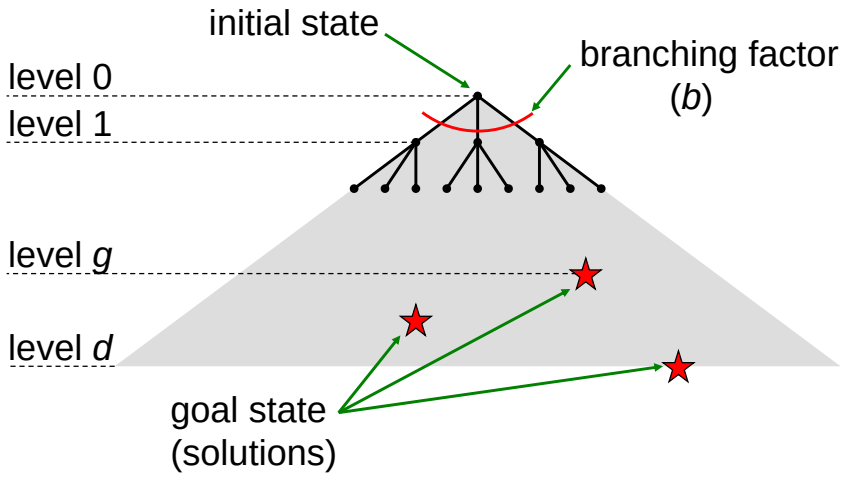
2007年度 計算知能論A
しらみつぶし探索

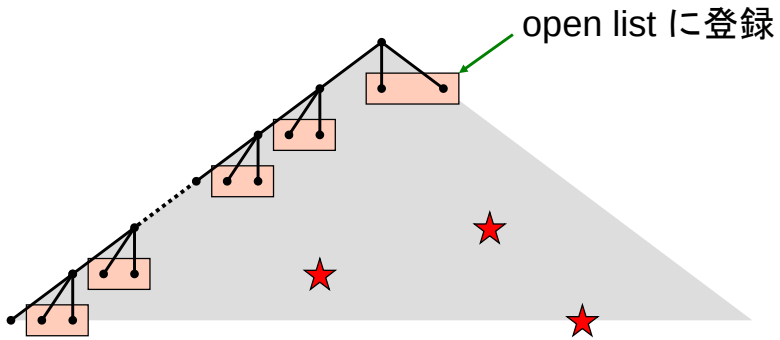
2007年5月21日

上田 和紀
早稲田大学理工学部CS学科

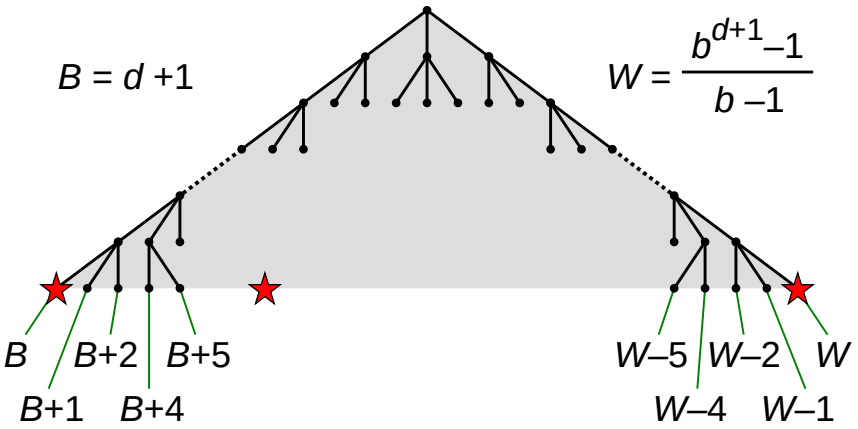
探索木 (search tree)



