DEIM 2021 A11-1

グラフに対する効率的なk最近傍検索のための索引構築手法

小林　瑞季†　　真次　彰平††　　塩川　浩昭†††

† 筑波大学情報学群情報科学類
†† 筑波大学システム情報工学研究科
††† 筑波大学計算科学研究センター，JST さきがけ
E-mail: †kobayashi@kde.cs.tsukuba.ac.jp, ††matsugu@kde.cs.tsukuba.ac.jp , †††shiokawa@cs.tsukuba.ac.jp

あらまし　k 最近傍検索はグラフ構造において特定の頂点から k 個の近傍を特定する検索である．近年，モバイル端末の普及により SNS やマップアプリの使用機会が増加しており，検索の高速化が重要となっている．代表的な高速化手法としてグラフを用いたインデックスを構築する方法があり，様々なインデックスが考案されている．しかし，それらのインデックスは主に交通ネットワークなどの単純なグラフに特化しており，ソーシャルネットワークなどの複雑なグラフに適用するのが難しい．そこで本稿では，複雑なグラフでも検索を高速に処理できるグラフ型インデックスを提案する．実データを用いた実験を行い，既存の手法に比べてインデックス構築時間を最大 26,945 倍高速に計算できることを確認した．

キーワード　グラフ，k 最近傍検索，グラフ型インデックス

1　序　　論

グラフ k 最近傍検索とは，クエリノード q，検索結果の数 k が与えられたとき，グラフ上の q から近くにある k 個のノードを求める検索であり，ソーシャルネットワークサービスやマップアプリなどで利用されている．ソーシャルネットワークサービスは，インターネットの普及につれて利用が盛んになっており，大量のユーザー情報や投稿などのデータが日々生み出されている．また，モバイル端末の普及によりマップアプリの利用機会も増えている．k 最近傍検索はソーシャルネットワーク上ではおすすめのユーザーや投稿の検索，マップアプリでは近隣の施設検索に主に用いられており，大量のデータの中から目的のデータを高速に検索する必要がある．

しかし，大規模なデータに対して k 最近傍検索を行うのには膨大な計算コストが必要であるという問題点がある．条件に適合するデータを検索するには対象となる全データを評価しなければならない．しかしながら，評価に要する計算時間はデータ量の増加とともに増加するため，大規模なデータに対しては現実的な時間で処理することができなくなる．この問題に対して，大規模なデータに対しての検索を高速化する代表的な手段として検索用のインデックスを構築する手法が近年提案されている [1], [2], [3], [5], [6]．これらの手法では検索用のインデックスを事前に構築することで，インデックスの構築コストが必要となるが，検索を簡略化することができるため高速に k 最近傍検索を処理することができる．

既存のインデックス構築手法の問題点として，対象を交通ネットワークに絞っているため，複雑なネットワークに用いることが難しいことが挙げられる．交通ネットワークの多くは平面グラフであるため，複雑なインデックスでも構築時間が小さくなりやすいが，現実のネットワーク構造には平面グラフではない複雑なネットワークが多く存在する．これらのグラフでは既存のインデックス構築手法は多くの計算時間が必要となる．なぜなら，既存の手法は多くの頂点間の最短距離を計算する必要があり，エッジ数が多いグラフでは最短距離の計算に大きなコストがかかるためである．代表的なグラフ型インデックスを構築する手法として，G-Tree [1] が挙げられる．G-Tree はグラフを再帰的に等しいサイズの部分グラフに分割し，部分グラフ間で距離行列を持たせることによって木構造型のインデックスを構築する．しかし，約 28,000 ノード規模のソーシャルネットワークに対しては 1 時間以内にインデックスを構築することができない．

この問題を解決するために本稿では，複雑なネットワークにおける k 最近傍検索を対象とした効率の良いインデックス構築手法を提案する．提案手法の目的は，大規模で複雑なグラフに対して高速にインデックスを構築し，k 最近傍検索を効率的に処理することである．既存手法ではインデックスを構築する際に多くの頂点間の最短距離の計算を行う必要があるが，ソーシャルネットワークなどの複雑で大規模なグラフではこの距離計算に大きなコストが必要となってしまう．提案手法では，グラフ上での最短距離計算をほとんど行わずにインデックスを構築する手法を提案する．グラフの中で木構造となっている部分グラフを集約したノードと，それ以外のグラフ上のノードをインデックスのノードとすることでインデックスを構築する．本手法は事前に多くの距離計算をする必要がないことと，複雑なネットワークには木構造が多く含まれるため集約できるノード数が多いことから，インデックス構築コストと検索コストを減らすことができる．

本研究の貢献として，以下の 2 つが挙げられる．

- 提案手法は交通ネットワーク，ソーシャルネットワークなどの複雑なグラフのどちらに対しても，既

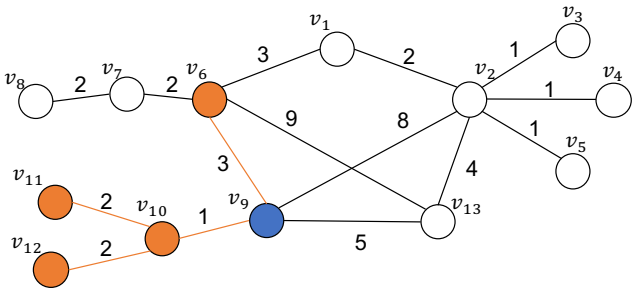


図 1　k 最近傍探索の例

存手法よりも高速にインデックスを構築することができる．実データを用いた実験を通じて，提案手法は既存手法と比較して約 26,945 倍高速であることを確認した．

- 提案手法は構築したインデックス上で k 最近傍検索を行うことで，既存手法よりも高速にクエリを処理することができる．実データを用いた実験を通じて，提案手法は既存手法と比較して約 426 倍高速であることを確認した．

