『アルゴリズム設計マニュアル (上)』要点まとめ

ぱろすけ

2013 年 3 月 19 日

0　この文書について

専ら自身の学習のために趣味で作成したものです。これさえ読み返せば内容が思い返せる（必要に応じて参照できる）ことを目指します。

1　アルゴリズム設計への導入

挿入ソートは、ひとつの要素から始め、まだソート済み列に加えられていない要素をひとつずつ適切な位置に挿入する。アルゴリズムの一例。

1.1　ロボットツアーの最適化

要するに巡回セールスマン問題。近い点に移動し続ける**最近接点**ヒューリスティックも、最も近いチェーンの端点を結んでゆく**最近接ペア**も、うまくいかない。最適解を求めるにはすべてのパターンを試す他ないが現実的な計算量ではない。

1.2　適切な仕事の選択

映画撮影スケジューリング問題。適切なヒューリスティックにより解ける例。

1.3　正しさの論証

1.3.1　アルゴリズムの表現

アルゴリズムは自然言語、擬似コード、プログラミング言語で表し得るが、アイディアが明確に現れるような表現法を選ぶこと。

1.3.2　問題と性質

一般化映画撮影スケジューリング問題には効率の良い解はない。問題を正しく形式化することが重要。定義が曖昧に過ぎたり、複雑すぎる目標を設定したりしやすい。

1.3.3 間違いを実証する

反例は**立証可能性**と**単純性**を持つべきである。そのためには小さな例を網羅的に考え、同点に持ち込んだり極端なものを探して弱点を見つけると良い。

1.3.4 帰納法と再帰

帰納法は証明に便利だよと書いているだけ。

1.3.5 総和

等差級数の場合、n 次式の和は $n+1$ 次式になる。等比級数は比が 1 より小さいとき収束するが、これはアルゴリズムの解析上便利なことがある。

1.4 問題のモデル化

モデル化とは応用問題を厳密な形式に変換すること。

1.4.1 組合せオブジェクト

用語と登場シーンの説明。順列、部分集合、木、グラフ、点、多角形、文字列。きちんと定義された構造とアルゴリズムの言葉でモデル化するよう心がける。

1.4.2 再帰的なオブジェクト

一部を削除しても同じものである。順列、部分集合、木、グラフ、点、多角形、文字列、すべて再帰的なオブジェクトといえる。

1.5 設計奮戦記について

1.6 ボクの設計奮戦記：超能力のモデル化

モデル化を正確に行うことは大事だよという話。いきなり間違ったモデル化をされて混乱する。

2 アルゴリズム解析

2.1 計算の RAM モデル

Random Access Machine は計算量を考えるときの単純なコンピュータのモデル。メモリは無限に持つ。アルゴリズムの話ではだいたい有用。

2.1.1 最悪、最良、そして平均の計算量

基本的には最悪計算量だけ考えれば良い。

2.2 ビッグオー記法

実際の計算量はコードに依存し、グラフのでこぼこも多い。しかし実用的にはオーダーで考えれば良い。$f(n) = O(g(n))$ は $c \cdot g(n)$ が $f(n)$ の上界であることを意味する。$f(n) = \Omega(g(n))$ は $c \cdot g(n)$ が $f(n)$ の下界であることを意味する。$f(n) = \Theta(g(n))$ は $c_1 \cdot g(n)$ が $f(n)$ の上界で $c_2 \cdot g(n)$ が $f(n)$ の下界であることを意味する。

