アルゴリズム 63－6
（1998．7．22）

正補混合表現によるグラフ探索時間の削減

豊橋技術科学大学 情報工学系
藤田 正人，伊藤 大雄，上原 秀幸，横山 光雄

概要

単純グラフは補グラフを用いても表現でき、枝数が密である時は補グラフ表現の方がデータ量が少なくてすむ。ただこの表現法によって、これまで線形時間アルゴリズムであるとされてきたものに対して、線形時間可解性が保証されない可能性が生じる。しかし著者らによって、単純グラフに対する既存の線形時間探索アルゴリズムにおいて、補グラフ表現を用いても線形時間で動作するアルゴリズムが提案された。

本報告では、グラフのそれぞれの節点の隣接枝数の密度に着目し、グラフ全体での枝数の密度に関係なく、グラフを表現するデータ量が削減できる単純グラフの表現法を提案する。そして、幅優先探索および深さ優先探索に対し、提案する表現法を用いても線形時間で動作するアルゴリズムを提案する。さらに、提案するアルゴリズムの計算時間を数値実験により評価し、その結果からグラフ探索時間の削減が可能であることが示された。

Decrease in the time of graph search on original-complement-mixed representations

Toyohasi University of Tecnology
Department of Information and Computer Sciences
FUJITA Masato, ITO Hiro, UEHARA Hideyuki, YOKOYAMA Mitsuo

abstract

Simple graphs can be represented by the complement graphs. When the complement graph includes smaller number of edges than an original one, using the complement-graph-representation reduces the space of memories for representing a graph. The authors have presented linear time breadth first search (BFS) and depth first serch (DFS) algorithms for the complement graph representations before.

This paper presents original-complement-mixed(OCM in short) representations, which decreases the space of memories for representing an original graph. The paper also shows that for a given OCM represented grpah, BFS tree and DFS tree of the original graph can be constructed in linear-time. Furthermore, it shows that the running time of BFS and DFS on the OCM representations are faster than ones of original representations, by computer examinations.

1 はじめに

節点集合 $V(|V|=n)$、枝集合 $E(|E|=m)$ の無向単純グラフ $G=(V,E)$ を表現するには、通常 $\Theta(n+m)$ のデータ量が必要であるという前提で、アルゴリズムの計算量が議論される。$n=O(m)$ で扱われるグラフが普通であるため、データの量は枝の数で決定される。ここで、グラフに関する問題としてとらえると、グラフが枝の密度の高いものであるならば、枝の存在する節点組でなく枝の存在しない節点組を、グラフを表現するデータとして用いる補グラフ表現の方がデータ量が少なくてすむ。このとき、枝の存在しない節点組 (v,w) を補枝とよび、

節点v,wは互いに補接するということにする。無向完全グラフの枝数は$\frac{n(n-1)}{2}$であることから、枝数において$m > \frac{n(n-1)}{4}$であれば、補グラフ表現を用いることによってデータ量が少なくてすむ。しかし、この表現法を用いることによって、従来の線形時間可解性を保証していたアルゴリズムにおいて、この保証が保たれるとは限らない可能性が生じる。この問題についての研究として文献[1][2]がある。その文献では、幅優先探索・深さ優先探索を含む既存の線形時間アルゴリズムに対して補グラフ表現を用いても、線形時間可解性を保証するアルゴリズムが提案されている。

本報告では、グラフ全体の枝密度ではなく任意の節点における隣接する節点数の密度に着目した。ここで、各節点vに隣接する節点は隣接リスト表現されていると考える。各節点の隣接節点数は$n-1$以下である。ここで、提案するグラフ表現では、節点vの隣接節点数が、$\lfloor\frac{n-1}{2}\rfloor$以下の時は$v$に隣接する節点のリストをグラフ表現のデータとして扱い、$\lfloor\frac{n-1}{2}\rfloor$より大きい時は節点$v$に補接する（すなわち隣接しない）節点のリストをグラフ表現のデータとして扱うことにする。このようにグラフのデータを記憶することによって、記憶に必要なデータ量が削減できる。ここで、補接する節点のリストによる表現を補接リスト表現と呼び、区別のために補接リスト表現と対称的である、隣接リスト表現を以下では正接リスト表現と呼ぶことにする。このように、単純グラフの各節点における隣接密度によって、補接リスト表現と正接リスト表現を選択して用いる表現法を、正補混合表現と呼ぶことにする。