本稿の構成は以下の通りである．2 節で本稿で取り扱う問題を説明し，3 節で関連研究を紹介する．4 節で提案手法について述べ，5 節で実データを用いた比較実験結果を示す．最後に，6 節で本稿のまとめを述べる．

2　問 題 提 議

本稿で対象とする問題の定義と，本稿で用いる記号の定義を行う．以下では，グラフ上のノードを頂点，インデックス上のノードをノードとして記述する．本稿ではグラフを $G = (V, E, W)$ と表し．V，E および W はそれぞれ頂点集合，エッジ集合およびエッジの重みからなる集合である u，v 間のエッジ $e(u, v) \in E$ は重み $w(u, v)$ を持つ．

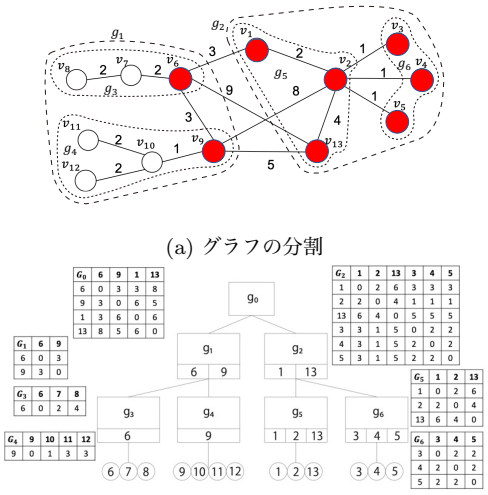
本稿で扱う k 最近傍検索を以下のように定義する．

定義 1 (k 最近傍検索)．グラフ $G = (V, E, W)$，クエリ頂点 $q \in V$ および検索結果数 k が与えられたとき，グラフ G において，クエリ頂点 q からの距離 $Dis(q, r_i)$ が最も小さい k 個の頂点集合 $R = \{r_1, r_2, \ldots, r_i, \ldots, r_k\} \in V$ を返す．ただし，距離 $Dis(q, r_i)$ はクエリ頂点 q から頂点 r_i までの最短経路長を表す距離関数である．

k 最近傍検索の例を図 1 に示す．図 1 のようなグラフとクエリ $\langle q, k\rangle$ が与えられたとき，k 最近傍検索は q からの距離が小さい順に k 個の頂点集合を返す．ここで，クエリ頂点を $q = v_9$，$k = 4$ とすると，q からの距離が 1 である v_{10}，距離が 3 である v_{11}，v_{12}，v_6 が q から最も距離が小さい 4 頂点となる．よって，$R = \{v_{10}, v_{11}, v_{12}, v_6\}$ が検索結果となる．

3　先 行 研 究

これまでに多くの k 最近傍検索の高速化手法が提案されている．本稿ではその中でも (1) 検索用のインデックスを構築する手法，ならびに (2) 検索用のインデックスを構築しない手法について述べる．



(a) グラフの分割

(b) G-Tree のインデックス

図 2　G-Tree の例

3.1　検索用のインデックスを構築する k 最近傍検索手法

3.1.1　G-Tree

文献 [1] では，G-Tree というインデックスを提案している．G-Tree は事前にグラフを *multilevel partitioning algorithm* [10] を用いて f 個の等しいサイズの部分グラフ ($|V_1| \approx \cdots \approx |V_f|$) に分割し，それらを再帰的に分割してできた部分グラフが τ 個以上の頂点を持たなくなるまでボーダーの数が少なくなるように再帰的に分割する．ここで，ボーダーという頂点を定義する．

定義 2 (ボーダー)．グラフ G の部分グラフ G_i において，$u \in V_i$ が $\exists e(u, v) \in E$ かつ $v \notin V_i$ を満たすとき，u を G_i のボーダーという

こうしてできた部分グラフをノードとし，隣接するノード間に距離行列を持たせることでインデックスを構築する．

図 2 を用いて $f = 2, \tau = 4$ とした場合の G-Tree の例を示す．元のグラフを G_0 とし，G_0 を頂点数が τ 個以下になるまで再帰的に分割したものが図 2 (a) である．G_0 を根ノードとし，親ノードのグラフを分割してできた部分グラフを子ノードとすることで図 2 (b) のような G-Tree ができる．

G-Tree の最下層のノードはノード内の全頂点と全ボーダーとの距離行列を持ち，それ以外のノードは隣接する子ノード間の全ボーダー同士の距離行列を持つ．

G-Tree を使用した探索アルゴリズムを Algorithm 1 に示す．クエリとして，始点を v_q，探索対象のリストを C，検索結果の数を k とした $q = \langle v_q, C, k\rangle$ が与えられる．C を用いて occurrence list を作成する．occurence list とは，探索対象の頂点がどのノードに属しているかを記録するリストである．また，現在の最上段のノードを記録するポインタ T_n を用いることで，探索を効率的に行うことができる．T_{min} は v_q から T_n までの最短経路長である．以上の要素を利用して，v_q が属する

