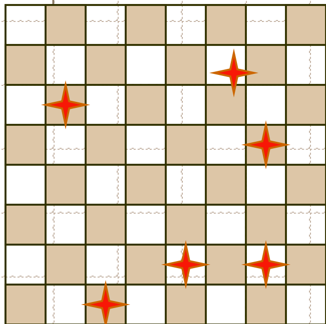
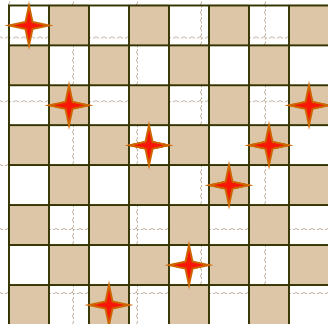
実装：状態対ノード(Implementation: states vs. nodes)

- 状態は物理構成(表現)である
- ノードは、状態、親ノード、アクション、パスコストg(x)、深さを含む検索木の一部を構成するデータ構造である。



- 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。

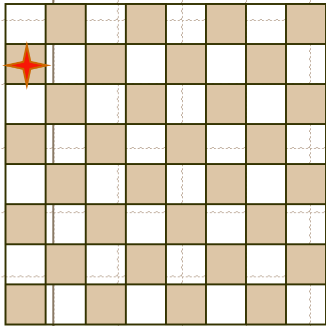
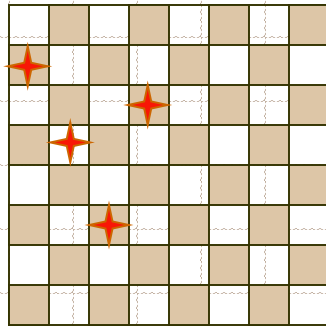
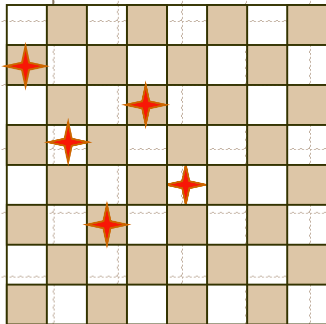
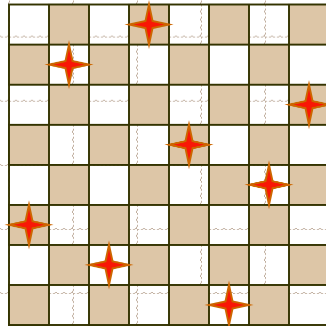
8クイーン問題定式化 #1

- **状態**: 0, 1, 2, …, 8 クイーンの盤上への任意の配置
- **初期状態**: 0 クイーン(一つも置かれていない盤)
- **後者関数**:クイーンを一つ、空いたマスに置く
- **コスト**: なし
- **ゴール検査**: 8クイーンが盤上にあり、どれも攻撃しない。

→ ~ 64×63×…×57 ~ 3×10^{14} 状態数＝ノード数

8クイーン問題定式化#2

- **状態**: 一番左からn列(0=<n=<8)目までに、n個のクイーンを1列に一つずつ置き、互いに攻撃しない配列が状態である。
- **初期状態**: 0 クイーン(一つも置かれていない盤)
- **後者関数**:クイーンが置かれていない最も左の列で、他のどのクイーンにも攻撃されない任意のマス一つにクイーンを加える。
- **コスト**: なし
- **ゴール検査**: 8クイーンが盤上にあり、どれも攻撃しない。