さらに本報告では、単純グラフに対する既存の線形時間探索アルゴリズムの代表的なものとして幅優先探索と深さ優先探索をあげ、正補混合表現を用いても、それに対して線形時間で動作する探索アルゴリズムを提案する。正補混合表現されたグラフでは、ほとんどのグラフにおいてデータ量が確実に削減されることから、ここで提案する探索アルゴリズムの実際の計算時間の削減にも有効であることが見込まれる。そこで本稿ではさらに、提案したアルゴリズムが通常の探索アルゴリズムと比べて、どの程度高速になるかを数値実験によって検証した。

また、本報告では無向単純グラフにおいて説明したが、同様の議論が有向単純グラフにおいても適用することができる。

2 諸定義

対象とするグラフは、自己閉路や並列枝を含まない無向単純グラフ$G=(V,E), |V|=n, |E|=m$とする。節点集合Vにおいて、各節点は連番で表されている。すなわち$V=\{1,2,\cdots,n\}$である。枝の表現については、その両端点を用いて(v,w)と表すものとする。

正補混合表現されたグラフにおいて、それぞれの節点が正接リストを持つか補接リストを持つかの判断は、前述のように元のグラフの隣接節点数によって決定され、以下の図のように定義されるものとする。ここで、元のグラフは隣接リスト表現されているものとし、リスト中の要素は節点番号順に並べられている。

節点vにおいて元のグラフでの隣接節点数が$\lfloor\frac{n-1}{2}\rfloor$以下のとき、元のグラフでの隣接リストをもとに、節点vの正接リストが作られる。（図1）

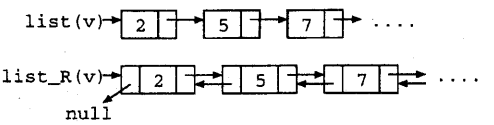

図1：正接リスト

ここで、正接リストは双方向リストであるものとし、元のグラフの隣接リスト同様、正接リストの要素も節点番号順である。

一方、節点vにおいて元のグラフでの隣接節点数が$\lfloor\frac{n-1}{2}\rfloor$より大きいとき、元のグラフでの隣接リストをもとに、節点vの補接リストが作られる。（図2）

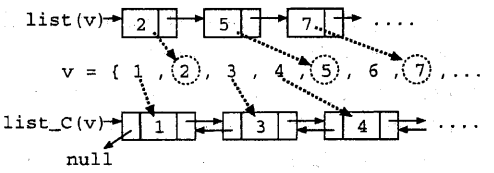

図2：補接リスト

補接リストも正接リスト同様、双方向リストで、リストの要素も節点番号順になっているものとする。

また、各節点$v \in V$が正接リストを持つか補接リストを持つかを区別するフラグをそれぞれの節点が持っているものとし、以下では正接リスト

を持つ時は $flag(v) = R$、補接リストを持つ時は $flag(v) = C$ としている。

このように隣接リスト表現されたグラフから正補混合表現されたグラフを作成するためには、$O(n^2)$ 時間かかる。以下では、始めから正補混合表現されたグラフが入力として与えられるという前提で、アルゴリズムの計算時間を解析する。

アルゴリズムの計算時間の解析のために正補混合表現されたグラフの枝数を定義する。このとき、各節点が持つリストの要素数の合計が、正補混合表現されたグラフの枝数に等しい。ここで、正補混合表現されたグラフ全体で、正接リストを持つ節点に隣接する節点数の合計を m_rとし、補接リストを持つ節点に補接する節点数の合計を m_cとする。そして、これらにより正補混合表現されたグラフ全体で各リストの持つ要素の合計μは、$m_r + m_c = \mu$と定義され、このとき$\mu \leq 2m$ である。

本報告では、幅優先探索と深さ優先探索を扱うが、どちらも無向単純グラフ $G = (V, E)$ を入力とし、従来のグラフ表現法に対して探索のための計算量は $O(n + m)$ 時間である。

