成蹊大学理工学研究報告
J. Fac. Sci.Tech., Seikei Univ.
Vol.58 No.2 (2021)pp.25-28

少ないメモリで動作するアルゴリズム

清見　礼[*1]

Algorithms Running with Small Memory

Masashi KIYOMI[*1]

ABSTRACT：Most of the researches in algorithms are for reducing computational time complexity. Such researches often neglect the amount of memory used, resulting in algorithms that require large amounts of memory and cannot be executed on ordinary PCs. On the other hand, there have been researches on reducing the amount of memory required for computation for a long time. However while most of them were theoretically interesting, practically too restrictive, such as whether some computation can be done with O(log n) bits for input size n.

In recent years, the use of big data has become widespread, and the size of the input to algorithms tends to increase compared to the past. In the past, it was natural to use the same amount of memory as the size of the input data, and this was not a problem. However, if we consider an example of input data filling a hard disk, it becomes unrealistic to use the same amount of memory as the input data, as the capacity of a hard disk is generally much larger than the capacity of memory.

Against this background, we consider algorithms that handle big data using less memory than the size of the input data. In this paper, the author outlines an example of such research that he has recently conducted.

Keywords：space efficient algorithm, longest increasing subsequence, longest common substring

(Received Dec. 10, 2021)

１．はじめに

アルゴリズムの研究の多くは、如何に計算の速度を上げるかを目標としている。そのような研究ではしばしば使用するメモリの量を度外視しており、結果として大量のメモリが必要で一般的なPCでは実行することのできないアルゴリズムとなってしまうことがある。一方で昔から計算に必要なメモリの量を減らす研究も行われてきたが、その多くは入力のサイズnに対してO(log n)ビットで計算が可能かというような、理論的には興味深いが現実的には制限がきつ過ぎるようなものが多かった。

近年、ビッグデータの活用が広く行われるようになり、アルゴリズムへの入力のサイズが以前と比較して増大する傾向にある。従来は入力のデータサイズと同程度のメモリを使用することは自然で、とくに問題となることはなかった。しかし、たとえば入力データがハードディスク一杯に詰まっているような例を考えると、入力データと同程度のサイズのメモリを使うことは現実的ではなくなる。一般にハードディスクの容量はメモリの容量よりはるかに大きいからである。

このような背景から、ビッグデータに対して入力データのサイズよりも少ないメモリで計算を行う研究というものを行っている。本稿では著者が最近行った、このような研究の例について概要を紹介する。

２．計算モデル

多くのアルゴリズムの研究ではRAM(Random Access Machine)モデルと呼ばれる計算モデルが使用される。このモデルではメモリにアドレスが振られており、任意のアドレスのメモリにO(1)の時間でアクセスができる計算モデルである。

[*1]：情報科学科教授(kiyomi@st.seikei.ac.jp)

メモリの使用量の制限を考える場合、このRAMモデルに加え、入力データは読み込み専用のメモリに置かれている、出力は書き込み専用メモリに行う、という制限をつけることが一般的である。本稿でもそのような計算モデルを採用する。

3．単調増加部分列問題

有限の長さの数列

$$a_1, a_2, \dots, a_n$$

が与えられたとき、その数列からいくつかの数を選んで得られる数列

$$b_1 = a_{i_1}, \dots, b_k = a_{i_k}\ (1 \le i_1 \le \cdots \le i_k \le n)$$

を$a_1, \dots, a_n$の部分列と呼ぶ。部分列のうち、単調増加になっているものを増加部分列と呼ぶ。さらに、増加部分列のうち長さが最大のものを最長増加部分列と呼ぶ。

最長増加部分列を求めることは古くから研究されており、理論的にも、実用的な視点からも非常に重要な問題である。たとえばUNIXで 2 つのファイルの差を求めるdiffコマンドの内部で解かれていたり、ゲノムの解析などに応用がある[2]。

最長増加部分列問題はペーシェンスソーティングと呼ばれる有名なアルゴリズム[5,8,9]を用いてO($n \log n$)時間で解くことが可能である。ただし、ペーシェンスソーティングのアルゴリズムは入力データと同程度のメモリを必要とする。

本稿では、最長増加部分列問題をO($\sqrt{n} \log n$)ビットのメモリでO($n^{1.5} \log n$)時間で解くアルゴリズム[6]を概説する。なお、最長増加部分列問題は最長増加部分列自体を出力する問題であるが、煩雑になるため、本稿では最長増加部分列の長さを答える問題のアルゴリズムを示す。

