DEIM Forum 2021 A25-5

計算ノートブック類似検索のための高速な検索アルゴリズム

堀内　美聡† 　山崎　翔平† 　佐々木勇和† 　肖　　川† 　鬼塚　　真†

† 大阪大学大学院情報科学研究科　〒 565–0871 大阪府吹田市山田丘 1-5

E-mail: †{horiuchi.misato,yamasaki.shohei,sasaki,chuanx,onizuka}@ist.osaka-u.ac.jp

あらまし　Jupyter Notebook をはじめとする計算ノートブックは，機械学習やデータ分析のために多くのユーザに利用され，インターネット上に公開されている．類似した計算ノートブックを高精度に検索できれば，再利用性が向上するが，計算ノートブックの類似検索に特化した検索技術は存在しない．本研究では，ソースコードや表形式データの内容が類似した計算ノートブックの検索技術を提案する．この手法では，計算ノートブックをワークフロー化しサブグラフマッチングを利用することによって，データ処理や表形式データ読み込みなどの順序を保持しつつ高速な検索を実現する．さらに，計算順序の選択やインデックスの利用により，検索を最適化し高速性を向上させる．実データを用いた実験により，提案手法は高精度かつ高速な検索が可能であることを示す．

キーワード　類似検索，計算ノートブック，サブグラフマッチング

1 はじめに

計算ノートブックは対話型のプログラミングインタフェースであり，代表例として Jupyter Notebook が挙げられる．機械学習やデータ分析を主な用途として多くのユーザに利用され，読み込んだデータセットの内容確認や，利用しやすい形へ整形するデータクリーニング，分析から視覚化に至るまで，様々な操作を行うことができる．多機能かつ実装上の利便性が高い計算ノートブックの循環利用が活発化しており，GitHub や Kaggle に公開されている計算ノートブックの数は膨大になっている [1].

インターネット上に公開されている既存の計算ノートブックを再利用する事で目的の機能を容易に実装できる．そのため，計算ノートブックを対象とした高精度かつ高速な類似検索技術の需要が高まっている．しかしながら，計算ノートブックの多機能性故に，内容に関する情報が適切なファイル名として保存されているケースは少なく，どのような処理を何の目的で作成したのかが分からない計算ノートブックも多い [2]．このような内容とファイル名が一致していない計算ノートブックが多数ある中で，キーワード検索は現実的でない．また，計算ノートブックのソースコードはセル単位で区切られており，表形式データや使用ライブラリ，セルの出力と関係し合っている．そのため，ソースコードに対する類似検索技術 [3, 4] や表形式データのみを対象とした検索技術 [5] により類似した計算ノートブックを検索することは難しい．これらのことから，計算ノートブックのソースコード，表形式データ，使用ライブラリ，およびセル出力を包括的に計算可能な検索技術が必要である．

計算ノートブックの各要素の総合的な類似度計算は，各要素の類似度の和で計算する方法が考えられる．この方法は，各要素の類似度をそれぞれ全て計算するため，計算量が膨大になる問題がある．また，各要素は関係し合っており，この関係性を崩さず高速化を実現することは困難である．例えば，表形式データはソースコードの実行によって編集され，セル出力の図は表形式データに基づいて生成される．これらの関係性の保持，各要素の類似度の計算，高速性の全てを両立した検索は容易ではなく，筆者らの知る限り既存研究も存在しない．

そこで本稿では，計算ノートブックのソースコード，表形式データ，使用ライブラリ，セル出力の各要素の類似度計算に加えて，要素間の関係性とセルの実行順序を捉えつつ計算コストを削減可能な新たな類似検索フレームワークを提案する．検索フレームワークでは，計算ノートブックを有向非巡回グラフ (DAG) に変換し，サブグラフマッチングを行うことにより，セルの実行順序や要素間の関係性を捉えた検索，および類似度計算のコストの削減が可能となる．さらなる効率化のために，インデックスの構築と要素の類似度計算順の最適化を行う．まず，インデックスでは，計算ノートブックのセル間の関係性に基づいてインデックスを構築し，サブグラフマッチングの候補空間の枝刈りを行う．次に，計算順の最適化では，検索結果に入り得ない候補の早期の枝刈りを行う．これらの高速化技術により，計算対象とする計算ノーブックと類似度計算のコストの削減による高速な類似度計算を実現する．

