アルゴリズム論（第7回）

2003.11.26
櫻井　彰人
algorithm@soft.ae.keio.ac.jp
http://www.sakurai.comp.ae.keio.ac.jp/

きょうの講義概要

- 探索
  - 探索のためのデータ構造
  - 2分木を使う探索
  - 縦型探索と横型探索
  - MIN-MAX探索
  - ハッシュ法
- 原理、アルゴリズム、計算量、正当性、実際問題での利用

4 探索

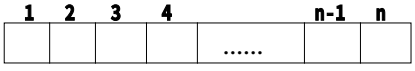
- 探索（Search）
  - 系統的にまたは試行錯誤して解を探す
    - » 解：初期状態から目標状態へ
  - 蓄えられた情報を効率よく探し出す
    - » 情報：キーのついたレコード
      - 名前＋電話番号、英単語＋説明、キーワード＋ポインタ、その他
  - 評価：見つかるまでの計算量（cf. 2分探索）

探索アルゴリズム

- 解の探索（Winston, P. H.: Artificial Intelligence, Addison-Wesley, Pub., 1984, p. 88より）
  - とにかく解を
    - » 深さ優先探索、山登り法、幅優先探索、Beam、Best-first
  - 最適解
    - » 分岐限定法、ダイナミックプログラミング、A*アルゴリズム
  - ゲーム
    - » ミニマックス法、$\alpha-\beta$ 法、ヒューリスティックな枝がり、Progressive deepening

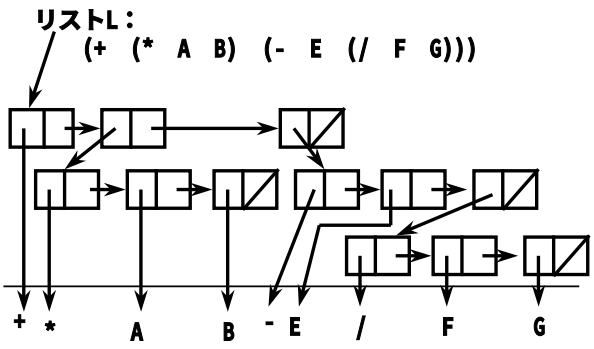
4.1 探索のためのデータ構造

- 配列（array）
  - 同一タイプのデータが1次元または2次元に並んだもの



- リスト（list）
  - データの組が一定の順序で集まったもの（入れ子構造を許す）

探索のためのデータ構造（2）



探索のためのデータ構造（3）

■表（table）

–複数個の欄（field）で１つのレコード（record）

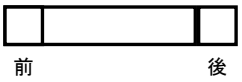
–各欄に名前がつき、レコードごとに欄に対応する値

–複数個のレコードの集合が表

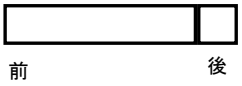
	欄1	欄2	欄3	…	欄n
1					
2					
…					
m					

探索のためのデータ構造（4）

◆ キュー（queue、待ち行列）：先入れ先出し
– FIFO: First In First Out

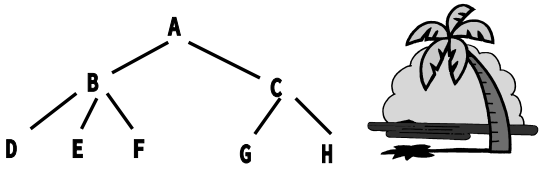


◆ スタック（stack、棚）：後入れ先出し
– LIFO: Last In First Out, FILO: First In Last Out



探索のためのデータ構造（5）

■ 木（tree）
– 階層的な構造で情報を表現

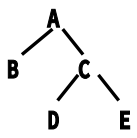


4.2 2分木の巡回

■ 2分木の節に蓄えられた情報の探索（2分木の再帰的巡回アルゴリズム）
– In-Order
» (1)左の部分木をSymmetric Order で巡回
» (2)根を巡回
» (3)右の部分木を Symmetric Order で巡回
– Preorder
» (1)根を巡回
» (2)左の部分木を Preorder で巡回
» (3)右の部分木を Preorder で巡回

2分木の巡回（2）

■ 2分木の巡回（つづき）
– Postorder
» (1)左の部分木を Postorder で巡回
» (2)右の部分木を Postorder で巡回
» (3)根を巡回



