アルゴリズムとデータ構造 プログラミング演習

森 立平 mori@c.titech.ac.jp

2016 年 7 月 8 日

今日の目標

- 動的計画法（「宝詰め込み問題」，最適解の探索も含む）のプログラムを書けるようになる．
- 「レポート」に必要なプログラムを書く．

今日の課題はなし

今日のワークフロー

1. 「レポート」に必要なプログラムを書く．
2. 余裕があれば「発展的な課題」をやる．
3. さらに余裕があれば「超発展的な課題」をやる．

1 宝詰め込み問題のプログラムの注意点

実行時エラー `Bus error`, `Segmentation fault` は配列のインデックスが配列のサイズ以上になっている場合に起き得る．また `Segmantation fault` は再帰呼び出しが無限に続いている場合にも起きる．

2 発展的な課題

A. 分割数の $O(n)$ 空間計算量のプログラムを参考に「宝詰め込み問題」の $O(W)$ 空間計算量のプログラムを書け．ただし，最適解の計算は必要なく最適値の計算だけでよい．プログラム名は `treasure_spaceW.c` とすること．

B. おそらく全員が $O(nW)$ 時間計算量のプログラムを書いていることと思う．補助関数の定義を変えることで $O(nP)$ 時間計算量のアルゴリズムが作れる．ここで P は「宝詰め込み問題」の最適値である．

   B.1 価値と重さの役割を逆にした補助関数を定義し，その補助関数の満たす再帰的な関係式を導出せよ．

   B.2 その新しい補助関数を用いてトップダウン法もしくはボトムアップ法で時間計算量が $O(nP)$ の動的計画法のプログラムを書け．プログラム名は `treasure_timeNP.c` とすること．

3 超発展的な課題: 実用的な高速アルゴリズム

3.1 はじめに

今までの課題では $O(nW)$ および $O(nP)$ 時間計算量のアルゴリズムを作った．これはある程度効率的（擬多項式時間アルゴリズム）であるが，W や P の値が大きいときには計算量が大きくなってしまう．しかし，実用的には理論的保証よりも大幅に効率的なアルゴリズムがいくつか知られている．ここではそのうちの一つであるトップダウン `minknap` アルゴリズムを実装してもらう．これはとても難しいので，多くても 3 人くらいしかできないと考えている．

3.2 「宝詰め込み問題」の最適解の上界と貪欲法

まず「宝詰め込み問題」の最適値の上界について考える．宝を袋に詰め込むときに，好きな重さにカットして詰め込むことができることにする．そのときに，カットされた宝の価値は重さに比例するとする（例えば重さが半分になるようにカットしたら，価値も半分になる）．以下この問題を「カットあり宝詰め込み問題」と呼ぶ．明らかに「カットあり宝詰め込み問題」の最適値は元々の「宝詰め込み問題」の最適値以上である．この「カットあり宝詰め込み問題」の最適解は以下のように与えられる．i 番目の宝の価値を v_i，重さを c_i とおく．重さあたりの価値 v_i/c_i が大きいものから順番に並べかえる．つまり，

$$\frac{v_1}{c_1} \geq \frac{v_2}{c_2} \geq \cdots \geq \frac{v_n}{c_n}$$

が成り立っているとする．ここで 1 番目の宝から順番に袋に詰め込んでいき，b 番目の宝が入りきらないときには袋にぴったり入るようにカットして詰め込む．この方法で得られる価値は

$$\sum_{i=1}^{b-1} v_i + \frac{W - \sum_{i=1}^{b-1} c_i}{c_b} v_b \qquad \text{ただし} \qquad b := \min\left\{ k \mid \sum_{i=1}^{k} c_i > W \right\}$$