→ 2,057 状態数＝ノード数

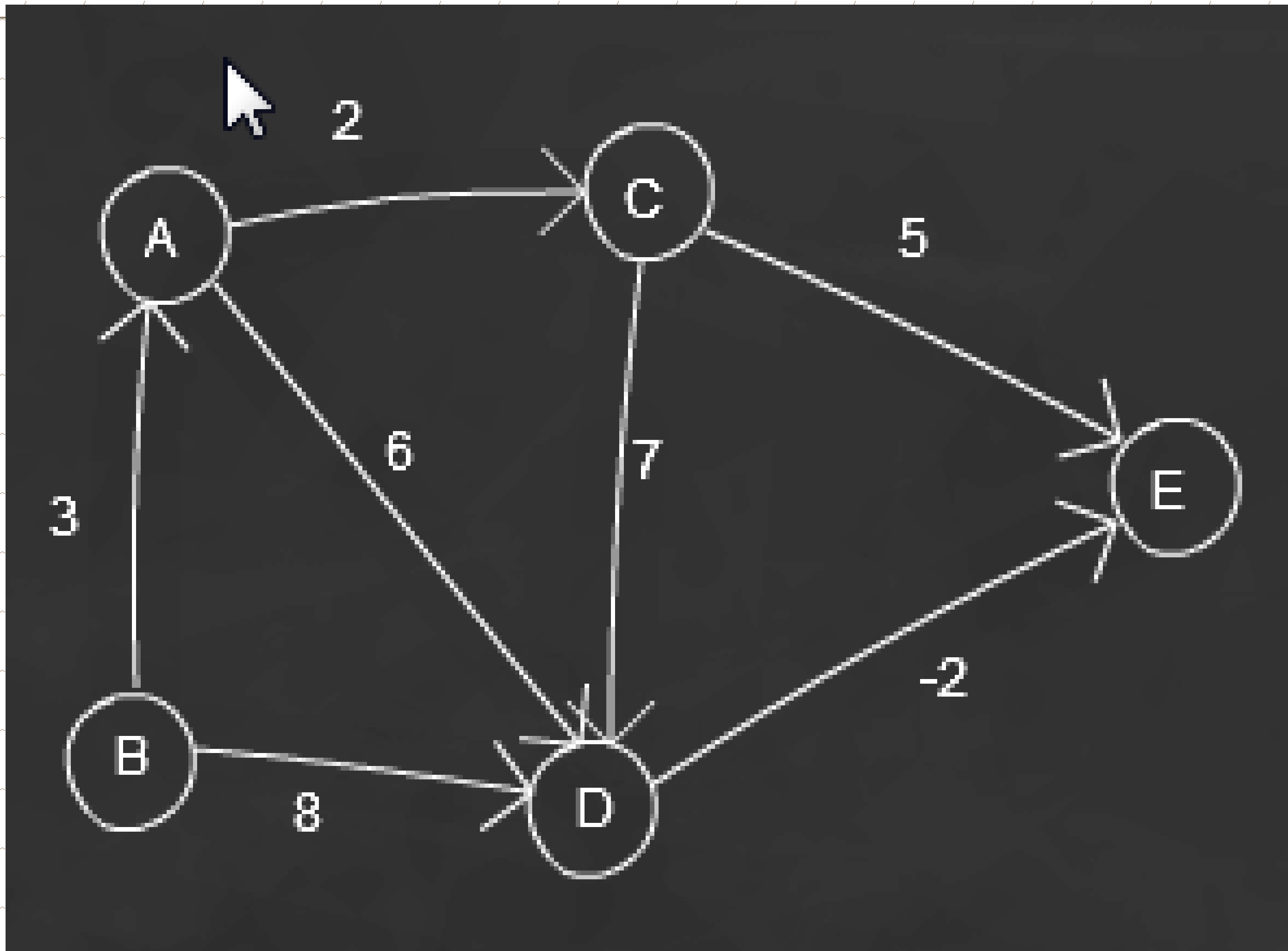
n-クイーン問題

- 解：ゴールノードである、パスではない。
- 状態空間のサイズ:
  - 8-クイーン → 2,057
  - 100-クイーン → 10^{52}
- 通常の探索アルゴリズムでは計算厳しい

均一コスト探索(Uniform-cost search)

- 経路コストg(n)によってコストが最も少ないノードを常に展開させることによって幅優先戦略を修正したものである。
- 実装:
  - *縁(fringe)* = 経路コストによるキュー
- 完全(Complete)? cost ≥ εのステップ数であれば、完全です
- 時間(Time)? *g* ≤ costである時の最適解がある, $O(b^{ceiling(C^*/\varepsilon)})$ *C** は最適解のコスト（ステップ数）
- メモリ(Space)? *g* ≤ costの最適解のステップ数, $O(b^{ceiling(C^*/\varepsilon)})$
- 最適（Optimal）? *g(n)による、ノードの数が増える。最適である*

