動的グラフにおけるk最近傍探索のための索引更新手法の提案

小林　瑞季†　　真次　彰平†　　塩川　浩昭††

† 筑波大学理工情報生命学術院システム情報工学研究群
†† 筑波大学計算科学研究センター
E-mail: †{kobayashi, 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索引を事前に構築する手法があるが，これらの手法はノードやエッジの追加・削除を伴う動的なグラフの更新に対応することが難しい．そこで本稿では，動的グラフに対する効率的な索引更新手法を提案する．本稿では実データを用いた評価実験により，提案手法が従来手法よりも効率的に索引構築・k 最近傍検索が可能であることを確認した．

キーワード　グラフデータベース，データ構造・索引，問合せ処理

1　はじめに

近年のソーシャルアプリケーションの発展に伴い，巨大なデータを表現するために大規模複雑グラフの重要性が増している [1], [2]．このようなグラフの分析において，k 最近傍検索 [3], [4], [5], [6] は様々なアプリケーション [7], [8] に不可欠な構成要素である．グラフ中のクエリノードが与えられると，k 最近傍検索はクエリノードからの最短距離が短い上位 k 個のノード集合を出力する．k 最近傍検索は，従来の距離ベースの問合わせ [5], [9] とは異なりクエリノード近傍を局所的に探索するため，応答に要する計算コストが小さい．この特徴を活用し，グラフ上での k 最近傍検索は，ライドシェアサービス [7] や，スパムやフェイクニュースの検出 [8] など様々な社会的応用がなされている重要な研究課題である．

k 最近傍検索は多くのアプリケーションで有用であるが，既存の k 最近傍検索手法には 2 つの問題点がある．1 つ目の問題点は複雑なグラフを対象としない点である．既存の最先端の k 最近傍検索手法は主に道路ネットワークのような平面グラフを前提としており，そのような密度の小さなグラフにおいては効率的な検索を実現する．しかしながら，近年注目を集めているソーシャルネットワークは局所的に非常に密な構造を持つため，既存手法は大きな計算コストを要し，効率的な検索ができない．2 つ目の問題点は大規模なグラフにおいて実用的でない点である．既存の k 最近傍検索手法は最大でも数千ノード程度の小さなグラフのみを対象に評価を行っており，それより大きなグラフを与えた場合には実用的な時間で応答できない．例えば，既存の最先端の k 最近傍検索手法 [3] は 7,000 ノード程度のグラフに対して前処理に 1 時間以上を要する．しかしながら，近年のソーシャルネットワークは数百万ノードを持つ大規模なグラフであり，それらを用いるアプリケーションにおいて k 最近傍検索を行うために既存手法を用いることは現実的でない．

1.1　既存研究と本研究の位置付け

k 最近傍検索における膨大な計算コストを克服するために，様々なアプローチが提案されてきた．最も効果的手法であるグラフ型索引を用いた手法 [3], [4], [10] はグラフをいくつかの部分グラフに分割し，各部分グラフ内の 2 ノード間の最短距離を索引として保存するアプローチである．グラフ型索引を用いた手法の代表的な例として，G-Tree [3] と ILBR [10] が挙げられる．G-Tree は Metis [11] を用いてグラフを不連続な部分グラフに分割し，それぞれの部分グラフに含まれる全ノード間の距離を事前に計算し索引とする手法である．ILBR はグラフからいくつかのランドマークノードを選択し，それらと全ノード間の最短距離を用いて A*法に基づく評価値 ALT [12] を事前計算し，索引を構築する．これらの手法は共通して，事前計算により構築した索引を用いることにより，不要なノードやエッジの計算を回避し，クエリに対して高速に k 最近傍ノードを検索する．

索引構築を行う k 最近傍探索に関する既存手法には次に示す 3 つの問題点がある．第 1 に，大規模複雑ネットワークに対する索引の構築に大きな計算コストを要するという点である．これは既存手法が道路ネットワークのような平面グラフのみを対象としており，より密度の大きな複雑ネットワークに対しては効率が悪いことに起因する．第 2 に，索引構築後の k 最近傍検索にも大きな計算コストを要するという点である．これは大規模複雑ネットワークに対しては索引を構築できる領域が小さく，各クエリに対して多くの計算が必要となるためである．第 3 に，ノードやエッジの更新に対応できないという点である．近年のグラフデータは更新が頻繁に行われるが，既存手法ではその差分のみに対する索引の再構築を行えない．そのためグラフの更新に応じて索引を一から再構築する必要があり，実用的でない．

以上のことから，k 最近傍検索に関する既存手法は (1) 索引構築の効率 (2) クエリ応答の効率 (3) 動的グラフへの対応の 3 点に問題点を持つ．そのため本稿ではこれら 3 つの問題点を解決する高速で実用的な k 最近傍検索手法を提案する．

1.2 本研究の貢献

本稿では大規模複雑ネットワークに対する高速な k 最近傍検索手法を提案する．提案手法は複雑ネットワークが持つ core-tree 特性に着目する．これは複雑ネットワークは密な部分グラフであるコアと疎で閉路を持たない木に分解でき，多くの場合全体の 9 割以上が木であるという性質である [13].

core-tree 特性に基づき，提案手法ではコアと木のそれぞれに最適化された索引 *Core-Tree-aware index (CT index)* を構築する．具体的には，提案手法はまずグラフから木構造を抽出し，その根ノードから各葉ノードまでの距離を格納する *tree-index*，および tree-index に含まれないノードをコアとし，コアに属するノード間の距離を格納する *core-index* を構築する．また検索時には core-index を用いてコアからコアまたは各根ノードまでの距離を計算し，tree-index を用いて根ノードから各葉ノードまでの距離を求める．さらには，CT index はグラフの更新が起きた際に根ノードから木を辿ることにより，索引に影響がある部分のみを修正することが可能である．そのため，提案手法はこの特性を利用して効率的な索引更新手法を設計する．

本研究の貢献は以下の通りである．

- **索引構築の効率性:** 我々の評価実験において，提案手法の索引構築はここ数年で提案された k 最近傍検索手法の索引構築と比較して最大約 10,000 倍高速である（5.2.1 節）.
- **k 最近傍検索の効率性:** 提案手法の k 最近傍検索クエリへの応答はここ数年で提案された k 最近傍検索手法のクエリ応答と比較して最大約 146 倍高速である（5.2.2 節）.
- **再構築不要な索引の更新:** 提案手法はグラフの更新に対して全体の再構築が不要であり，提案手法の索引の差分計算は再構築した場合と比較して約 4,000 倍高速である（5.2.3 節）.
- **正確性:** 提案手法が用いる索引構築およびクエリ応答アルゴリズムは，計算コストが大幅に減少するにもかかわらず，常に正確な k 最近傍ノードを出力する（定理 1）.