実験では Kaggle で共有されている計算ノートブック 111 個を用いて検索フレームワークの検索精度と検索時間の評価を行う．検索精度の評価実験では，ユーザ実験により，検索目的に沿った内容の計算ノートブックが検索されることを示す．検索時間の評価実験では，提案手法が高速に検索できることを示す．また，表形式データをクエリに含むケースにおいて，提案手法はサブグラフマッチングを利用しない検索よりも検索時間を大幅に短縮できることを示す．

本稿の構成は次の通りである．まず，2 章で関連研究を示し，3 章にて事前知識を説明する．4 章に計算ノートブックの類似検索の問題定義および類似度の計算方法を，5 章にて提案する検索フレームワークとその素朴なアルゴリズムの問題点を示す．その問題点の対処として，6 章に提案する高速なアルゴリズムを示す．7 章にて実験結果を示し，8 章にて本稿をまとめる．

2 関連研究

計算ノートブックは，頻繁に利用・公開されている一方で，生産性の観点から問題性が指摘されている [2]．特に重要な問題として，実行の再現性の低さが挙げられる．これは，同一計算ノートブックであってもセルの実行順で結果が異なる上，セルの再実行と同時にそのセルの識別子も変わってしまい実行順を追うことができないためである [6]．この問題を解決するために，計算ノートブックのカーネルを含むシステムの拡張 [7] や，独自の識別子の利用などによりセル間の依存関係を明確に保持する手法 [6]，計算ノートブックのセルや変数に関してワークフロー化することで実行結果の来歴を追う [8] などの対処法が研究されている．これらは，計算ノートブックをグラフで表現することによって，セルの実行順序を捉えたまま保持することの有用性を示している．

計算ノートブックでは，Pandas DataFrame をはじめとする表形式データが多く取り扱われており，計算ノートブック上の表形式データに着目した技術が活発に開発されている．Zhang らは，計算ノートブックに含まれる表形式データの高速な検索技術 Juneau を提案した [5, 9]．さらに，表形式データへの操作の研究として， Yan らは，データサイエンティストがどのようにデータの加工や変換などの操作を行なっているかを学習し，データへの最適な操作を自動提案するシステムを開発した [1]．また，Jupyter Notebook の対話性を向上させる研究として， Chen らは Pandas DataFrame の実行を最適化することで Jupyter Notebook セルを実行してから結果が出るまでの時間を高速化した [10]．計算ノートブックの類似検索においても，表形式データを検索内容に含むことは適当である．上記の技術において，Juneau [5, 9] は，表形式データの類似性の評価が可能であるが，ソースコードや出力などの他の計算ノートブックの内容に関する検索には適応していない．そのため，表形式データの検索技術を計算ノートブックの高速検索に適用することはできない．

ソースコードの類似性についてはソフトウェア工学の分野で盛んに研究されており，ソースコードを有向グラフに変換して類似サブグラフの探索に基づいた類似ソースコード検出技術 [3]，コードクローン検出技術 [4] などが研究されている．これらの技術はソースコードのみに特化しており，表形式データやライブラリ，セルの出力を含めた複合的な類似度計算や，類似部分の関係性・実行順の認識ができないため，計算ノートブックの類似検索には適していない．

3 事前知識

本章では計算ノートブックについて説明する．計算ノートブックは，セルという数行のソースコードのまとまりで構成され，ソースコードの実行はセルごとに行われる．セルはひとつずつ順に実行されていき，同時に実行されることはない．また，ライブラリは通常のソースコードと同様に “import” でインポート操作でき，“from” を使用すると関数や変数，クラスなどを指定した操作ができる．計算ノートブックで一度インポート操作をしたライブラリは，他のセルの実行にも利用できる．各セルでは DataFrame で与えられる表形式データの読み込みやデータクリーニングなどの操作が行われる場合がある．さらに，各セルでは matplotlib や seaborn といったライブラリを利用した図示や，DataFrame の内容および文字列を出力することができる．

In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import plotly.express as px
import plotly.graph_objects as go
df = pd.read_csv("input/indian_food.csv")
```

In [2]:
```
df=df.replace(-1,np.nan)
df=df.replace('-1',np.nan)
```

In [3]:
```
Veg = (df['diet'].str[0].str.lower() == 'v').sum()