Algorithm 1 G-Tree の探索

Input: インデックス：I，クエリ：$q = \langle v_q, C, k \rangle$
Output: k 最近傍探索の結果リスト:R

Compute the occurrence list L based C
Initialize 優先度付きキュー $Q = \phi$，結果リスト $R = \phi$
foreach $v \in L(leaf(v_q))$ **do**
  Q.Enqueue($\langle Dis(v_q, v), v_q \rangle$)
end for
Initialize $T_n = leaf(v_q)$ and T_{min}
$d_{Emax} = d_N(q, p_k)$
while R.Size()$< k$ and ($Q \neq \phi$ or $T_n \neq root$) **do**
  if $Q = \phi$ **then**
    UpdateT(T_n, T_{min}, Q)
  end if
  $\langle dis, e \rangle \leftarrow$ Q.Dequeue()
  if $dis > T_{min}$ **then**
    UpdateT(T_n, T_{min}, Q)
    Q.Enqueue($\langle dis, e \rangle$)
  else if e is an object **then**
    Insert e into R
  else if e is a node **then**
    foreach $c \in L(e)$ **do**
      Q.Enqueue($\langle Dis(v_q, c), c \rangle$)
    end for
  end if
end while

Algorithm 2 UpdateT (T_n, T_{min}, Q)

$T_n \leftarrow T_n.father$ and update T_{min}
foreach $c \in L(T_n)$ **do**
  Q.Enqueue($\langle Dis(v_q, c), c \rangle$)
end for

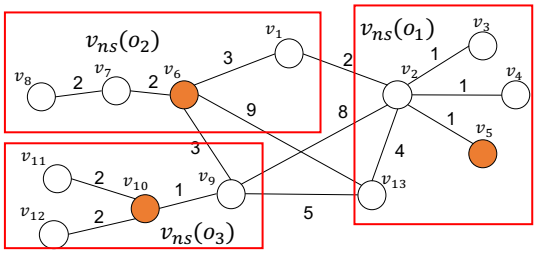
G-Tree の最下層のノードから順に探索することで，k 最近傍頂点を求める．

また，G-Tree を拡張した手法として文献 [2] があり，G*-Tree というインデックスが提案されている．G-Tree の離れた最下層のノード間にショートカットとして距離行列を導入することでインデックスを構築する．どちらの手法も k 最近傍検索を効率的に処理することができるだけでなく，最短距離検索にも利用できるという利点がある．

3.1.2 ランドマーク距離を用いた手法

文献 [3] では，ランドマークとネットワークボロノイ図 [8] を使ったインデックスを提案している．ランドマークとは全ての頂点との距離をあらかじめ求めておくいくつかの頂点のことであり，任意の 2 頂点間の距離の下界を与えるために用いられる．ランドマークを用いることで，任意の 2 頂点間の良い下界を得ることができるようになり，検索の際に検索範囲を絞ることが可能となる．ここで，ランドマーク間距離とボロノイノードセットについて以下のように定義する．

定義 3 (ランドマーク間距離)．ランドマークを l_i としたとき，ある頂点 o とクエリ頂点 q とのランドマーク間距離は



(a) ネットワークボロノイ図への分割

$v_{ns}(o_2)$	$v_6, 0$	$v_7, 2$	$v_1, 3$	$v_8, 4$

(b) $v_{ns}(o_2)$ の構成

図 3　ランドマーク距離を用いた手法の例

$$LB_{l_i}(q, o) = |d(l_i, q) - d(l_i, o)| \leqq d(q, o)$$

で表される．ここで，$d(u, v)$ はグラフ上での u, v 間の最短経路長を表している．

定義 4 (ボロノイノードセット)．ボロノイノードの中心を o_i, o_j としたとき，ボロノイノードセット $V_{ns}(o_i)$ は

$$v_{ns}(o_i) = \{v \mid v \in V, d(v, o_i) \leqq d(v, o_j) \forall o_j \backslash o_i\}$$

で表される．

ネットワークボロノイ図を用いたインデックスの例を図 3 に示す．ボロノイノードの中心を $O = \{v_5, v_6, v_{10}\}$ としてネットワークボロノイ図を構築したものが図 3 (a) である．$v_{ns}(o_2)$ を例とすると，ボロノイノードセットは図 3 (b) のように構築されており，それぞれの要素には頂点とランドマークから頂点までの距離 $d(o_i, v)$ が格納されている．k 最近傍検索をユークリッド距離を基準に用いず，インデックス構築時に計算した $d(o_i, v)$ を用いて計算できるランドマーク間距離を基準として探索を行う．例えば，図 3 (a) において v_5 から近い頂点の探索を行うとする．探索している頂点を o とすると，通常は $d(v_5, o)$ を基準にして頂点を探索するが，この手法では $LB_{l_i}(v_5, o) = |d(l_i, v_5) - d(l_i, o)|$ を基準にして探索を行う．このような探索を行うことで誤った候補頂点を探索することが少なくなり，効率的な探索を行うことができる．

3.2 検索用のインデックスを構築しない k 最近傍検索手法

検索用のインデックスを構築しない手法として文献 [4] がある．文献 [4] では，Incremental Euclidean Restriction (IER) というダイクストラアルゴリズムを拡張した手法を提案している．IER はあらかじめ k 個の頂点を候補集合として取得し，ダイクストラアルゴリズムなどを用いて残りの頂点集合を探索し候補集合を更新することで k 最近傍検索の結果を求める．