2.3 増加率と支配関係

2.3.1 支配関係

よく現れるクラスと支配関係の話。特に難しい話はない。

2.4 ビッグオーを使いこなす

特に難しい話はない。

2.4.1 関数の加算

2.4.2 関数の乗算

2.5 効率に関する議論

2.5.1 選択ソート

選択ソートは残りの要素のうち最も小さいものをソート済み列に加える。計算量は $\Theta(n^2)$ になる。簡単。

2.5.2 挿入ソート

挿入ソートの計算量は $O(n^2)$ になる。簡単。

2.5.3 文字列パターンマッチング

単純な文字列パターンマッチングについての計算量の細かな議論。落ち着けば何も難しいことはない。

2.5.4 行列の乗算

3 乗のアルゴリズムであることは明らか。

2.6 対数とその応用

対数が現れる例をいくつか示す。

2.6.1 対数と 2 分探索

2.6.2 対数と木

2.6.3 対数とビット

2.6.4 対数と乗算

任意の a と b について $a^b = \exp(b \ln a)$ と計算できる。

2.6.5 高速な指数計算

大きな n について a^n を計算するときは $a^n = a(a^{[n/2]})^2$ とすると高速。これは分割統治法の一種。

2.6.6 対数と総和

調和数について $H(n) = \sum_{i=1}^{n} 1/i \sim \ln n$ が成り立つ。これは計算量の解析に使えることがある。クイックソートなど。

2.6.7 対数と刑事裁判

合衆国では詐欺罪の損害額と量刑は対数的な関係にある。だから一気にたくさん騙そう。

2.7 対数の性質

アルゴリズムの解析では、定数倍の違いにしかならないため、対数の底が何であるかは無視して良い。

2.8 ボクの設計奮戦記：ピラミッドの謎

ナップザック問題をスパコンで力ずくで解こうとしたけど無理だったけれど、アルゴリズムを工夫したら手元のコンピュータでも容易に解けた、という話。アルゴリズム的には『プログラミングコンテストチャレンジブック』の最初に出てきたものと酷似している。

2.9 高度な解析

高度な話題であり、上巻には他に現れないが、下巻で有用であるもの。

2.9.1 難解な関数

逆アッカーマン関数は、きわめてゆっくり増加する関数であり、かなり大きな n についても $\alpha(n) < 5$ である。$\log n / \log\log n$ は次数が $\log n$ である木の高さとして現れる。

2.9.2 極限と支配関係

$n! \gg c^n \gg n^3 \gg n^2 \gg n^{1+\varepsilon} \gg n\log n \gg n \gg \sqrt{n} \gg$
$\log^2 n \gg \log n \gg \log n/\log\log n \gg \log\log n \gg \alpha(n) \gg 1$

3 データ構造

コンテナ、辞書、優先順位付きキューについて、配列とリストでの実装を詳説する。

3.1 連続データ構造と連結データ構造

連続データ構造とはメモリ上で連続しているデータ構造であり、配列、行列、ヒープ、ハッシュ表などが含まれる。連結データ構造とはいくつかのメモリ領域がポインタで結ばれる構造であり、リスト、木、隣接リストなどが含まれる。

3.1.1 配列

配列はメモリの局所性により高速キャッシュメモリを活かすことができる。配列に対し合計 n 回の挿入を行うとして、動的配列で実現するとその総移動回数は $2n$ 回であるから、この計算量はなんと $O(n)$ となり、静的配列の場合と等しい。

3.1.2 ポインタと連結構造

C 言語によるリスト構造の実現の話。

3.1.3 比較

連結リストはメモリを食うが挿入と削除は簡単。リストも配列も再帰的なオブジェクトであることに留意すべき。

3.2 スタックとキュー

コンテナとはデータを内容に無関係に出し入れするデータ構造のこと。スタックでもキューでも平均待ち時間は同じ。

3.3 辞書

辞書の概要と、実装の違いによる基本操作の計算量の違いについて述べている。長いけれども特に難しい点はない。

3.4 2 分探索木

探索が高速でかつ更新も柔軟なものの例としての 2 分探索木。ある頂点について、左部分木のすべての頂点はその頂点より小さく、右部分木のすべての頂点はその頂点より大きい。このような木は複数の形で存在し得る。

