Vol.2018-AL-169 No.10
2018/9/3

最大次数3のグラフにおけるnビット未満の作業領域を用いた深さ優先探索

川端 祐也[1]　泉 泰介[1,a)]

概要：一般的なスタックを用いた深さ優先探索では，入力で与えられるグラフネットワーク以外に，$O(n \log n)$ビットの作業領域を必要とする．本研究では，必要な時間計算量を多項式時間で抑えつつ，使用する追加の作業領域をより少ないスペースで深さ優先探索を達成することを目標とする．先行研究では，一般のグラフにおいて$n + o(n)$ビットの作業領域で深さ優先探索を達成するアルゴリズムが提案されている．本稿では，最大次数3の無向グラフに対して$0.984n + o(n)$ビットの作業領域を用いて深さ優先探索を行うアルゴリズムを提案する．著者らの知る限り，このアルゴリズムは非自明なグラフクラスに対してnビット未満の作業領域を用いて深さ優先探索を行う初めてのアルゴリズムである．

キーワード：領域制限アルゴリズム，空間複雑性，深さ優先探索

Depth First Search Using Less than n Bits for Graphs with Maximum Degree Three

Yuya Kawabata[1]　Taisuke Izumi[1,a)]

1. はじめに

近年，理論計算機科学において空間複雑性についての議論が活発化している．これは実世界で扱うデータが大規模化していることに動機付けられており，古典的な計算問題に対する様々なモデルについて，空間複雑性の観点からアルゴリズムを見直す動きがみられる．本研究では，特に作業領域を制限した深さ優先探索アルゴリズムについて検討を行う．作業領域を制限したモデルでは，入力データは読み取り専用のメモリに内蔵されているものとし，計算プロセスでは追加の作業領域を用いることができるものとする．作業領域を制限したモデル上ののでのアルゴリズム設計における理論的な興味は，主に入力データサイズに対して準線形規模の作業領域を用いて対象としている問題を解くことができるかどうかという点にある．特に，計算量理論の分野においては，対数サイズ作業領域（作業領域が入力データサイズの対数サイズ）の場合において解くことができる問題について (クラス L) の問題について古くから研究がされており，L≠P が成立するかどうかが大きな未解決問題として認識されている．本研究で検討する辞書式順深さ優先探索問題 (Lex-DFS) は，クラス P に完全に属しており，対数作業領域では解くことができないと予想されている問題の一つである．この観点から見た場合，Lex-DFS の空間複雑性の解明は理論的には非常に興味深い問題であるが，一方で上界に関してはナイーブな解法 ($O(n \log n)$ ビットの作業領域を使用) に対する改善は大きくない．近年，いくつかのグループによりこの上界を $O(n)$ ビットへと下げる試みが示されている [1, 2, 7]．一般のグラフにたいして現時点で知られているもっとも作業領域の少ないアルゴリズムは $n + O(\log n)$ ビットで Lex-DFS を実現するが [1]，n ビットの壁を破るアルゴリズムの構成可能性に関しては未だに未解決である．

本研究では，最大次数が 3 の無向グラフに対して，この問題をわずかではあるが肯定的に解決する．具体的に

[1] Nagoya Institute of Technology, Gokiso-cho, Showa-ku, Nagoya, Aichi, 466-8555, Japan
[a)] t-izumi@nitech.ac.jp

は，$0.984n + o(n)$ ビットの作業領域を用いて多項式時間で Lex-DFS 問題を解くアルゴリズムを提案する．著者らの知る限り，このアルゴリズムは非自明なグラフクラスに対して n ビット未満の作業領域を用いて多項式時間で深さ優先探索を行う初めてのアルゴリズムである．

1.1 関連研究

Lex-DFS の空間複雑性に関しては，初期は主に計算量理論とアルゴリズムの並列化可能性の観点からの限界解明が積極的に行われていた．その端緒は，Lex-DFS の判定問題版に対して，対数領域帰着のもとでの P 完全性が示されたことによるところが大きい [10]．この事実は，Lex-DFS を対数作業領域で計算するアルゴリズムがおそらく存在しないことを示している．一方で，空間複雑性の上界に関しては，ナイーブな $O(n \log n)$ ビット領域を用いたアルゴリズムを上回るものは長らく提示されてこなかった．線形作業領域を用いた深さ優先探索アルゴリズムの研究の端緒は，浅野らによる $O(n)$ ビット作業領域を用いた Lex-DFS アルゴリズムの提示による [1]．この論文においては，$O(n)$ ビットの作業領域を用いた一般のグラフに対する Lex-DFS アルゴリズムがいくつか提示されている．もっとも省スペースなアルゴリズムは $n + o(n)$ ビットの作業領域を用いて Lex-DFS を多項式時間で実現しているが，このアルゴリズムは Reingold の対数作業領域 *s-t* 連結性判定アルゴリズム [11](無向グラフの場合) あるいは Barnes らによる準線形領域 *s-t* 連結性判定アルゴリズム [4]（有向グラフの場合) をサブルーチンとして利用しており，その計算時間は非常に遅い（なお補足しておくと，我々が提案するアルゴリズムはこれらの *s-t* 連結性判定アルゴリズムには依拠していない)．また，DAG，木に関しては準線形作業領域で Lex-DFS を実現するアルゴリズムが示されている．ほぼ同時期に，Elmasry らも同様の線形作業領域アルゴリズムを提案している [7]．このアルゴリズムは $O(n)$ ビットの作業領域を用いている点では浅野らのアルゴリズムと同様であるが，実行時間に関してはより高速である．その後，2-連結成分分解，幅優先探索等の関連問題を含めて，線形作業領域アルゴリズムの検討が数多くなされている [2, 5, 6, 8, 9]．ただし，いずれの研究も主に速度向上，あるいは速度と作業領域サイズのトレードオフの検討が主であり，作業領域の限界追及という意味では，現状においても浅野らによる $n + o(n)$ ビットのアルゴリズムを上回るものは存在しない．