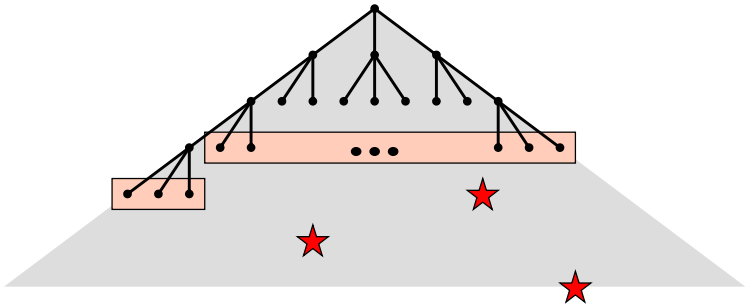
深さ優先探索（縦型探索, depth-first search）



深さ優先探索（縦型探索, depth-first search）



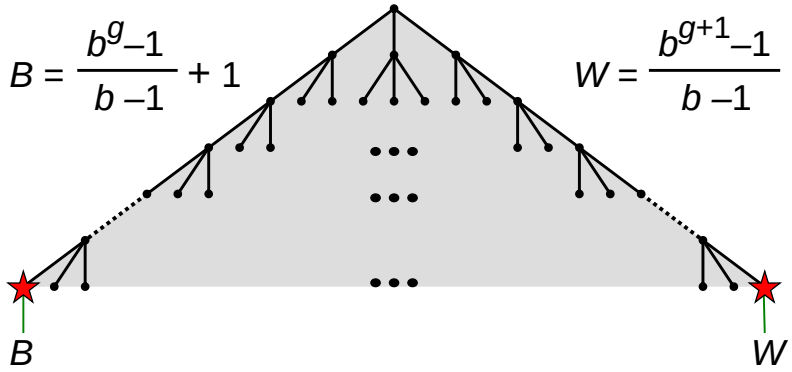
幅優先探索（横型探索, breadth-first search）



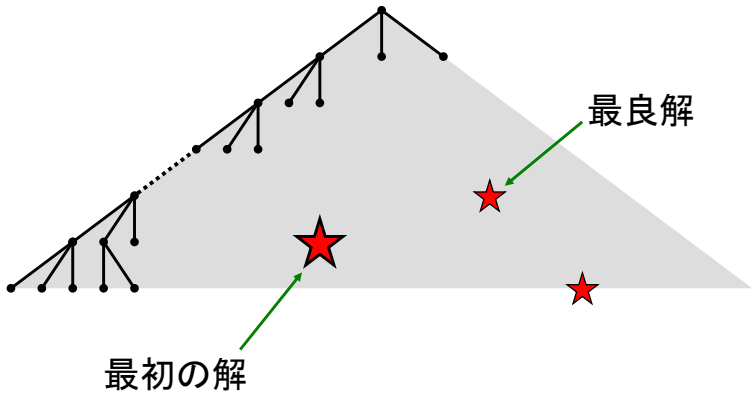
: *exponential w.r.t. g*
(open list の大きさ)

幅優先探索（横型探索, breadth-first search）

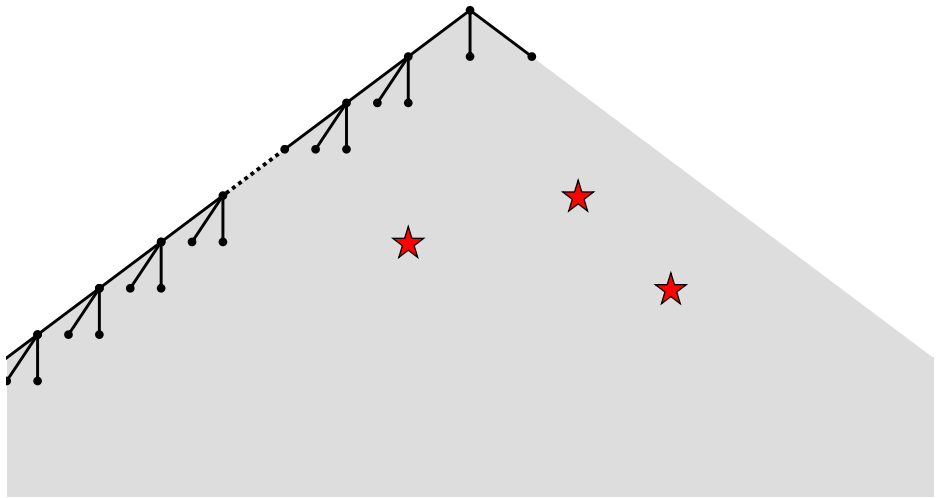
$$B = \frac{b^g - 1}{b - 1} + 1 \qquad W = \frac{b^{g+1} - 1}{b - 1}$$