3.4.1 2 分探索木の実装

木の高さを h とすると、ある要素の探索は $O(h)$、最小要素の探索も $O(h)$ である。すべての頂点と辺を横断する際は、各頂点について左部分木、自身、右部分木の順で処理すれば良く、$O(n)$ で済む。挿入については、可能な場所はただひとつであり、それは探索により見つかり、定数時間で追加を行え、$O(h)$ である。削除の場合は、その頂点をその次の値で置き換えると考え、右部分木の最小値をそこに割り当てる。

3.4.2 2 分探索木はどれほどよいか？

基本操作はすべて $O(h)$ で行え、挿入順がランダムの場合は平均の木の高さは $O(\log n)$ になる。

3.4.3 平衡探索木

高さが常に $O(\log n)$ になる平衡探索木が提案されている。詳細についてはここでは触れられない。平衡探索木を使うことで $O(n \log n)$ の様々なソートが提案される。

3.5 優先順位付きキュー

優先順位付きキューもいろいろな実装があるが、最小要素の位置を別に持つことでどのような実装でも定数時間で見つけることができるようになる。ただしこのときは削除に余計な時間がかかる。

3.6 ボクの設計奮戦記：三角形を連ねる

きちんとデータ構造を選びましょうねという話。

3.7 ハッシングと文字列

3.7.1 衝突の回避

ハッシュ値が衝突することがある。連鎖法はハッシュの各要素を隣接リストとすることで回避する。開アドレス法では、既に埋まっていた場合、その次の位置に要素を配置する。探索するときは、本来あるべき位置から連続してある要素をすべてチェックする。削除する場合は、そのあとに連続する要素をすべて再挿入する。ぐぐると、削除済みというフラグをつければ良いという記述もある。

3.7.2 ハッシングを用いた効率的な文字列マッチング

Rabin-Karp のアルゴリズム。部分文字列パターンマッチング問題において、テキスト文字列のすべての位置からの文字列をハッシュ化しておくことにより、ハッシュ表をチェックする問題になる。n 文字と m 文字のマッチングでは $O(n)$ の表を構成することになり、ハッシュの計算に $O(m)$ かかる。しかし、ハッシュは実は計算結果を活かして定数時間で行うことが可能である。それゆえ、だいたい $O(n+m)$ 時間で走らせることが可能になる。

3.7.3 ハッシングによる重複検出

ハッシングは、重複文書の検出や、盗作の検出、ファイルのハッシュ化による暗号化などの応用がある。ハッシングはランダム化アルゴリズムにおける基本的なアイディアで、$\Theta(n \log n)$ や $\Theta(n^2)$ となるような問題にも線形時間のアルゴリズムをもたらす。

3.8 特定目的のデータ構造

文字列にはサフィックス木とサフィックス配列があり、パターンマッチングに用いられる。幾何的データ構造には kd 木があり、高速な探索を可能にする。集合データにはビットベクトルがある。これらは下巻で詳しく述べられる。

3.9 ボクの設計奮戦記：数珠つなぎ

2 分探索木で探す、ハッシュを用いる、サフィックス木、圧縮サフィックス木、と改良することで問題を解けたよという話。サフィックス木の詳細は下巻。

4　ソートと探索

4.1　ソートの応用

配列中に任意の要素 k が現れる回数を知る良い方法は、配列をソートし、$k-\varepsilon$ と $k+\varepsilon$ の位置を探すことである。また、**凸包**を構成する良い方法は、ある軸ですべての点をソートし、その順で凸包に点を加えてゆくことである。新たな点は必ず凸包をなし、かつ削除すべき点は新たな点によって作られる領域にあるため効率が良い。ふたつの配列の共通要素を持つか判定するには小さい方をソートしたのち大きい方の各要素を探すことが考えられるが、実用的にはハッシュを用いるのが最良である。

4.2　ソートの実際

もとの並びの相対的な順序を崩さないものを**安定ソート**と呼ぶ。

4.3　ヒープソート：データ構造による高速ソート

選択ソートは素直に実装すると $O(n^2)$ だが、最小要素の検索にヒープや平衡 2 分木を用いると $O(n\log n)$ となる。ヒープソートは、実際のところはヒープを用いた選択ソートに過ぎない。