一般に Lex-DFS はスタックを用いて実現することが可能であるが，一つのスタックと準線形作業領域のみで実現可能なあるアルゴリズムについて，その作業領域を削減する一般的な枠組みが Barba らにより提案されている [3]．ただし，この枠組みは入力データの走査があらかじめ決められた順序に従って行われるようなアルゴリズムについてのみ適用可能であり，Lex-DFS アルゴリズムの作業領域圧縮に対して利用することはできない．

1.2 論文構成

本論文は全 5 節で構成される．2 節では，深さ優先探索問題を考えるにあたっての諸定義について述べる．3 節では，提案アルゴリズムのベースとなる Lex-DFS アルゴリズムを述べる．4 節では，提案アルゴリズムの概要について述べる．5 節では，まとめと今後の課題について述べる．

2. 諸定義

2.1 モデル定義

本稿を通して，$[a, b]$ を a 以上 b 以下の自然数の値からなる集合とする．深さ探索問題は，単純無向連結グラフ $G = (V, E)$ 上で考えられる．V は頂点の集合で，$|V|$ を n と表す．以下，$V = [1, n]$ とし，個々の頂点は $[1, n]$ 中の値で表される相違な識別子を持つ．E は辺の集合であり，$E = m$ とする．$N(v)$ を頂点 v に接続している頂点の集合とする．また，$N_d(v)$ を v から距離 d 以下である頂点の集合と定義する．$N(v)$ は v 自身を含まないのに対し，v と v の距離は 0 であるため $N_d(v)$ は v 自身を含むことに注意されたい．

2.2 (辞書式順) 深さ優先探索問題

本稿において，深さ優先探索問題とは，入力として無向グラフ $G = (V, E)$ と探索を開始する頂点 $r \in V$ を受け取り，深さ優先探索における訪問順序に従って頂点列を出力する問題と定義する．本稿では，より強い定義である，辞書式順深さ優先探索 (Lexicographic Depth-First Search: Lex-DFS) 問題をを考える．一般に，グラフ $G = (V, E)$ および始点 $r \in V$ が与えられたとき，複数の深さ優先探索順序が存在するが，Lex-DFS は，この探索順序に制限を課した深さ優先探索である．Lex-DFS においては，現在のの滞在頂点 v において複数の隣接未訪問ノードが存在する場合，v の隣接頂点のリストにおいて先に現れる未訪問頂点を優先的に訪問しなければならないという制約が課される．本稿では以降簡単のため，新たな頂点を訪問するとき，頂点番号が最小の未訪問隣接ノードに必ず訪問しなければならない，すなわち，各頂点の隣接頂点集合は，その頂点番号に従ってリスト中に昇順で格納されているとする．なお詳細は省くが，この仮定は容易に除去することが可能である．

辞書式順深さ優先探索問題の出力は，より形式的には，Algorithm 1 の実行に沿って得られる頂点出力列であると定義される．我々が提案するアルゴリズムの説明のため，Algorithm 1 の記述においては，以下のような手続きを備えた抽象スタックの存在を仮定している．

- Push(v) スタックの先頭に，頂点 v を追加する．
- Pop スタックの先頭から頂点を削除する．
- Top スタックの先頭の頂点を取得する．
- Find(v, x) $N(v)$ に含まれる頂点のうち，未訪問の頂点

(その時点においてまだ一度も Push されたことのない頂点) で頂点番号が x より大きいものを一つ返り値として返す．そのような頂点が $N(v)$ 中に存在しない場合は-1 を返す．

以降本稿では，これらの操作を備えた抽象スタックを DFS スタックと呼ぶこととする．また，$N_{\geq x}(v) = \{u \mid u \in N(v), u \geq x\}$ と定義する．

2.3 グラフの最大次数

ある頂点 v に接続する辺の本数を次数 $\Delta(v)$ と表す．また，グラフの最大次数を $\Delta = \max_{v \in V} \Delta(v)$ と定義する．本論文で提案するアルゴリズムは，$\Delta \leq 3$ であるようなグラフのみを考える．