In-Order: B, A, D, C, E
Pre: A, B, C, D, E
Post: B, D, E, C, A

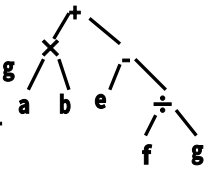
アルゴリズム4.1(Postorder)

◆ 入力：2分木（節ごとに子供の節へのポインタ）
◆ 出力：Postorderでの巡回結果
– (1)根の節をスタックに入れる（プッシュダウン）
– (2)スタックが空なら終了
– (3)スタックの一番上にある節pに調べるべき子供の節がなければ、pをスタックから取り出し（ポップアップ）、出力エリアに送って(2)へ
– (4)pに調べていない子供の節cがあれば、このcをプッシュダウンして(3)へ

数式の木の巡回とポーランド記法

■ 数式：根を演算子、左右の部分木を被演算子とした2分木
- Preorder がポーランド記法、Postorder が逆ポーランド記法
- ×と÷の方が＋と－より優先度が高い

例：a×b + (e - f÷g)
（1）Preorderによる巡回
+, ×, a, b, -, e, ÷, f, g
（2）Postorederによる巡回
a, b, ×, e, f, g, ÷, -, +



アルゴリズム4.2（逆ポーランド記法）

◆ 入力：カッコと単項演算子を含まない数式
◆ 出力：逆ポーランド記法の文字列
- （1）i ←1
- （2）i番目の入力を調べる
  - » (2.1) 変数なら出力エリアに送り、i を1増やして（2）へ
  - » (2.2) 入力の終了であれば、スタックにある演算子をポップアップしながら出力エリアに送って終了
  - » (2.3) 演算子のときは、
    - (2.3.1) スタックが空またはスタックの一番上にある演算子より優先度が高ければプッシュダウンし、i を1増やして（2）へ
    - (2.3.2) スタックの一番上の演算子の優先度より低いか同じときは、ポップアップして出力エリアに送り、この条件が成り立つ間この操作を繰り返して(2.3.1)へ

ポーランド記法と英語
逆ポーランド記法と日本語

例：a×b + (e - f÷g)
(1) Preorder + × a b – e ÷ f g
sum of “product of a and b” and “difference of e and division of f by g”
(2) Postoreder a b × e f g ÷ - +
a と b との積と e と f と g の商の差の和
(a と b との積)と (e と f と g の商の差)の和

4.3 木を使う探索

◆ 問題全体が見通せない
◆ 解析的あるいはアルゴリズム的解法がわからない
◆ 特定の状態のすぐ次に続く状態は分かる

⇩

試行錯誤で問題解決
探索のための木（探索木）を作りながら
根が問題の最初の状態

木による探索の例（8-パズル）

◆ はじめの状態

2	8	3
1	6	4
7		5

◆ 最後（目標）の状態

1	2	3
8		4
7	6	5

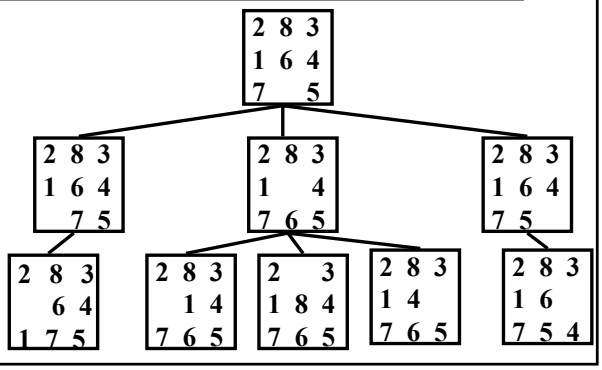
◆ はじめの状態のすぐ次の状態

2	8	3
1	6	4
	7	5

2	8	3
1		4
7	6	5

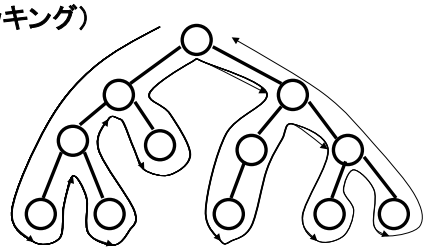
2	8	3
1	6	4
7	5	

8-パズルの探索木（一部）



4.3.1 縦型(深さ優先)探索

- Depth First Search
- まずは深く、深く、、、進めなくなったら一歩戻って、再び、深く、深く、、、(cf. バックトラッキング)



