1　38 ページ目翻訳

我々はアルゴリズムという用語を、有限のステップ数で特定の入力から特定の出力を得るための、よく定義された規則または命令のセットを意味するために使用します。例えば、おなじみの正の整数の割り算で行われる一連の手順がアルゴリズムである。数学におけるアルゴリズムの利用や関心は、主にコンピュータの普及に刺激され、ここ数年で急激に高まっている。これから述べるように、アルゴリズムはグラフ理論の研究において重要な役割を担っている。いくつかの基本的なアルゴリズムを調べる前に、まず、アルゴリズムの中心的な話題である「複雑さ」について説明します。

アルゴリズムの複雑さは、コンピュータがそのアルゴリズムを使って問題を解くときに費やされる計算量の大きさを測るものである。この尺度は、計算ステップの数、実行時間、または記憶領域を指すことがあります。しかし、これら最初の 2 つには実質的な違いはなく、計算ステップ数が複雑さの解釈として使用されることになる。アルゴリズムの複雑さは、一般に、入力データのサイズとプレゼンテーションの関数である。

ある問題を解決するためのアルゴリズムは 1 つだけではないことが多い。例えば、一般的な計算問題は、(非特異) 行列の逆行列を求めることである。アルゴリズム A は $n \times n$ 行列の逆行列を $0.5n^4$ 単位の時間で求めるとします。別のアルゴリズム B は、$2.5n^3$ 個の時間単位で $n \times n$ 個の行列の逆行列を求めます。したがって、アルゴリズム A は 4×4 行列の逆行列を 128 時間単位で求めるが、アルゴリズム B は 25%長くかかる。実際、おなじみのガウス消去法を用いれば、$n \times n$ 行列の逆行列を最大でも cn^3 単位の時間で求めることができる (ここで c は定数)。アルゴリズム A が $n \times n$ 行列の逆行列を求めるのに $0.5n^4$ 時間単位かかると言うのは、多くても $0.5n^4$ 時間単位という意味である。実際には、比較的少数の $n \times n$ 行列が必要とする。$.5n^4$ 回単位です。

2　39 ページ目翻訳

つまり、最悪の場合にのみ、その時間が必要となる。このような複雑さの指標をワーストケースの複雑さと呼ぶ。対角行列の逆行列を求める場合など、最悪の場合の複雑さよりはるかに少ない時間で済む行列の逆行列問題の例があることは間違いない。ワーストケース複雑度は最も一般的な複雑度の尺度であるが、他の尺度もある。例えば、平均ケース複雑度は、すべての $n \times n$ 個の行列に対するアルゴリズムの平均実行時間を表します。アルゴリズムは、その複雑さが入力サイズ（例えば n）の多項式である場合、効率的または「高速」である。例えば、その複雑さがデータの入力サイズ（例えば n）の多項式であるか、または n の多項式で境界付けられている場合、アルゴリズムは効率的または「高速」である、のアルゴリズムが効率的であり、2^n, $n!$, n^n のアルゴリズムは効率的ではありません。計算問題は、その問題を解くための効率的なアルゴリズムが存在する場合、扱いやすいと呼ばれる。計算問題は、その問題を解くための効率的なアルゴリズムが存在しないことが証明されれば、難解である。本文中では、扱いやすい問題の例を見ていくことにする。しかし，多くの問題では，扱いやすいかどうかが分からない．(この問題については 2.4 節で説明する) 節で述べる)。扱いにくい問題の例として、いわゆるタイリング問題 (Wilf 9 参照) がある。ある多角形に対して、平面全体を多角形の形の床タイルで敷き詰めることができるか？ 図 2-1 は、長方形、正六角形、ひし形、および他の 2 つの多角形を示している。

図 2-2 は、図 2-1 の長方形を使った平面のタイル貼りと、正六角形を使ったタイル貼りを示しています。

3　40 ページ目翻訳

タイリング問題は難解であることが Berger [2] によって証明されている。つまり、与えられたポリゴンで平面をタイリングできるかどうかを決定するための効率的なアルゴリズムは存在しないのである。もちろん、この問題のいくつかの例については、効率的なアルゴリズムが存在する。実際、タイリング問題は、長方形と正六角形については簡単に解ける。また、正五角形についても簡単に解くことができます。正五角形では平面をタイル状にすることはできません（問題 1 参照)。アルゴリズムの複雑さをより効果的に比較するために、次のように説明します。関数の「次数」を説明する。f と g を正整数の集合で定義された 2 つの関数とする。正整数の集合に定義された 2 つの関数があるとする。f の次数は g の次数より低いか等しいとする。が存在するとき、f の次数は g の次数以下であるという。が存在する。すべての $n > n_o$ に対して。f の次数が g の次数以下である場合、$f(n) = C(g(n))$ と書いたり、$f(n)$ は $O(g(n))$ ($f(n)$ は ”big oh” の $g(n)$ と読みます) と言ったりします。これは、f が g より速く成長しないことを意味します。関数 f は g よりゆっくり成長するか、同じ速度で成長するかです。関数 f と g は、$f(n) = O(g(n)$ と $g(n) = Of(n))$ であれば同じ次数である。$f(n) = Cg(n)$ である。$f(n) = 3n + 5$, $g(n) = n^2$ であるとする。すると、$f(n) = O(g(n))$ となる。定数 $C = 1$ に対してすべての $n > 4 = n_o$ に対して $3n + 5 \leq 1n^2$ となる。この不等式は数学的帰納法で検証することができるが、別の方法を用いる。$n > 5$ のとき $3/n \leq 3/5$, $n \leq 5$ のとき $5/n^2 \leq 1/5$ であることに注目する。つまり、$3/n + 5/n^2 \leq 3/5 + 1/5 < 1$ は $n > 4$ のとき、n^2 に $(3/n + 5/n^2) < 1$ を掛けると、目的の結果が得られる。また、C と n_0 を他の値にすることで、$f(n) = O(g(n))$ を検証することができた。例えば、$C = 3$ とする。すると、$n > 2$ に対して $3n + 5 \leq 3n^2$ となることがわかります。