4.3.1　ヒープ

ヒープは 2 分木であり、min ヒープでは親は 2 つの子より小さな値を持つ。これは配列を用いることにより効率よく実現できる。ヒープの高さは常に $[\lg n]$ である。

4.3.2　ヒープを構成する

ヒープに要素を追加するときは、配列の空きの最初の要素に配置する。もしその値が親より小さければ親と要素を入れ替える。これは再帰的に行う。

4.3.3　最小要素を取り出す

最小要素は根であるから単にそれを参照すれば良い。空きはもっとも右端にある要素で埋める。もしこれが 2 つの子より大きければそれと入れ替える。これも再帰的に行う。ヒープソートは**インプレースソート**であり、ソートする要素を格納する配列以外に余分なメモリを使わない。

4.3.4　ヒープの高速な構成法

ひとつずつ要素を加えることでヒープを構成するのではなく、すべての要素を配置してからヒープ内を整理するほうが効率が良い。子を持つものだけ整合性をチェックすればよく、かつそのとき木は概ね低いからである。これはほぼ線形時間で済む。

ヒープの k 番目に小さい要素が x 以上かを判定したいとき、これを x より小さいものが k 個以上かと読み替える。そして、x より小さい節を次々と訪問し、その数を数える。これが k を超えたら終了する。訪問する節の数は親と子 2 つで高々 $3k$ なのでこれは $O(k)$ のアルゴリズムである。頭良い。

4.3.5 逐次挿入によるソート

挿入ソートのような方法を**逐次挿入**と呼ぶ。これもデータ構造に平衡 2 分木を用いると $O(n \log n)$ で済む。

4.4 ボクの設計奮戦記：飛行機のチケットをくれないか

とりあえず飛行機のチケットの値段はものすごい複雑な感じになってることはわかった。

4.5 マージソート：分割統治法によるソート

マージソートはランダムアクセスに頼ることがないのが利点である。一方で、配列をソートするための補助バッファが必要となることが欠点である。部分配列はキューにコピーすると良い。

4.6 クイックソート：ランダム化によるソート

実装について書いてあるのでさすがに書けるようにしとくとよさそう。

4.6.1 直観：クイックソートの平均の場合

ランダムな 2 分探索木では n 回の挿入の後の平均の高さは $2 \ln n$ である（証明なし）。これは $1.386 \lg n$ なのでそれほど高くはない。ゆえに計算量は $O(n \log n)$ であるといえる。

4.6.2 ランダム化アルゴリズム

決定的なクイックソートのアルゴリズムでは必ず最悪の入力例が存在する。そこで、入力を予め**ランダム化**しておくことで計算量の期待値を $\Theta(n \log n)$ とできる。ランダム化はサンプリングやハッシング、探索などでも用いられる。

4.6.3 クイックソートは本当にクイック？

適切に実装されたクイックソートはマージソートやヒープソートよりたいてい 2,3 倍早い、らしい。最も内側のループでの操作がより単純だから。

4.7 分配ソート：バケットを用いたソート

名前をソートするとき、まず頭文字別に分け、それぞれをソートすると効率的である。これを**バケットソート**あるいは**分配ソート**といい、データが均一のときに強い。実装などの細かい話はなし。

4.7.1 ソートの下界

ソートは $n!$ の順列のそれぞれについて異なる動きをしなければならない。そのような木を考えると高さは $\Theta(n \log n)$ になる。これがソートの下界である。このように自明でない下界が得られるアルゴリズムは少ない。

4.8 ボクの設計奮戦記：スキーナの抗弁

非常に大きなデータをソートするときはいろいろ話が違うよねという話。

4.9 2 分探索と関連アルゴリズム

4.9.1 出現の数え上げ

ソートされた配列中で要素 k の出現回数を求める場合、2 分探索を改造して他の要素との境界を求めるようにすると高速に求まる。

4.9.2 片側 2 分探索

配列の要素数の上限がわからないとして、配列中の要素の転換点を求めたい。その場合は $A[1], A[2], A[4], A[8], ...$ と探索して 2 分探索の窓を決定すると早い。