我々の知る限り提案手法は索引構築と k 最近傍検索の高速性を兼ね備えた最初の手法である．例えば，160 万ノードのソーシャルネットワークに対して，提案手法は 5 秒以内に索引を構築し，1 秒以内に正確な kNN ノードを発見できる．本研究を通じて，グラフ k 最近傍検索は様々なアプリケーションの品質を向上させるのみならず，その応用範囲の拡大が可能となる．

2 事 前 準 備

本稿で用いる主な記号を表 1 に定義する．本稿では重み付き無向グラフ $G = (V, E, W)$ を考える．ここで，V, E, W はそれぞれノード，エッジ，エッジの重みの集合である．2 ノード $u, v \in V$ 間にエッジが存在する場合，そのエッジを $e(u, v) \in E$ と表し，任意のエッジ $e(u, v) \in E$ に対して対応するエッジの重み $w(u, v) \in W$ が存在する．ただし，$w(u, v) \in \mathbb{N}$ である．

k 最近傍検索はグラフ中のクエリノードから近い上位 k 件のノードを見つける問題である．まず最短距離を定義する．

定義 1 (最短距離)**.** G におけるパス $u = u_0 \to u_1 \to u_2 \to \cdots \to u_i = v$ が，$u, v \in V$ の最短経路であるとする．このとき，2 ノード u, v 間の最短距離を $dist(u, v)$ と表し，以下のように定義する．

表 1: 本稿が用いる記号の定義

記号	定義
G	重みあり無向グラフ.
V, E, W	ノード集合，エッジ集合，エッジの重み集合.
$e(u, v)$	2 ノード u, v 間のエッジ.
$w(u, v)$	$e(u, v)$ の重み．$w(u, v) \in \mathbb{N}$.
q	k 最近傍検索におけるクエリノード．$q \in V$.
k	k 最近傍検索における出力ノード数．$k \in \mathbb{N}$.
$dist(u, v)$	2 ノード u, v 間の最短距離（定義 1）.
$dist_k(q, V)$	ノード q に対して k 番目に小さい最短距離.
$V_k(q)$	k 最近傍検索の結果ノード集合.
$\mathcal{I}$	k 最近傍検索のための CT index（3 節）.
$\mathcal{T}$	tree-index（定義 2）.
$\mathcal{C}$	core-index（定義 3）.
$f_l(u)$	ラベル関数（定義 4）.
$\overline{d}(r)$	r からの最短経路距離の上限値（定義 5）.
$u.parent$	ノード u の親ノード.

$$dist(u, v) = \textstyle\sum_{i=0}^{i-1} w(u_i, u_{i+1}).$$

また，$dist_k(q, V)$ を k 番目に近いノードまでの最短距離，すなわち集合 $\{dist(q, v) \mid v \in V\}$ における k 番目に小さい要素として定義する．

定義 1 に基づき，k 最近傍検索を以下のように定義する:

問題定義 1 (k 最近傍検索)**.** グラフ $G = (V, E, W)$，クエリノード $q \in V$，出力ノード数 $k \in \mathbb{N}$ が与えられたとき，k 最近傍検索は次に示すノード集合 $V_k(q)$ を見つける問題である．

$$V_k(q) = \{v \in V \mid dist(q, v) \leqq dist_k(q, V)\}.$$

3 Core-tree-aware index 構築

本節では，大規模複雑ネットワークにおける k 最近傍検索処理問題（問題定義 1）を効率的に解決するためのグラフ型索引手法，*Core-Tree-aware index (CT index)* を提案する．3.1 節において，提案手法 CT index の基本的な考え方を示し，その後，CT index の詳細について述べる．具体的には，3.2 節で索引構築手法，3.3 節で k 最近傍検索アルゴリズムを示す.

3.1 基本アイデア

1 節で述べたように，G-Tree [3] などの既存の索引構築手法は索引を構築する際に高い計算コストを要する．これは既存手法におけるグラフ分割が道路ネットワークのような平面グラフのみを対象としているためである．しかしながら，実世界のネットワークはより複雑であり，例えば，[14], [15] では複雑ネットワークは平面グラフと比較して密度が高く，多様な構造を持つことが報告されている．さらには，複雑ネットワークはスモールワールド性を持つため，分割された各部分グラフの間に介在するエッジが多い．このようなエッジは索引構築時の効果的な分割を妨げるため，索引上でのクエリ応答性能が悪くなる．その結果として，既存手法は複雑ネットワークに対して索引構築とクエリ応答の双方において効率性を損なう．以上のことから，既存のグラフ分割に基づく索引構築手法は複雑ネットワークに対して良好な分割を得ることが難しく，実用的な時間内に索引構築およびクエリ応答ができない [16].

Algorithm 1 CT INDEX 構築

Require: グラフ $G = (V, E, W)$;
Ensure: CT index $\mathcal{I} = \langle \mathcal{T}, \mathcal{C} \rangle$;

▷ **(Step 1) Tree-indexing:**
1: $\mathbb{T} \leftarrow$ EXTRACTTREES(G) and $\mathbb{D} \leftarrow \emptyset$;
2: **for each** $T_i \in \mathbb{T}$ **do**
3: $r \leftarrow root(T_i)$ and $D_i \leftarrow \emptyset$;
4: **for each** $v \in T_i$ **do**
5: $D_i \leftarrow D_i \cup \{\langle v, dist(r, v)\rangle\}$;
6: **end for**
7: $\mathbb{D} \leftarrow \mathbb{D} \cup \{D_i\}$;
8: **end for**
9: $\mathcal{T} = (\mathbb{T}, \mathbb{D})$;

▷ **(Step 2) Core-indexing:**
10: **for each** $T_i \in \mathbb{T}$ **do**
11: $r \leftarrow root(T_i)$ and $V_c \leftarrow (V \backslash T_i) \cup \{r\}$;
12: **end for**
13: $E_c = \{e(u, v) \in E \mid u, v \in V_c\}$ and $W_c \leftarrow \emptyset$;
14: **for each** $e(u, v) \in E_c$ **do**
15: $W_c \leftarrow W_c \cup \{w(u, v)\}$;
16: **end for**
17: $\mathcal{C} = (V_c, E_c, W_c)$;
18: **return** $\mathcal{I} = \langle \mathcal{T}, \mathcal{C} \rangle$;