グラフ探索の例：Bスタート,Eゴール



OpenListとClosedList(径路の解)

- OpenList

1. (B,0)
2. [(A,3),(D,8)]
3. [(C,5),(D,8)]
4. [(D,8),(E,10)]
5. [(E,6)]

- ClosedList (パス)

1. {}
2. {(B,0)}
3. {B->A, g(A)=3}
4. {B->A->C, g(C)=5}
5. {B->D, g(D)=8}
6. {B->D->E, g(E)=6}

Solution={B->D->E}

深さ制限探索(Depth-limited search)

= depth-first search with depth limit *l*,
i.e., nodes at depth *l* have no successors

```
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
```

深さ制限探索

- 深さ優先探索の欠点を補うために，探索の深さに一定の制限を置く方法.
- 深さの限界は，ある場合には，問題の知識に基づいて選ぶことが出来る. たとえば，ルーマニアの地図では，20都市しかないので，もし解があれば，それは最長でも長さが19でなければならない.
- 深さ制限探索は，つぎの反復深化探索で，繰り返し利用される.
- 深さ制限探索では、最後のノードを展開して、それが解でなかったら、探索は失敗する. そのため、時間計算量は、幅優先探索の時間計算量＋1である.

反復深化探索

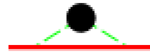
- 反復深化探索は，最初は深さ0，次に深さ1，さらに深さ2，というようにすべての可能な深さを試すことによって，最良の深さ制限を選ぶ問題を避ける戦略である．
- 反復深化探索は，幅優先探索と同じように，最適で完全であり，かつ，深さ優先探索と同様のメモリの消費量である．
- 反復深化探索の時間計算量は，分岐度bが10のとき，幅優先探索より11%多いだけである．分岐度が2のときでも，約2倍である．

反復深化探索(Iterative deepening search)

```
function ITERATIVE-DEEPENING-SEARCH( problem) returns a solution, or fail-
ure
    inputs: problem, a problem
    for depth ← 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH( problem, depth)
        if result ≠ cutoff then return result
```