2.4 簡潔ビットベクトル

本稿で提案するアルゴリズムでは，バイナリ列に関する簡潔データ構造をしばしば用いる．本節ではその手法を説明する．あるバイナリ列の重みを，その列に含まれる 1 の個数と定義する．また，長さ l のバイナリ列 B に対して，$B[i]$ の値を取得するクエリを $access(B, i)$ とする．バイナリ列 B に対して，$access$ をサポートした圧縮データ構造を簡潔ビットベクトルと呼ぶ．一般に，重みが $l/2$ の場合，自明な情報理論的下界より B をそのまま記録することが記憶領域的には最適となるが，重みが $l/2$ よりも十分に小さい場合，データアクセスの可用性を保ったままデータ圧縮により使用領域を削減することが可能である．具体的には，以下の定理が成り立つ．

Theorem 1. 長さ l，重み d であるバイナリ列 B に対して，空間計算量が $d \log \frac{l}{d} + 2d$，$access(B, i)$ に必要な時間計算量が $O(d)$ であるような簡潔ビットベクトルの構築方法が存在する．

以降の提案アルゴリズムの説明のため，上記定理の構築方法の概要について簡単に述べる．$b[i] = \operatorname{argmin}_j \left\{ \sum_{k=1}^{j} B[j] = i \right\}$ とする要素数 d の整数列 b を考える．$b[i]$ はバイナリ列 B において先頭から見て i 番目に現れた 1 の場所を指しているため，$b[1, d]$ の値が全て取得できれば，$access(B, i)$ を実現することができる．ここで，$t = \lceil \log \frac{l}{d} \rceil$ として，配列 b の情報を以下のような 2 つの配列 L, H に分離する．

- $L[i] := b[i]$ の下位 α ビット
- $H[i] := b[i]$ の上位 $\lceil \log n \rceil - t$ ビット

明らかに，$L[i]$ および $H[i]$ の値を得ることができれば $b[i]$ を復元することができる．L について，これを格納するために必要な空間計算量は $dt = d \log \frac{l}{d}$ ビットであるため，当該データ構造の構成のためには H を $2d$ ビットの記憶領域で格納することができればよい．いま，H の階差数列 D を次のように定義する．

$$D[i] = \begin{cases} H[1] & (i = 1) \\ H[i] - H[i-1] & (otherwise) \end{cases}$$

このようにしたとき，$H[i] = \sum_{j=1}^{i} D[i]$ が成り立つ．H が非減少列であることから，D は非負整数列となる．そこで，D の単進表記をした列を E とする．ここで非負整数列の単身表記とは，i 番目の要素の数だけ 0 を並べ，直後に 1 を区切り文字として配置するバイナリ列のことを指す．例として，$\{2, 5, 0, 1\}$ という列を単身表記した場合は 001000001101 というバイナリ列になる．E に含まれる 1 の個数は D の長さと一致するため，ちょうど d 個存在する．0 の個数は $\sum_{j=1}^{d} D[i]$ と一致する．以下の式変形により，0 の個数は d 個以下であることが示せる．

$$\sum_{j=1}^{d} D[i] = H[d] \leq \lfloor \frac{l}{2^t} \rfloor \leq d$$

よって，バイナリ列 E の長さは高々 $2d$ である．任意の i について $H[i]$ を復元することは，E を先頭から確認していくことで $O(d)$ 時間で容易に実現できる．

3. 最大次数 3 のグラフに対して $n + o(n)$ ビットで Lex-DFS 問題を解くアルゴリズム

本節では，提案アルゴリズムのベースとなる，最大次数 3 のグラフに対する Lex-DFS 問題を解くアルゴリズムを提案する (以下このアルゴリズムを 3LDFS と呼ぶ)．2.2 節の深さ優先探索の疑似コードを見てわかるように，DFS スタックの内部を除き，利用されるメモリ領域は頂点番号を記憶するための一時変数のみであり，そのために必要なメモリ量は明らかに $O(\log n)$ ビットである．したがって，本節ではアルゴリズム 1 で用いられる DFS スタックの各手続きについて，そのメモリ効率のよい実装方法を提案する．

3.1 基本アイデア

Lex-DFS 問題におけるスタックを $n + o(n)$ ビットで実現するための基本的なアイデアは，スタックに格納された頂点を根から辿るパス (すなわち，根から現時点での訪問頂点への DFS 木上のパス) を管理することである．スタックに含まれる根以外の頂点の次数が高々 3 である場合，直前に辿ってきた辺を除くと残る辺の本数は 2 以下，すなわち次に向かう頂点の候補は高々 2 であるため，その頂点番号の大小のみを 1 ビットのフラグを用いることでパス全体を保持することができる．スタックに格納されるパスの長さは $n - 1$ 以下であることから，根に接続する辺以外の情報は $n - 2$ ビットあれば格納できる．また，根のみは次頂点の候補として 3 頂点が存在しうるので，$\log n$ ビットを用いて次頂点の番号を陽に保持しておくことで辺情報を格納できる．以上より，合計 $n + O(\log n)$ ビットを用いてスタックを実現することが可能となる．

Algorithm 1 深さ優先探索アルゴリズム

```
1: s ← −1                                  ▷ s = −1 は DFS 木を根から下る方向に探索している状態に相当
2: Push(r)                                 ▷ スタックに始点 r を追加する．
3: r を出力する．
4: while スタックのサイズが空でない do
5:     v_next ← Find(Top(), s)
6:     if v_next = −1 then                 ▷ N(v) に未訪問の頂点が含まれない場合バックトラックを実行
7:         s ← Top()                       ▷ バックトラック元の記録
8:         Pop()                           ▷ スタックから先頭の頂点を削除する．
9:     else
10:        s ← −1
11:        Push(v_next)                    ▷ 未訪問の隣接頂点を一つスタックに追加する
12:        v_next を出力する．
```