本研究では複雑なネットワークの構造的性質に着目し，k 最近傍検索に対する高速なグラフ型索引構築手法および高速なクエリ応答手法を提案する．そのために，本稿では複雑ネットワークの core-tree 特性を考慮した新たな索引である，CT index を設計する．これは，複雑ネットワークのコア部分に対する網羅的な最短距離計算を効率化するための索引 core-index と，ツリー部分対する根ノードから各葉ノードまでの距離を効率的に格納する索引 tree-index の 2 つからなる索引である．さらには，CT index を用いた k 最近傍検索手法として，tree-index に基づいて木に対する不要な計算を枝刈りし，core-index 上のノードを逐次的に探索する．これにより提案手法はコアから各根ノードまでの経路を効率的に探索し，根ノード以降の木構造の探索は省略することが可能となる．

3.2 索引構築アルゴリズム

core-tree 特性に従い，提案手法は CT index $\mathcal{I} = \langle \mathcal{T}, \mathcal{C} \rangle$ を構築する．ここで，$\mathcal{T}$ と $\mathcal{C}$ はそれぞれ tree-index と core-index を表す．本稿ではまず tree-index を以下に定義する．

定義 2 (Tree-index $\mathcal{T}$)**.** $\{T_1, T_2, \dots\}$ をグラフに含まれる木の集合とし，木 T_i の根ノードを r_i とおく．ノード $v \in T_i$ と v から r_i までの最短距離 $dist(r_i, v)$ のペア $\langle v, dist(r_i, v)\rangle$ の集合を D_i とする．つまり，$D_i = \bigcup_{v \in T_i} \langle v, dist(r_i, v)\rangle$ である．tree-index を $\mathcal{T} = (\mathbb{T}, \mathbb{D})$ として定義し，$\mathbb{T}$ と $\mathbb{D}$ はそれぞれ T_i と D_i の集合，すなわち，$\mathbb{T} = \{T_1, T_2, \dots\}$ と $\mathbb{D} = \{D_1, D_2, \dots\}$ を表す．

定義 2 に示すように，tree-index $\mathcal{T}$ は G に含まれる木について，それぞれの根ノードからその木に属す各ノードまでの最短距離を格納する．次に，core-index を以下のように定義する．

定義 3 (Core-index $\mathcal{C}$)**.** V_c を G に含まれるコアノードと根ノードの集合とする．*core-index* は $\mathcal{C} = (V_c, E_c, W_c)$ とする．ここで，$E_c = \{e(u, v) \in E \mid u, v \in V_c\}$，$W_c = \{dist(u, v) \mid e(u, v) \in E_c\}$ である．

定義 3 に示すように，core-index $\mathcal{C}$ は G に木に含まれないノードとそれぞれの木の根ノードの接続関係を格納する．

定義 2 および定義 3 に基づき，全てのノードは core-index および tree-index のいずれか一方に格納される．ここで各ノードの所属を識別するために，次に示すラベル関数 f_l を定義する．

定義 4 (Label function f_l)**.** ラベル関数 f_l は，$u \in T$ のとき，$f_l(u) = tree$ であり，そうでない場合 $f_l(u) = core$ である．

アルゴリズムの概要: Algorithm 1 は，$\mathcal{I} = \langle \mathcal{T}, \mathcal{C} \rangle$ を構築する疑似コードである．このアルゴリズムは，tree-index（1–7 行目）の構築と core-index の構築（8–14 行目）の二つのステップからなる．tree-index の構築について，グラフ G が与えられたとき Tree-indexing は定義 2 に基づき，まず G に含まれる全ての木 $\mathbb{T} = \{T_1, T_2, \dots\}$ を取り出す．そのために，Algorithm 1 は EXTRACTTREES 関数（1 行目）を呼び出すが，この関数では incremental aggregation method [17] 法を用いる．この抽出手法の概要は以下の通りである．

手順 1． $u \in V$ について，次数が 1 であるものを選択する．
手順 2． u を隣接ノードに集約する．
手順 3． 全てのノードが少なくとも 2 つの隣接ノードを持つまで，手順 1 と手順 2 を繰り返す．
手順 4． 集約されたノードを $\mathbb{T}$ として出力する．

手順 1 と手順 2 によりノード $u \in V$ が隣接ノード $v \in V$ に集約されると，v の次数は減少する．例えば，u と v の次数がそれぞれ 1 と 2 である場合，集約後の v の次数は 1 となる．したがって，この集約は全てのノードが少なくとも 2 つの隣接ノードを持つまで手順 1 と手順 2 を繰り返すことにより，木の集合 $\mathbb{T} = \{T_1, T_2, \dots\}$ を求めることができる．[17] で述べられているように，この集約は木のノード数に対して線形の時間計算量を要する．したがって，α を木に含まれるノードの割合とすると，この集約は $O(\alpha|V|)$ 時間で $\mathbb{T}$ を構築できる．

ここで，木の集合 $\mathbb{T} = \{T_1, T_2, \dots\}$ について，それぞれの木は木に含まれるノード集合を保持している．Algorithm 1 では，各木 $T_i \in \mathbb{T}$ について定義 2 の D_i を計算する（2–6 行目）．最後に，tree-index $\mathcal{T} = (\mathbb{T}, \mathbb{D})$ を出力する（9 行目）．

次に，core-index $\mathcal{C}$ を構築する（10–17 行目）．定義 3 に示したように，$\mathcal{C}$ は V_c，E_c，W_c から構成される．まず，$\mathbb{T}$ の木に含まれない全てのノードとすべての根ノードを含む集合 V_c を構築する（10–11 行目）．次に，V_c のノードが E にエッジを持つ場合，それらを隣接させる（13–15 行目）．

3.3 k 最近傍検索処理

本節では CT index $\mathcal{I} = \langle \mathcal{T}, \mathcal{C} \rangle$ を用いて，クエリに対して提案手法がどのように k 最近傍検索を行うかを説明する．提案手法は木に含まれるノードの計算を省略することで効率的な検索を実現する．具体的には，まず $\mathcal{C}$ を用いてコアノードを始点とし，クエリノード q に近い k 件のノードが見つかるまで探索を行う．その過程で木 T_i の根ノード r_i に到達した場合，T_i 全体が q に対する k 最近傍ノードになり得るか，tree-index $\mathcal{T}$ を用いて上限値推定を行い判定する．これにより，T_i 全体が k 最近傍ノードになる場合，提案手法は T_i の内部の探索を省略でき，高速なクエリ応答が可能となる．

上記のアイデアを実現するために，上限値に関して次の定義を導入する．

Algorithm 2 k 最近傍検索処理

Require: CT index $\mathcal{I} = \langle \mathcal{T}, \mathcal{C} \rangle$, クエリノード q, 検索結果の数 k;
Ensure: kNN ノード集合 $V_k(q)$;