4.9.3 平方根とその他の根

2 分探索で平方根や方程式の解を求められるよねという話。

4.10 分割統治法

分割統治法は、問題を（たとえば）半分に分割しそれぞれを解く。

4.10.1 再帰式

4.10.2 分割統治法の再帰式

Strassen は $n \times n$ 行列の積を $n/2 \times n/2$ 行列の積 7 つに分割することにより、積を $O(n^{2.81})$ で計算するアルゴリズムを得た。

4.10.3 分割統治法の再帰式を解く

分割統治法の再帰式 $T(n) = aT(n/b) + f(n)$ を簡単に解く**マスター定理**の紹介。ぐぐって使えれば良い。ここでは直感的な解釈も示されている。

5 グラフ横断

5.1 グラフの特徴

自己ループや多重辺を持つグラフを**非単純グラフ**という。非閉路な有向グラフを **DAG(Directed Acyclic Graph)** といい、スケジューリング問題などに現れる。バックトラック探索などではグラフは**明示**的には示されない。

5.1.1 交友グラフ

ソーシャルネットワークを題材にグラフの用語の実用例を挙げる。新しく登場する用語はグラフの**次数**で、それは頂点に接続する辺の本数を意味する。すべての頂点で次数が等しいグラフを**正則グラフ**という。

5.2 グラフのデータ構造

グラフは隣接行列か隣接リストで表されるが、大概は隣接リストのほうが良い。C 言語による実装が示されている。実際には *LEDA* や *Boost* などのライブラリを用いると良い。

5.3 ボクの設計奮戦記：ボクはムーアの法則の犠牲者だった

ハードウェアの性能向上を見越したアルゴリズムを設計すると良いよねという話。

5.4 ボクの設計奮戦記：グラフを手に入れる

三角形の集合から**双対グラフ**を作るのに $O(n^2)$ の時間をかけていたのを、改良により線形時間にした話。この場合の双対グラフとは、各三角形に対し頂点をひとつ設定し、隣接する三角形の頂点同士を結んだものをいう。データの初期化も線形でなければならない、ということが要点である。

5.5 グラフの横断

グラフの**横断**とは、すべての頂点を「未発見→発見済み→処理済み」と遷移させる処理である。

5.6 幅優先探索

キューを用いて実装する。

5.6.1 横断を活用する

探索する関数と頂点や辺を諸理する関数は分けておくと便利。

5.6.2 経路の発見

探索の途中で各頂点の親を記録しておけばそれを辿ることで最短経路を見つけることができる。

5.7 幅優先探索の応用

5.7.1 連結成分

グラフが**連結**であるとは、任意の 2 つの頂点の間に経路が存在すること。**連結成分**とはグラフの連結している部分のうち極大のもの。驚くほどたくさんの問題が連結成分を見つけたり数えたりする問題に帰着する。これは幅優先探索を複数回走らせることで実現できる。

5.7.2 2 彩色グラフ

2 **部グラフ**は 2 **彩色問題**を解けば見つけられる。2 彩色問題は、頂点を見つけるたびに親と反対の色で塗り、辺を見つけるたびにその正当性をチェックすることにより解かれる。

5.8 深さ優先探索

深さ優先探索で頂点の出入りの時刻を記録すると、それを用いて先祖関係の発見や子孫の数のカウントを行うことができる。また、深さ優先探索は無向グラフの辺を**木辺**と**逆辺**に分類する。

5.9 深さ優先探索の応用

深さ優先探索では、同じ辺を 2 度目に訪れるとき、その辺の先は必ず直接の先祖である・

5.9.1 閉路を見つける

探索中に逆辺があれば、それは閉路を構成する。

5.9.2 関節点

削除するとグラフの連結成分が分断されるような頂点を**関節点**あるいは**切断点**という。**連結度**とはそれを削除するとグラフを非連結にするような頂点の個数をいう。