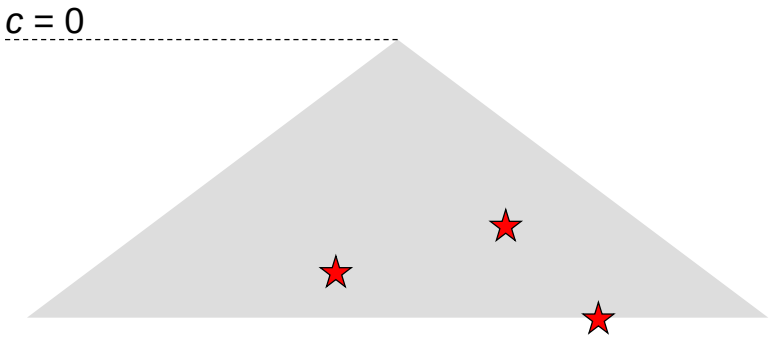
深さ優先探索 — 問題点 1



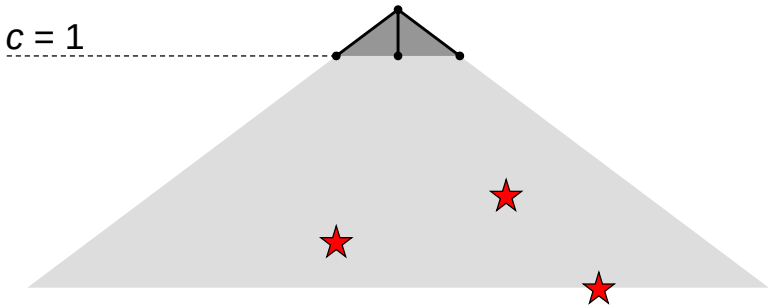
深さ優先探索 — 問題点 2



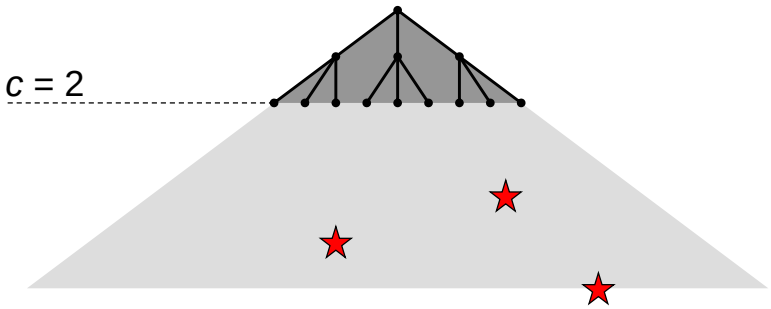
反復深化 (iterative deepening)



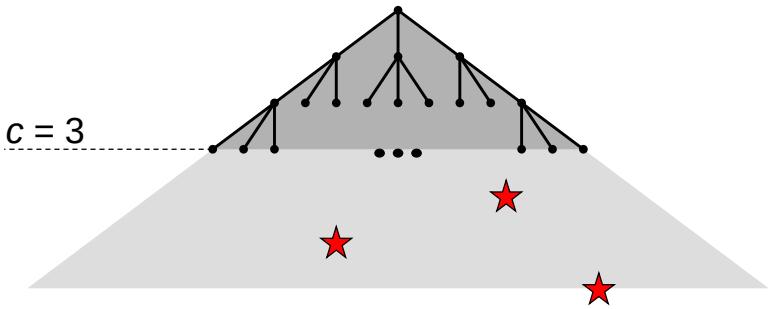
反復深化 (iterative deepening)



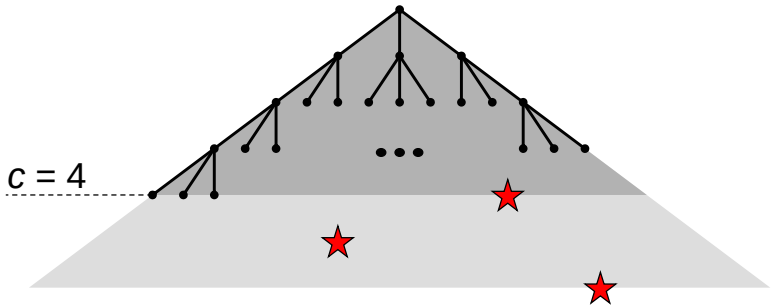
反復深化 (iterative deepening)



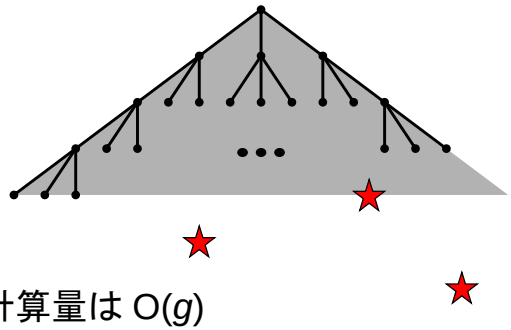
反復深化 (iterative deepening)