▷ **(Step 1) Initialization:**
1: $V_k(q) \leftarrow \emptyset$, and priority queue $Q \leftarrow \emptyset$;
2: **if** $f_l(q) = tree$ **then**
3: Obtain T_i *such that* $q \in T_i$ and $r \leftarrow root(T_i)$;
4: Obtain $dist(q, r)$ from D_i;
5: $\langle V_k(q), Q \rangle \leftarrow$ TreePruning$(\langle r, dist(q, r) \rangle)$;
6: **else**
7: $Q \leftarrow \{\langle q, 0 \rangle\}$;
8: **end if**
▷ **(Step 2)** k**NN search:**
9: **while** $\max\{dist(q, v') \mid v' \in V_k\} < dist_{k+1}(q, V)$ **do**
10: $\langle u, dist(q, u) \rangle \leftarrow Q.dequeue()$;
11: **if** $f_l(u) = tree$ **then**
12: $\langle V_k(q), Q \rangle \leftarrow$ TreePruning$(\langle u, dist(q, u) \rangle)$;
13: **else**
14: $V_k(q) \leftarrow V_k(q) \cup \{u\}$;
15: **for each** $\{e(u, v) \in E_c \mid v \notin V_k(q)\}$ **do**
16: $dist(q, v) \leftarrow dist(q, u) + w(u, v)$;
17: Q.enqueue$(\langle v, dist(q, v) \rangle)$;
18: **end for**
19: **end if**
20: **end while**
21: **return** $V_k(q)$;
▷ **Subroutine for tree pruning:**
22: **procedure** TreePruning$(\langle r, dist(q, r) \rangle)$
23: Obtain T_i *such that* $r \in T_i$;
24: **for each** $\{e(r, v) \in E_c \mid v \notin V_k(q)\}$ **do**
25: $dist(q, v) \leftarrow dist(q, r) + w(r, v)$;
26: $Q \leftarrow Q \cup \{\langle v, dist(q, v) \rangle\}$;
27: **end for**
28: $d_{\min} \leftarrow \min\{dist(q, v) \mid \langle u, dist(q, v) \rangle \in Q\}$;
29: **if** $|V_k(q)| + |T_i| \leqq k$ and $\overline{d}(r) \leqq d_{\min}$ **then**
30: $V_k(q) \leftarrow V_k(q) \cup T_i$;
31: **else**
32: **for each** $v \in T_i$ **do**
33: $f_l(v) \leftarrow core$, $dist(q, v) \leftarrow dist(q, r) + dist(v, r)$;
34: $Q \leftarrow Q \cup \{\langle v, dist(q, v) \rangle\}$;
35: **end for**
36: **end if**
37: **end procedure**

定義 5 (上限値 $\overline{d}$)**.** T_i のルートノードを r_i とすると，r_i から T_i 中のノードの最短距離の上限値 $\overline{d}(r_i)$ を以下に定義する．

$$\overline{d}(r_i) = \begin{cases} dist_{\max}(T_i) - dist(q, r_i) & (q \in T_i) \\ dist(q, r_i) + dist_{\max}(T_i) & (Otherwise) \end{cases}$$

ここで，$dist_{\max}(T_i) = \max\{D_i\}$ である．

定義 5 は $\overline{d}(r_i)$ がクエリノード q と T_i 内のノードとの最長距離であることを示す．つまり，$\overline{d}(r_i)$ は q と T_i 間の距離の上限値である．定義 5 より，次の補題が成り立つ．

補題 1. T_i の根ノードを r_i とし，$d_{\min} = \min\{dist(q, v)\}$ とする，ここで，$v \in Q \cup \{v \mid e(r_i, v) \in E_c, v \notin V_k(q)\}$ である．$|V_k(q)| + |T_i| \leqq k$ と $\overline{d}(r_i) \leqq d_{\min}$ が成り立つならば，$T_i \subseteq V_k(q)$ が成り立つ．

証明. $q \in T_i$ のとき補題 1 は自明に成り立つため，$q \notin T_i$ の場合について背理法により補題を示す．ノード u が $u \in T_i$ かつ $V_k(q)$ であると仮定する．このとき，補題の前提条件より $|V_k(q)| + |T_i| \leqq k$ であり，少なくとも 1 つのノード $u' \in V \backslash \{V_k(q) \cup T_i\}$ について $u \notin V_k(q)$ であるから，$dist(q, u') < \overline{d}(r)$ である．これは，$\overline{d}(r) \leqq d_{\min}$ に矛盾する．よって，補題 1 は成り立つ． □

補題 1 により，補題に示される条件を満たす木は計算を省略できることがわかる．

アルゴリズムの概要: $\mathcal{I}$ を用いた k 最近傍検索の疑似コードを Algorithm 2 に示す．Algorithm 2 は主探索アルゴリズム（1–17 行目）と木構造を探索するサブルーチンアルゴリズム（22–34 行目）からなる．主探索アルゴリズムは索引 $\mathcal{I}$ を用いて q に対する k 最近傍ノードを探索する．そのために，まず優先度付きキュー Q をラベル関数 $f_l(q)$ に基づいて初期化する（1–7 行目）．具体的には，$f_l(q) =$*tree* である場合にはサブルーチン TreePruning によって枝刈りできるかを評価し（2–5 行目），そうでない場合には q を Q に挿入する（6，7 行目）．なお，Q はクエリノード q からの最短距離によって昇順に優先度付けされる．

k 最近傍検索ステップ（9–17 行目）では，Q から $\langle u, dist(q, u) \rangle$ を取り出し（10 行目），$V_k(q)$ が距離 $dist_{k+1}(q, V)$ より小さい全てのノードを含むまで k 最近傍検索が続けられる．このとき，$f_l(u) = tree$ ならば，初期化ステップと同様に TreePruning（11–12 行目）を呼び出す．そうでなければ，コアノードの距離を更新して探索を継続する（14–17 行目）．

TreePruning（23–30 行目）は根ノード r に対して，補題 1 を用いて $V_k(q)$ に T_i が含まれるかどうかを調べ，含まれる場合は T_i を探索せずに k 最近傍ノードに含める（29–30 行目）．それ以外の場合は，T_i 内のノードをコアノードとみなし（32–34 行目），k 最近傍検索ステップでコアノードと同様に探索する．