IER のアルゴリズムを Algorithm 3 に示す．クエリ $q = \langle v_q, k \rangle$ が与えられたとき，R-Tree [9] などを用いて k 個の頂点を候補集合 $R = \{p_1, \ldots, p_i, \ldots, p_k\}$ として取得する．その後，q と R 内のすべてのノード $p_i \in R$ について，q から p_i までの距離 $d(q, p_i)$ を計算し，1 番遠いノードまでの距離を上限 D_{Emax} として保存する．そして，D_{Emax} を下回る $d(q, p)$ を持つ隣接ノード p が見つかる間探索を繰り返すことで k 最近傍頂点を求める．

Algorithm 3 IER

Input: グラフ：$G(V, E, W)$，クエリ：$q = \langle v_q, k \rangle$
Output: k 最近傍探索の結果リスト:R

```
Initialize 結果リスト R = φ
R ← {p_1, ... p_k} = Euclidean_NN(q, k)
foreach p_i ∈ R do
    d_N(q, p_i) = compute_ND(q, p_i)
end for
sort R in ascending order of d_N(q, p_i)
d_Emax = d_N(q, p_k)
while d_E(q, p) > d_Emax do
    (p, d_e(q, p)) = next_Euclidean_NN(q)
    if d_N(q, p) < d_N(q, p_k) then
        remove p_k from R
        insert p into R
        d_Emax = d_N(q, p_k)
    end if
end while
```

3.3 既存手法の問題点

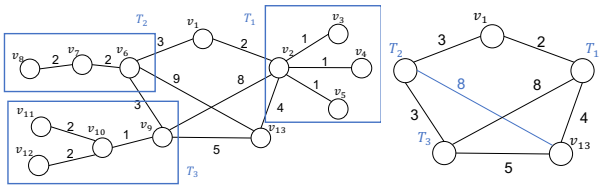
既存のインデックス構築手法の問題点として，ソーシャルネットワークなどの複雑なグラフにおいてインデックス構築に大きなコストがかかることが挙げられる．G-Tree について，ボーダーの数が少なくなるようにグラフ分割を行うが，複雑なグラフではエッジが多いためボーダーの数が多くなってしまう．そのため，距離行列のサイズが大きくなってしまうので，インデックスの構築に膨大な時間がかかってしまう．ランドマーク距離を用いた手法について，ランドマークとの距離を計算するのにあらかじめ ALT [7] を用いて全ノード間の距離を計算しておかなくてはならないため，複雑なグラフでは事前準備に膨大な時間がかかるという問題点がある．インデックスを構築しない手法について，IER は逐次探索を行うことから k 最近傍ではない頂点を多く探索してしまうため，検索時のオーバヘッドが大きいという問題点がある．

4 提 案 手 法

本節では提案手法について説明する．

4.1 基本アイデア

本節で提案する木構造集約を用いる手法の基本アイデアは，ソーシャルネットワークには木構造部分が多く含まれるという特性から，木構造を 1 つのノードに集約することでノード数を減らすことである．インデックスの例を図 4 に示す．本手法は図 4 (a) のようにグラフの中で木構造になっている部分グラフを探索する．木構造の探索後，図 4 (b) のように木構造の頂点集合を集約し，集約した頂点集合とその他の頂点をノードとしてインデックスを構築する．このとき，隣接ノード間の最短経路長を計算し，エッジの重みを最短経路長に置き換える．このインデックス上で k 最近傍検索を行うことで，効率的にクエリを処理することができる．



(a) 木構造の探索　　(b) 木構造を集約

図 4　インデックス構築の例

表 1　ノードが持つ情報

記号	定義
NUM	ノードに含まれる頂点数
MAXDIS	根から葉までの最大距離
UPPER	クエリノードから葉までの最大距離
ELEMENTS	集約された頂点と根からの距離集合
ADJNDS	隣接ノードリスト
ADJDIS	隣接ノードまでの距離リスト

4.2 インデックス構築

本節ではインデックス構築について説明する．まず，インデックスのノードについて以下のように定義する．また，それぞれのノードは表 1 の情報を持つものとする．

定義 5 (T-Node)．グラフ上で木構造となっている頂点集合を集約したノードを *T-Node* とする．

定義 6 (O-Node)．グラフ上で *T-Node* に属していない頂点を *O-Node* とする．

T-Node と O-Node の具体例として，それぞれ図 4 における T_1 と v_1 が挙げられる．図 4 (a) において，部分グラフ v_2, v_3, v_4, v_5 は木構造となっており，この部分を集約すると図 4 (b) の T_1 となる．また，図 4 (a) において v_1 は木構造となっているどの部分グラフにも属していないため，図 4 (b) の v_1 となる．

Algorithm 4 インデックス構築

Input: グラフ：$G(V, E)$
Output: インデックス：I

```
1: Initialize インデックス I = φ
2: SearchTree(G)
3: foreach v ∈ V do
4:     if v is Index.Node then
5:         foreach a ∈ v.Adjacent do
6:             I[v].ADJNDES ← a
7:             I[v].ADJDIS ← Dijkstra(v, a)
8:         end for
9:     end if
10: end for
```

提案手法の詳細なアルゴリズムを Algorithm 4 に示す．まず，インデックス I を作成する (1 行目)．SearchTree(G) は Algorithm 5 に示されている．グラフの端点から探索を開始し，木構造を満たす間グラフ上を辿っていくことで木構造に属する頂点集合を T-Node の $ELEMENTS$ に集約する．このとき，