3.1.1 アルゴリズムの詳細

本節では，前節で提示した基本アイデアに基づいた DFS スタックの実現について述べる．提案手法において，DFS スタックの内部はバイナリ列 S で管理され，また現在の要素数管理する変数 len を持つとする．列 S の i 番目の要素 $S[i]$ は，$i+1$ 番目の頂点から $i+2$ 番目の頂点へとパスを辿るときにおける次頂点の選択を記録しているものとする $(1 \leq i \leq n-2)$. また $len \geq 2$ であるとき，スタックに格納されるパスにおいて根と隣接する頂点の番号が他の変数 s に記録されているものとする．これらの情報から DFS スタックの各種手続きを実現するために，まず，DFS スタックに頂点 $v \in V$ が含まれるかどうか判定する補助手続き InStack(v) を設計する．図 2 に，手続き InStack(v) の疑似コードを示す．手続き InStack は，スタック情報を利用して根からパスのトレースを逐次的に行うことで v がスタックに含まれるかどうかを判定する．ここで，スタックにおける i 番目の頂点を v_{cur},$i-1$ 番目の頂点を v_{parent} としたとき，$i+1$ 番目の頂点 v_{next} は以下のような処理により決定することができる．

- $\Delta(v_{cur}) = 2$ であるならば，$N(v_{cur})$ のうち v_{parent} でない頂点が v_{next} となる．
- $\Delta(v_{cur}) = 3$ である場合，v_{next} となる頂点の候補は v_{parent} を除いた 2 頂点に絞られるため，$S[j] = 0$ ならば頂点番号の小さい方，$S[i-1] = 1$ ならば頂点番号の大きい方を v_{next} とする．

これを逐次的に繰り返すことで，スタックに含まれる任意の頂点の列挙を行うことができるため，v がスタックに含まれているかどうかの判定が可能となる．

手続き InStack を用いると，DFS スタックにおける手続き Push, Pop, Top は容易に実現することができる．Push, Top の基本実装は手続き InStack と同じ要領で，根からスタックを辿りスタック末尾の頂点 u を特定すればよい．Top の場合はそのまま u を出力として返し，Push の場合は u からみて追加された頂点 v がどちらの方向にあるかを調べ，

Algorithm 2 InStack(v)

```
1: if len ≥ 1 then
2:     if r = v then
3:         Return True.
4: if len ≥ 2 then
5:     if s = v then
6:         Return True.
7: v_parent ← r
8: v_cur ← s
9: i ← 2
10: while i < len do
11:     if S[i − 1] = 0 then
12:         v_next ← min {u ∈ N(v_cur) | u ≠ v_parent}
13:     else
14:         v_next ← max {u ∈ N(v_cur) | u ≠ v_parent}
15:     if v_next = v then
16:         Return True.
17:     v_parent ← v_cur
18:     v_cur ← v_parent
19:     i ← i + 1
20: Return False.
```

0 or 1 をバイナリ列 S の末尾に追加したのち，len の値をインクリメントする．これらの操作はスタックのトレースを一度行うだけで実現できるため，その計算時間は $O(n)$ となる．Pop の場合は S の末尾を除去し，len をデクリメントすればよいため，$O(1)$ 時間での実装が可能である．

残る手続きは Find だけであるが，その実現には，第 1 引数として与えられた頂点の各隣接頂点が既訪問かどうかを記録しておく必要がある．ナイーブな実装として，各頂点に訪問済みかどうかを表す 1 ビットフラグを用意しておくという方法を用いることができるが，その場合 Find の実装には（スタック内情報とは別に）n ビットのメモリを必要とするため，所望の領域計算量を達成できない．本提案アルゴリズムでは，既訪問頂点集合中の一部の頂点のみにフ

ラグを立てることでこの問題を解決する．

3.2 手続き Find の実現

グラフ $G = (V, E)$ において頂点集合 $U \subset V$ が以下を満たすとき，U は G の d-支配集合であると定義する．

- 任意の頂点 $v \in V$ について，$U \cap N_d(v) \neq \phi$ が成り立つ．

提案アルゴリズムにおける Find の実装は，以下のアイデアを用いて既訪問頂点を (ストリーム的に) 重複を許して全列挙することである．

アイデア 1 DFS スタックの内部において，既訪問頂点 (一度 Push された頂点) の集合に誘導される G の部分グラフ G' に対して，頂点数 $n/\log n$ 以下の $2\log n$-支配集合 U を常に維持する．

アイデア 2 グラフ $G(U)$ において，アイデア 1 で構成された支配集合の各頂点に対して，そこを中心とした $2\log n$-近傍中の頂点を全列挙する．

d-支配集合の定義から，上記アイデアが実現されると，すべての既訪問頂点について (重複を許した) その全列挙を行うことができるため，頂点 v が現在の既訪問頂点の集合 U に含まれているかどうかの判定を行うことができる．よって，引数のとして与えられた頂点 v に対して $N_{\geq x}(v)$ を順次訪問済みかどうか確認していくことで，Find(v, x) は実現可能である．