***k* 最近傍検索の正確性:** 補題 1 に従う Algorithm 2 に示す k 最近傍検索処理方法は，終了後に以下の性質を持つ．

定理 1 (正確性)**.** Algorithm 2 で得られた kNN ノード $V_k(q)$ は，G 上で探索された kNN ノードと等価である．

証明. 補題 1 により，TreePruning は T_i の全てのノードが $V_k(q)$ に含まれる場合のみ T_i を枝刈りする．そうでない場合は，Algorithm 2 によりノードを *core* としてラベル付けする（28 行目）．$V_k(q)$ に距離が $dist_{k+1}(q, V)$ より小さいノードが全て含まれるまで，全てのコアノードを探索する．以上のことから，定理 1 が成り立つ．

定理 1 は本研究の索引構築と k 最近傍検索処理のアプローチが，k 最近傍検索の品質を犠牲にしないことを示している．

4 索引の更新

本節では CT index を動的に更新する手法を提案する．グラフは更新される際，ノードの追加と削除，エッジの追加と削除といった処理が行われる．ここで，ノードの追加と削除についてはそれらに隣接するエッジの追加や削除で表されるため，エッジの追加と削除によって本質的に等価な処理に置き換えることが可能である．そのため，本提案手法ではエッジの追加と削除の 2 つの動作を対象に CT index の更新を考える．

索引の更新にあたっては，エッジの追加や削除が発生する領域がコアと木のどちらであるかによって必要な処理が異なる．そのため，4.1 節に示す 4 つの追加に関するケースと，4.2 節に示す 2 つの削除に関するケースについて場合分けを行い，順

に説明する．簡単のために，以降の節では木に含まれる根以外の各ノードについて，自身に隣接する根ノードに近い側のノードを親ノードと呼ぶ．

4.1 エッジの追加

CT index 上でのエッジの追加については以下の 4 つの場合に分けられる．

1． 同じ木に所属するノード間への追加．
2． 違う木に所属するノード間への追加．
3． コアノードと木に所属するノード間への追加．
4． コアノード間への追加．

同じ木に所属するノード間への追加：追加されるエッジの両端のノードを v_1, v_2 とし，これらのノードが属する木を T とおく．このとき，T は $e(v_1, v_2)$ によって閉路が作られ，木構造とならない．そのため，根ノード r から $e(v_1, v_2)$ を通るサイクルを新たにコアとする．その後残った木について，葉ノードから親ノードを辿り新たな根ノードを見つけることで木を更新する．

違う木に所属するノード間への追加：追加されるエッジの両端のノードを v_1, v_2 とし，これらのノードが属する木を T_1, T_2 とする．このとき，T_1 と T_2 の間にまたがって閉路が作られ，T_1, T_2 は木構造とならない．そのため，T_1 の根ノード r_1 から v_1 までのパス，および T_2 の根ノード r_2 から v_2 までのパスを新たにコアとする．その後それぞれの木の残った木について，葉ノードから親ノードを辿り新たな根ノードを見つけることで木を更新する．

コアノードと木に所属するノード間への追加：追加されるエッジの両端のノードの内，木に所属するノードを v_1，コアノードを v_2 とし，v_1 が属する木を T とおく．このとき，T は $e(v_1, v_2)$ によって閉路が作られ，木構造とならない．そのため，根ノード r から v_1 までのパスを新たにコアとする．その後，残った木について，葉ノードから親ノードを辿り，新たな根ノードを見つけることで木を更新する．

コアノード間への追加：単に 2 ノード間にエッジを追加する．

アルゴリズムの概要: エッジが追加された場合に $\mathcal{I}$ を更新するアルゴリズムを Algorithm 3 に示す．Algorithm 3 は主更新アルゴリズム（1–18 行目）と木構造を再構築するサブルーチンアルゴリズム（19–34 行目）からなる．主更新アルゴリズムは，エッジがどのノード間に追加されるかによって場合分けを行い，それぞれの場合について更新を行う．どちらのノードも同じ木に所属する場合（4–6 行目），それらが所属する木に対してサブルーチン TreeRestructuring を呼び出す．それぞれのノードが異なる木に所属する場合（8–10 行目），それぞれが所属する木に対してサブルーチン TreeRestructuring を呼び出す．片方のノードのみが木に所属する場合（13，14 行目），片方のノードが所属する木に対してサブルーチン TreeRestructuring を呼び出す．どちらも木構造に所属しない場合は，それぞれのノード間にエッジを追加して終了する（16，17 行目）．

ノード v_1, v_2 と木 T が与えられたとき，TreeRestructuring は以下の手順で $\mathbb{T}$ を更新する．

Algorithm 3 エッジの追加処理

Require: CT index $\mathcal{I} = \langle \mathcal{T}, \mathcal{C} \rangle$，追加されるエッジ (v_1, v_2, w);
Ensure: 更新された CT index $\mathcal{I} = \langle \mathcal{T}, \mathcal{C} \rangle$;
1: $E_c \leftarrow E_c \cup e(v_1, v_2)$;
2: a Set $E \leftarrow \emptyset$;
3: **if** $f_l(v_1)$ and $f_l(v_2) = tree$ **then**
4: **if** $SameTree(v_1, v_2)$ **then**
5: $T \leftarrow v_1, v_2$ が所属する木;
6: TreeRestructuring($\langle v_1, v_2, T \rangle$);
7: **else**
8: $T_1 \leftarrow v_1$ が所属する木，$T_2 \leftarrow v_2$ が所属する木;
9: TreeRestructuring($\langle v_1, v_2, T_1 \rangle$);
10: TreeRestructuring($\langle v_1, v_2, T_2 \rangle$);
11: **end if**
12: **else if** ($f_l(v_1) = tree$ and $f_l(v_2) = core$) or ($f_l(v_1) = core$ and $f_l(v_2) = tree$) **then**
13: $T \leftarrow v_1, v_2$ どちらかが所属する木;
14: TreeRestructuring($\langle v_1, v_2, T \rangle$);
15: **else**
16: $E_c \leftarrow E_c \cup e(u, v)$;
17: $W_c \leftarrow W_c \cup w$;
18: **end if**
▷ **Subroutine for tree restructuring:**
19: **procedure** TreeRestructuring(v_1, v_2, T)
20: **if** $v_1 \in T$ **then**
21: v_1 から $root(T)$ までのノードをコアに変更;
22: **end if**
23: **if** $v_2 \in T$ **then**
24: v_2 から $root(T)$ までのノードをコアに変更;
25: **end if**
26: $\mathbb{T} \leftarrow \mathbb{T} \backslash T$;
27: $\mathbb{T} \leftarrow \mathbb{T} \cup$ Extract($T_{v_{1,2}}$);
28: **end procedure**