NonVeg = (df['diet'].str[0].str.lower() == 'n').sum()

names = ["Vegetarian", "Non-Vegetarian"]
values = [Veg, NonVeg]

fig, axs = plt.subplots(1, 1, figsize=(8, 6), sharey=True)
axs.bar(names, values)
```

Out[3]:

```
<BarContainer object of 2 artists>
```

図 1　計算ノートブックの例

上記に基づき，計算ノートブックをその構成要素の組み合わせとして以下のように定義する．

定義 1 (計算ノートブック)　セルを $c = (\mathcal{S}, \mathcal{D}, \mathcal{O})$ とする．$\mathcal{S}$ は c のソースコードを表し，$\mathcal{S} \neq \phi$ である．$\mathcal{D}$ は c に含まれる表形式データの集合を表す．$\mathcal{O}$ は c に含まれる出力形式の多重集合を表す．c の集合を $\mathcal{C}$ とし，計算ノートブックを組 $(\mathcal{L}, \mathcal{C})$ で定義する．$\mathcal{L}$ はインポートされているライブラリ名の集合を表す．

計算ノートブックの例を図 1 に示す．それぞれの四角はセルを表し，セル毎にソースコードを持っている．1 つ目のセルでは，ライブラリのインポートに加えて DataFrame 型の変数を読み込み，2 つ目のセルはその変数に操作を行っている．3 つ目のセルは変数の内容を文字列で出力している．

4 計算ノートブックの類似検索

本章では，本稿で取り組む計算ノートブックの類似検索に関して問題定義および類似度計算の説明を行う．類似検索では，ソースコード，表形式データ，ライブラリ，セルの出力の形式を入力とし，類似性が上位 k 個の計算ノートブックを出力する．類似性が高い計算ノートブックは，ソースコード，表形式データ，使用するライブラリがそれぞれ類似しており，セルの出力の形式が多く一致しているものとする．さらに，ソースコードの実行順や，出力順序などの順序も類似している．本研究では，セルの順序に基づいた検索のためにサブグラフマッチングを用い，サブグラフマッチングの結果を利用して各要素および計算ノートブックの類似性を計算する．以降，それぞれについて説明する．

本稿で提案する検索フレームワークでは，計算ノートブックにおける処理順や表形式データの出力順序を保持しつつ類似度計算コストを削減するために，検索対象の計算ノートブックおよびクエリを DAG で表現する．これをそれぞれワークフローグラフとクエリグラフと呼ぶ．以下で詳細を定義する．

定義 2 (ワークフローグラフ) 計算ノートブックのワークフローを DAG $W = (V_w, E_w, L_w, A_w, \mathcal{L}_w)$ として表現し，W をワークフローグラフと呼ぶ．V_w はノードの集合，$E_w \subseteq V_w \times V_w$ は $v \in V_w$ から $v' \in V_w$ へ向かう有向辺 $e(v, v')$ の集合，L_w は各ノードからラベル集合へのマッピング，A_w は各ノードから属性集合へのマッピング，$\mathcal{L}_w$ はライブラリ集合である．$v \in V_w$ に対して $L_w(v)$ および $A_w(v)$ はそれぞれ v のラベルと属性である．ワークフローグラフのノードラベルとして，セルのソースコードに対応するノードはラベル l_S，表形式データに対応するノードはラベル l_D，セルの出力に対応するノードはラベル l_O とする．したがって $L_w : V_w \to \{l_S, l_D, l_O\}$ である．ノード v はラベル $L_w(v)$ に対応して属性 $A_w(v)$ を持つ．$L_w(v)$ が l_S, l_O，および l_D の場合，$A_w(v)$ はそれぞれ出力形式の集合，ソースコード，表形式データとなる．

なお，計算ノートブック 1 つは，ワークフローグラフ 1 つに対応する．

検索内容の順序を捉えた類似度計算のために，クエリも同様に DAG で表現し，クエリとワークフローグラフ間でサブグラフマッチングを利用する．クエリに対応するグラフ（クエリグラフ）を以下のように定義する．