アルゴリズム4.3 縦型探索

- 入力:問題の初期状態、各状態からすぐ次の状態を作り出す手続き、ゴールの状態
- 出力:問題が解ければ「成功」、解けなければ「失敗」
- 内容
  - (1)根だけを要素とするキューを作る
  - (2)キューが空になるか、ゴールに到達するまで、キューの先頭要素がゴールかどうか調べ続ける
    - » (2.1)先頭要素がゴールなら何もしない
    - » (2.2)ゴールでないならキューから先頭要素を取り除き、その子供たちの節があればキューの先頭に追加する

アルゴリズム4.3 縦型探索(続き)

  - (3)ゴールの節が見つかれば「成功」を返し、そうでなければ「失敗」を返す
- 「成功」したときの経路も知りたいときは、各節で子供の節を作り出すとき、「親」にあたる節を記憶しておく
- 縦形探索は「運」がよければ速く解に到達するが、場合によっては無限ループ
- 解が複数個あるとき、最初の解は必ずしも最適解であるとは限らない

縦型探索の変形(山登り法)

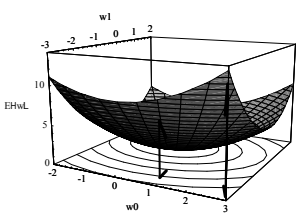
- Hill Climbing
- 問題解決の観点から、節の望ましさを評価
- アルゴリズム4.1の(2.2)で
  - キューの先頭要素がゴールでないとき
    - » その子供たちの節があれば、望ましい順に整列させてキューの前に追加する

山登り法の利用

- 全体を表現する関数は不明だが、各地点での微係数がわかるときに使える
  - 勾配法(gradient method)
- 山が複数個あると、出発地点によって極大(極小)の地点であっても最大(最小)の地点でないことがある
- 先頭要素の子どもたちの節だけでなく、まだ調べていないすべての節(探索木の葉)も含めた全体の中で最良のものを選ぶのがBest-First Search

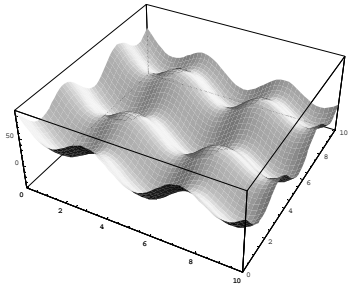
勾配法

- 様々な方法が提案されている。
- 中でも最も単純なものが、最急降下法
  - 最大値を求めるなら、最急上昇法(あまり使わない)。

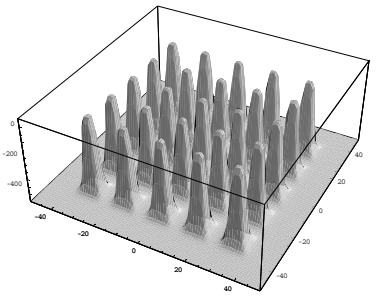


最急降下方向と等高線とのなす角度に注目!

多峰性関数の例



多峰性関数の例 (Shekel's Foxholes)

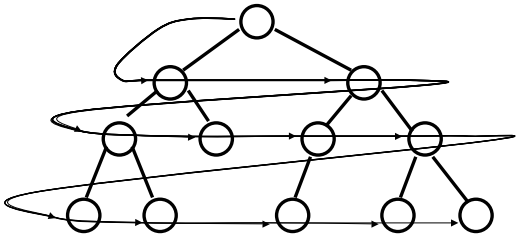


◆ Shekel’s Foxholes を正負逆転したもの

4.3.2 横型(幅優先)探索

■ **Breadth First Search**

■ **同じレベル(深さ、高さ)の節をまず探索**



アルゴリズム4.4 横型探索

◆ **入力:問題の初期状態、各状態からすぐ次の状態を作り出す手続き、ゴールの状態**

◆ **出力:問題が解ければ「成功」、解けなければ「失敗」**

◆ **内容**

– **(1)根だけを要素とする行列(キュー)を作る**

– **(2)キューが空になるか、ゴールに到達するまで、キューの先頭要素がゴールかどうか調べ続ける**

» **(2.1)先頭要素がゴールなら何もしない**

» **(2.2)ゴールでないならキューから先頭要素を取り除き、その子供たちの節があればキューの最後に追加する**

アルゴリズム4.4 横型探索(続き)