例えば、$C = 11$、$n_0 = 0$。$f(n) = .5n^4$, $g(n) = 2.5n^2$ と仮定する。すると、$g(n) = Of(n)$ となり、$g(n)$ は $O(n)$ となる。また、$f(n)$ は $O(n)$ であるが、$f(n)$ は $O(g(n))$ である。$h(n)$ i が次数 k の任意の n の多項式であれば、$h(n) = 0(n*)$ となる。アルゴリズム A と B がそれぞれ複雑度関数 $f(n)$ と $g(n)$ を持つ場合、$f(n) = O(g(n))$、$g(n) = O(f(n))$ のとき、アルゴリズム B より $1s$ 効率が高いと言う。関数の増加順序の階層を知るために、7 つの一般的な順序を列挙する。$a, b > 1$ であれば $\log n$ は $\log n$ の定数倍であるから、対数の底は重要ではない。

増加する順序のヒエラルキー：

すべての $n \leq 1$ に対して $\log n < n$ なので、$logn$ は $O(n^2), O(n^3), O(2^n)$ です。

4　41 ページ目翻訳

そして、あらゆる正の実定数 C に対して、$2 > C_n^k$ の不等式が十分である。しかしながら、2^n はあらゆる正の整数 k 大きな n に対して成立するので多項式オーダーではない。同じ問題を解くのに 2 つのアルゴリズム A、B が使われ、A は時間複雑度 $O(n!)$ B は時間複雑度 $O(g(n))$ ここで $\log n = O(n)$ であるとする。図 2-3 は、様々な n の値に対して、いくつかの一般的な複雑関数の実行時間を示したもので、時間の単位は 1 秒である。表中の実行時間が秒単位で与えられていない場合、その時間は概算である。

5　43 ページ目翻訳

一般的な代入表記 $A \to B$ は、A の（現在の）値を B の値で置き換えることを示す。A および B が単語の場合、$A > B$ と書き、アルファベット順に A が B に先行することを示す。アルゴリズム 2.1. [単語のリスト $W(1), W(2), ..., W(n)$ の中でアルファベット順に最初の単語を決定し、この単語とリスト内の位置を出力する] 。1.[初期状態では、FIRST は現在の最初の単語をアルファベット順に表す] 。FIRST $W(1)$. 2.[目的の単語の位置を与える p を初期化する] p 1. 3.[FIRST の前に単語 $W(k)$ がある場合、FIRST は $W(k)$ に置き換えられ、p は k に置き換えられる。は k に置き換えられる]。$k = 2$〜n の場合 (a) $W(k) < FIRST$ であれば () $FIRSTW(k)$ とする。(i) $p - k$ となる。4.FIRST、p を出力する。アルゴリズム 2.1 の複雑さを、必要な比較の回数で評価する。を比較する必要がある。この数は $n - 1$ であるので、アルゴリズム 2.1 の複雑さは $n - l$ である、または $O(n)$ となる。

6　44 ページ目翻訳

2.2 探索アルゴリズム

このセクションでは、基本的な問題と、その問題を解決する 2 つのアルゴリズムを説明する。また、これらのアルゴリズムの効率も比較する。アルファベット順の単語リストが与えられたとき、特定の単語がリスト上のどこに現れるかどうかを判断したいと思う。例えば、ある飛行機に乗る予定の乗客のアルファベット順のリストに、個人の名前があるかどうかを知りたい場合がある。このような場合、私たち（あるいはコンピュータ）は、どのようにリストを検索して、その名前が含まれているかどうかを判断するのでしょうか？(単語のリストをアルファベット順に並べる 2 つのアルゴリズムが 2.3 節で紹介されています)。簡単な方法は、リストの先頭から順番に、名前が見つかるか、リストの最後に来るまで検索することです。このアルゴリズムは、しばしば逐次探索アルゴリズムと呼ばれる。リスト上の k 番目の位置にある単語を $w(k)$ とする。リスト上に n 個の単語がある場合、$w(n)$ とする。目的の単語が KEY だとすると、ある k、$1 < k < n$ に対して $W(k)$ =KEY かどうかが問題となる。すると、単語はアルファベット順に、$W(1), W(2), ...$ となる。アルゴリズム 2.2（逐次探索アルゴリズム）。[T_b は、与えられた単語 KEY がアルファベット順のリストに現れるかどうかを決定する $W(I), W(2), ..., W(n)$ の単語のアルファベット順のリストに KEY が現れるかどうかを判定し、現れる場合はリスト上の位置を表示する。リスト上の位置を表示する]。1. [リスト上の k 番目の単語が KEY である場合、その位置を出力し、アルゴリズムを終了する。] $k = 1 - n$ まで (a) $W(k) = KEY$ なら k を出力して停止する。2.Not on list を出力する。確かに、アルゴリズム 2.2 が終了するまでに、n 回もの比較 (リスト上の最新の単語が KEY であるかどうかのチェック) を要するかもしれない。したがって、複雑さを直接 ne 単位で測るのではなく、比較の回数で測るなら、このアルゴリズムの（最悪の場合の）複雑さは、n または $O(n)$ となる。この逐次探索アルゴリズムは、リスト上の単語がアルファベット順に与えられているという事実を利用していないことに注意されたい。次に、アルファベット順のリストの中から与えられた単語を検索する「分割統治（divide-and-conquer)」技法を説明する。分割統治とは、問題をより小さなサブ問題に分割することで、より簡単に解決できるようにすることである。より簡単に解くことができる、という意味である。バイナリサーチアルゴリズムは、アルゴリズム 2,2 と同じ課題を達成するが、より効率的である。つまり、最悪の場合、大きい数に対して比較回数が大幅に少なくなる。x を実数とする。x の階 x は、x より小さい最大の整数である。

7　45 ページ目翻訳

を意味し、x の天井（x]）は x 以上の最小の整数である。x がそれ自体整数である場合、x] =—x=X である。例えば、〜 さらに、〜である。逐次探索アルゴリズムは、リスト上の最初の単語から開始する。これに対して、バイナリサーチアルゴリズムは、リストの途中から開始する。バイナリサーチアルゴリズムは、リストの「mniddle」単語を比較して、それが KEY かどうかを判断する。 この単語が KEY でなく、KEY がこの真ん中の単語にアルファベット的に先行する場合、KEY はリストの最初の単語と中間単語の中間にある単語と比較され、それ以外は、KEY はリストの中間単語と最後の単語の中間にある単語と比較される。リストが n 個の単語で構成されている場合、中間単語とは、—の位置に現れる単語を意味する。仮に $n = 25$ とする。すると、$W(13)$ はリストの真ん中の単語であり、$W(13)$ と KEY を比較することになる。もちろん、$W(13) = KEY$ であれば、KEY はリスト上にあり、13 番目の単語である。$W(13)$ KEY の場合、KEY が $W(13)$ よりもアルファベット的に先行しているかどうかをチェックします。これを KEY< $W(13)$ と書きます。もし、実際に KEY > $W(13)$ であれば、次に KEY と $W(19)$ を比較します。例えば、DOOR という単語が次のリストにあるかどうかを判断したいとします。という単語があるかどうかを調べたいとする：