手順 1． v_1 と v_2 の内，T に所属するノードから T の根ノードまでをコアノードとする．

手順 2． $u \in T$ について，次数が 1 であるものを選択する．

手順 3． u を親ノードに集約する．

手順 4． 全てのノードがコアノードに集約されるまで，手順 2 と手順 3 を繰り返す．

手順 5． 更新された木集合を新たに $\mathbb{T}$ として出力する．

TreeRestructuring は Algorithm 1 で用いた集約関数 ExtractTrees に類似する処理を T に用いて木の更新を行う（19–28 行目）．ExtractTrees と異なる点として，TreeRestructuring はどのノードがコアノードか既知である点がある．したがって，葉ノードから根ノードに到着するまで順に親ノードへ集約するだけで木を構築することができる．

4.2 エッジの削除

CT index 上でのエッジの削除については以下の 2 つの場合に分けられる．

1． コアノード間での削除．
2． 同じ木に所属するノード間での削除．

コアノード間での削除：削除されるエッジの両端のノードを v_1, v_2 とする．このとき，エッジが削除された後に，v_1，v_2 共に他の 2 つ以上のノードと索引上で隣接している場合は，エッジを削除して終了する．しかし，いずれかのノードが一つのノードとしか接続していない場合，$\mathbb{T}$ の更新が必要となる．ここで，エッジが削除された後にひとつのノードとしか接続していないノードを n_t，n_t が接続しているノードを n_r とする．n_r が木を集約したノードである場合，新たに n_r に集約されている

Algorithm 4 エッジの削除処理

Require: CT index $\mathcal{I} = \langle \mathcal{T}, \mathcal{C} \rangle$，削除されるエッジ (v_1, v_2);
Ensure: 更新された CT index $\mathcal{I} = \langle \mathcal{T}, \mathcal{C} \rangle$;
1: $E_c \leftarrow E_c \backslash e(v_1, v_2)$;
2: **if** $f_l(v_1)$ and $f_l(v_2) = tree$ **then**
3: $T \leftarrow v_1, v_2$ が所属する木;
4: $n_c \leftarrow v_1, v_2$ のうち，一方がもう一方の子ノードとなっている方のノード;
5: $T \leftarrow T \backslash n_c$ 以下のノード;
6: **else if** $f_l(v_1)$ and $f_l(v_2) = core$ **then**
7: $E_c \leftarrow E_c \backslash e(v_1, v_2)$;
8: **if** $|v_1.adjacent| = 1$ **then**
9: $n_r \leftarrow v_1.adjacent$;
10: $T_{n_r} \leftarrow T_{n_r} \cup v_1$;
11: **else if** $|v_2.adjacent| = 1$ **then**
12: $n_r \leftarrow v_2.adjacent$;
13: $T_{n_r} \leftarrow T_{n_r} \cup v_2$;
14: **else**
15: $E_c \leftarrow E_c \backslash e(v_1, v_2)$;
16: **end if**
17: **end if**

表 2: k 最近傍検索の実験に用いる実データセット (問題定義 1)

Name	$\|V\|$	$\|E\|$	Type	Source
CAL	21,048	21,693	道路ネットワーク	[3]
NY	264,346	366,923	道路ネットワーク	[4]
FLA	1,070,376	2,712,798	道路ネットワーク	[18]
TV	3,892	17,262	ソーシャルネットワーク	[14]
GV	7,057	89,455	ソーシャルネットワーク	[14]
NS	27,917	206,259	ソーシャルネットワーク	[14]
AT	50,515	819,306	ソーシャルネットワーク	[14]
SP	1,632,803	22,301,964	ソーシャルネットワーク	[14]

ノード集合に n_t を加える．n_r が木を集約していないコアノードである場合，n_r を新しく根ノードとし，n_r に集約されているノード集合に n_t を加える．

同じ木に所属するノード間での削除：削除されるエッジの両端のノードを v_1, v_2 とし，これらのノードが属する木を T とおく．ここで，v_1 と v_2 のうち，一方がもう一方の子ノードとなっている方のノードを n_c とする．T に含まれているノード集合から，n_c 以下の木となっているノード集合を取り除くことで，木を更新する．

アルゴリズムの概要: エッジが削除された場合に $\mathcal{I}$ を更新するアルゴリズムを Algorithm 4 に示す．アルゴリズムは，エッジがどのノード間で削除されるかによって場合分けを行い，それぞれの場合について更新を行う．どちらのノードも同じ木に所属する場合（3–5 行目），2 ノードの内，子ノード以下のノードをそれらの所属する木から削除する．どちらもコアノードでエッジを削除した後にいずれかのノードの次数が 1 である場合（7–13 行目），それぞれの隣接ノードからもう一方のノードを削除し，次数が 1 であるノードを隣接しているノードの木の要素にする．エッジを削除した後に両方のノードの次数が 1 より大きい場合（15 行目），それぞれの隣接ノードからもう一方のノードを削除する．

5 評価実験

本章では，提案手法の効率を索引構築時間と k 最近傍検索の処理時間の観点から実験的に考察する．

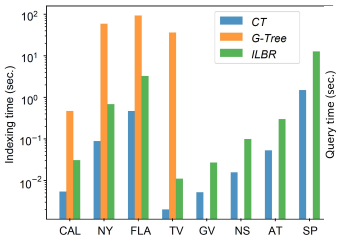


図 1: 索引構築時間

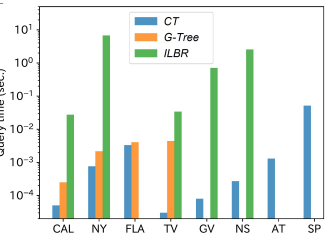


図 2: k 最近傍検索の処理時間

5.1 実験設定

比較手法: 提案手法を以下に示す既存の最先端の索引構築アルゴリズムを含むベースライン手法と実験的に比較する．

- **CT:** 複雑ネットワークにおける core-tree 特性を利用したグラフ索引構築手法を提案する（3 節）．まず Algorithm 1 に基づいて索引を構築し，k 最近傍検索に対して Algorithm 2 を実行する．
- **G-Tree:** 最新鋭のグラフ索引構築手法である [3]．索引を構築するために，G-Tree は Metis [11] を用いてグラフを階層的な部分グラフに分割する．
- **ILBR:** ランドマークを用いたグラフ索引構築手法 [10]．ILBR はネットワークボロノイ図 [19] を索引として利用する．

更新処理手法の評価実験については以下 2 つの設定を比較する．

- **動的更新手法:** 4 節で説明した動的更新手法を用いて CT index の更新を行う．
- **索引再構築法:** 一定本数分の更新エッジの情報が到着する度に CT index を再構築する．