木辺と逆辺を用いて到達可能なもっとも早い先祖を保存すると、これを用いて関節点の判定ができる。自分自身までしか到達できないときその親は関節点。親までしか到達できないときその親は関節点。

1 本の辺を削除するとグラフが非連結になるとき、その辺は**ブリッジ**と呼ばれる。この判定は、その辺が木辺であり、かつどの逆辺もその上下を結ばないことを確認することで行える。

5.10 有向グラフでの深さ優先探索

有向グラフでは、辺は**木辺**、**前進辺**、**逆辺**、**交差辺**に分類される。

5.10.1 位相的ソート

位相的ソートとは、すべての有向辺が左から右を向くように頂点を一直線に並べることである。頂点が処理済みのラベルをつけられた順序を逆にすると位相的ソートになる。

5.10.2 強連結成分

グラフが**強連結**であるとは、どの 2 つの頂点の間にも有向経路があることである。ある頂点からグラフを横断し、そして辺の方向をひっくり返したグラフについても同じように横断し、ともにすべての点に到達できればそのグラフは強連結である。閉路をすべて圧縮してひとつの点にすることにより強連結成分とそれをつなぐ辺が得られる。

6 重み付きグラフのアルゴリズム

6.1 最小スパニング木

最小スパニング木とは、グラフの辺の部分集合で頂点をすべて連結するもののうち、重みが最小のものをいう。

6.1.1 プリムのアルゴリズム

プリムのアルゴリズムでは、木の頂点の数を増すような辺で重みが最小のものを木に加え続ける。実装の工夫により $O(nm)$ から $O(m + n \ln n)$ まで減らせるらしい。

6.1.2 クラスカルのアルゴリズム

疎なグラフで効率が良い。残ってる辺でもっとも軽いものを取り上げ、両端が同一の連結成分に入ってないければその辺を加える。連結成分の管理を ***union-field*** で行うことにより $O(m \log m)$ で走る。

6.1.3 union-field データ構造

union-field データ構造は配列で効率よく表される。配列の各要素は親のインデックスを指し、自分自身が指定されるときそれが根であることを意味する。

6.1.4 最小スパニング木の変種

最大スパニング木は単にすべての辺に符号をつければ得られる。最大積スパニング木は辺の重みを対数と取り替えることにより得られる。シュタイナー木（重み最小化のために頂点を自由に加えて良い）と低次数スパニング木（頂点の最高次数を最小にする）はここでは解くことができない。

6.2 ボクの設計奮戦記：ネットさえあればよい

最小スパニング木を用いたクラスタリングで問題を解決した話。

6.3 最短経路

6.3.1 ダイクストラのアルゴリズム

お馴染みのダイクストラ法。辺ではなく頂点にコストがあるときは、辺にコストを課すよう解釈し直すことでダイクストラ法がそのまま適用できる。

6.3.2 全ペア最短経路

$W[i,j]^k$ を、k 番目までの頂点を通って i から j に行くときの最小コストとする。すると $W[i,j]^k = \min(W[i,j]^{k-1}, W[i,k]^{k-1} + W[k,j]^{k-1})$ が成立し、これを利用して効率的に全ペア最短経路が求められる。これは $O(n^3)$ ではあるがダイクストラ法を n 回呼び出すよりは早く走る。これをフロイド-ワーシャル法という。

6.3.3 推移的閉包

最大次数の頂点を求めるときはフロイトのアルゴリズムの結果を用いれば良い。

6.4 ボクの設計奮戦記：電話で文書を作る

日本語入力みたいな話。

6.5 ネットワークフローと 2 部マッチング

ネットワークフロー問題とは、重み付きグラフのある頂点からある頂点まで、各パイプの最大容量（辺の重み）を守って送ることのできる最大フローを求める問題である。

6.5.1 2 部マッチング

2 部グラフの最大マッチング問題はネットワークフロー問題として解かれる。

6.5.2 ネットワークフローの計算

s から t への最大フローは常に s-t 最小カットの重みに等しい。残余フローグラフとは、i から j へのフローの量 $f(i,j)$ について、重みが $f(i,j)$ の辺 (j,i) と、重みが $c(i,j)-f(i,j)$ の辺 (i,j) を持つグラフである。残余フローグラフに s から t への経路がなくなるまでフローを増加させることによりネットワークフロー問題は解かれる。