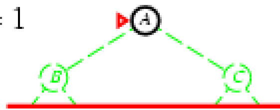
反復深化探索(Iterative deepening search) / =0

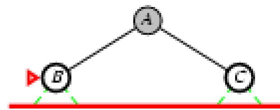
Limit = 0

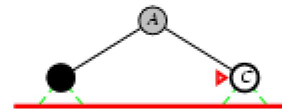


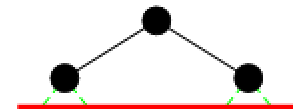
反復深化探索(Iterative deepening search) / =1

Limit = 1

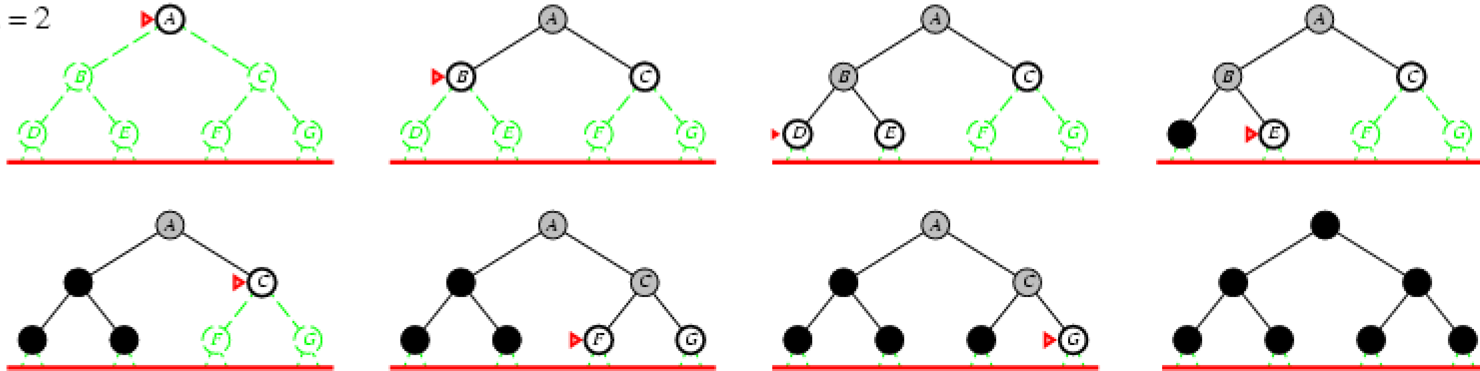




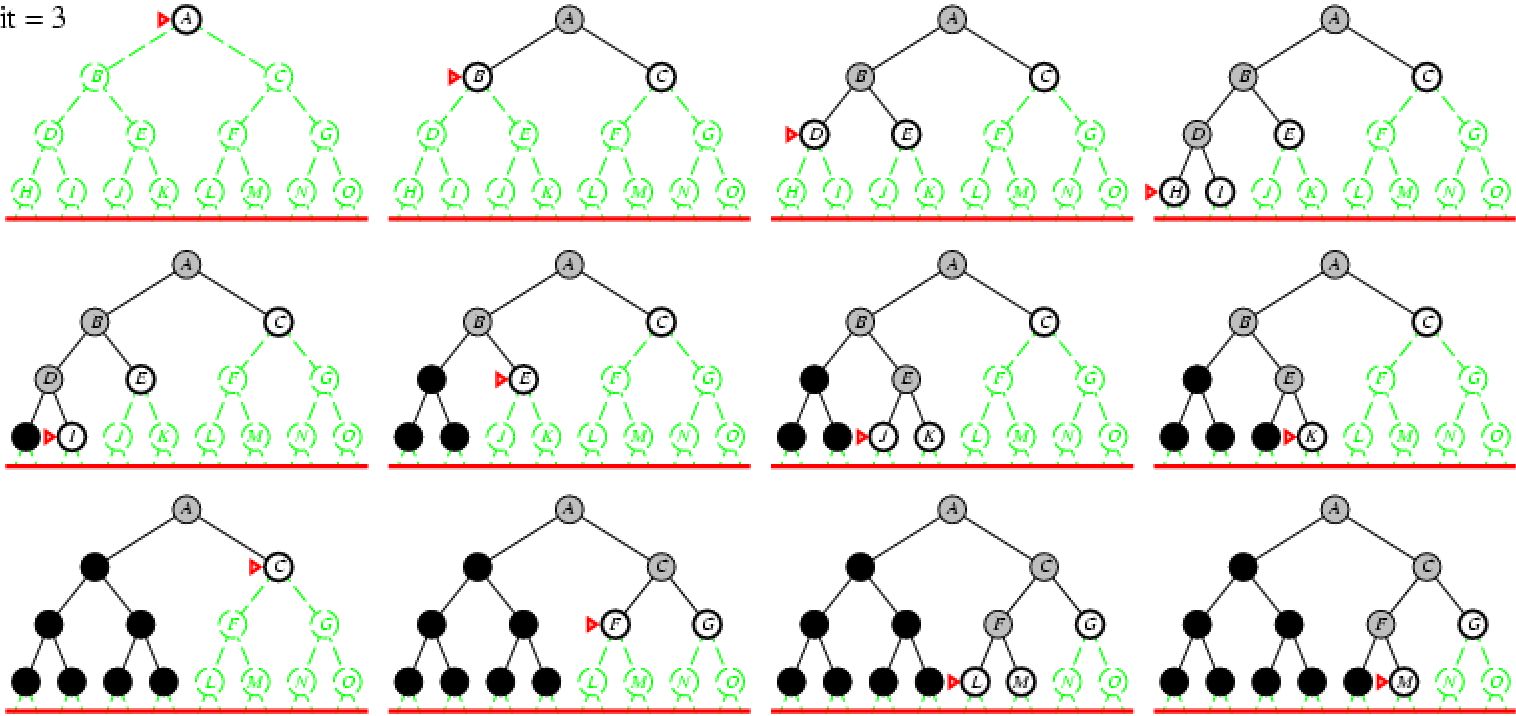




反復深化探索(Iterative deepening search) *l* =2



反復深化探索(Iterative deepening search) $l=3$



反復深化探索(Iterative deepening search)