特に断らない限り，G-Tree と ILBR のパラメータ設定は原著論文で推奨されているものと同じものを使用した [3], [10]．また，デフォルトの k の設定として，$k = 0.01 \times |V|$ を使用した．全てのアルゴリズムは C++で実装し，gcc 9.2.0 で-O2 オプションを使ってコンパイルした．全ての実験は Intel Xeon CPU（2.60 GHz）と 128 GiB RAM を搭載したサーバー上で行った．k 最近傍検索について，各データセットに対して，ランダムに 30 個のクエリノードを選択した実行時間を平均したものを結果とする．更新処理について，各データセットに対して，ランダムに $m/2$ 本のエッジを追加し，$m/2$ 本のエッジを削除した場合，つまり，合計 m 本のエッジが更新された場合の実行時間を結果とする．

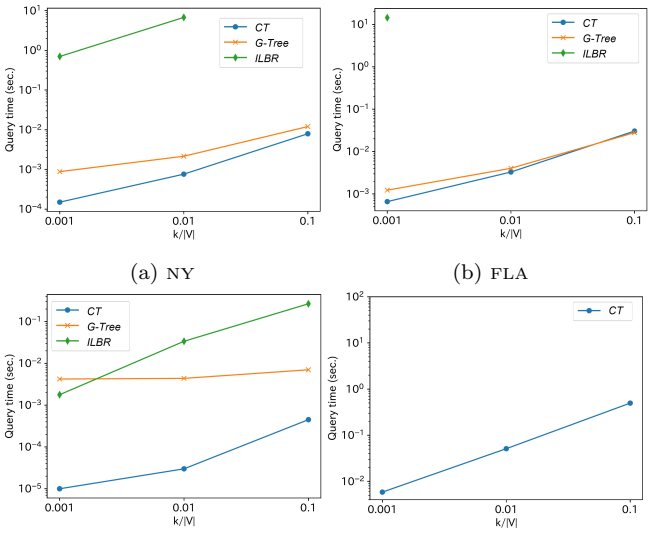
データセット: k 最近傍検索の実験的解析（問題定義 1）には，先行研究 [3], [4] および複数の公開リポジトリ [14], [18] で公開されている 8 つの実世界グラフを用いた．表 2 はそれぞれのグラフの詳細を示す．CT index の有効性を実験的に議論するために，我々は道路ネットワークとソーシャルネットワークの 2 種類の実世界グラフを採用した．表中，CAL，NY，FLA は道路ネットワークを，残りはソーシャルネットワークを表す．

5.2 k 最近傍検索の処理効率

まず，k 最近傍検索問題（問題定義 1）に対する提案手法の実行時間を評価する．

5.2.1 索引構築時間

図 1 は表 2 に示した実データセットについての索引構築時間



(a) NY (b) FLA

(c) TV (d) SP

図 3: k 最近傍検索における k の変化の影響

を示している．GV，NS，AT，SP について，1 時間以内に索引構築が終了しなかったため，G-Tree の結果は省略している．提案手法は，すべての条件下で G-Tree と ILBR を大幅に上回る性能を示し，前述の最新手法と比較して最大で 4 桁の差となることを実証した．例えば，TV に対して提案手法は G-Tree の 18,074 倍高速である．これは，グラフに多くの木が含まれる場合，Algorithm 1 の実行時間を短縮でき，各木 T_i が距離の事前計算のために $O(|T_i|)$ 時間で処理できるためである．ソーシャルネットワークは core-tree 特性を持っているため，部分グラフの大部分は木である．従って，提案手法のアルゴリズムはソーシャルネットワークの索引構築時間を大幅に短縮することができる．

5.2.2 k 最近傍検索の処理時間

図 2 は $k = 0.01 \times |V|$ のときの k 最近傍検索の処理時間である．図 2 において，k 最近傍検索が 1 分以内に終了しない場合，またはその索引が利用できない場合は結果を省略している．提案手法は，ソーシャルネットワークにおいて他の手法を大幅に上回る性能を示しつつ，既存手法が対象としている道路ネットワークに対しても同程度の性能を保持している．これは，提案手法のアルゴリズムが補題 1 により探索木の計算を省略できるためである．このため，ソーシャルネットワークにおいては，前述の最新手法と比較して最大で 2 桁の k 最近傍検索の処理時間を向上させることができる．また，NS，AT，SP のような既存手法が検索に失敗するデータセットに対しても，提案手法は高速な k 最近傍検索の処理を実現した．

図 3 は，検索時間に対する k の影響を示したものである．k は $0.001 \times |V|$，$0.01 \times |V|$，$0.1 \times |V|$ と変化させた．すべてのデータセットに対して類似する結果が得られたが，本稿では紙面の都合上 NY，FLA，TV，SP についてのみ議論する．また，k 最近傍検索が 1 分以内に終了しない場合，またはその索引が利用できない場合は結果を省略している．提案手法は k に関係なくソーシャルネットワーク上で他の手法より有意に高速でありつつ，既存手法が対象とする道路ネットワークに対して

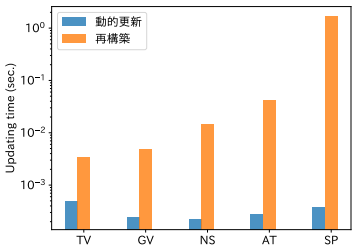


図 4: 索引更新の処理時間

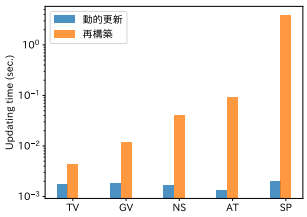


図 5: エッジの追加のみをした場合の処理時間

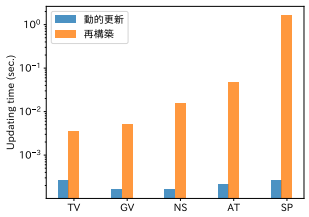
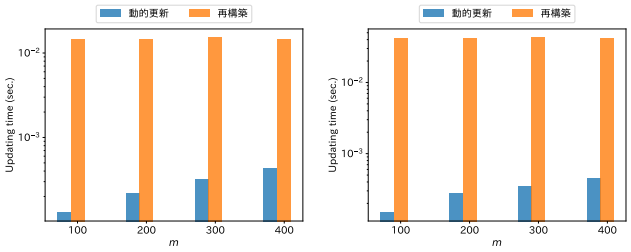


図 6: エッジの削除のみをした場合の処理時間



(a) NS (b) AT

図 7: 索引更新における n の変化の影響

も同程度の性能を示している．これは，ソーシャルネットワークのように木が多いグラフでは提案手法が補題 1 を用いてノードを探索せずに木を大幅に枝刈りできたことに起因する．以上のことから，提案手法は最先端の既存手法よりも大規模で複雑なネットワークに適しているといえる．