W(1) W(2) W(3) W(4) W(5) W(6) W(7) W(8) W(9) W(10) = アロー= ボール= 車= ドア= 足= ハンド= ラダー= ネット= パン= TENT

二値探索アルゴリズムを用いて、まず $W(5)$ =FOOT がキーワードの DOOR であるかどうかを確認する。そうではありません。D0OR<FO0T なので、次にサブリスト $W(1), W(2), W(3), W(4)$ の中間語である $W(2)$ =BALL を考える。BALL D0OR だが DOOR>BALL なので、次に DOOR とサブリスト $W(3), W(4)$ の中間の単語、すなわち $W(3)$ =CAR を比較してみる。CAR∗DOOR、しかし DOOR > CAR なので、残る単語はただ一つ、すなわち $W(4)$ となり、単語を発見したことになる。ここで、バイナリサーチアルゴリズムを正式に提示する。アルゴリズム 2.3(バイナリサーチアルゴリズム)。WOD、We、...、Wn の n 個の単語を検索し、もし検索できたなら、リスト上の位置を表示する)。[10 与えられた単語 KEY がアルファベット順のリストに現れるかどうかを判定する。1. f と l は、検討中のサブリスト上の単語の最初と最後の位置を表す。初期状態では、$f = l$、$l = n$ である]。1.1 $f - 1$. 1.2 l n.

8　46 ページ目翻訳

サーチアルゴリズム 2.$W(k)$ は $W()$ と $W(e)$ の中間の単語である] While fs l do (a) k -. (b) [リストの k 番目の単語が KEY の場合 (b)[リストの k 番目の単語が KEY であれば、その lo cation を出力してアルゴリズムを終了する。] W(k) = KEY なら、k を出力して停止する。(c) [KEY が現在のサブリストの中央の単語の前にある場合、新しいサブリストは $W(f)$ と $W(k-1)$ の間の単語で構成される]。KEY $< W(k)$ であれば、$-k-1$ である。(a) U KEY が現在のサブリストの中央の単語に続く場合、新しいサブリストは $W(k+1)$ と $W(e)$ の間の単語からなる]。KEY$> W(k)$ であれば、$f-k+1$ である。3.Not on list を出力する。ここで、バイナリーサーチアルゴリズムの（ワーストケースの）複雑さを決定する。ステップ 1 と 3 はそれぞれ一度だけ実行され、どちらも一定の数の演算を伴うことに注意してください。したがって、これらの複雑さは $O(1)$ である。ステップ 2 が実行される最大回数を $B(n)$ とすると、$B(n)$ は、ステップ 2 が実行される最大回数を表す。確かに、この最大回数は、キーワードがリストに現れないときに発生する。ステップ 2 を初めて実行した後、検討すべき単語は最大で $5J$ 個のリストとなる。したがって $B(n)s1+B()$ となる。強引な帰納法を用いて、次のことを証明する。$B(n)=1+log, n=1+logn$ となる。(2.1) (2.2) $n=1$ の場合、ステップ 2 は一度しか行われないので、この場合、不等式 (2.2) が満たされる。ここで、(2.2) が 1¡ n s k の整数 n について、ある正の整数 k について成立すると仮定し、次のことを示す。B(k + 1) ¡1+log(k + 1) である。B(k + 1) s1+B(—) である。B(k1) s 1 + log(—!) I +log()、最初の不等式は帰納的仮説から導かれる。したがって、B(k + 1)s2+ log(k + 1)- log 2 =1 + log(k +) となり、B(n) = O(log n) となる。ステップ 2 を実行するたびに、最大で 2 つの比較と 2 つの代入演算が必要になるのでの演算が行われるので、ステップ 2 の複雑さは B(n) のオーダーになる、すなわち、O(log n) となる。(2.1) による、ただし

となり、最初の不等式は帰納的仮説から導かれる。したがって

B(k+1) ¡= 2 + log (k+1) - log 2 = 1 + log (k+1) です。

となり、所望の結果が得られる。その結果、B(n) = O(log n) となる。Step2 を実行するたびに、最大で 2 つの比較が必要となり、2 つの割り当て操作が行われるため、結果として Step2 の複雑さは B（n）のオーダー、すなわち O（log n）である。

9　47 ページ目翻訳

2.3 ソートアルゴリズム

セクション 2.2 で説明した検索アルゴリズムでは、与えられたリストがアルファベット順であると仮定した。この節では、与えられた単語のリストをアルファベット順に並べるための 2 つのアルゴリズムについて説明する。最初のものは、順次ソートまたは選択ソートアルゴリズムと呼ばれ、アルゴリズム 2.1 を用いて、リスト上のアルファベット順の最初の単語を探し出すものである。アルゴリズム 2.4（逐次ソートアルゴリズム n)。[n 個の単語からなるリスト Wl)、W(2)、...、W(n) をアルファベット順に並べる]。1.k = 1 から n -1 まで (a) アルゴリズム 2.1 を適用して、サブリスト W(k), W(k+ 1),..., W(n) から、最初の単語である FIRST = W(p) を、アルファベット順に求める。(b) ［サブリストのアルファベット順の最初の単語と、サブリストの最初の単語を入れ替える］。W(p) - W(k). W(k) ¡ FIRST.2.[単語はアルファベット順に出力される] 。i = 1〜n の場合 W(i) を出力する。順次ソートのアルゴリズムをリストで説明する：w(1) = fence, w(2) = side, w(3) = car, w(4) = gate.n = 4 なので、k = 1、k = 2、k = 3 に対応するように、ステップ la と 1b を 3 回実行する。

オリジナルリスト（=) フェンスサイドカーゲートステップ 1（k CAR サイドフェンスゲート=) ステップ 1 (k = 2) CAR フェンスサイドゲートステップ 1 (k = 3) CAR フェンスゲート 側面アルゴリズム 2.4 の複雑さを理解するために、まず、アルゴリズム 21 は最大で n - 1 個の比較を必要とすることを思い出してください。アルゴリズム 2.1 を 2 回目に適用した場合、サブリストは n - l 個の単語を持つ。このようにすると、必要な比較は最大でも (n - 1) + (n -2)+ **+3+2+1 = () = n(n 1)/2 比較となる。(この式は数学的帰納法で検証できる。) したがって、アルゴリズム 2.4 の計算量は $O(n)$ である。単語のリストをアルファベット順に並べるもう一つの一般的なアルゴリズムは、リストの最後から始めて、リストを通過するたびに連続する 2 つの単語を比較します。2 つの単語がアルファベット順でない場合、単語は交換される。同じリストを用いて、この方法を説明します。