– **(3)ゴールの節が見つかれば「成功」を返し、そうでなければ「失敗」を返す**

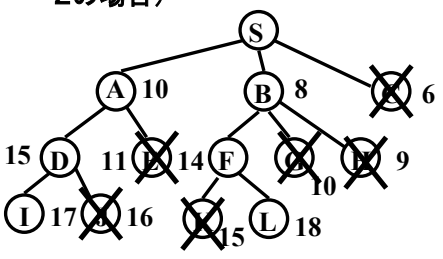
◆ **「成功」したときの経路も知りたいときは、各節で子供の節を作り出すとき、「親」にあたる節を記憶しておく**

◆ **解の候補を全部作り出すので探索木が大きくなる**

◆ **最適解を見つけやすい**

横型探索の変形(Beam Search)

◆ **各レベルにおいて、望ましさの評価値が最良のw個の節だけで探索を進める(例はw＝2の場合)**

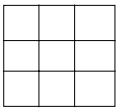


4.4 ゲームの木の探索（MINI-MAX戦略）

- 2人の対戦ゲーム（将棋、チェス、囲碁など）や経済学でのゲームの理論
  - 状況の有利・不利が数値（評価関数）で表現できれば
    - » MINI-MAX戦略（自分は自分に有利な、相手も相手にとって有利な「次の一手」を選ぶ）

最初の状況
（三目並べ）

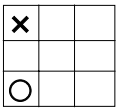
X：自分
○：相手



例：三目並べ（1）

- 評価関数E(s)：自分（X）にとって有利になれば最大値、相手（○）が有利なら最小値
  - » E(s) =（自分の勝つ可能性が何通り）ー（相手の勝つ可能性が何通り）

状況
s1

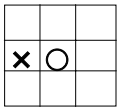


×が勝つ勝ち方：５通り
○が勝つ勝ち方：５通り

E(s1) = 5 - 5 = 0

例：三目並べ（2）

状況
s2



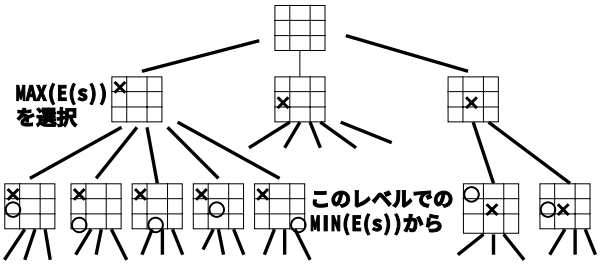
×が勝つ勝ち方：４通り
○が勝つ勝ち方：６通り

E(s2) = 4 - 6 = -2

自分の手：
評価関数値が最大
相手の手：
評価関数値が最小

次の一手のために
「先読み」が必要

MINI-MAXの原理



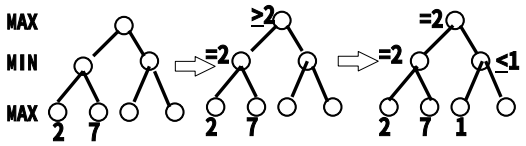
このレベルでのE(s)のMAXから上のレベルのE(s)が決定
（自分も相手も評価関数上で最適な手を打つと仮定）

アルゴリズム4.5 ミニマックス法 minimax(s,level)

- 引数：状況s、ゲームの木のレベル
- 値：評価値
- 内容
  - （1）sがゲームの終わりなら自分の勝ち、引き分け、負けを判定し、＋1、0、ー1を返す。そうでなければ
    - » (1.1) level が先読みの限界なら E(s)
    - » (1.2) s から一手進めた状況を $s_1, s_2, ..., s_k$ とする
    - » (1.3) level が偶数なら max(minimax(s_1,level+1), minimax(s_2 ,level+1), ..., minimax(s_k ,level+1)) そうでなければ、min(minimax(s_1,level+1), minimax(s_2 ,level+1), ..., minimax(s_k ,level+1))