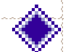
- 分岐係数bと深さdまでの深さ制限検索で生成されたノードの数:

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

- 分岐係数bと深さdへの反復深化探索で生成されたノードの数:

$N_{IDS} = (d+1)b^0 + d\ b\char`^{1} + (d-1)b\char`^{2} + \ldots + 3b^{d-2} + 2b^{d-1} + 1b^d$

- For $b = 10$, $d = 5$,



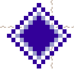
  - N_{DLS} = 1 + 10 + 100 + 1,000 + 10,000 + 100,000 = 111,111

  - N_{IDS} = 6 + 50 + 400 + 3,000 + 20,000 + 100,000 = 123,456

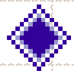
- DLS Overhead = (123,456 - 111,111)/111,111 = 11%

反復深化探索の性質(Properties of iterative deepening search)

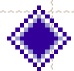
- 完全(Complete)? Yes



- 時間計算量(Time)? $(d+1)b^0 + d\,b^1 + (d-1)b^2 + \dots + b^d = O(b^d)$



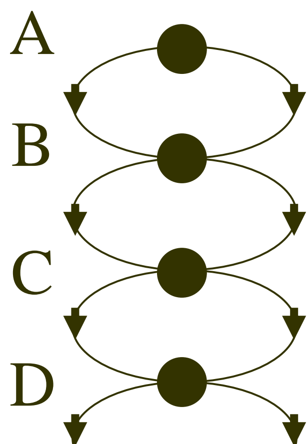
- メモリ計算量(Space)? $O(bd)$

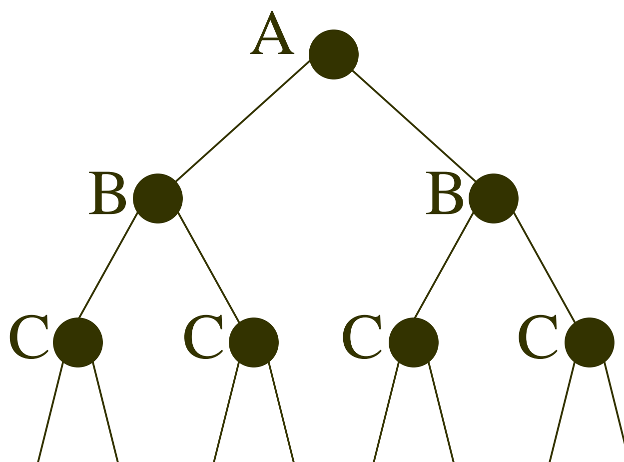


- 最適(Optimal)? Yes, if step cost = 1

繰り返し状態の回避

- 複数の経路を通して，ある状態に到達できるとき，同じ状態を繰り返し展開してしまうのを防ぎたい．
- 繰り返し展開を防ぐ3つの方法：
  - 親ノードに戻らない．
  - ルートノードからの経路上のノードに戻らない．
  - 以前に生成したどんなノードにも戻らない．





探索戦略の比較 (Comparison of algorithms)

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

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．

まとめ(つづき)

- **幅優先探索**は，探索木の浅いノード順に展開する．それは，完全で，(通常)最適であるが，時間，空間計算量は，膨大である．
- **深さ優先探索**は，探索木の最も深いノードを最初に展開する．それは，完全でも最適でもない．時間計算量は膨大であるが，空間計算量は小さい．
- **深さ制限探索**は，深さ優先探索での深さの限界を設ける．
- **反復深化探索**は，ゴールが見つかるまで限界を増やしながら深さ制限探索を呼び出す．それは，幅優先探索と深さ優先探索の長所を併せ持つ．