6.6 アルゴリズムではなくグラフの設計

いくつかのグラフの応用例の紹介。古典的なグラフアルゴリズムが使えるように問題を解釈するとよい。

7 組合せ探索とヒューリスティックな方法

7.1 バックトラック法

バックトラック法は、組合せアルゴリズム問題のすべての可能な解を列挙する。バックトラック法では、深さ優先探索のような形ですべての可能な解を列挙する。解候補がベクトルで表されるとき、k 次元の解候補をもとに $k+1$ 次元の解候補をつくる。

7.1.1 すべての部分集合を構成する

真偽値の配列を全パターン生成する例。

7.1.2 すべての部分集合を構成する

順列を全パターン生成する例。

7.1.3 グラフのすべての経路を構成する

ふつうに深さ優先探索をしているだけに見える。

7.2 探索の枝刈り

探索について、枝刈りや対称性の利用で大幅に計算量を減らせることがある。

7.3 数独

バックトラック法で数独を解く。解がない場合は枝刈りし、制約の多いマスを優先的に選ぶことでかなり早くなる。

7.4 ボクの設計奮戦記：チェスボードを被覆する

チェスボードを被覆する。枝刈りにより計算量を大幅に減らした話。

7.5 ヒューリスティックな探索

7.5.1 ランダムサンプリング

ランダムサンプリング、あるいはモンテカルロ法は、解空間からランダムにひとつを選び評価することを繰り返す。これは、解が高い確率で存在し、探索中に解に接近しているかどうか判断できない場合に有用である。

7.5.2 局所探索

山登り法について。「排水溝に落ちた学部長」って言葉が太字で出てくるんだけど何なのか。評価関数が凸なときや、逐次増加的な評価関数の計算が楽なときには有用である。

7.5.3 シミュレーテッドアニーリング

物理学を真似て組合せ最適化問題を解く。熱力学では粒子は高エネルギー状態へも遷移し得る。典型的には、コストが下がるときは必ず遷移し、コストが上がるときは $e^{-\frac{(C(s_i)-C(s_{i+1}))}{k\cdot t_i}} \geq r$ のとき遷移する。ここで $t_1 = 1$、$t_k = \alpha \cdot t_{k-1}$、$0.8 \leq \alpha \leq 0.99$、$r$ は $0 \leq r \leq 1$ の乱数とする。これは巡回セールスマン問題をかなりうまく解く。