定義 3 (クエリグラフ) クエリグラフ Q はワークフローグラフと同様の 5 つ組 $(V_q, E_q, L_q, A_q, \mathcal{L}_q)$ とする．クエリグラフのノードラベルは $\{l_S, l_D, l_O, *\}$ とする．$*$ は，到達可能性のラベルを示し，節点間にパスがあることを表す．ただし，隣接するノード $v_i, v_j \in V_q$, $e(v_i, v_j) \in E_q$ に対し，$L_q(v_i) = L_q(v_j) = *$ であるならば，$v_i = v_j$ とみなせるため，ラベルが $*$ のノードは隣接しないものとする．

ワークフローグラフと異なり，クエリグラフは到達可能性をラベルとして含むことができる．これにより，要素間の関係性をより柔軟に指定可能とする．

順序を捉えた検索のために，検索の際にサブグラフマッチングを利用する．ここで，検索対象である計算ノートブックのワークフローグラフは通常の DAG であるが，検索の入力であるクエリグラフは到達可能性を含む．本研究が対象とするサブグラフマッチングを以下で定義する．

定義 4 (サブグラフマッチング) ワークフローグラフ $W = (V_w, E_w, L_w, A_w, \mathcal{L}_w)$ に対して，クエリグラフ $Q = (V_q, E_q, L_q, A_q, \mathcal{L}_q)$ がマッチしているとは以下の 3 つの条件を満たすことである．

（*1*） $v \in V_q$ のラベル $L_q(v) \neq *$ ならば，v はある一つのノード $u \in V_w$ とマッピング M で対応し，$M(v) = u$ である．

（*2*） $v \in V_q$ のラベル $L_q(v) \neq *$ ならば，$L_q(v) = L_w(M(v))$ である．

（*3*） $v_i, v_j \in V_q$ 間に辺 $e(v_i, v_j) \in E_q$ が存在し，$L(v_i) \neq *, L(v_j) \neq *$ のとき，$\exists e; e(M(v_i), M(v_j)) \in E_w$.

（*4*） $v_i, v_*, v_j \in V_q$ において，辺 $e(v_i, v_*)$ および $e(v_*, v_j)$ が存在し，$L(v_i) \neq *, L(v_j) \neq *, L(v_*) = *$ のとき，$M(v_i) \rightsquigarrow M(v_j)$．ここで，$\rightsquigarrow$ は到達可能を表す．

このとき，マッチした部分は W のサブグラフ $W' \subseteq W$ であり，W および Q が与えられた時に全てのマッチするサブグラフを検出することをサブグラフマッチングと呼ぶ．また，このとき検出された各サブグラフに対するノードのマッピング $M : V_q \to V_w$ の集合を $\mathbb{M}$ とする．

次に，クエリ Q とワークフローグラフ W の類似度について述べる．Q と W の類似度 $Sim(Q, W)$ の計算は 3 つのステップからなる．まず，V_q と $M(V_q)$ の対応ノード間（つまり，v と $M(v)$）の属性の関連度 $Rel_l\,(v, M(v))$ およびライブラリの類似度を計算し，Q とサブグラフの類似度 $Sim(Q, W, M)$ を計算，最後に Q と W の類似度を計算する．詳細を以下で定義する．

定義 5 (関連度) クエリ Q とワークフローグラフ W 間で同じラベルのノード $v, M(v)$,$v \in V_q$, $M \in \mathbb{M}$ 間の類似性のスコアを関連度 $Rel_{L_q(v)}\,(v, M(v))$，ライブラリについての類似性のスコアを関連度 $Rel_l\,(Q, W)$ と定義する．関連度の値の範囲は $[0, 1]$ とし，値が大きいほど類似しているとする．

また，どのような場合に類似しているかを以下に示すが，各関連度計算の具体的な計算方法は問わない．

- ソースコードが類似しているとは，比較するソースコードの内容が同様の操作を同様の手順で行なっているということである．これは，ソースコードの文字列の比較や，変数間の操作関係を DAG で表現し編集距離で比較するなどによって，類似度を計算する．ソースコードには，ライブラリのインポートを行うための文字列が含まれる場合もある．
- 表形式データが類似しているとは，同様の値で構成される列を多数共有しているということである．列単位の類似度に基づいて，表の総合的な類似度を計算する．