元のリストフェンスサイド車のゲート一次通過フェンスカーサイドゲートセカンドパス CAR フェンスゲートサイドカーフェンスゲートサイドサードパス CAR フェンスゲートサイドこのアルゴリズムは、アルファベット順の最初の単語がリストの先頭に「バブルアップ」することから、バブルソートアルゴリズムと呼ばれています。また、時折、交換ソートアルゴリズムとも呼ばれることがある。アルゴリズム 2.5(bubblesort アルゴリズム)。[n 個の単語のリスト W)、W（2)、...W（n）をアルファベット順に並べる]。1.

j = l から n 1 まで [最初の j -1 個の単語をアルファベット順に並べると、WU), WO + 1), ..., W(n) の単語だけがソートされる必要がある。]i=1〜n-j の場合 [リスト Wy)、Wy + 1)、..... の連続した単語をチェックする。W(n) を下から順にチェックし、所望の交換を行う]。W(n +1 - i)¡W(n -i) であれば、(a) W(n - i) を TEMP する。(b) W(n - i) W(n +1 - i). () W(n + 1 -)=TEMP.2. (単語はアルファベット順に出力される。)k = 1〜n の場合 W(k) を出力する。フェンスサイド車ゲート車フェンスサイドゲート

10　48 ページ目翻訳

bubblesort アルゴリズムを使用する場合、n 個の単語のリストを最初に通過させるには、n -1 個の比較が必要です。アルファベット順の最初の単語がリストの一番上に置かれると、次のパスでは一番下の n - l 個の単語を見て、n 2 個の比較を行う。このように続けると、比較の総数は最大で n(n-1) 個になることがわかる。2 + 3 + 2 + 1 =なので、複雑さは O(n) となる。これは、逐次ソートと同じ複雑さである (n- 1) + (n-2) + となります。のアルゴリズムと同じ複雑さです。別のソートアルゴリズムは分割統治法を用いており、ムネージソートとして知られている。n¿ 2 の場合、このアルゴリズムはまず n 個の単語からなるリスト L: W(), W(2), ..., W(n) を、L の約半分の長さの 2 つのサブリスト Li と L2 に分割します。より具体的には L—:W(1), W(2),..., W(3I) と Lz:W（l+1)、W（L3—+2)、...、W（n）である。次に、アルゴリズムは、自分自身を呼び出すことによって、Li と L をアルファベット順にソートする。最後に、2 つの（アルファベット順の）リストが「マージ」される。これらのリストのそれぞれから最初の要素が比較される。これらの 2 つの単語の間で最初にアルファベット順に並んだ単語は、アルファベット順に L の最初の単語となる。この手順を残りの 2 つのサブリストで繰り返し、L 上のアルファベット順の次の単語を決定する。なお、2 つのサブリスト Li と L2 をマージするためには、最大で n l 回の比較が必要である。問題 5 では、マージソートに必要な比較回数は最大 O(n log n) であることを示せということだ．他にも多くのソートアルゴリズムがある。その中で興味深いのは、クイックソート（[1, 4] を参照）とヒープソート（[1] を参照）である。クイックソートの複雑さは O (n) であり、シーケンシャルソートやバブルソートと同じであるため、「クイックソート」という名称は少し語弊があるような気がします。一方、ヒープソートは複雑さが O(n log n) である。もちろん、wrono Mse のこれらのソートアルゴリズムは定義上すべて効率的である。第 1.5 節では、p 個の非負整数の並びがグラフであるかどうかを判定する手順を説明した。ここで、これをアルゴリズムとして正式に再定義する。アルゴリズム 2.6. [p (2 1) 個の非負整数の並びが図形的であるかどうかを判定する] 。1. もし数列に p -1 を超える項があれば、その数列は図形的でない。図式的でない。2. 数列のすべての整数が 0 である場合、数列は図形的である。もし、数列に負の整数が含まれていれば、その数列は図形的でない。そうでない場合は、ステップ 3 に進みます。必要であれば、現在の数列の数字を並べ替え、nou 4. 最初の数字（n）を数列から削除し、次の数列から 1 を減算する。n 個の数字から 1 を引く、ステップ 2 に戻る。ステップ 1、2 の複雑さは O(p) である。ステップ 3 では、P 個の数字の並べ替えが必要な場合がある。例えば、バブルソートアルゴリズムを使用する場合、ステップ 3 では 0(p) 個の比較が必要になる。ステップ 4 が発生するたびに、最大で p - 1 の減算が発生する。

11　49 ページ目翻訳

ステップ 4 が発生するたびに、最大で $p-1$ の減算が行われ、つまり、ステップ 4 では $O(p)$ の演算が行われる。ステップ 2 〜 4 では、バブルソートアルゴリズムを使用する場合、$O(p^2)$ 個の比較が必要である。ステップ 2-4 は p 回繰り返すことができるので、このアルゴリズムの複雑さは $O(p^3)$ となる。もちろん、複雑度が低次のソートアルゴリズムを使用すれば、アルゴリズム 2.6 の複雑度は改善されるだろう。

2.4 NP-completeness の導入

2.1 節で指摘したように、すべての問題は扱いやすいか扱いにくいかのどちらかである。しかし、多くの問題では、その問題がこの 2 つのうちどちらに属するかはわからない。このような問題の中に、「NP 完全」問題のクラスがある。これらの問題は、これらの問題の 1 つを解く効率的なアルゴリズムが、これらの問題のすべてに対する効率的なアルゴリズムを保証するという意味で、すべての問題が本質的に同じである。つまり、これらの問題のうち 1 つが難解であることが知られている場合、他の問題も同様である。なお、これは NP 完全問題の性質に過ぎず、定義ではない。

12　50 ページ目翻訳

なぜこれらの問題が研究に値するのか疑問に思う人もいるかもしれません。この分野の多くの専門家は、すべての NP 完全問題は解決困難であると考えているため、これらの問題の 1 つを解決するための効率的なアルゴリズムを見つける試みは優先順位が低いはずです。ただし、問題が NP 完全であることがわかったからといって、問題を放棄する必要があるわけではありません。むしろ、この知識が自分のアプローチの方向性を導くはずです。たとえば、問題の特定の特殊なケースを解決する効率的なアルゴリズムを見つけようとしたり、条件を緩和して効率的に解決できる新しい問題を取得したりすることができます。最適化問題の場合、常に「最適に近い」解を保証する効率的なアルゴリズムの開発を試みることができます。採用されているいくつかのアプローチについては、後続の章で説明します。