4．アルゴリズム

4．1　ペーシェンスソーティング

まず、最長増加部分列問題を解く既知のアルゴリズムである、ペーシェンスソーティングを説明する。

ペーシェンスソーティングでは、与えられた数をパイルと呼ばれるデータ構造の配列に格納していく。パイルはリンクリストで構成される。パイルpの最後部をtop(p)で表すことにする。Pが空の場合はtop(p)=∞と約束する。以下にペーシェンスソーティングのアルゴリズムを示す。

1．パイル p[1], …, p[n] を空に初期化する
2．for i = 1, 2, …, n:
3．　top(p[j]) ≤ a_iを満たす最小のjを見つける
4．　p[j]にa_iを追加

このアルゴリズムによって得られる、空でないパイルの数が最長増加部分列の長さである。top(p[j])は単調増加になることが簡単に示せるので、3．の操作は二分探索を用いてO(log n)時間でできる。

4．2　最長増加部分列問題のO($\sqrt{n} \log n$)ビットのアルゴリズム

ペーシェンスソーティングのアルゴリズムを見ると、各パイルについてtop の値のみを覚えておけば十分であることがわかる。ただし、入力データの数列がもともと単調増加である場合など、topだけを覚えていても$\Theta(n)$個分の数値を覚えなくてはいけない可能性はある。当然、topのみを覚えているだけでもO($n \log n$)時間のアルゴリズムであることに変わりはない。

一方、p[j]の要素がすべてわかっている場合、p[j+1]の要素を計算することができる。これは簡単なケースアナリシスであるが、詳細については参考文献[6]を参照。P[j]からp[j+1]を求めるのにかかる計算時間はO(n)時間、必要なメモリはp[j]とp[j+1]を覚えるのに必要なメモリとO(log n)ビットである。そこでまずp[1]を求め、その後p[2]、p[3]、…と求めていけば最長増加部分列の長さを求めることができる。必要なメモリは要素数最大のパイルを覚えるのに必要なメモリ量（の 2 倍）程度である。ただしこの場合も、入力データが降順に並んでいるような場合、$\Theta(n)$個分の数値を覚えなくてはいけない可能性がある。

以上２つの事実を念頭に置くと、以下のようなアルゴリズムを考えることができる。すなわち、要素数$\sqrt{n}$以下のp[j]を知っているときp[j+1],…,p[j+2$\sqrt{n}$]のtopを覚えておきながらp[j+1],…,p[j+2$\sqrt{n}$]のサイズを求める。この際、要素数が$\sqrt{n}$を超えるようなパイルは高々$\sqrt{n}$個しかないから、$\sqrt{n} \le k \le 2\sqrt{n}$をうまく選べば必ずp[j+k]の要素数を$\sqrt{n}$以下にできる。よってp[j]からp[j+1],…,p[j+k-1]のtopを覚えておくことにより、p[j+k]を求める。これを空でないパイルの数がわかるまで（高々$\sqrt{n}$回）繰り返す。

このアルゴリズム全体で必要なメモリの量は明らかにO($\sqrt{n} \log n$)ビットに抑えることができる。計算時間はパイルを計算するのが高々$\sqrt{n}$回であること、パイルを１つ計算するのに必要な時間がO($n \log \sqrt{n}$)時間であることを考えるとO($\sqrt{n} \cdot n \log \sqrt{n}$) = O($n^{1.5} \log n$)時間のである。

５．その他の例

もう一つ、最近行ったアルゴリズムのメモリを減らす研究[7]を紹介すると、２つの文字列

$$a_1, a_2, \ldots, a_n$$

と

$$b_1, b_2, \ldots, b_n$$

が与えられたとき、それら２つの共通の部分列で長さが最長なもの（最長共通部分文字列）を求めるアルゴリズムがある。最長共通部分文字列自体はよく知られた動的計画法を用いて$O(n^2)$時間で簡単に求められる。しかしこのアルゴリズムは$O(n)$ビットのメモリを使用する。