7.5.4 シミュレーテッドアニーリングの応用

グラフの最大カットの問題、独立集合を求める問題、回路基板の配置などはシミュレーテッドアニーリングにより解かれる。

7.6 ボクの設計奮戦記：ラジオでないだけ

ヒューリスティックを使ってビンパッキング問題の変形版を解いた話。

7.7 ボクの設計奮戦記：配列のアニーリング

シミュレーテッドアニーリングを応用した話。

7.8 他のヒューリスティックな探索法

遺伝的アルゴリズム、ニューラルネットワーク、蟻コロニー最適化。これらは一般に手間のわりに良い解が得られない。

7.9 並列アルゴリズム

安易に並列化しても大概は良いことはなく、アルゴリズム的工夫をしたほうが良いと述べている。

7.10 ボクの設計奮戦記：速くったってどうしようもない

並列化の負荷バランスをミスって可哀想なことになった話。

8 動的計画法

8.1 キャッシング 対 計算

動的計画法は本質的には記憶領域と時間とのトレードオフである。

8.1.1 再帰によるフィボナッチ数

再帰で書くと計算量は最低でも 1.6^n になってしまう。

8.1.2 キャッシングによるフィボナッチ数

計算結果をメモしておけばフィボナッチ数は線形時間で求まる。

8.1.3 動的計画法によるフィボナッチ数

ここでは計算結果をすべて記憶しなくとも良いので記憶容量も定数にできる。

8.1.4 2 項係数

2 項係数 ${}_n\mathrm{C}_x$ は ${}_n\mathrm{C}_k = {}_{n-1}\mathrm{C}_{k-1} + {}_{n-1}\mathrm{C}_k$ であるから動的計画法で求められる。

8.2 近似文字列マッチング

近似文字列マッチングでは、スペルミスを補って与えられた文字列にもっとも近い文字列を探す。

8.2.1 再帰による編集距離

再帰で編集距離を求めることができる。間違いにコストを課し、P と T の比較について $D[i,j]$ を $P_1, P_2, P_3, ..., P_i$ と $T_1, T_2, T_3, ..., T_j$ の最小コストとする。すると $D[i,j]$ は、$D[i-1, j-1]$ または $D[i-1, j-1]+1$（置き換え）、$D[i-1, j]+1$（挿入）、$D[i, j-1]+1$（削除）のうち最小のものとなる。

8.2.2 動的計画法による編集距離

上記は再帰で書くと遅いが動的計画法で書くと早い。

8.2.3 経路の再構成

行った操作を配列に記録しておくことで、操作の経路を容易に再構成することができる。

8.2.4 さまざまな編集距離

多くの問題が近似文字列マッチングの特別な場合として解かれる。部分文字列マッチングはゴールとするセルの指定を変えれば良い。最長共通部分文字列は代入を禁止すれば良い。最大単調部分列問題では、もとの列をソートしたものを比較対象とすれば良い。

8.3 最長増加列

動的計画法を設計してみる例。配列 s の最長増加列は、終端が s_i の最長増加列の長さを配列として保持することにより簡単に求められる。また、それぞれについて直前要素を保持することにより再構成も可能である。

8.4 ボクの設計奮戦記：ロブスターの進化

動的計画法により画像のモーフィングを行う話。

8.5 分割問題

線形分割問題とは、与えられた整数列を、順序を変えずに各領域での合計値の最大値を最小にするように分割する問題であり、並列処理によく現れる。これは、$M[n,k]$ を $s_1, s_2, s_3, ..., s_n$ を k 個の区間に分割したときの最適なコストとする。すると $M[n,k] = \min_{i=1}^{n} \max(M[i,k-1], \sum_{j=i+1}^{n} s_j)$ である。

8.6 文脈自由文法の構文解析

チョムスキー標準形とは、自明でない各規則が $X \to YZ$ あるいは $X \to \alpha$ のどちらかで定義される文法のことをいう。任意の文脈自由文法はチョムスキー標準形に変換できる。$M[i,j,X]$ を部分文字列 $S[i,j]$ が非終端記号 X により生成されるとき真であるブール関数とする。すると $M[i,j,k] = \bigvee_{(X \to YX) \in G} (\bigvee_{i=k}^{j} M[i,k,Y] \cdot M[k+1,j,Z])$ である。これを用いて動的計画法により構文解析が行える。

8.6.1 最小重み三角分割

よくわからない。

8.7 動的計画法の限界：TSP

8.7.1 動的計画法のアルゴリズムが正しいのはどのようなときか？

動的計画法は、現在までにどのような操作列が実行されてきたかを知る必要がない問題にのみ適する。

8.7.2 動的計画法はどのようなときに効率がよいのか？

動的計画法は、オブジェクトに順序がないとき、たいてい指数的な時間と記憶領域を必要とする。

8.8 ボクの設計奮戦記：過去はただの Prolog

無理難題かと思ったら動的計画法で解けたよという話。

8.9 ボクの設計奮戦記：バーコードのためのテキスト圧縮

こちらも同じような話。