ここで、NP 完全問題について非公式に紹介します。正式なアプローチでは、計算モデルとして「チューリング マシン」を使用する必要があります。これは、Garey と Johnson の書籍に記載されています [7. 第 2 章]、イーブン [6]、ウィルフ [9]。NP 完全問題をより注意深く説明するには、次の用語が必要です。問題のインスタンスとは、問題のパラメーター (つまり、入力) を完全に記述する特定の値または数量のセットを意味します。問題を記述するために使用される方法は、エンコード スキームとも呼ばれます。グラフに適用される問題の場合、適切なエンコード スキームは通常、長さが次の多項式である文字列によって問題のインスタンスを記述するものです。

順序とサイズ。たとえば、グラフ同型問題は、与えられた 2 つのグラフ G と G2 が同型であるかどうかを判断することです。この問題の例は、G1 と G2 の頂点とエッジのセットを指定することによって与えられます。これは合理的なエンコード スキームです。

問題のすべてのインスタンスの出力が「はい」または「いいえ」のいずれかであるという特性を持つ問題は、決定問題と呼ばれます。の「はいインスタンス」または「いいえインスタンス」は、出力がそれぞれ「はい」または「いいえ」になるインスタンスです。したがって、決定問題 は、D- で示される問題 のすべてのインスタンスと、すべての Yes インスタンスのサブセット Y、CD で構成されます。

たとえば、問題: 「G G は (2 つの与えられたグラフ G1 と G2)?” は決定問題です。この例では、Y はすべてのペア G1、G2 で構成されます。

G1 = G2 のようなグラフ - 意思決定問題 の非決定的アルゴリズムには 2 つの段階があります。 の特定のインスタンス I に対して、最初の段階では、証明書 C(I) とも呼ばれる構造 (または潜在的な解決策) を推測します。第 2 段階では、入力 I と C(I) を使用して、I が Y にあるかどうかを C(I) で判断できるかどうかを効率的にチェックします。直感的に、証明書を推測することは、答えを推測することと同じです。たとえば、π をグラフ同型問題 (決定問題として記述) とします。 G1 と G2 が同じ次数列を持つ 2 つのグラフであると仮定します。 の証明書の 1 つの可能性は、V(G) から V(G2) への 1 対 1 関数です。このような証明書が推測されると、if uv E E(G) の場合および du v E E(G2) の場合に限り、それが同型写像であるかどうかを効率的にチェックできます。この場合、チェック段階の複雑さは O(g) であることに注意してください。

証明書の選択には意味がある必要があります。特に、π が決定問題である場合、D のすべてのインスタンス に対して、次のことが当てはまります: (1) I EY の場合、その後

13　51 ページ目翻訳

(i) と (ii) の両方が成立する場合、非決定的アルゴリズムが π を解決すると言います。決定問題 π を解く非決定的アルゴリズムのチェック段階を (効率的に) 実行できるコンピューターまたはマシンがあると仮定します。また、これらのコンピュータが無制限に存在すると仮定します。 π のインスタンス I に対して考えられる各証明書 $C(I)$ が少なくとも 1 回推測され、これらのコンピューターのいずれかへの入力として I とともに提供された場合、π は 効率的に解決できるようになります。グラフ同型問題と上記の証明書では、$V(G_1)$ から $V(G_2)$ までの $p(G_1)!$ 全単射関数があるため、$p(G_1)!$ の可能な証明書が存在します。

NP 問題のクラスは、合理的な符号化スキームの下で非決定的アルゴリズムによって解決できるすべての決定問題のクラスとして定義されます。NP という用語は、非決定的多項式を表します。 NP 問題のセットは NP で示されます。

π_1 のすべてのインスタンス I をインスタンスに変換する、多項式変換と呼ばれる効率的なアルゴリズム A が存在する場合、問題 π_1 は問題 π_2 に多項式変換できると言います。 π_2 の $A(I)$ なので、I IN Y_{π_1} となり、pi_1 のすべてのインスタンス I が π_2 のインスタンス $A(I)$ に変換されます。 $A(I) \in Y_{\pi_2}$ がある場合に限り、$I \in Y_{\pi_1}$ になります。 π_1 を π_2 に多項式変換できる場合は、π_1 inif π_2 と書き、π_1 を π_2 に多項式に変換できるとします。

π_1 が問題であるとします。「与えられたグラフ G に対して、$V(G)$ には、S の 2 つの頂点が隣接しないように、k 個の頂点のサブセット S が含まれていますか?」 π_2 を問題とします: 「与えられたグラフ G に対して、$V(G)$ には S の 2 つの頂点がすべて隣接するように k 個の頂点のセット S が含まれていますか?」 π_1 inif π_2 は、π_1 のすべてのインスタンス G を π_2 のインスタンス、つまり Gb に多項式変換できるためです。G の次数が p の場合、この変換には $O(p^2)$ ステップが必要です。

NP 完全問題のクラスは、NP 内の他のすべての問題が π に多項式変換できるという特性を持つすべての NP 問題 π で構成されます。 NP 完全問題を定義したので、そのような問題が存在するかどうかを尋ねることもできます。結局のところ、NP のすべての問題が多項式で NP に還元できるという性質に問題が生じるのはなぜでしょうか? しかし、1971 年に Cook[3] は最初の NP 完全問題の存在を確立しました。 彼は、次に説明する「充足可能性」問題が NP 完全であることを示しました。

ブール変数は、値 true (T) または false (F) を取ることができる変数です。 x_1、x_2、...、x_n をブール変数とします。ブール変数 x_i の否定 x_ib もブール変数で、x_i が F の場合に限り T になります。リテラルは、変数 x_i またはその否定 x_ib のいずれかです。句はリテラルのセットです。 少なくとも 1 つのリテラルが変数 $x_1, x_2, \ldots, x_n$ に対する真理値を持っている場合、節は変数 $x_1, x_2, \ldots, x_n$ への真理値の特定の代入に対して満たされていると言います。 少なくとも 1 つのリテラルが真理値 T を持つ場合。節のセットは、各節が満たされるように変数 x_1、x_2、$\cdots$、x_n に真理値が割り当てられている場合に満たされます。直感的には、各節のリテラルの間に or を配置し、節の間に and を配置することを考えてください。 次に、この複合ステートメントを true にする、変数 x_1、x_2、$\cdots$、x_n への真理値の割り当てがあるかどうかを判断します。