3 幅優先探索

この節では、正補混合表現されたグラフを入力とし、幅優先探索木 T_Bを出力する線形時間アルゴリズムを与える。まず、3.1 節で正補混合表現されたグラフの例を与え、提案する幅優先探索 (Breadth First Search: BFS) アルゴリズムがどのように実行されるのかを示す。その後、提案する BFS アルゴリズムを正確に記述する。3.2 節では BFS アルゴリズムの計算量について議論する。

3.1 BFS アルゴリズムの実行例

例として与えるグラフを図 3.(a) としたとき、隣接リスト表現されたグラフは図 3.(b) であり、これより正補混合表現されたグラフは図 3.(c) のようになる。(ここでは 2 章で図示したリストよりも簡略化して表現している)

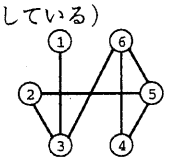

図 3：(a) グラフ例

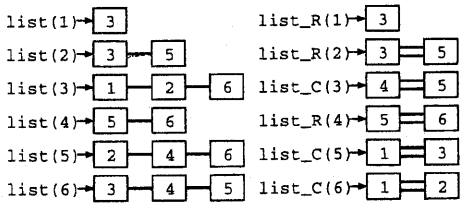

図 3：(b) 隣接リスト表現　図 3：(c) 正補混合表現

次に変数やリストを定義する。

label(i) : 節点 i の状態を示す変数. キューに格納されていないとき$= F$、格納されたとき$= T$. 初期値は *label*$(i) = F$, $i = 1, \cdots, n$.

N_f : 走査されていない節点数. 初期値 $N_f = n$.

scan(i) : 節点 i のリストにおけるスキャンポインタ. 初期値は節点 $i(i = 1, \cdots, n)$ のリストの先頭要素を指すポインタ.

free : 木に含まれない節点の双方向リスト. セルを 2 つ持ち、1 つは節点番号を格納し、1 つは補接節点参照用に用いられる. 初期状態は節点番号順に節点 1 から n までの要素を持つ。

que : 次の走査点を格納するためのキュー. 初期状態は要素なし.

実行例を図 4.(a)-(c) に示す。ここでは最初の走査点を節点 2 としている。

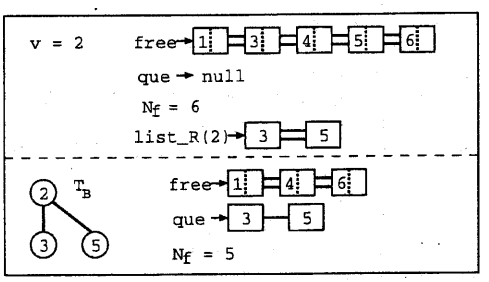

図 4.(a)：走査点 $v = 2$ のとき

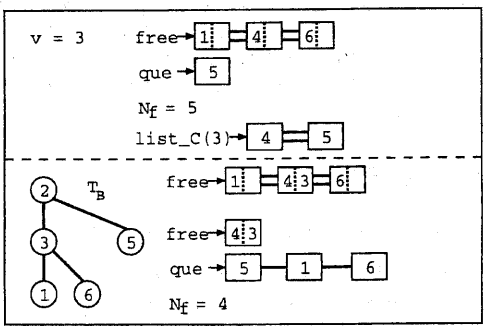

図 4.(b)：走査点 $v = 3$ のとき

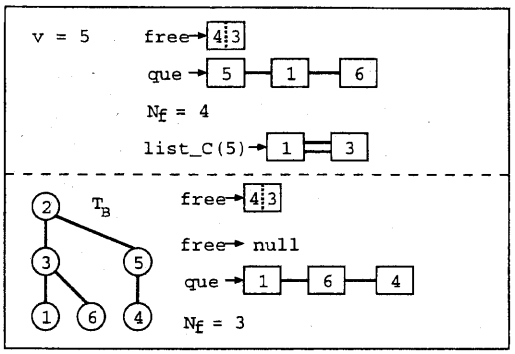

図 4.(c)：走査点 $v = 5$ のとき

節点 2 が走査される時のそれぞれのリスト・変数の状態は図 4.(a) 上部のようになっている。走査点 2 は正接リストを持つので、節点 3 と 5 が隣接することがわかる。そして、節点 3 と 5 はまだ木に含まれていないので、枝 $(2,3),(2,5)$ が存在し、これらは木 T_B に加えられる。節点 3 と 5 を *que* に格納し、リスト *free* から除去する。N_f は $N_f = 5$ に更新される。(図 4.(a) 下部)