グラフ $G = (V, E)$ に対する DFS における頂点の出力順を $v_1, v_2, ..., v_n$ としたとき，$V_k = \{v_1, v_2, \cdots, v_k\}$ とし，T_k を V_k により誘導される G の Lex-DFS 木の部分木とする (これは必ず連結であることが保証される)．今，G' を，頂点集合が V_k でかつ T_k の辺をすべて含むような G の任意の部分グラフとする．ある定数 d に対して頂点集合 $U = \{v_i \mid i = 1 + cd, c \in \mathbb{N}\}$ は G' の $2d$-支配集合となることは容易に示すことができる．この事実を用いるとアイデア 1 の実現は容易である．すなわち，DFS の出力頂点列のうち，k 個ごとに一つの頂点を支配集合に含めていけばよい．支配集合の記録のためには，2.4 節で示した簡潔ビットベクトルを用いる．$k = \log n$ とすると $|U| \leq n/\log n$ となり，これはサイズ n, 密度 $n/\log n$ 以下のビットベクトルとして表すことができるため，2.4 節で紹介した簡潔ビットベクトル表現により記録するならば，高々 $O(\frac{n \log\log n}{\log n}) = o(n)$ ビットを用いて管理することが可能である．

アイデア 2 では，提案アルゴリズムのスタックの管理方法と同様に，最大次数 3 のグラフではある頂点 v を始点とする任意の頂点への最短経路が距離と同じ大きさのバイナリ列にエンコードできる点を利用する．このことから，ある頂点 v から距離 d 以下の頂点は，$O(2^d)$ 通りのパスを全探索することで列挙することが可能となる．支配集合に属する各頂点を始点とした距離 d 以下の頂点を全列挙すると，d-支配集合の性質より，グラフ中の全頂点を列挙できることがいえる．提案手法においては $d = 2\log n$ としているので，各頂点における近傍列挙の要する時間は多項式時間 ($O(2^d) = O(n^2)$ 時間) に収まる．この方法を用いて既訪問頂点の全列挙を行うためには，支配集合に属する各頂点からのパスを探索する際に未訪問頂点と既訪問頂点の間のカット C に属する辺を使わないようにする必要がある．今，次のような辺集合 F を，現時点でのスタックに含まれている頂点と接している辺のうち，一度も参照されていない辺の集合と定義する．すなわち F は，スタックで管理しているパスに使われていない辺でかつ，使われている辺よりも辞書順で後に現れるような辺の集合である．G から F を取り除いたグラフを考えると，既訪問頂点は一つの連結成分を成しており，その連結成分は既訪問頂点集合により誘導される DFS 木の部分木を含んでいる．なぜならば，F に含まれる辺は，C に含まれている辺であるか，DFS 木における前進辺および後退辺のいずれかであることがいえ，なおかつ F は C を真に含んでいるためである．そのため，F 中の辺を禁止辺としてアイデア 2 の全列挙アルゴリズムを実行することで，既訪問頂点の全列挙が達成できる．ある辺が F に含まれているかどうかは，手続き InStack と同じ要領でスタックのパスとそれに接続する辺を全て調べることで容易に判定が可能であるため，以上より $o(n)$ ビットの追加の作業領域のみで既訪問頂点の全列挙を達成できることがいえる．このときの時間計算量は，

- 支配集合の頂点数が $O(n/\log n)$
- 支配集合に属する各頂点からのパスは $O(2^{2\log n}) = O(n^2)$ 通り
- 辺が分離集合 F に含まれているか判定するために $O(n)$ 時間

であるため，全体としては手続き Find の実行は $O(n^4/\log n)$ 時間で完了する．

4. n ビット未満の作業領域への改善

本節では，3 節で提案したアルゴリズム 3LDFS をベースとして，最大次数 3 のグラフの Lex-DFS 問題を $cn + o(n)$ ビット ($c < 1$) で解くアルゴリズムを提案する．前述の通り，3LDFS においてはパスを管理するスタックを $n + o(n)$ ビットの作業領域を用いて陽に管理することができるため，バックトラックなどの問題を容易に解決することができた．本節で提案する Lex-DFS アルゴリズム LM3LDFS の基本となるアイデアは，簡潔ビットベクトルと Lex-DFS における問題構造を利用して，DFS スタック内の情報を圧縮することである．