Algorithm 5 SearchTree

Input: グラフ：$G(V,E)$，インデックス：I

1: Initialize 葉頂点集合：$leaves = \phi$，親ごとの子集合：$parents = \phi$
2: **foreach** $v \in V$ **do**
3: **if** $v.Adjacent$.Size() $= 1$ **then**
4: Insert v into $leaves$
5: **end if**
6: **end for**
7: **while** $leaves$.Size() $\neq 0$ **do**
8: **foreach** $l \in leaves$ **do**
9: Insert l into $parents[l.parent]$
10: l がどのノードに入るか記録
11: **end for**
12: $leaves$.Clear()
13: **foreach** $p \in parents$ **do**
14: **if** $p.Adjacent$.Size() $-parents[p]$.Size() $= 1$ **then**
15: Insert p into $leaves$
16: **else**
17: p is root
18: **end if**
19: **end for**
20: $parents$.Clear()
21: **end while**
22: **foreach** $v \in V$ **do**
23: **if** v is root **then**
24: Insert v into I
25: ノードの記録から $v.ELEMENTS$ に追加
26: Calculate $I[v].MAXDIS$
27: **else if** v is not in tree **then**
28: Insert v into I
29: **end if**
30: **end for**

根から葉までの最大距離を計算し，$MAXDIS$ に格納する (2 行目)．その後，T-Node の根頂点と O-Node について，グラフ上をダイクストラアルゴリズムを用いて探索を行い，隣接ノードを $ADJNDS$ に，隣接ノード間の最小距離を $ADJDIS$ に格納する (3 行目から 10 行目)．以上のようにして，インデックスを構築する．

インデックスの構築を図 4 を用いて説明する．グラフのそれぞれの端点から木構造の頂点集合を探索すると，図 4 (a) のようにグラフの中で青い長方形で囲まれた T_1，T_2，T_3 が見つかる．それぞれの木構造を 1 つのノードに集約し，集約した頂点集合とその他の頂点をノードとする．それぞれのノードと隣接集合に対して最短距離を計算し，エッジの重みを最短距離に置き換えたものが図 4 (b) である．インデックス構築時のノード T_1 が保持している情報を表 2 に示す．NUM と $COUNT$，$ELEMENTS$ は集約時に計算し，$ADJNDS$ と $ADJDIS$ は隣接ノードとの最短距離計算時に計算して代入する．同様に，このような情報を他のノードも保持している．

表 2　T_1 が持つ情報

記号	値
NUM	4
MAXDIS	1
UPPER	NULL
ELEMENTS	$\{\langle v_2,0\rangle,\langle v_3,1\rangle,\langle v_4,1\rangle,\langle v_5,1\rangle\}$
ADJNDS	$\{v_1,v_{13}\}$
ADJDIS	$\{2,4\}$

4.3 探索アルゴリズム

本節では探索アルゴリズムについて説明する．クエリとして，始点を v_q, 検索結果の数を k とした $q = \langle v_q, k\rangle$ が与えられる．探索アルゴリズムとして，基本的には優先度付きキュー Q を用いて行う．まず，v_q と 0 のペアを Q に挿入する．そして，先頭要素の隣接ノードと v_q から隣接ノードまでの距離を Q に追加し，先頭要素を k 最近傍探索の結果リスト R に追加するという動作を R が k 個のノードを格納するまで繰り返すことで探索を行う．

v_q が，T-Node に集約されている頂点の場合，その T-Node に含まれるすべての頂点を優先度付きキュー TQ に挿入し，Q の先頭要素が入れ替わるたびに TQ の要素と比較し，Q の先頭要素よりも TQ の要素が小さければ R に TQ の要素を追加する．そうすることで，最初の T-Node の要素を探索することができる．探索の過程で，T-Node が見つかったときは探索を簡略化することができる可能性がある．とある T-Node t とクエリまでの距離を $dis(v_q,t)$ としたとき，$t.MAXDIS$ を用いて，$t.UPPER = dis(v_q,t) + t.MAXDIS$ と計算することができる．ここで，以下の定理を用いる．

定理 1. Q の先頭の距離要素 $TopDis = Q.Top().dis$ と，クエリノード v_q から探索ノード t の根からの最遠頂点までの距離 $Ndis = t.UPPER$ があるとき，$TopDis \geq Ndis$ の場合，探索ノード t に含まれるすべての頂点は探索結果 R に追加することができる．

証明. t に含まれるすべての頂点を $E = \{e_1,\ldots,e_i,\ldots,e_n\}$ とする．t の根 e_r からの最遠頂点を e_d とすると，任意の $e_i \in E$ について，$dis(e_r,e_i) \leqq dis(e_r,e_d)$ となる．このことから，$dis(v_q,e_r) + dis(e_r,e_i) \leqq t.UPPER$ となるので，$t.UPPER \leqq TopDis$ の場合，t に含まれるすべての頂点は探索結果 R に追加することができるといえる．　□

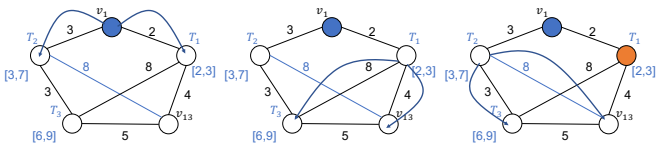
定理 1 により，T-Node のすべての要素を 1 度の探索で R に追加することができることがあるので，探索範囲を減らすことが期待できる．