次に *que* の先頭要素が次の走査点となるので、図 4.(a) 下部 *que* より節点 3 が走査点 $v = 3$ となる。節点 3 が走査される時のそれぞれのリスト・変数の状態は図 4.(b) 上部のようになっている。走査点 3 は補接リストを持つので、補接する節点 4 と 5 は、元のグラフでは走査点 3 と隣接しない。これより、走査点 3 の補接リストの要素をリスト *free* から除いたものが、走査点 3 に実際に隣接する節点となる。この隣接する節点を求めるために、走査点 3 に補接する節点 4 と 5 のリスト *free* での補接節点参照用のセルに、このときの走査点 3 を格納する。このように操作することにより、リスト *free* を参照するときに、補接節点参照用セルにそのときの走査点が格納されていない要素が、実際に隣接する節点として求めることができる。このとき、節点 1 と 6 が走査点 3 に隣接する節点としてあげられる。ゆえに、枝 $(3,1),(3,6)$ は木 T_B に加えられ、節点 1 と 6 を *que* に格納し、リスト *free* から除去する。N_f は $N_f = 4$ に更新される。(図 4.(b) 下部)

図 4.(b) 下部の *que* から、次の走査点は $v = 5$ となり、走査点 5 は補接リストを持つ。節点 3 の時と同様の処理を行なうことで、図 4.(c) 下部のようになる。以下、このように $N_f = 0$ となるまで幅優先探索が行なわれ、BFS 木 T_B が得られる。

最後に、提案する BFS アルゴリズムを正確に記述する。

1. グラフ上の任意の節点を走査点 v とする.
2. v をキューに格納し、リスト *free* から v を除き、$label(v) := T$.
3. while($N_f > 0$)
4. キューの先頭要素 v を取りだし、$N_f := N_f - 1$.
5. if(v が正接リストを持つ)
6. while(v の正接リストで $scan(v)$ の指す点が存在する)
7. v の正接リストにおいて $scan(v)$ の指す点を w とする.
8. if($label(w) = F$)
9. 枝 (v,w) を木 T_B に加え、リスト *free* から w を除去する. w を *que* に格納し、$label(w) := T$.
10. $scan(v) := scan(v) + 1$.
11. if(v が補接リストを持つ)
12. 補接リストの要素のリスト *free* の補接節点参照用セルに、このときの走査点 v を格納する.
13. リスト *free* の先頭要素を w とする.
14. while(リスト *free* の要素の中でまだ参照していない節点が存在する)
15. if(リスト *free* の w の補接節点参照用セルの要素 $\neq v$)
16. 枝 (v,w) を木 T_B に加え、w を *que* に格納し、$label(w) := T$.
17. リスト *free* から w を除去し、リスト *free* での w の次の要素を新たに w とする.
18. 木 T_B を出力する.

3.2 計算量

提案する BFS アルゴリズムでの計算量を考える。3 行目の *while* によって繰り返される 4 行目から 17 行目までの部分において、8 行目から 9 行目までと 15 行目から 17 行目までは木 T_B に枝が加えられる部分なので、アルゴリズム全体で計算時間は $O(n)$ 時間である。

次に、走査点が正接リストを持つ時の 5 行目から 10 行目までのアルゴリズム全体での計算時間について考える。6 行目から 10 行目までの *while* 文によ

る繰り返しは、*while* 文の条件から節点 v における正接リストの要素数の分だけ実行される。ゆえに、アルゴリズム全体でのこの部分の計算時間は、グラフ全体での正接リストの全要素数 m_r より、$O(m_r)$ 時間である。