4.1 アルゴリズムの概要

3 節の 3LDFS においてスタックに対して行われている操作として，以下の 3 つに分けることができる．

(1) スタックに含まれている頂点を根から順に列挙する．
(2) スタックに頂点（辺）を追加する．

(3) スタックから頂点（辺）を削除する．

よって，$cn + o(n)$ ビット $(c < 1)$ の作業領域のみを用いてこれらの操作をサポートするデータ構造の設計ができればよい．アルゴリズム LM3LDFS では，3LDFS と同様にパスを管理するスタックを構築する．3LDFS との明確な違いは，スタックの長さが長くなりすぎる可能性のあるタイミングにおいて，現在管理しているスタックを圧縮することにある．ここで，時刻という概念を導入する．LM3LDFS において，既訪問頂点の個数が t になったタイミングを，時刻 t と定義する．ある時刻 t においてスタックが管理している根からのパスを σ_t で表すとする．明らかに，任意の $t' \geq t$ について，$\sigma_{t'}$ は σ_t のある接頭部に，時刻 t 以降に構成されたパスが付随している形をとる．この事実を利用して，提案アルゴリズムではある時刻 α におけるスタック σ_α の情報と，その後新たに追加した情報からなるパスの接尾部 σ' を組み合わせて現在のスタック情報を復元する．具体的には，σ_α は時刻 α において後述の圧縮アルゴリズムで圧縮をしておき，任意のタイミングで根から順に列挙をすることが可能な状態にする．このとき，現在の DFS スタックに占める σ_α の接頭部の長さを記録しておくためのカウンタ変数を用意しておくことで，σ_α から必要な長さの接頭部のみを抽出することができる．また，σ' に関しては 3 節のアルゴリズム同様未圧縮の状態で陽にスタックを管理しておく．σ' の長さは時刻 α の時点における未訪問頂点の個数 $(= n - \alpha)$ を超えることはないので，この方法を用いると，時刻 α 以前では，α ビット以下のメモリで $LDFS$ 同様に Lex-DFS を行うことができ，時刻 α 以降では圧縮したスタックに必要な記憶領域と，$n - \alpha$ ビットのメモリの合計の空間計算量で Lex-DFS を実行することができる．以上を踏まえて，DFS スタックの圧縮方法及びパラメータ α の適切な設定について議論を行う．なお，以降の議論では，時刻 t における DFS スタックの内容と，それが管理するパス σ_t を混乱の生じない範囲において同一視して議論する．

4.2 DFS スタックの圧縮

本小節では，事前に設定したパラメータ α に基づいて，時刻 α における DFS スタック σ_α を圧縮する方法について述べる．DFS スタックの圧縮方法を述べるにあたって，次のような頂点集合を定義する．$Lex-DFS$ における頂点の出力順を $v_1, v_2, ..., v_n$ としたとき，

$$A = \{v_i \mid i < \lfloor \alpha/2 \rfloor\}, B = \{v_i \mid i \geq \lfloor \alpha/2 \rfloor\}$$

さらに，スタック σ_α では，接頭辞が全て A に属している頂点で，接尾辞が全て B に属している頂点となるような境界が必ず存在する．このときの接頭辞を σ^*_A，接尾辞を σ^*_B と定義する．また，A と B の辺カットを C と定義する．我々が提案する Lex-DFS アルゴリズムでは，時刻 α 未満のときに，3LDFS の操作に加えて C に含まれる辺の本数を数える操作を追加する．この操作は，$O(\log n)$ ビットのカウンタを用いて既訪問頂点の個数を記録し，スタックに A に含まれる頂点がいくつあるかをカウントしておけば容易に実装できる．なぜならば，DFS における後退辺の性質により，このような辺は $v \in B$ である頂点からみたとき，必ずもう一方の端点はスタック上に存在するからである．そのため，B に属する頂点 v を Push した際に隣接頂点集合 $N(v)$ に対して，v に接続されている辺の反対側の端点が A に属しているかどうか判定することで，カット C の辺の本数をカウントすることができる．

スタックの圧縮方法は C の大きさによって 2 つの方法を使い分ける．具体的には，事前に用意したパラメータ $\beta (0 \leq \beta \leq n)$ を用いて，次のように場合分けを行う．

- $|C| < \beta$ のとき
  この場合は，B-A 間にまたがる辺が少ないため，この辺のみを記憶して σ^*_A と σ^*_B を独立に復元する方針をとる．実際には，C の辺における，A 側に属する端点の集合を簡潔ビットベクトル表現したものと，σ^*_A が管理するパスの末尾ノード (以降 v_A とする),σ^*_B が管理するパスの先頭ノード (以降 v_B とする) の頂点番号を保持しておき，必要に応じて σ^*_A, σ^*_B を復元する．簡潔ビットベクトルの構築は，時刻 α までの DFS をもう一度最初からやり直し，C に含まれる辺を検出するごとに簡潔ビットベクトルへの頂点の動的な追加を行うことで実現できる．
  次に，σ^*_A, σ^*_B の復元方法について述べる．σ^*_A は始点ノード r からの 3LDFS の実行を，頂点 v_A に最小に訪問するまで実行することで復元される (v_A 訪問時点での DFS スタックの状態が σ^*_A に相当する)．この復元は $\frac{\alpha}{2}$ 頂点のグラフに対する 3LDFS の実行に対して必要とするメモリ領域，すなわち $\frac{\alpha}{2} + o(n)$ ビットの作業領域で実行可能である．σ^*_B の復元も基本方針は同様である．頂点 v_B を始点として，C 中の辺を禁止辺として 3LDFS を実行することで復元することが可能である．C 中の辺を禁止辺とするには，訪問先が簡潔ビットベクトルに保持された頂点であるかを判定することで実現できる．
- $|C| \geq \beta$ のとき
  この場合も，先の例と同様に σ^*_A, σ^*_B をそれぞれ独立に復元するという方針をとる．本ケースでは σ^*_B のスタックはそのまま保持するが，σ^*_A については後退辺の性質を利用していくつかの情報を省いて保持するという方針をとる．σ^*_A に属するある頂点 v が，C に属する辺 c に接続しているとする．このとき，c は DFS 木に含まれておらず，かつ c の v ではない側の端点は B に属することより Lex-DFS において v より後に訪問されている．すなわち，c は v 側から見た場合前進