提案手法の詳細なアルゴリズムを Algorithm 6 に示す．まず，優先度付きキュー Q, TQ と結果リスト R を作成する (1 行目)．v_q が O-Node の場合は，v_q を Q に追加する (3 行目)．v_q が T-Node に集約されている頂点の場合は，その T-Node に含まれるすべての頂点を TQ に挿入し，その T-Node を Q に挿入する (5 行目から 8 行目)．その後，Q の先頭要素を取り出し (4 行目)，そのノードの隣接ノード $n \in v.Adjacent$ を探索

Algorithm 6 探索アルゴリズム

Input: インデックス：I, クエリ：$q = \langle v_q, k \rangle$
Output: k 最近傍探索の結果リスト:R

1: Initialize 優先度付きキュー $Q = \phi, TQ = \phi$，結果リスト $R = \phi$
2: **if** v_q is O-Node **then**
3: Q.Enqueue($\langle 0, v_q \rangle$)
4: **else**
5: **foreach** $v \in T$-Node(v_q) **do**
6: TQ.Enqueue($\langle dis(v_q, v), v \rangle$)
7: Q.Enqueue($\langle dis(v_q, T\text{-Node}(v_q))$, T-Node(v_q) $\rangle$)
8: **end for**
9: **end if**
10: **while** R.Size()$< k$ and $Q \neq \phi$ **do**
11: $\langle dis, e \rangle \leftarrow$ Q.Dequeue()
12: **foreach** $n \in I[e].ADJNDS$ **do**
13: **if** n is not searched **then**
14: Q.Enqueue($\langle dis(v_q, n), v \rangle$)
15: $n.UPPER = dis(v_q, n) + n.MAXDIS$
16: **end if**
17: **end for**
18: **while** $TQ \neq \phi$ and Q.Mindis$\geqq TQ$.Mindis and R.Size()$< k$ **do**
19: Insert TQ.Dequeue().e into R
20: **end while**
21: **if** $I[e].UPPPER <= Q$.Mindis and R.Size()$+I[e].NUM \leqq k$ **then**
22: **foreach** $v \in I[e]$ **do**
23: Insert v into R
24: **end for**
25: **else**
26: **foreach** $v \in I[e]$ **do**
27: TQ.Enqueue($\langle dis(v_q, v), v \rangle$)
28: **end for**
29: **end if**
30: **end while**



(a) 手順 1. (b) 手順 2.〜3. (c) 手順 4.〜

図 5 探索の例

していく．n が探索済みでなかった場合，n と v_q から n までの距離を Q に挿入し，$n.UPPER$ を更新する (12 行目から 17 行目)．探索中に Q の先頭要素と TQ の要素を比較することで，TQ の要素の探索を行う (18 行目と 19 行目)．また，探索ノードの $UPPER$ が Q の先頭要素より小さかった場合，ノードの全要素を R に追加することができる (21 行目から 24 行目)．大きかった場合は，TQ に全要素を挿入することで，次の探索対象に移ったときに T-Node の要素を順に R に追加することができる (26 行目から 28 行目)．

探索手順の具体例を $v_q = v_1$，$k = 6$ として図 5 を用いて説明する．

手順 1. v_1 の隣接ノードを Q に挿入．$Q = \boxed{\langle T_1, 2 \rangle \mid \langle T_2, 3 \rangle}$，$TQ, R = \phi$

手順 2. $\langle T_1, 2 \rangle$ を取り出し，その隣接ノードを Q に挿入．$Q = \boxed{\langle T_2, 3 \rangle \mid \langle v_{13}, 6 \rangle \mid \langle T_3, 10 \rangle}$，$TQ, R = \phi$

手順 3. T_1 の UPPER が 3，Q の先頭の距離要素が 3，$R.Size() + T_1.COUNT = 4 \leqq k = 6$ より，T_1 に含まれる全頂点は R に追加できる．$Q = \boxed{\langle T_2, 3 \rangle \mid \langle v_{13}, 6 \rangle \mid \langle T_3, 10 \rangle}$，$TQ = \phi$，$R = \{v_2, v_3, v_4, v_5\}$

手順 4. $\langle T_2, 3 \rangle$ を取り出し，その隣接ノードを Q に挿入．$Q = \boxed{\langle v_{13}, 6 \rangle \mid \langle T_3, 6 \rangle \mid \langle T_3, 10 \rangle \mid \langle v_{13}, 12 \rangle}$，$TQ = \phi$，$R = \{v_2, v_3, v_4, v_5\}$

手順 5. T_2 の UPPER が 7，Q の先頭の距離要素が 6 より，T_2 の全頂点を TQ に挿入．

$Q = \boxed{\langle v_{13}, 6 \rangle \mid \langle T_3, 6 \rangle \mid \langle T_3, 10 \rangle \mid \langle v_{13}, 12 \rangle}$，

$TQ = \boxed{\langle v_6, 3 \rangle \mid \langle v_7, 5 \rangle \mid \langle v_8, 7 \rangle}$，$R = \{v_2, v_3, v_4, v_5\}$

手順 6. $\langle v_{13}, 6 \rangle$ を取り出し，その隣接ノードを Q に挿入．

$Q = \boxed{\langle T_3, 6 \rangle \mid \langle T_3, 10 \rangle \mid \langle T_3, 11 \rangle \mid \langle v_{13}, 12 \rangle}$，

$TQ = \boxed{\langle v_6, 3 \rangle \mid \langle v_7, 5 \rangle \mid \langle v_8, 7 \rangle}$，$R = \{v_2, v_3, v_4, v_5\}$