さらに、走査点が補接リストを持つ時の 11 行目から 17 行目までのアルゴリズム全体での計算時間を考える。12 行目は補接リストの要素数分だけ実行される。また、14 行目から 17 行目までの *while* 文の繰り返し部分の計算時間において、15 行目から 17 行目までが実行されない時は、補接リストの要素数分だけ *while* 文の繰り返しが実行される可能性がある。これより、すでに考慮している 15 行目から 17 行目までの部分を除いて考えると、11 行目から 17 行目までのアルゴリズム全体での計算時間は、グラフ全体での補接リストの全要素数 m_c より、$O(m_c)$ 時間である。

よって、提案する BFS アルゴリズム全体で必要な計算時間は、

$$O(n + m_r + m_c) = O(n + \mu)$$

となる。以上により、提案アルゴリズムが線形時間アルゴリズムであることが示された。

4 深さ優先探索

この節では、正補混合表現されたグラフを入力とし、深さ優先探索木 T_D を出力する線形時間アルゴリズムを与える。3 章と同様、4.1 節で正補混合されたグラフの例を与え、提案する深さ優先探索 (Depth First Search: DFS) アルゴリズムがどのように実行されるのかを示す。その後、提案する DFS アルゴリズムを正確に記述する。4.2 節では DFS アルゴリズムの計算量について議論する。

4.1 DFS アルゴリズムの実行例

正補混合表現されたグラフの例として 3 章の図 3.(a) を与えるものとし、アルゴリズムの簡略化のために補接リストの先頭要素にはダミーの節点 0 を挿入したものを用いることにする。ゆえに、図 3.(a) の正補混合表現されたグラフは図 5 のようになる。

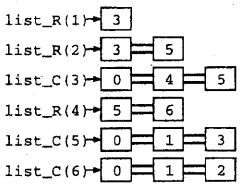

図 5：正補混合表現されたグラフ

次に変数やリストを定義する。

v_c : 参照している補接節点.

$parent(i)$: 節点 i における親. 初期値は $parent(i) = null, i = 1 \cdots n$.

$scan(i)$: 節点 i のリストにおけるスキャンポインタ. 初期値は節点 $i(i = 1 \cdots n)$ のリストの先頭要素を指すポインタ.

$label(i)$: 節点 i の状態を示す. 木に含まれる時= T、含まれない時= F. 初期値は $label(i) = F, i = 1 \cdots n, label(0) = label(n+1) = T$.

また、提案する DFS アルゴリズムでは、上の変数やリストの他に関数 $find$ を用意する。用意する関数 $find(i)$ は、節点番号 i 以上の節点でまだ探索木 T_D に含まれていない最小の節点 j を出力するものである。

$$find(i) = min\{j | j \geq i \text{ でかつ、} j \notin T_D\}$$

ただし、$j \geq i$ で $j \notin T_D$ をみたす j が存在しない時は、ダミーとして加えた節点 7 が返される。

以下の例では視覚的にわかりやすくするために、関数 $find$ の値をポインタの形で図示している。この関数 $find$ の実現と計算時間については 4.2 節で述べる。