である．これは「カットあり宝詰め込み問題」の最適値である（証明は省略する）．

この方法は元々の「宝詰め込み問題」に利用できないだろうか？つまり 1 番目から $b-1$ 番目まで詰め込むと「宝詰め込み問題」のそこそこ良い解になっているのではないか？この 1 番目から $b-1$ 番目まで詰め込むアルゴリズムのことを「**貪欲法**」と呼ぶ（厳密には貪欲法では全ての宝を 1 番目から n 番目まで順番に見て，袋に入るものを入れていく）．この貪欲法はいい線はいっているが，最悪のケースではとても悪い．例えば $c_1 = 1$，$v_1 = 1$，$c_2 = W$，$v_2 = W - 1$ の場合，貪欲法では 1 番目の宝だけを袋に入れる．そして価値 1 が得られるが，2 番目の宝だけを袋に入れた場合，価値 $W - 1$ が得られる．袋の容量 W が大きいと，貪欲法は最適解に比べてとても低い価値しか得られない．そこで貪欲法を次のように改良する．

> アルゴリズム D: 貪欲法の解と「最も価値が高い宝を一つだけ入れる」という解で得られる価値が高い方を選ぶ（ただし重さが袋の容量を超えているような宝はあらかじめ除かれているものとする）

定理 1. アルゴリズム D は「宝詰め込み問題」の最適値の**半分以上**の価値の解を与える．

証明. アルゴリズム D で得られる価値は少なくとも

$$\max\left\{ \sum_{i=1}^{b-1} v_i,\, v_{\max} \right\}$$

以上である．ここで $v_{\max} := \max_{i=1}^{n} v_i$ とする．また

$$\max\left\{ \sum_{i=1}^{b-1} v_i,\, v_{\max} \right\} \geq \frac{\sum_{i=1}^{b-1} v_i + v_{\max}}{2} \geq \frac{\sum_{i=1}^{b-1} v_i + v_b}{2}$$

が成り立つ．ここで，$\sum_{i=1}^{b} v_i$ は「カットあり宝詰め込み問題」の最適値以上である（b 番目の宝の価値がそのまま足されているため）．既に見たように「カットあり宝詰め込み問題」の最適値は「宝詰め込み問題」の最適値以上であるので，上の式は「宝詰め込み問題」の最適値の半分以上である． □

3.3 アルゴリズム: トップダウン `minknap`

アルゴリズム D は「最適値の半分以上」というそこそこ良い解を与えることが分かった．もしも飛び抜けて価値や重さが大きい宝が存在しないとすると，アルゴリズム D は貪欲法の解を選ぶ．つまり貪欲法の解，もしくは「1 番目から $b-1$ 番目の宝を選ぶ」という解は実用上（飛び抜けて大きな価値や重さを持つ宝が存在しない場合）そこそこ良い解である．また，実際の最

適解は「1 番目から $b-1$ 番目の宝を選ぶ」という解にとても近く，多くの場合 b 番目前後の宝の詰め方が違うだけであるという実験結果がある．

これらの考察に基づいた動的計画法を考えよう．補助関数 $f(m,U)$ を「重さの総和が U 以下であり，1 番目から $S(m)-1$ 番目までの宝を袋に入れて，$T(m)+1$ 番目から n 番目までの宝を袋に入れないような解の価値の最大値」と定義する．ただし，

$$S(m) := \max\{1, b - \lfloor m/2 \rfloor\}$$
$$T(m) := \min\{n, b - 1 + \lceil m/2 \rceil\}$$

とする．すると次の再帰的な関係式が満たされる．

$$f(0,U) = \begin{cases} 0, & \text{if } U < C^* \\ V^*, & \text{if } U \geq C^* \end{cases}$$

また，$m \geq 1$ のとき

$$f(m,U) = \max \begin{cases} f(m-1,U) \\ f(m-1,U+c_{S(m)}) - v_{S(m)}, & m \text{ が偶数かつ } b-1 \geq \lfloor m/2 \rfloor \\ f(m-1,U-c_{T(m)}) + v_{T(m)}, & m \text{ が奇数かつ } U \geq c_{T(m)} \text{ かつ } n-b+1 \geq \lceil m/2 \rceil. \end{cases}$$

ここで $C^* := \sum_{i=1}^{b-1} c_i$，$V^* := \sum_{i=1}^{b-1} v_i$ とする．また，「宝詰め込み問題」の最適値は $f(2n,W)$ で与えられる．この補助関数を用いた動的計画法の時間計算量を考えよう．補助関数 f の最初の引数 m は高々 $2n$ で二番目の引数 U は高々 $W + C^* \leq 2W$ である．よって時間計算量は $O(nW)$ である．時間計算量のオーダーは一番最初の「宝詰め込み問題」のアルゴリズムと同じであるが，実際のプログラムの実行時間はこの新しいアルゴリズムの方が**かなり遅い**．しかし，ここからさらに「**枝刈り**」という工夫をしたトップダウン法を考えることで高速化できる．