手順 7. $\langle v_{12}, 6 \rangle$ と TQ の要素を，TQ の要素が小さく $R.Size()$ が k 以下の間比較．v_6 と v_7 を R に追加したところで探索終了．$R = \{v_2, v_3, v_4, v_5, v_6, v_7\}$

表 3 データセット

データセット	グラフの種類	$\|V\|$	$\|E\|$
CAL	交通ネットワーク	21,048	21,693
NY	交通ネットワーク	264,346	366,923
TV	ソーシャルネットワーク	3,892	17,262
GV	ソーシャルネットワーク	7,057	89,455
NS	ソーシャルネットワーク	27,917	206,259
AT	ソーシャルネットワーク	50,515	819,306
SP	ソーシャルネットワーク	1,632,803	22,301,964

5 評価実験

本節では，提案手法の有効性について実データを用いて評価を行う．

5.1 実験設定

本章では，実データに対して提案手法と既存手法を実行することで，実行速度の観点から提案手法の有効性を検証する．

実験は既存手法である G-Tree とインデックス構築時間とクエリ処理時間について比較を行う．データセットには以下の 6 つの実データを用いた．

- **CAL**
  - カリフォルニアの道路ネットワークのデータ
- **NY**
  - ニューヨークの道路ネットワークのデータ
- **TV, GV, NS, AT**
  - Facebook のソーシャルネットワークのデータ

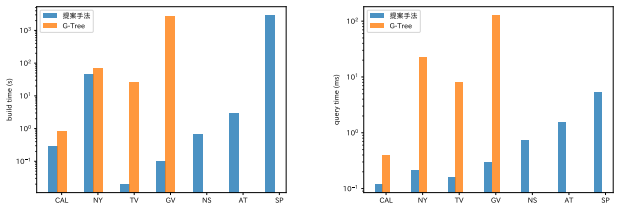


図 6　インデックス作成時間の比較

図 7　クエリ処理時間の比較

- **SP**
  - Pokec のソーシャルネットワークのデータ

CAL は文献 [1] の著者らが公開しているもの [1] を使用し，NY は文献 [5] で使用されているもの [2] を使用した．また，TV，GV，NS，AT および SP は SNAP で公開されているもの [3] を使用した．使用したグラフの詳細は表 3 に示す．本実験はインデックスの構築について，それぞれのデータセットにおいてインデックスを構築した時間を 10 回測ったものを平均している．クエリ処理時間について，それぞれのデータセットにおいてクエリノードをランダムに決定し，$k = 1000$ で実験を行った時間を 30 回測ったものを平均している．

提案手法のアルゴリズムは C++で実装し，G-Tree については GitHub[1] に公開されているものを用いた．コンパイルは -O2 オプションを使用し，gcc 9.3.0 で行った．すべての実験は Intel(R) Core(TM) i5-8279U CPU 2.40GHz，および 16GB RAM で構成される PC 上で行った．

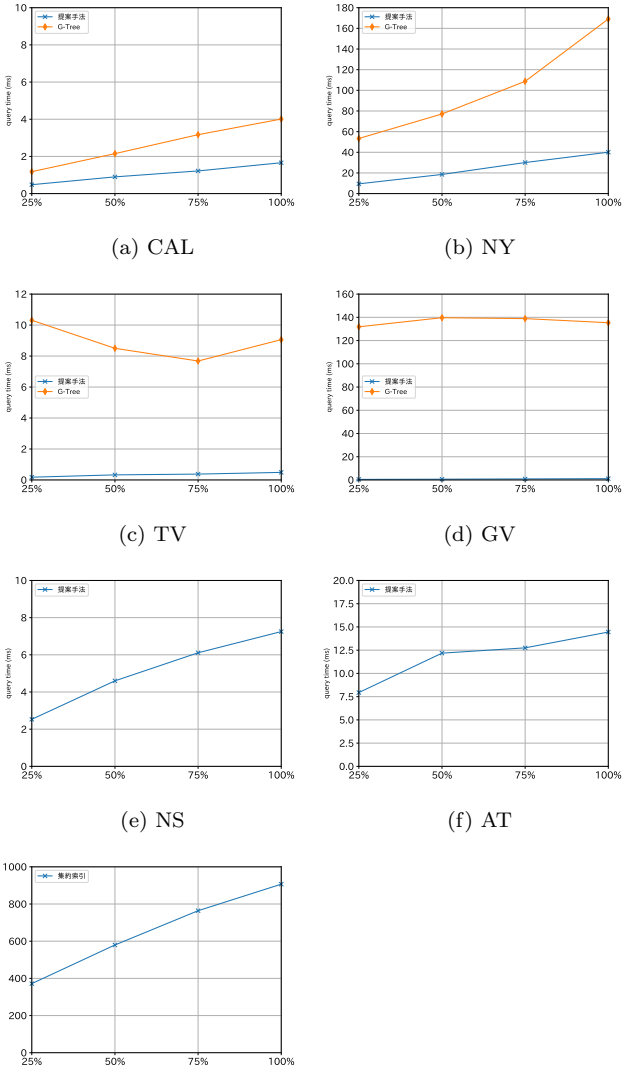
入力するクエリは，q をランダムとし，$k = 1000$ で 30 回実験を行ったものの平均を結果としている．