反復深化 (iterative deepening)

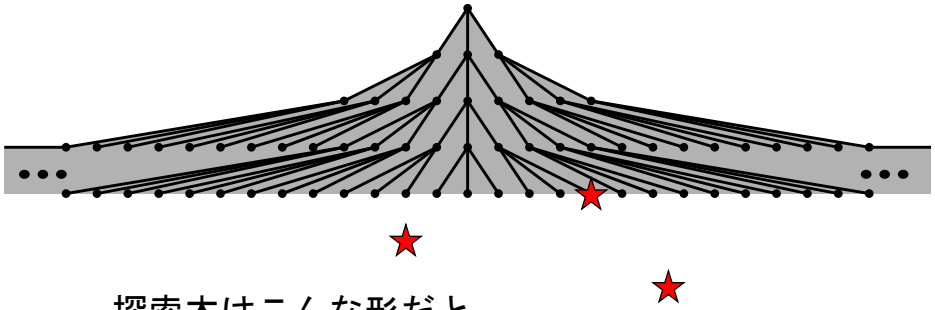


反復深化 (iterative deepening)



空間計算量は O(*g*)
幅優先に比べて時間がかからないのか？

反復深化 (iterative deepening)



探索木はこんな形だと
考えるほうが正しい.

反復深化 (iterative deepening)

- 幅優先探索との手間（訪れる節点数）の比

b	反復深化／幅優先
2	2
3	1.5
5	1.25
10	1.11
25	1.04

- アルゴリズムの空間局所性次第でコストは逆転

反復深化 (iterative deepening)

- 反復深化探索は「最適」なしらみつぶし探索
  - いかなるしらみつぶし探索アルゴリズムも，反復深化探索の空間計算量と時間計算量を改善することはできない。
  - 理由
    - 空間計算量：
    - 時間計算量：

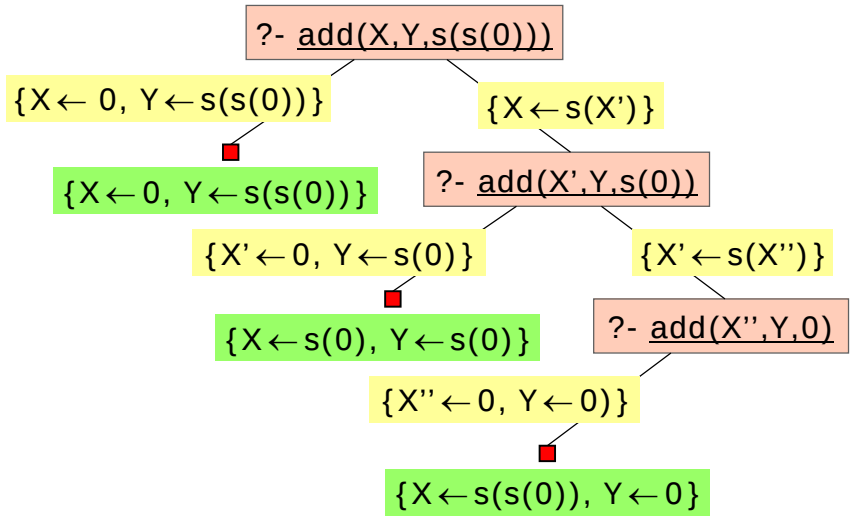
メタインタプリタ

- 言語 $\mathcal{L}$ で書いた言語 $\mathcal{L}$ のインタプリタもしくは Virtual Machine (VM)
  - $\mathcal{L}$ = Lisp, Prolog, . . .（記号処理言語）
  - プログラムとデータがほぼ同じ形式なので作りやすい
  - でも，Prologプログラムは論理式，Prologプログラムが扱うデータは項
  - Prologプログラムをデータとして表現する方法をまず決める

メタインタプリタにかけるプログラム

もとの表現	メタインタプリタ用表現
論理演算子 ，	関数記号 & (infix)
論理演算子 :-	述語記号 <= (infix)
節（規則） h :- b_1, ..., b_n	原子論理式 h <= b_1 & ... & b_n
節（事実） h	原子論理式 h <= true

SLD木（復習）



Prologの実行はSLD木を縦型探索

- 問題空間
  - 状態 ＝ ゴール（質問），すなわち述語呼出しの並び
  - 操作 ＝ ゴール中の最左の述語呼出しを，規則（プログラム節）を用いて展開すること
- 初期状態 ＝ 初期ゴール
- 目標状態 ＝ 空ゴール (true)