14　52 ページ目翻訳

たとえば、句 $\{x_1, \overline{x}_3\}$、$\{x_1, \overline{x}_3\}$、$\{\overline{x}_1, x_3\}$ は満たされます。これを確認するには、$x_1 = x_2 = x_3 = T$ とします。 ただし、句 $\{x_1, \overline{x}_2, \overline{x}_3\}$、$\{x_1, x_2\}$、$\{\overline{x}_1, x_2\}$、$\{\overline{x}_2, \overline{x}_1\}$、および $\{x_3\}$ は使用できません。 満足できる。 そうであれば、$x_3 = T$ となります。さらに、$\{x_1, x_2\}$ と $\{\overline{x}_1, x_2\}$ は両方とも文節であり、x_1 と $\overline{x}_1$ は異なる真理値を持っているため、$x_2 = T$、つまり $\overline{x}_2 = F$。 $\{\overline{x}_2, \overline{x}_1\}$ は句であるため、$x_1 = F$ になります。 ただし、$\{x_1, \overline{x}_2, \overline{x}_3\}$ は満たされません。

SAT で示される充足可能性問題では、次のように述べられています。条項のセットが与えられた場合、それらは充足可能ですか? したがって、充足可能性の問題は決定問題です。 SAT の適切なエンコード スキームは、SAT のインスタンス I を、I 内のリテラルと句の数の多項式となる文字列で記述するものです。

値 x_1、x_2、. . .、x_n への真理値の割り当ては 2^n 通りあることに注意してください。これらすべての可能性を試すことで、SAT を解決できます。 ただし、文節の数が n の多項式である場合、このアルゴリズムは指数関数的に複雑になるため、(このような場合には) 効率的ではありません。I が SAT のインスタンス、つまり節のセットであり、$C(I)$ が変数への真理値の代入である場合、$C(I)$ が I の各節が次のようになるかどうかを効率的にチェックできます。 満足。各節が満たされるように、真理値 $C(I)$ が I の変数に割り当てられる場合に限り、I は充足可能であるため、SAT は NP に属します。 ただし、SAT が NP 完全問題であることを示すのは非常に複雑です。したがって、この結果を証拠なしで述べます。

定理 2.1。 SAT は NP 完全です。

定理 2.1 で使用されているテクニックのいくつかを説明するために、例を考えてみましょう。π を決定問題とします: 与えられたグラフ G について、V_i の 2 つの頂点が隣接しないように $V(G)$ を 3 つのセット V_1、V_2、V_3 に分割することは可能ですか G では、$1 \leq i \leq 3$? G を図 2-4 に示す π のインスタンスとします。

ここで、この π のインスタンスを SAT のインスタンスに変換します。$x_{i,j}$ をステートメントとします: 頂点 i は $V_j (1 \leq i \leq 6$ および $1 \leq j \leq 3)$ に属します。各 $i (1 \leq i \leq 6)$ に対して、次の句を構築します。

$$A(i) = \{x_{i,1}, x_{i,2}, x_{i,3}\}$$

$$B(i) = \{\overline{x}_{i,1}, \overline{x}_{i,2}\}$$

$$C(i) = \{\overline{x}_{i,1}, \overline{x}_{1,3}\}$$

$$D(i) = \{\overline{x}_{i,2}, \overline{x}_{i,3}\}$$

15　53 ページ目翻訳

これらの節が満足できる場合、節 A(i) は頂点 i が少なくとも 1 つの V_j に属することを保証し、節 $B(i)$、$C(i)$、および $D(i)$ は次のことを保証します。 各頂点 i は最大 1 つの V_j に属します。したがって、これらの条項は合わせて、それぞれの

各辺 $e = uv$ および各 $j = 1$、2、3 について、次のようにします。

$$F(e, j) = \{\overline{x}_{u,j}、\overline{x}_{v,j}\}$$

。

これらの節が満足できる場合、G の隣接する 2 つの頂点が同じ V_j に属さないことが保証されます。したがって、上記の 54 の節が満たされる場合に限り、同じ V_j に属する 2 つの頂点が隣接しないように、$V(G)$ を 3 つのセット V_1、V_2、V_3 に分割できます。

1971 年に最初の NP 完全問題の存在が確立されて以来、豊富な NP 完全問題が呼び出されてきました。これらの問題の多くについての良い情報源は、Garey と Johnson の本です [7]。NP 内の π 問題が、NP 内のすべての π'' に対して $\pi' \propto \pi$ となるような NP 完全問題であることを示します。 したがって、π は NP 完全です。

2 番目に特定された NP 完全問題という栄誉は、3 つの充足可能性問題に与えられました。または 3SAT: それぞれ最大 3 つのリテラルを含む特定の文節のセットは満たされますか? 前述のアプローチを適用して、この問題が NP 完全であることを示します。

定理 2.2。 3SAT は NP 完全です。

証拠。SAT は NP の問題であるため、3SAT も同様です。ここで、SAT inf 3SAT を示します。 I を SAT のインスタンスとします。少なくとも 4 つのリテラルを持つ各句を、それぞれに 3 つのリテラルが正確に含まれる句のコレクションに置き換えます。

$x_1^*, x_2^*, \ldots, x_k^*$ が C を満たすように $x_1, x_2, \ldots, x_k$ に真理値を代入するとします。 次に、少なくとも 1 つの $r(1 \leq r \leq k)$ について、$x_r^* = T$ となります。$z_1^*, z_2^*, \ldots, z_k^*$ を $z_1, z_2, \ldots, z_k$ への真理値の代入とし、$z_i^* = Tfor 1 \leq i \leq r$ とします。-2 および $z_i^* = Fforr - 2 < i \leq k - 3$。このようにリテラル $x_i(1 \leq i \leq k)$ および $z_j(1 \leq j \leq k - 3)$ に真理値を代入すると、新しい各節の少なくとも 1 つのリテラルが真理値を持ちます。 T.