補助関数 f をトップダウン法で再帰的に評価するときに，できるだけ再帰呼び出しを避ける方法を考えよう．再帰的な関係式を使って $f(m,U)$ を評価するときに 2 回補助関数を再帰呼び出しして，そこから計算されるものの最大値をもとめることになる．もし，この 2 つの式のうちのどちらが最大値を与えるのかあらかじめ分かっていれば，再帰呼び出しの回数を 1 回に減らすことができる．前述の考察より，最適解は「1 番目から $b-1$ 番目の宝を選ぶ」という解に近いことが期待されている．よって，**2 つの式のうち最大値を与えるものとして有望なのは $f(m-1,U)$ の方である**．この考察に基づき，$f(m,U)$ を計算する際にはまず $f(m-1,U)$ を評価し，もう片方の計算結果が $f(m-1,U)$ より大きくなる可能性がある場合に限り，そちらも計算することにしよう．

そのために補助関数 f の上界を考える．そしてその上界を使って $f(m-1,U)$ より大きな値が得られる可能性があるかどうか判定する．自明な上界として以下が成り立つ．

$$f(m,U) \leq \sum_{i=1}^{S(m)-1} v_i + \frac{U - \sum_{i=1}^{S(m)-1} c_i}{c_{S(m)}} v_{S(m)} \tag{1}$$

これは「カットあり宝詰め込み問題」の考え方と同じであるが (1)で $v_{S(m)}$ の係数は 1 を超えることもある．また一方で m が偶数のとき

$$f(m-1,U+c_{S(m)}) - v_{S(m)} \leq \sum_{i=1}^{S(m)-1} v_i + \frac{U - \sum_{i=1}^{S(m)-1} c_i}{c_{S(m)+1}} v_{S(m)+1} \tag{2}$$

が成り立つ．これらの上界の中の 2 つの和（Σ 記号のところ）はあらかじめ計算しておいて配列に入れておけばよいので，$O(1)$ 時間計算量でこれらの上界が評価できることになる．

動的計画法の枝刈りに話を戻そう．$f(m,U)$ の計算の際には，まず $f(m-1,U)$ を評価する．次に m が偶数の場合は上界 (2)を，奇数の場合は (1)を用いた $f(m-1,U-c_{T(m)})+v_{T(m)}$ の上界を評価する．得られた上界が $f(m-1,U)$ より大きい場合に限り，再帰呼び出しをする．この改良により再帰呼び出し回数を削減することができ，「実用上」とても高速なプログラムが書ける（最悪時間計算量が何らかの入力に対する仮定のもとで $O(nW)$ から改善するかどうかは分からない）．このアルゴリズムのことを「トップダウン `minknap` アルゴリズム」と呼ぶことにする．

3.4 超発展的な課題

I. 補助関数 $f(m,U)$ が満たす再帰的な関係式（$m=0$ の場合も含む）を説明せよ.

II. 式 (2)を示せ.

III. トップダウン `minknap` アルゴリズム（最適値の計算のみ）の擬似コードを書け.

IV. トップダウン `minknap` アルゴリズム（最適解の計算含む）の擬似コードを書け (II を飛ばしてこちらのみ擬似コード書いてもよい).

V. トップダウン `minknap` アルゴリズムのプログラムを書け.プログラム名は `treasure_minknap.c` とすること.

VI. 枝刈りをしない場合とする場合とでそれぞれ f の関数呼び出しの回数をカウントせよ．ただし，計算済みのパラメータに対する関数呼び出しの回数はカウントしなくてよい．それぞれの場合について，サンプル入力に対する f の関数呼び出しの回数を書け（サンプル入力は火曜日までに用意する）．またそれぞれの場合について，プログラムの実行時間を書け.

VII. その他，さらにトップダウン `minknap` アルゴリズムを改良するアイデアがあれば書け．実際にプログラムを書いて実行時間が改善したか報告せよ.