Prolog のバニラメタインタプリタ

- “small-step” version

```
solve(A) :– solve(A,true).
solve(A, A).
solve(A0,A) :– reduce(A0,A1), solve(A1,A).

reduce((true&B), C) :– reduce(B,C).
reduce((A0&B), (A&B)) :– reduce(A0,A).
reduce(A,true) :– builtin(A), call(A).
reduce(A,B) :– (A <= B).

builtin(_ is _).  builtin(dif(_, _)).  . . .
```


反復深化メタインタプリタ

```
solve(A) :– solve(A,true).
solve(A, A).
solve(A0,A) :– solve(A0,A1), reduce(A1,A).

reduce((true&B), C) :– reduce(B,C).
reduce((A0&B), (A&B)) :– reduce(A0,A).
reduce(A,true) :– builtin(A), call(A).
reduce(A,B) :– (A <= B).

builtin(_ is _).  builtin(dif(_, _)).  . . .
```


打切り機構の導入


```
solve(A,N) :- solve(A,true,N,C).
solve(A, A,N, N).
solve(A0,A,N0,N) :–
    N0>0, N1 is N0–1,
    solve(A0,A1,N1,N),
    reduce(A1,A).
```

打切り機構の導入

カウンタ　　使い残し

```
solve_id(A,N) :- solve_id(A,true,N,C).
solve_id(A, A,N, N).
solve_id(A0,A,N0,N) :-
    N0>0, N1 is N0–1,
    solve_id(A0,A1,N1,N),
    reduce(A1,A).
```


メタインタプリタの得失

- バニラメタインタプリタ自体には付加価値はない.
- しかし，少し手を加えることで機能が容易に変更・増強できる
  - 例１: 実行戦略の変更（例：反復深化）
  - 例２: 証明過程の表示（＝ 理由の説明）
    - cf. エキスパートシステム
  - 例３: 述語呼出しの遅延
    - cf. 仮説推論
- １桁程度の性能低下

Prologのバニラメタインタプリタ (2)

- “big-step” version

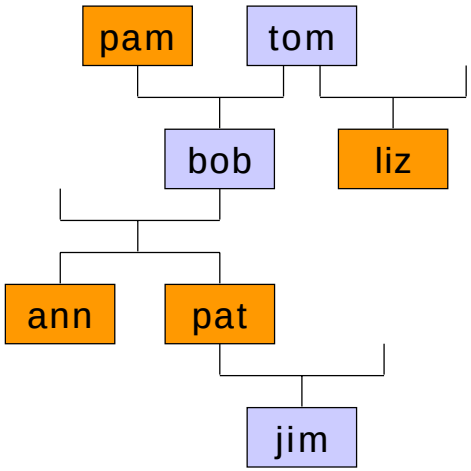
```
solve(true).
solve(A & B) :– solve(A), solve(B).
solve(A) :– builtin(A), call(A).
solve(A) :– (A <= B), solve(B).
builtin(_ is _).   builtin(dif(_, _)).  . . .
```


メタインタプリタの他の応用

- ゴールが成功した理由（＝ 証明過程）を表示するメタインタプリタ
  - cf. エキスパートシステム

```
solve(true, true).
solve((A & B), and(PA,PB)) :–
      solve(A, PA), solve(B, PB).
solve(A, known(A)) :– builtin(A), call(A).
solve(A, because(A,PB)) :–
      (A <= B), solve(B, PB).
```

家族関係，系図の登録 (family.pl)




```
is_parent_of(pam, bob).
is_parent_of(tom, bob).
is_parent_of(tom, liz).
is_parent_of(bob, ann).
is_parent_of(bob, pat).
is_parent_of(pat, jim).

female(pam).
female(liz).
female(pat).
female(ann).
male(tom).
male(bob).
male(jim).
```


メタインタプリタの他の応用

- 述語呼出しの遅延機能をもつメタインタプリタ
  - delay 宣言のある呼出しには手をつけず，残りの部分を実行
  - 最初のゴールが成功するための十分条件（仮説）を計算
    - ➔ 仮説推論，アブダクション (abduction)

メタインタプリタの他の応用

- 述語呼出しの遅延機能をもつメタインタプリタ

```
dsolve(true, D,D).
dsolve(A & B, D0,D) :–
      dsolve(A, D0,D1), dsolve(B, D1,D).
dsolve(A, [A|D],D) :- delay(A).
dsolve(A, D,D) :– builtin(A), call(A).
dsolve(A, D0,D) :–
      (A <= B), dsolve(B, D0,D).
```