逆に、(2.3) の条件が満たされるとします。次に、少なくとも 1 つの x_i に真理値 T が割り当てられます。そうでない場合は、$\{x_1, x_2, z_1\}$ が満たされるため、z_1 には真理値 T を割り当てる必要があります。このように続けると、z_i はすべての $i(1 \leq i \leq k - 3)$ に対して真理値 T を持たなければならないことがわかります。ただし、$\{x_{k-1}, x_k, \overline{z}_{k-3}\}$ 節は満たされません。

16　54 ページ目翻訳

SAT のインスタンス I から 3SAT のインスタンス I' への変換が多項式変換であることを示すことはまだ残っています。I に m 個の節と n 個の変数があるとします。 この場合、I' には最大 $(2n-2)m$ 個の節が含まれます。これは、I の各節には最大 $2n$ 個のリテラルがあり、したがって最大 $2n-2$ の新しい節で置き換えられるという事実からわかります。さらに、I' には最大でも $n+(2n-3)m$ 個の変数があります。

P 問題のクラスは、すべての扱いやすい決定問題で構成される集合であり、この集合を P で表します。P 内のすべての問題も NP に属します。 推測の段階は実際には冗長です。単に問題を解決して証明書を無視するだけでチェック段階を効率的に実行できるため、推測として空の証明書で十分です。NP $\subseteq$ P かどうか、つまり NP = P かどうかは、おそらく今日の主題における最も重要な未解決 (決定) 問題です。NP に属する決定問題に限定していることに言及しておく必要があります。 $\pi_1 \propto \pi$ のような NP 完全問題 π_1 が存在する決定問題 π は、NP 困難と呼ばれます。したがって、NP 困難問題は、少なくとも NP 完全問題と同じくらい解決が困難です。

2.5 貪欲なアルゴリズム

このセクションでは、共通の特性を持つアルゴリズムのクラスについて説明します。貪欲なアルゴリズムとは、その選択のその後の影響に関係なく、各ステップで可能な限り最善の選択を行うアルゴリズムです。後で、グラフ理論における貪欲アルゴリズムの非常に強力な例に遭遇します。ここでは、貪欲アルゴリズムの簡単な例について説明します。

旅行者が海外旅行のためにスーツケースに荷物を詰めています。彼女はいくつかの品物を持ち歩きたいと考えていますが、スーツケースごとに r ポンドの重量制限があります。彼女が梱包したい n 個のアイテム 1、2、. . . 、n があります。各項目 i は重み w(i) と値または重要度係数 $v(i)$ を持ちます。彼女の問題は、最大でも r ポンドまで梱包し、梱包された品物の価値を最大化することです。これはナップザック問題とも呼ばれます。

この問題を解決するために、貪欲なアルゴリズムを使用します。項目の値が昇順でないように項目がリストされているとします $(v(1) \leq v(2) \leq \cdots \leq v(n))$。旅行者はまず、重量が最大でも r ポンドである最も価値のある品物を選択します。さらにアイテムを追加すると重量制限を超えるまで、この方法を続けます。このアルゴリズムを正式に紹介します。

17　55 ページ目翻訳

アルゴリズム 2.7。

[スーツケースに詰めるアイテムを n の中から選択する貪欲なアルゴリズム。 項目は 1、2、. . .、n で表され、項目 i は重み $w(i)$ と値 $v(i)$ を持ちます。 さらに、r は重量制限を示し、S は梱包されるアイテムのセットを示します。]

1. $v(1) \leq v(2) \leq \cdots \leq v(n)$ になるように項目を並べ替えます。

2. [梱包するアイテムの集合 S を Φ として初期化する。 変数合計重量と合計値は 0 に初期化されます。]

2.1S ←Φ。

2.2 総重量 ← 0.

2.3 合計値 ← 0.

3. i = 1 〜 n の場合

[新しい総重量が重量制限を超えない場合、梱包する品目のセット S に品目 i を含めます。 S に i を追加すると、合計重量と合計値が増加します。] 総重量 $+w(i) \leq r$ の場合、

(a) $S \leftarrow S \cup \{i\}$。

(b) 総重量 ← $+\, w(i)$

(c) 合計値 ←合計値 $+\, v(i)$。

4. 出力 S、合計値。

ステップ 1 では、値が増加しない順に n 項目を並べ替える複雑さは、もちろん並べ替えアルゴリズムの選択によって異なります。 たとえば、バブルソート アルゴリズムを使用する場合、ステップ 1 の複雑さは $O(n^2)$ になります。 ステップ 3 と 4 の複雑さは $O(n)$ であるため、アルゴリズム 2.7 の複雑さは $O(n^2)$ になります。 したがって、アルゴリズム 2.7 は効率的なアルゴリズムです。具体的な例として、8 つの商品を梱包したいとします。 それらの重みと値を図 2-5 に示します。 さらに、スーツケースの制限重量が 44 ポンドであるとします。つまり、$r = 44$ となります。 次に、アルゴリズム 2.7 に従って梱包される品目を図 2-5 に示します。したがって、アルゴリズムはアイテム 1、2、5、および 8 を選択しました。これらのアイテムの総重量は 44 ポンド、合計値は 25 です。ただし、この選択は最適な解決策ではありません。 たとえば、アイテム 1、3、4、および 6 を選択した場合、合計の重みは 44 になり、合計値は 28 になります。これは、貪欲なアルゴリズムは効率的ですが、可能な限り最高の値を与える必要はないことを示しています。 問題の解決策。 最適なソリューションを段階的に導き出す唯一の方法が非効率なアルゴリズムを使用することである場合さえあります。

18　56 ページ目翻訳

実際、C が正の定数である場合、ナップサックの問題のインスタンス I が存在するように、$V(I)$ がアイテムの最大値になるように示すことができます（問題 3 を参照）これは梱包でき、$V'(I)$ は、アルゴリズム 2.7 で選択されたアイテムの値であり、$CV'(I) < V(I)$ です。 したがって、効率的なアルゴリズムの使用を主張する場合、最高のアルゴリズムに満足する必要がある場合があります。

19　57 ページ目翻訳

コンピューター内のグラフ表現

この章ではさまざまなアルゴリズムについて説明しましたが、グラフを扱ったのは 1 つだけです。 ただし、残りのテキストでは、紹介するアルゴリズムにはすべてグラフが含まれ、コンピューターでそれらを表現する方法を知る必要があります。

まず、コンピューターに情報を保存するために一般的に使用されるいくつかのデータ構造について説明します。特に、頻繁に使用する 3 種類のリストがあります。 「リスト」という言葉を非公式に使用して、オブジェクトの配置を意味しました。より正式には、リストは多くの要素で構成されており、それぞれが単一の「フィールド」であるか、いくつかのフィールドに分割されます。フィールドは一般に、頂点、エッジ、または「ポインター」を表します。