4.5ゲームの木の探索（α－β法）

- MINI-MAXの原理だと調べなくてよい手がある



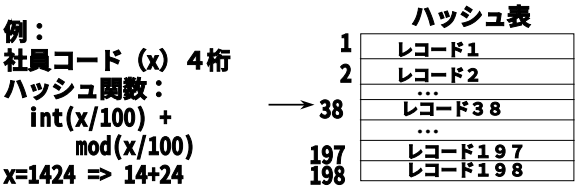
最近のゲームプログラミング

- 良い評価関数と、先読み数の増加
  - ゲームによっては、かなり強い
- IBM社のDeep Blue
  - チェスで人間の世界チャンピオンを負かす
- 将棋
  - チェスより複雑、プロに近づく
- 囲碁
  - 奥が深くまだまだ

4.6 ハッシュ表の探索

ハッシュ表：
キーを引数とする算術式（ハッシュ関数）で格納位置を決定する表

例：
社員コード（x）４桁
ハッシュ関数：
int(x/100) + mod(x/100)
x=1424 => 14+24



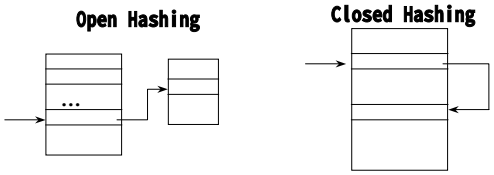
hash: 切り刻む (cf. hashed rice) ばらばらにすることが肝要

ハッシュ関数

- ハッシュ関数 $h(x_k)$
  - 定義域：キーの値の最小値$\leq x_k \leq$キーの値の最大値
  - 値域：$1 \leq h(x_k) \leq$ ハッシュ表の大きさ
  - 計算時間が短い
  - 値が一様に分布（これが肝要）
  - 違うキーには異なる値 ⇐ かち合い（collision）がおこることも

かち合いへの対応

- オープン・ハッシング
  - 予備の表（オーバーフローエリア）に
- クローズド・ハッシング
  - 同じハッシュ表内の空き領域に



アルゴリズム4.6 線形アドレシング（クローズド、検索）

- 入力：キーk、表の大きさMAX
- 出力：検索位置p
- 内容
  - （1）flag ← false; p← h(k); base ← p; i ← 1
  - （2）i < MAX である間以下を繰り返す
    - » (2.1) h(p) が探しているキーならば、flag ← true として（3）へ
    - » (2.2) p ← mod(p+1,M) ; i ← i+1
  - （3）flag = false なら、情報が見つからなかった。そうでなければ、p が格納位置

アルゴリズム4.7 クローズド・ハッシング（挿入）

- 入力：キーk、表の大きさMAX
- 出力：挿入位置i
- 内容
  - （1）flag ← false; p← h(k); base ← p; i ← 1
  - （2）i < MAX である間以下を繰り返す
    - » (2.1) h(p) が探しているキーならば、flag ← true として（3）へ
    - » (2.2) p ← mod(p+1,M) ; i ← i+1
  - （3）flag = false & i = MAX+1 ならば、ハッシュ表が一杯。そうでなければ、i に格納

探索時間の解析（1）

■ 平均探索長
- ハッシュ表の大きさ n、はいっているデータの個数 k
  - » かちあいが起こる確率 $p = k/n$
- あるレコードが i回の探索で見つかる確率
  - » $p^{i-1}(1 - p)$
- 平均探索長 $L = \Sigma_{i=1,k}\, i\, p^{i-1}(1 - p)$
  $\leq \Sigma_{i=1,\infty}\, i\, p^{i-1}(1 - p)$
  $= (1 - p)\, 1/(1 - p)^2$
  $= 1/(1 - p)$
  $= n/(n - k)$

探索時間の解析（2）

■ $L = n/(n - k)$
- n が k より十分大きいとき、線形探索の n/2 より小

■ ハッシュ法
- 探索効率がよい
- 表に空きができる
- ハッシュ関数に依存