辺であり，σ_A^* の復元においてこの辺は存在しないものとして取り扱っても問題は生じない．そのため，σ_A^* の復元においては，v は仮想的に次数が 2 の頂点として扱うことができる．次数 2 の頂点は，直前に辿った辺を除くと次に進むべき辺が一意に定まるため，v から次に辿るべき辺の情報を DFS スタック内に記憶しておく必要がない．そのためこの辺情報を省略（DFS スタック内部の S が保持するバイナリ列から，当該部分の 1 ビット情報を削除）することで圧縮を行うことができる．

次に，圧縮したスタックから σ_A^* を復元することを考える．基本の操作は手続き InStack と同様で，スタックに書き込まれた辺情報を参照して次の頂点を訪問する．ただし，次の頂点へ向かうための辺情報が圧縮により削除されている可能性がある．次訪問先への辺情報が圧縮により省略されているかどうかの判定は，現在の頂点に隣接した頂点が B に属しているか確認をすればよい．これは，σ_B^* が未圧縮であることから σ_B^* に属する頂点を v_B を始点として手続き InStack を用いて列挙することで容易に判定可能である，隣接頂点が B に属しているときは，そこへ向かう辺は前進辺であるため次訪問先はもう片方の隣接頂点を選ぶことで，正しい次訪問先へと到達できる．

次節において，本アルゴリズムの計算量の解析を行う．

4.3 メモリ使用量および計算時間の解析

圧縮されたスタックの構築及び頂点の列挙に必要なメモリ使用量および計算時間について，場合分けをして議論を行う．

- $|C| < \beta$ のとき
  簡潔ビットベクトルの保持に必要な空間計算量は，密度が n/β 以下であることが保証されるため，$\beta(2+\log n/\beta)$ ビットとなる．圧縮スタックの構築において，C における A 側の端点を記録するために時刻 α までの 3LDFS を再度行うため，構築にかかる時間計算量は 3LDFS の実行と同じく $O(n^4/\log n)$．構築時の空間計算量は，3LDFS に必要な作業領域と簡潔ビットベクトルのメモリ，$o(n)$ ビットの作業領域であるため，$\beta(2+\log n/\beta)+\alpha+o(n)$ ビットとなる．また，頂点の列挙に必要な時間計算量は，3LDFS の計算量に加えて簡潔ビットベクトルへの参照が辺の参照のたびに発生するので $O(n^5/\log n)$ となる．σ_A^*, σ_B^* の各スタックの復元に必要な追加の作業領域は $\frac{\alpha}{2}+o(n)$ ビットであるため，簡潔ビットベクトルの保持に必要なメモリも合わせると，合計で $\beta(2+\log\frac{n}{\beta})+\frac{\alpha}{2}+o(n)$ ビットとなる．
- $|C| \geq \beta$ のとき C に属する辺 $e=(u,v)$ を以下のように分類する．一般性を失うことなく，u を A に属する頂点，v を B に属する頂点とする．なお，DFS の性質から σ_A^* に属しない頂点と σ_B^* に属する頂点を結ぶ辺は C に含まれることはない．
  - 端点 u,v のいずれも σ_A^*, σ_B^* に属しない：時刻 α において，u,v はそれぞれ既訪問であることから，u,v のいずれも今後スタックに追加されることのない頂点であることがわかる．このような辺の本数を x とする．
  - 端点 u が σ_A^* に属するが，v は σ_B^* に属しない：時刻 α において，v は既訪問であることから，v は今後スタックに追加されることのない頂点であることがわかる．このような辺の本数を y とする．
  - 端点 u,v がそれぞれ σ_A^*, σ_B^* に属する：先の圧縮方法により，スタックをこれらの辺の本数分だけ圧縮できるため，このような辺の本数を z とすると，z ビットだけ辺の本数を縮められることが分かる．

  A に属する頂点に着目すると，既訪問でかつ既に Pop によってスタックから除外された頂点に接する辺が x 本あることがわかる，最大の次数が 3 であるため，鳩ノ巣原理より，時刻 α の時点でスタックから除外された頂点が A 中に少なくとも $\lceil\frac{x}{3}\rceil$ 個存在することが示せる．同様に B に属する頂点に着目すると，$\lceil\frac{x+y}{3}\rceil$ 個の頂点がスタックから除外されていることが示せる．これらより，σ_α の長さ L と圧縮後のスタックの長さ L' について，次のような関係が成り立つ．

$$\alpha - L \geq \lceil\frac{x}{3}\rceil + \lceil\frac{x+y}{3}\rceil$$
$$L - L' = z$$