提案する DFS アルゴリズムの実行例を図 6(a)-(d) に示す。ここでは最初の走査点を節点 2 としている。

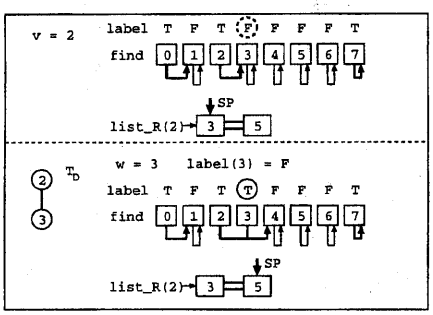

図 6.(a)：走査点 $v = 2$ のとき

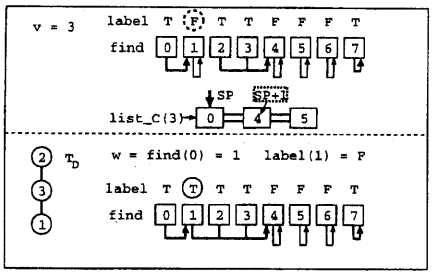

図 6.(b)：走査点 $v = 3$ のとき

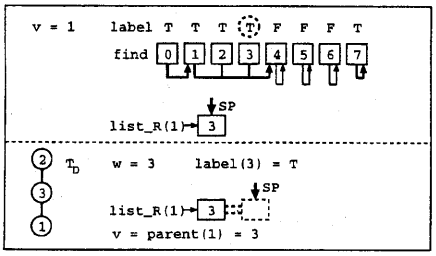

図 6.(c)：走査点 $v = 1$ のとき

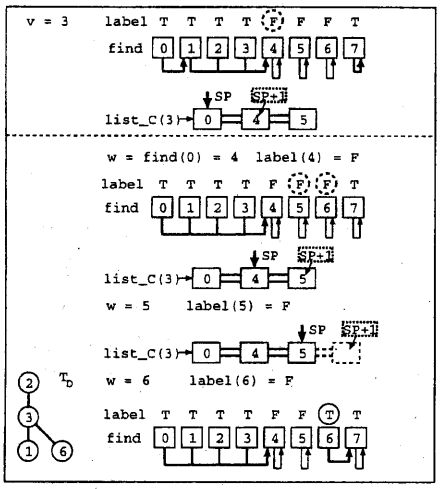

図 6.(d)：走査点 $v = 3$ のとき

実行例の解説をする。節点 2 が走査される時の変数・リストの状態を図 6.(a) 上部に示す。走査点 2 は正接リストを持ち、スキャンポインタ（以下 SP とする）の指す節点 3 は、走査点 2 と枝を持つ候補点 w である。候補点 $w = 3$ において $label(3) = F$ であることから、枝 $(2,3)$ は木 T_D に加えられ、$label(3) = T$ に更新される。そして次の要素を指す様、SP も更新される。（図 6.(a) 下部）

次の走査点は $v = 3$ で、この時の変数・リストの状態は図 6.(b) 上部のようになっている。走査点 3 は補接リストを持ち、SP の指す節点はダミーの節点 0 となっている。関数 $find(0)$ は、節点番号 0 以上の節点の中で、まだ木 T_D に含まれていない最小節点を返す。このとき関数 $find(0) = 1$ で、節点 1 は走査点 3 と枝を持つ候補点 $w = 1$ となる。ここで $label(1) = F$ で、補接リストの SP+1 の要素と比較することにより、節点 1 が補接リストの要素でないことがいえる。ゆえに枝 $(3,1)$ は木 T_D に加えられ、$label(1) = T$ に更新される。（図 6.(b) 下部）

次の走査点は $v = 1$ で、この時の変数・リストの状態は図 6.(c) 上部のようになっている。節点 1 は正接リストを持ち、SP の指す節点は 3 である。走査点 1 と枝を持つ候補点は $w = 3$ となるが、$label(3) = T$ より節点 3 はすでに木 T_D に含まれているので、正接リストの SP を更新し次の要素を参照する。しかし SP の指す節点が存在しないので、走査点 1 の走査は終了となり、走査点 v を節点 1 の親節点 3 に更新する。（図 6.(c) 下部）

走査点が再び $v = 3$ となり、この時の変数・リストの状態は図 6.(d) 上部のようになっている。走査点 3 は補接リストを持つので、関数 $find$ を用いて SP の指す点から新たに候補点を求める。このとき候補点は $w = 4$ である。$label(4) = F$ から、節点 4 はまだ木 T_D に含まれていないが、補接リストの SP+1 の要素と比較すると節点 4 は走査点 3 に補接する節点であることがわかるので、枝 $(3,4)$ は存在せず、木 T_D に加えることはできない。そこで候補点 $w = 4$ は、$w + 1 = 5$ の含まれる $union$ 集合の代表元に更新され、このとき候補点は $w = 5$ となる。また、SP も SP:=SP+1 に更新される。候補点 $w = 5$ も木 T_D に含まれていないが、補接リストの SP+1 の指す要素との比較により、節点 5 も走査点 3 に補接する節点であるため、枝 $(3,5)$ も存在しない。このとき、上述と同様に候補点 w と SP は更新され、$w = 6$ となる。$label(6) = F$ で、補接リストの SP+1 の指す要素との比較により、節点 6 は走査点 3 に補接する節点でないことがいえるので、枝 $(3,6)$ は存在する。ゆえに枝 $(3,6)$ は木 T_D に加えられ、$label(6) = T$ に更新される。（図 6.(d) 下部）