たとえば、データ構造の要素が次のような 3 つの数量のリストに関連付けられているとします。

名前ポインター

フィールド「名前」には、文字列の文字列が含まれています。 フィールド「番号」には正の整数が含まれ、フィールド「ポインター」には非陰性整数、つまり、データ構造の別の要素のアドレスであるポインターが含まれています。したがって、ポインターは別の要素を「ポイント」します。 このタイプのリストは、リンクリストと呼ばれます。ポインターは通常、データ構造の図に矢印として描かれます。

要素には、データ構造の他の要素を指している場合、null ポインターがあります。null ポインターの値に 0 を使用します。 上記の例では、name（first）= v_1、name（pointer（first））= v_2、および

番号（pointer（pointer（first）））= 6

この最後のケースでは、「最初の」は最初のリストに私たちを配置し、ポインター（最初の）が 2 番目のリストを指し、ポインター（ポインター（最初））ポンツを 3 番目のリストに指します。このリストのフィールド番号の要素は 6 です。

スタックは、上部と呼ばれる挿入と削除が常に一端に作成されるリストです。スタックの上部アイテムは、最近挿入されたものです。 スタックは、「最初のアウトで最後」を意味する AD LIFO リストと呼ばれることがあります。 図 2-6 はスタックを示しています。次に、番号 3 が挿入されます。 次に、3 が削除されます（削除できる唯一の数字です）。次の 4 が挿入され、次に 5 が続きます。その後、5 が削除されます。

20　58 ページ目翻訳

キューは、すべての挿入が「バック」と呼ばれる一方の端で行われ、すべての削除が「フロント」と呼ばれるもう一方の端で行われるリストです。 したがって、キューは「先入れ先出し」リストです。 図 2-7 はキューを示しています。これらの定義を使用して、コンピューターにおけるグラフの表現について説明する準備が整いました。 表現の選択は、アルゴリズムの効率に影響を与えることがよくあります。

グラフの可能な表現の 1 つは、隣接行列を使用することです。G を $V(G) = \{v_1, v_2, \ldots, v_p\}$ の (p, q) グラフとする。G の隣接行列 $A = [a_{i,j}]$ は、次のように定義される $p \times p$ 行列です。

$$a_{i,j} = \begin{cases} 1 & if v_i v_j \in E(G) \\ 0 & otherwise. \end{cases} \tag{1}$$

したがって、A は、主対角線上のすべてのエントリが 0 である対称行列です。 グラフとその隣接行列を図 2-8 に示します。A は $p \times p$ 行列であるため、p^2 のメモリ位置をそのエントリに割り当てる必要があります。G が比較的辺の少ないグラフ (つまり、数値 q/p が小さい) の場合、その隣接行列の多くの位置に 0 が含まれます。したがって、比較的少数の辺には異常に大量のメモリ スペースが必要になります。ただし、この問題は、グラフを隣接リストで表すことによって修正できます。

$V(G) = \{v_1, v_2, \ldots, v_p\}$ のグラフ G を考えます。G の隣接リスト表現。便宜上、v_i は単に $i (1 \leq i \leq 6)$。i 番目の隣接リストでは、i に隣接する頂点が番号順にリストされます。 実際、特定の頂点に隣接する頂点は常に数値順またはアルファベット順にリストされると仮定します。 行は NULL ポインターで終了します。

一般に、(p, q) グラフ G の隣接リストは、$(p + 2q) \times 2$ テーブル T で記述することもできます (図 2-8 を参照)。T の (n, j) エントリは $T(n, j)$ で表されます。ここで、$1 \leq n \leq p + 2q$ および $j = 1$ または 2 です。$1 \leq n \leq p$ の場合、$T(n, 1)$ は空白になります。 $p + 1 \leq n \leq p + 2q$ の場合、$T(n, 1)$ は隣接リストに表示される頂点を表します。

21　59 ページ目翻訳

$(1 \leq n \leq p + 2q)$ の場合、エントリ $T(n, 2)$ は 0 (隣接リストの終わりを示す) または n より大きい行番号を指します。

隣接リスト テーブル T を使用して、グラフ G 内の頂点 $v_i (1 \leq i \leq 6)$ の隣接リストを決定できます。まず $T(i, 2)$ について考えてみましょう。$T(i, 2) = 0$ の場合、v_i は G にあります。 それ以外の場合、$j > i$ の場合は $T(i, 2) = j$。 v_i の最初の頂点の隣接リストが完成します。それ以外の場合、$T(j, 2) = j + 1$ となり、$T(j + 1, 1)$ は v_i に隣接する次の頂点になります。 この手順は、いくつかの $n \leq p + 2q$ について $T(n, 2) = 0$ になるまで続きます。

22　60 ページ目翻訳

具体的には、$\deg v_i = d > 0$ とします。 $T(i,2) = j$ の場合、$j > 0$ となり、v_i に隣接する最初の頂点は $T(j,1)$ になります。 v_i に隣接する d 頂点は、実際には $T(n,1)$ $(n = j, j+1, \ldots, j+d-1)$ です。$T(j+d-1,2) = 0$ のとき、v_i の隣接リストが完成します。

たとえば、図 2-8 のグラフ G の頂点 v_2 について考えてみましょう。 エントリ $T(2,1)$ は空白ですが、$T(2,2)$ はポインタ 9 です。 そこで $T(9,1)$ を見てみましょう。 $T(9,1)$ は 1 なので、頂点 v_2 は v_1 に隣接します。 ここで $T(9,2)$ はポインタ 10 です。エントリ $T(10,1)$ は 3 です。 したがって、v_2 も v_3 に隣接しています。 エントリ $T(10,2)$ はヌル ポインタ 0 です。

隣接リストを使用して (p,q) グラフ G を表現するには、$2p + 4q$ の場所が必要です。 G に含まれるエッジが比較的少ない場合、たとえば q が p 内で線形である場合、コンピューターでの G の表現には $O(p)$ の位置が必要です。 これにより、$O(p^2)$ の位置を使用する隣接行列表現が改善されます。

これらの表現はダイグラフにも適用されます。 隣接行列 $V(D) = \{v_1, v_2, \ldots, v_p\}$ を持つ (p,q) 有向グラフ D の $A = [a_{i,j}]$ は p によって定義される p 行列の倍

もし...

有向グラフの隣接リストは、i 番目の隣接リストで、v_i が隣接する頂点が (番号順に) リストされることを除いて、グラフの隣接リストと同じです。 繰り返しますが、各リストは null ポインターで終わります。(p,q) ダイグラフの隣接リスト テーブルには $p + q$ 行があります。