我々は既知の動的計画法で得られる表に注目し、この表を別のやり方で求める方法を開発した。

動的計画法ではサイズ$n \times n$の有向グリッドグラフ（の各グリッドに対角線を追加したグラフ）の左上の点から右下の点へ至る最長路を求めるものであるが、x座標が中間の場所で最長路が通る点を決め打ちすると、メモリが半分で済む動的計画法を２回実行することで、その中間点を通る最長路が求まる。そこで、通る中間点をあり得るn通りすべてについてこれを行えば（かかる時間はn倍になるが）半分のメモリで最長路を求めることができる。さらにこれをx座標について再帰的に行えば、$O(\log n)$ビットのメモリで最長路を求めることができる。ただし、このようなことを行うと、アルゴリズムの実行には指数的な時間がかかってしまう。それでは計算に時間がかかり過ぎ実用的とは言えない。そこで、通る場所を決め打ちする際、点を決め打つのではなく、範囲を指定してその範囲内を通るというように決め打ちをするアルゴリズムを開発した。計算が煩雑になるので詳細は参考文献[7]に譲るが、開発したやり方では計算時間を$O(n^3)$時間に抑えながら、使用するメモリの量を$O\left(\frac{n(\log n)^{1.5}}{2\sqrt{\log n}}\right) = o(n)$ビットに抑えることに成功している。

６．未解決な問題

グラフとその中の１つの頂点が与えられたとき、与えられた頂点からそのグラフを深さ優先探索することを考える。よく知られた深さ優先探索のアルゴリズムでは$\omega(n)$ビットのメモリを使用する。

我々は深さ優先探索を行ったときに訪れる順で頂点を多項式時間で出力する$n + O(\log n)$ビットのアルゴリズムを開発した[1]。しかし、$o(n)$ビットのメモリで動作する多項式時間アルゴリズムが存在するかどうかは未解決である。

７．まとめ

本稿では最近我々が研究した、入力データより少ないメモリで動作するアルゴリズムを２つ紹介した。

１つ目のアルゴリズムの肝は、パイルが１つずつ次々に生成できるということと、要素数が$\sqrt{n}$以上のパイルは高々$\sqrt{n}$個しかないという事実である。これだけの観察で、非常に有名なペーシェンスソーティングのアルゴリズムを改良し、使用するメモリを大幅に減らすことが可能である。入力データがハードディスク一杯に詰まっているようなデータに対しても、使用するメモリがその平方根程度であれば十分実用的なアルゴリズムとなる。

１つ目のアルゴリズムでは既知のアルゴリズムで用いるパイルの配列を別のやり方で構築し、２つ目のアルゴリズムでも既知の動的計画法の表を別のやり方で構築している。これら２つの既知のアルゴリズムは教科書に載るような非常に有名なものであるが、それらについてさえ、メモリを減らすという研究は十分に行われておらず、今回のような重要な結果を得ることができた。ビッグデータが益々幅広く用いられるようになることで、アルゴリズムの使用メモリを減らす研究は今後さらに重要性を増すものと信じる。

参考文献

1) Tetsuo Asano, Taisuke Izumi, Masashi Kiyomi, Matsuo Konagaya, Hirotaka Ono, Yota Otachi, Pascal Schweitzer, Jun Tarui, Ryuhei Uehara: Depth-First Search Using O(n) Bits. ISAAC 2014: 553-564
2) A. L. Delcher, S. Kasif, R. D. Fleischmann, J. Peterson, O. White, S. L. Salzberg: Alignment of whole genomes. Nucleic Acids Research. **27** (11): pp. 2369–2376(1999)
3) John Hammersley: A few seedlings of research. Proc. Sixth Berkeley Symp. Math. Statist. and Probability. **1**: pp. 345–394 (1972)
4) D. S. Hirschberg: "A linear space algorithm for computing maximal common subsequences". Communications of the ACM. **18** (6): 341–343 (1975)
5) J. Hunt, T. Szymanski: "A fast algorithm for computing longest common subsequences", Communications of the ACM, **20** (5): 350–353 (1977)

6) Masashi Kiyomi, Hirotaka Ono, Yota Otachi, Pascal Schweitzer, Jun Tarui: Space-Efficient Algorithms for Longest Increasing Subsequence. Theory Comput. Syst. 64(3): 522-541 (2020)
7) Masashi Kiyomi, Takashi Horiyama, Yota Otachi: Longest common subsequence in sublinear space. Inf. Process. Lett. 168: 106084 (2021)
8) David Maier: The Complexity of Some Problems on Subsequences and Supersequences. J. ACM. ACM Press. **25** (2): 322–336 (1978)
9) C. L. Mallows: Patience sorting. Bull. Inst. Math. Appl. **9**: 216–224 (1973)