次の走査点は $v = 6$ となり、以下これまでのように深さ優先探索が行なわれ、最初の走査点 2 の走査が終了するまでアルゴリズムは実行され、DFS 木 T_D が得られる。

以下に、提案する DFS アルゴリズムを正確に記述する。

1. グラフ上の任意の節点を走査点 v とし、$label(v) := T, parent(v) := 0$.
2. while($v \neq 0$)
3. if(v が正接リストを持つ)
4. v の正接リストにおいて $scan(v)$ の指す点を w とする. $scan(v)$ の指す点がないとき、$w := n+1$.
5. if($scan(v)$ の指す点が存在する)
6. if($label(w) = F$)
7. 枝 (v, w) を木 T_D に加え、$label(v) := T, parent(w) := v, v := w$.
8. $scan(v) := scan(v) + 1$.
9. else $v := parent(v)$.
10. if(v が補接リストを持つ)
11. v の補接リストにおいて $scan(v)$ の指す点を v_c とする.
12. if($scan(v)$ の指す点が存在しない) $w := n+1, v := parent(v)$.
13. else $w := find(v_c)$.
14. while($w \leq n$) $\Rightarrow v$ との枝を持つ候補節点 w が存在する.
15. if($label(w) = F$)
16. $scan(v)$+1 の指す要素を v_c とし、$scan(v)$+1 の指す点がないとき、$v_c := n+1$.
17. if($w \geq v_c$)
18. if($w = v_c$) $w := find(w+1)$.
19. if($v_c \neq n+1$) $scan(v) := scan(v) + 1$.
20. else
21. 枝 (v, w) を木 T_D に加え、$label(w) := T, parent(w) := v, v := w, w := n+1$.
22. else $v := parent(v), w := n+1$.
23. 木 T_D を出力する.

4.2 計算量

提案する DFS アルゴリズムの計算量を考える。木 T_D に枝が加えられる部分は、6 行目から 7 行目までと 20 行目から 21 行目までで、アルゴリズム全体で $O(n)$ 時間で実行される。また、走査点 v の走査が終了し、木 T_D での親をたどって走査点 v の更新がされる部分は、9 行目と 12,22 行目で、これもアルゴリズム全体で $O(n)$ 時間で実行される。

次に、走査点が正接リストを持つ時の 3 行目から 9 行目までのアルゴリズム全体での実行時間を考える。すでに述べた 6,7,9 行目の処理を考慮して考えると、5 行目から 8 行目までの if 文の処理は正接リストの要素の参照なので、アルゴリズム全体での計算時間は、グラフ全体での正接リストの全要素数 m_r から、$O(m_r)$ 時間であることがいえる。

さらに、走査点が補接リストを持つ時の 10 行目から 22 行目までのアルゴリズム全体での実行時間を考える。13,18 行目の関数 $find$ は、Gabow and Tarjan の union-find のアルゴリズム [3] を利用することにより実現される。Gabow and Tarjan[3] の union-find を用いて、線形時間で DFS を実行する手法として Imai and Asano[4] があり、本アルゴリズムは基本的にその手法に基づいている。このことから、DFS アルゴリズム中で用いる関数 $find$ のアルゴリズム全体での計算時間は、$O(n)$ 時間として扱うことができる。すでに述べた 12,13,18,20-22 行目の処理を考慮して考えると、14 行目から 22 行目までの $while$ 文中での残っている処理は、補接リストの要素の参照のための処理である。ゆえに、アルゴリズム全体でのこの部分の計算時間は、グラフ全体での補接リストの全要素数 m_c から、$O(m_c)$ 時間であることがいえる。