  これら 2 つより，$\alpha - L' \geq \lceil\frac{x+y+z}{3}\rceil$ を得ることができるため，圧縮後のスタックの長さは高々 $\alpha-\beta/3$，すなわち $\alpha-\beta/3$ ビットの作業領域で格納することができる．スタックの構築及び属する頂点の列挙に必要な時間計算量はともに，$O(n^2)$ である．

提案アルゴリズムにおいて，時刻 α 以前に必要な作業領域は $\alpha+o(n)$ ビットである．時刻 α 以降に必要なメモリ量は，時刻 α 以後に追加された頂点からなるスタックを保持するためのメモリ領域 $n-\alpha+o(n)$ ビットと，圧縮されたスタックから頂点を列挙する際に必要な作業領域の合計である．これらと先の圧縮スタックの構築に必要な作業領域を合わせると，次の補題が成り立つ．

Lemma 1. 提案アルゴリズム *LM3LDFS* は，定数 $\alpha,\beta\in[1,n]$ に対して，$\max\left\{\beta(2+\log\frac{n}{\beta})+\max\left\{\alpha, n-\frac{\alpha}{2}\right\}, n-\frac{\beta}{3}\right\}+o(n)$ ビットの作業領域で最大次数 *3* のグラフにおける *Lex-DFS* 問題を解く．

数値計算により，この作業領域を最小化する α,β を求めると，$\alpha=\frac{2}{3}n, \beta=0.0501\cdots n$ のときに空間計算量が最も小さくなることが求められる．この値を代入することで，以下の定理を得る．

Theorem 2. 提案アルゴリズム *LM3LDFS* は，$0.983728n+o(n)$

ビットの作業領域で最大次数 *3* のグラフにおける *Lex-DFS* 問題を解く．

5. まとめと今後の課題

本論文では，$c < 1$ を満たす定数 c に対して，$cn + o(n)$ ビットの追加の作業領域で最大次数 3 のグラフにおける Lex-DFS 問題を解くアルゴリズムを提案した．今後の課題としては，$c < 1$ を満たす定数 c に対して，作業領域 $cn + o(n)$ ビットで次数制約のない一般グラフにおける DFS アルゴリズムが存在するかどうかという問題が挙げられる．また，最大次数 3 のグラフや一般グラフに対して準線形作業領域で動作する DFS アルゴリズムが存在するかどうかという問題も挙げられる．

参考文献

[1] Asano, T., Izumi, T., Kiyomi, M., Konagaya, M., Ono, H., Otachi, Y., Schweitzer, P., Tarui, J. and Uehara, R.: Depth-First Search Using $O(n)$ Bits, *International Symposium on Algorithms and Computation (ISAAC)*, pp. 553–564 (2014).

[2] Banerjee, N., Chakraborty, S. and Raman, V.: Improved Space Efficient Algorithms for BFS, DFS and Applications, *International Conference on Computing and Combinatorics(COCOON)*, pp. 119–130 (2016).

[3] Barba, L., Korman, M., Langerman, S., Sadakane, K. and Silveira, R. I.: Space–Time Trade-offs for Stack-Based Algorithms, *Algorithmica*, Vol. 72, No. 4, pp. 1097–1129 (2015).

[4] Barnes, G., Buss, J., Ruzzo, W. and Schieber, B.: A Sublinear Space, Polynomial Time Algorithm for Directed s-t Connectivity, *SIAM Journal on Computing*, Vol. 27, No. 5, pp. 1273–1282 (1998).

[5] Chakraborty, S., Raman, V. and Satti, S. R.: Biconnectivity, Chain Decomposition and st-Numbering Using O(n) Bits, *27th International Symposium on Algorithms and Computation (ISAAC2016)*, Leibniz International Proceedings in Informatics (LIPIcs), Vol. 64, pp. 22:1–22:13 (2016).

[6] Chakraborty, S. and Satti, S. R.: Space-Efficient Algorithms for Maximum Cardinality Search, Stack BFS, Queue BFS and Applications, *International Conference on Computing and Combinatorics(COCOON)*, pp. 87–98 (2017).

[7] Elmasry, A., Hagerup, T. and Kammer, F.: Space-efficient Basic Graph Algorithms, *32nd International Symposium on Theoretical Aspects of Computer Science (STACS 2015)*, Leibniz International Proceedings in Informatics (LIPIcs), Vol. 30, pp. 288–301 (2015).

[8] Hagerup, T.: Space-Efficient DFS and Applications: Simpler, Leaner, Faster, *CoRR*, (online), available from ⟨http://arxiv.org/abs/1805.11864⟩ (2018).

[9] Kammer, F., Kratsch, D. and Laudahn, M.: Space-Efficient Biconnected Components and Recognition of Outerplanar Graphs, *41st International Symposium on Mathematical Foundations of Computer Science (MFCS 2016)*, Leibniz International Proceedings in Informatics (LIPIcs), Vol. 58, pp. 56:1–56:14 (2016).

[10] Reif, J. H.: Depth-first search is inherently sequential, *Information Processing Letters*, Vol. 20, No. 5, pp. 229 – 234 (1985).

[11] Reingold, O.: Undirected Connectivity in Log-space, *Journal of ACM*, Vol. 55, No. 4, pp. 17:1–17:24 (2008).