5.2.3 更新の処理時間

図 4 は表 2 に示した実データセットについて，$m = 200$ のときの更新処理時間を示している．図 4 より，提案手法は動的な索引更新を用いて再構築するコストを削減したことにより，ネットワークの更新に対して非常に高速な索引実現したといえる．

図 5， 6 は，グラフの更新がそれぞれエッジの追加のみである場合とエッジの削除のみである場合の更新処理時間を示している．これらの実験結果についてはエッジの追加がわずかにエッジの削除よりも処理時間が大きいのみで大きな差異は無く，どちらの更新手法も有効に働いていることがわかる．

図 7 は，更新時間に対する m の影響を示したものであり，本稿では m を $100, 200, 300, 400$ と変化させた．紙面の都合上 NS，AT についてのみ議論するが，他のデータセットについても同様の結果が得られた．図 7 より，提案手法は m に関係なく高速に索引を更新できているといえる．特に，$|E|$ と m の差が大きくなるほど更新時間の差が大きく，提案手法は大規模なグ

ラフに対してより有効であるということがわかる．以上のことから，提案手法は更新が頻繁に行われるような実世界のネットワークに複数の条件で対応可能であり，実用的であるといえる．

6 結　　論

本論文では大規模な複雑ネットワークに対する k 最近傍検索を効率的に計算するための新しい高速索引構築アルゴリズムを提案した．提案手法はグラフのコアと木の索引を別々に構築することにより，グラフに対する k 最近傍検索の品質を落とすことなく索引構築と k 最近傍検索の効率性を兼ね備える．我々の実験において，提案手法は索引構築と k 最近傍検索の処理時間において，最先端の既存手法を最大で 4 桁上回る性能を示した．これらの結果は，本研究の core-tree 特性を考慮した索引構築手法が複雑なネットワークに対するグラフ k 最近傍検索のコストの削減に有効であることを示している．さらには，我々はグラフの動的な変化に対応する効率的な索引の更新アルゴリズムを提案し，提案した索引更新手法は大規模なネットワークでより効率的に索引を更新できることを示した．

謝　　辞

本研究の一部は，JST さきがけ (JPMJPR2033) ならびに JSPS 科研費 (JP22K17894) の支援を受けたものである

文　　献

[1] Hiroaki Shiokawa, Toshiyuki Amagasa, and Hiroyuki Kitagawa. Scaling Fine-grained Modularity Clustering for Massive Graphs. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence (IJCAI 2019)*, pp. 4597–4604, 7 2019.

[2] Hiroaki Shiokawa. Scalable Affinity Propagation for Massive Datasets. *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI 2021)*, Vol. 35, No. 11, pp. 9639–9646, May 2021.

[3] 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, pp. 2175–2189, August 2015.

[4] Zijian Li, Lei Chen, and Yue Wang. G*-Tree: An Efficient Spatial Index on Road Networks. In *Proceedings of the 35th IEEE International Conference on Data Engineering (ICDE 2019)*, pp. 268–279, 2019.

[5] Hanan Samet, Jagan Sankaranarayanan, and Houman Alborzi. Scalable Network Distance Browsing in Spatial Databases. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pp. 43–54, 2008.

[6] Ken C. K. Lee, Wang-Chien Lee, Baihua Zheng, and Yuan Tian. ROAD: A New Spatial Object Search Framework for Road Networks. *IEEE Transactions on Knowledge and Data Engineering*, No. 3, pp. 545–560, November 2012.

[7] Zeqiang Chen, Peng Li, Junlei Xiao, Lei Nie, and Yu Liu. An Order Dispatch System Based on Reinforcement Learning for Ride Sharing Services. In *Proceedings of 2020 IEEE 22nd International Conference on High Performance Computing and Communications; IEEE 18th International Conference on Smart City; IEEE 6th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pp. 758–763, 2020.

[8] Zulfikar Alom, Barbara Carminati, and Elena Ferrari. Detecting Spam Accounts on Twitter. In *2018 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM 2018)*, pp. 1191–1198, 2018.

[9] Hannah Bast, Stefan Funke, and Domagoj Matijevic. Ultrafast Shortest-Path Queries via Transit Nodes. In *Proceedings of a DIMACS Workshop of The Shortest Path Problem*, Vol. 74, pp. 175–192, 2006.

[10] Tenindra Abeywickrama and Muhammad Aamir Cheema. Efficient Landmark-Based Candidate Generation for kNN Queries on Road Networks. In *Proceedings of the 22nd International Conference on Database Systems for Advanced Applications (DASFAA 2017)*, pp. 425–440, 2017.

[11] George Karypis and Vipin Kumar. Analysis of Multilevel Graph Partitioning. In *Proceedings of the IEEE/ACM SC95 Conference (SC 1995)*, 1995.

[12] Andrew V. Goldberg and Chris Harrelson. Computing the Shortest Path: A Search Meets Graph Theory. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 156–165, 2005.

[13] Austin Benson and Jon Kleinberg. Link Prediction in Networks with Core-Fringe Data. In *Proceedings of The Web Conference 2019 (WWW 2019)*, pp. 94–104, 2019.

[14] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford Large Network Dataset Collection. `http://snap.stanford.edu/data`, June 2014.

[15] Hiroaki Shiokawa, Yasuhiro Fujiwara, and Makoto Onizuka. SCAN++: Efficient Algorithm for Finding Clusters, Hubs and Outliers on Large-Scale Graphs. *PVLDB*, Vol. 8, No. 11, pp. 1178–1189, July 2015.

[16] Makoto Onizuka, Toshimasa Fujimori, and Hiroaki Shiokawa. Graph Partitioning for Distributed Graph Processing. *Data Science Engineering*, Vol. 2, No. 1, pp. 94–105, 2017.

[17] Hiroaki Shiokawa, Yasuhiro Fujiwra, and Makoto Onizuka. Fast Algorithm for Modularity-based Graph Clustering. In *Proceedings of the 27th AAAI Conference on Artificial Intelligence (AAAI 2013)*, pp. 1170–1176, 2013.

[18] Camil Demetrescu. The 9th DIMACS Implementation Challenge. `http://users.diag.uniroma1.it/challenge9/download.shtml`, June 2010.

[19] Atsuyuki Okabe, Barry Boots, Kokichi Sugihara, and Sung Nok Chiu. *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams*. Series in Probability and Statistics. John Wiley and Sons, Inc., 605 Third Ave. New York, NY, United States, 2nd edition, 2000.