よって、提案する DFS アルゴリズム全体で必要な計算時間は、

$$O(n + m_r + m_c) = O(n + \mu)$$

となる。以上により、提案アルゴリズムが線形時間であることが示された。

5 数値実験

正補混合表現されたグラフを入力とする深さ優先探索について数値実験を行なった。比較に用いる従来の DFS・補グラフを入力とする DFS と、提案する DFS を C 言語でプログラミングした。ここで提案する DFS のプログラミングに関して、実験で用いたプログラムは、文献 [3] で提案している線形時間 $union-find$ の手法を用いないで、アルゴリズム全体で $union - find$ の操作が $O(n\alpha(n,n))$ 時間のものを用いた。しかし、アッカーマン逆関数 $\alpha(n,n)$ は実用上、定数とみなすことができることから、本

実験における計算時間に影響はないと思われる。このことは計算結果にも表れている。入力するグラフは、節点数 $|V| = 100$ とし、ランダムに発生させたものを用いた。実験では 30 個のランダムグラフを用意し、それぞれの DFS アルゴリズムを用いて深さ優先探索を行なった。得られた計算時間を平均したものを、以下では結果として扱っている。

従来の DFS と、文献 [1][2] で提案されている補グラフを入力とする DFS と、本報告で提案する正補混合表現されたグラフを入力とする DFS の、それぞれの探索木 T_D を出力するまでの計算時間を比較した結果を図 7.(a) に示す。

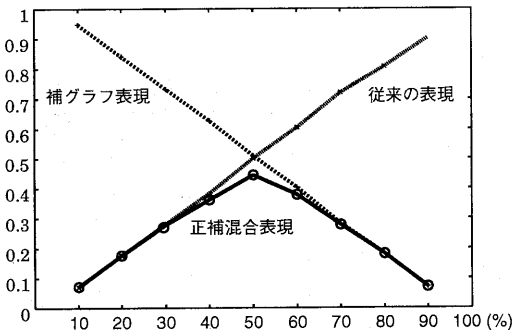

図 7.(a)：各表現による DFS の計算時間

横軸は元のグラフの枝密度 (%) であり、縦軸は従来の DFS の枝密度 50%のときの計算時間を 0.5 として、正規化したものである。この結果より、本報告で提案する正補混合表現されたグラフを入力とする DFS アルゴリズムによって、計算時間が削減されることが示された。

また、提案する DFS アルゴリズムの線形性を示すために、正補混合表現されたグラフの枝数 μ が最大のときを 100%として横軸とし、DFS アルゴリズムでの計算時間を枝数 μ が最大のときを 1 として正規化したときの計算結果を図 7.(b) に示す。

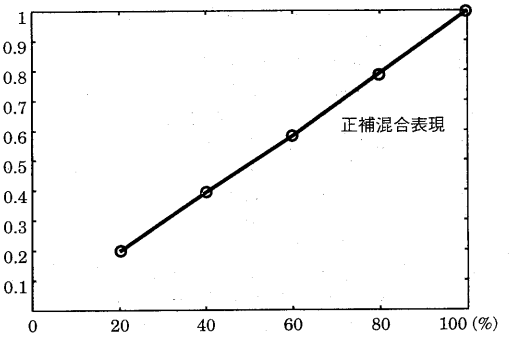

図 7.(b)：実験結果

6 まとめ

単純グラフは正補混合表現によって表現することができ、枝密度の大小に関わらず、データ量を削減することができる。この表現法を入力とした線形時間探索アルゴリズムを提案することによって、データ量の削減が実際の計算時間において活かされ、計算時間の削減を実現したことが数値実験によって検証された。

参考文献

[1] 伊藤 大雄, "補グラフ入力に対する線形時間グラフ探索アルゴリズム," 信学技報 COMP95-12, pp. 9-16, (1995).

[2] Ito. H and Yokoyama. M, "Linear time algorithms for graph search and connectivity determination on complement graphs," Information Processing Letters, vol. 66, no. 4, pp. 209-213, (1998).

[3] Gabow H. N. and Tarjan R. E., "A Linear time algorithm for a special case of disjoint set union," J. of Computer and System Science, vol. 30, pp. 209-221, (1985).

[4] Imai. H and Asano. T, "Efficient algorithm for geometric graph search problems," SIAM J. Comput., vol. 15, pp. 478-494, (1991).