5.2　インデックス構築時間

インデックス構築時間について，G-Tree と比較した結果を図 6 に示す．NS と AT および SP の実験結果について，G-Tree は 1 時間以内に実行が終わらなかったため実行を打ち切った．また，提案手法 2 では G-Tree と比べて，交通ネットワークでは最大約 2.8 倍，ソーシャルネットワークでは最大約 26,945 倍高速にインデックスを構築することができるとわかる．提案手法 2 において，インデックス構築が木構造部分の探索のみで済むため，単純なネットワークでも複雑なネットワークでも高速化が可能となったことがわかる．この実験により，単純なネットワークと複雑なネットワークにおいて，提案手法 2 は既存手法よりも高速にインデックスを構築できることがわかる．

5.3　クエリ処理時間

クエリ処理時間について，G-Tree と比較した結果を図 7 に示す．NS と AT および SP の実験結果について，G-Tree は 1 時間以内にインデックス構築が終わらず実行を打ち切ったため，クエリの実行を行っていない．また，提案手法 2 では G-Tree と比べて，交通ネットワークでは最大約 85 倍，ソーシャルネットワークでは最大約 426 倍高速にクエリを処理することができるとわかる．G-Tree はグラフ分割によってはグラフ上で近くにある頂点がインデックス上では離れてしまう可能性があることや，複雑なネットワークでは距離行列が大きくなってしまうことで探索に時間がかかってしまう．それに対し提案手法はクエリノードを中心に探索を行っていくため，クエリノードの近隣のノードに対して効率的にクエリを処理することができることがわかる．



(a) CAL　(b) NY

(c) TV　(d) GV

(e) NS　(f) AT

(f) SP

図 8　k を変化させた場合のクエリ処理時間の比較

k をグラフの頂点数に対して 25%，50%，75%および 100%と変化させ，それぞれのデータセットについて実験を行った結果を図 8 に示す．NS と AT および SP の実験結果について，G-Tree は 1 時間以内にインデックス構築が終わらず実行を打ち切ったため，クエリの実行を行っていない．G-Tree は交通ネットワークでは，距離行列あたりのボーダーの数が少なく探索範囲が広がるにつれて多くの距離行列を探索する必要があるため，探索範囲の増加とともに検索時間が増加しているのではないかと考えられる．また，複雑なネットワークではデータが重みなし無向グラフのため，検索したノード内のほとんどの頂

1：https://github.com/TsinghuaDatabaseGroup/GTree
2：http://users.diag.uniroma1.it/challenge9/download.shtml
3：http://snap.stanford.edu/data

点が結果リストに追加できるため，同じノードの検索を繰り返す必要がなく探索時間にあまり変化がないのではないかと考えられる．これに対して提案手法は枝刈り済みのインデックス上の探索を逐次的に行うため，交通ネットワークでも複雑ネットワークでも検索時間の増加に同様な傾向が見られる．

6 結　　論

本稿では k 最近傍検索を高速に処理するグラフ型インデックスを提案した．提案手法は既存手法の問題点であった大規模で複雑なグラフからでも高速にインデックスを構築することができ，クエリ処理も高速に行うことができる．また，交通ネットワークでも同様にインデックス構築とクエリ処理を高速に行うことができる．実データを用いた実験の結果から，提案手法の有効性を確認することができた．

今後の課題として，より大規模なデータセットに対しても効率的にインデックス構築とクエリ処理を行うことができるかの確認を行うことと，最新の手法との比較がまだできていないため，最新の手法との比較実験を行うことが挙げられる．

謝　　辞

本研究の一部は JST ACT-I，JST さきがけ（JPMJPR2033）及び JSPS 科研費 JP18K18057 による支援を受けたものである．

文　　献

[1] Ruicheng Zhong, Guoliang Li, Kian-Lee Tan, Lizhu Zhou, and Zhiguo Gong, G-Tree: An Efficient and Scalable Index for Spatial Search on Road Networks, IEEE Transactions on Knowledge and Data Engineering, VOL.27 NO.8, August, 2015, pp.2175-2189

[2] Z. Li and L. Chen and Y. Wang, G*-Tree: An Efficient Spatial Index on Road Networks, IEEE International Conference on Data Engineering, 2019, pp.268-279

[3] Abeywickrama, Tenindra and Cheema, Muhammad, Efficient Landmark-Based Candidate Generation for kNN Queries on Road Networks, Database Systems for Advanced Applications, March, 2017, pp.425-440

[4] Dimitris Papadias, Jun Zhang, Nikos Mamoulis, Yufei Tao, Query Processing in Spatial Network Databases, International Conference on Very Large Data Bases, 2003, pp.802-813

[5] K. C. K. Lee and W. Lee and B. Zheng and Y. Tian, Query Processing in Spatial Network Databases, IEEE Transactions on Knowledge and Data Engineering, VOL.24 NO.3, 2012, pp.547-560

[6] Samet, Hanan and Sankaranarayanan, Jagan and Alborzi, Houman, Scalable Network Distance Browsing in Spatial Databases, Special Interest Group on Management of Data, 2008, pp.43-54

[7] Goldberg, Andrew V., and Chris Harrelson., Computing the shortest path: A search meets graph theory., SODA, 2005, pp.156–165

[8] Okabe, A., Boots, B., Sugihara, K, Spatial Tessellations: Concepts and Applications of Voronoi Diagrams., John Wiley and Sons, Inc., 2nd edn., 2000

[9] Guttman, Antonin, R-Trees: A Dynamic Index Structure for Spatial Searching, Special Interest Group on Management of Data, 1984, pp.47–57

[10] G. Karypis and V. Kumar, Analysis of multilevel graph partitioning Association for Computing Machinery/IEEE, 1995, p.29