- 使用ライブラリが類似しているとは，指定したライブラリ名の集合に対し，計算ノートブック全体でインポートしたライブラリ名の集合が同様のもので構成されているということである．
- 出力形式が一致しているとは，指定した出力形式に対し，計算ノートブックのあるセルの出力形式が一致しているということである．“DataFrame” は DataFrame の内容の出力，“png” は matplotlib や seaborn といったライブラリなどによるグラフや図の出力，“text” は文字列の出力である．

上記の関連度計算に基づき，ワークフローグラフを利用した計算ノートブックの類似度を以下で定義する．

定義 6 (計算ノートブックの類似度) クエリ Q に対して，W

のサブグラフマッチング結果のそれぞれの類似度 $Sim(Q, W, M)$, $M \in \mathbb{M}$ は,

$$Sim(Q, W, M) = \sum_{v \in V_q} w_{L_q(v)} Rel_{L_q(v)}(v, M(v)) + w_l Rel_l(Q, W) \quad (1)$$

とする．ただし，$w_{L_q(v)} \in \{w_{l_S}, w_{l_D}, w_{l_O}\}$ であり，w_{l_S}, w_{l_D}, w_{l_O}, w_l はそれぞれラベル l_S, l_D, l_O のノードおよびライブラリの関連度に対応した任意の重みとする．また，サブグラフマッチングの結果のサブグラフが複数存在する場合は，サブグラフの類似度のうち最も高い類似度をそのワークフローグラフ W の類似度 $Sim(Q, W)$ とし，

$$Sim(Q, W) = \max_{M \in \mathbb{M}} Sim(Q, W, M) \quad (2)$$

とする．ただし，$\mathbb{M} = \phi$ の場合，$Sim(Q, W) = 0$ とする．

以上より，ワークフローグラフを利用した計算ノートブックの類似検索について，問題定義を以下とする．

問題定義 （計算ノートブックの類似検索） 検索対象となる計算ノートブックの集合，検索クエリ Q，および関連度の重み w_{l_S}, w_{l_D}, w_{l_O}, w_l が与えられた時，クエリとの類似度 $Sim(Q, W)$ が高い上位 k 個の計算ノートブックを抽出する．

5 類似検索フレームワーク

本章では提案する類似検索フレームワークについて説明する．フレームワークの全体像を図 2 に示す．検索フレームワークでは，クエリグラフ $Q = (V_q, E_q, L_q, A_q, \mathcal{L}_q)$，および各関連度の重み w_{l_S}, w_{l_D}, w_{l_O}, w_l を入力とする．クエリグラフに対してデータベースに格納されているワークフローグラフの類似度を計算し，出力は類似度 $Sim(Q, W)$ が高い上位 k 個の計算ノートブックである．フレームワークの検索部分は大きく二つのコンポーネントで構成されており，一つ目はサブグラフマッチング，二つ目は類似度計算である．

5.1 節にワークフローグラフの構築方法を，5.2 節に類似検索フレームワークを簡単に実装した手法と，検索速度の観点から素朴な手法の問題点を示す．次章の 6 章に素朴な手法の問題点に対応するため高速化の工夫を組み込んだ手法のアルゴリズム（高速な手法）を示す．

5.1 ワークフローグラフの構築

計算ノートブック $(\mathcal{L}, \mathcal{C})$ からワークフローグラフ $W = (V_w, E_w, L_w, A_w, \mathcal{L}_w)$ への変換について説明する．全てのセル c の $\mathcal{S}$，および集合 $\mathcal{D}$, $\mathcal{O}$ の全ての要素をそれぞれをノードとするノード集合 V_w を構成し，ノードラベルはそれぞれ l_S, l_D, l_O とする．ノード属性 $A_w(v)$ はノード $v \in V_w$ に対応する $\mathcal{S}$, $\mathcal{D}$, $\mathcal{O}$ の要素とする．$\mathcal{L}_w$ は $\mathcal{L}$ とする．以下で辺の張り方を説明する．i 番目に実行されるセルを c_i とし，c_i のコード $\mathcal{S}$ に対応するノードを $v_i \in V_w$ $(L_w(v_i) = l_S)$ とする．ラベル l_S のノードは直列に繋いだ直鎖のグラフを構成し，

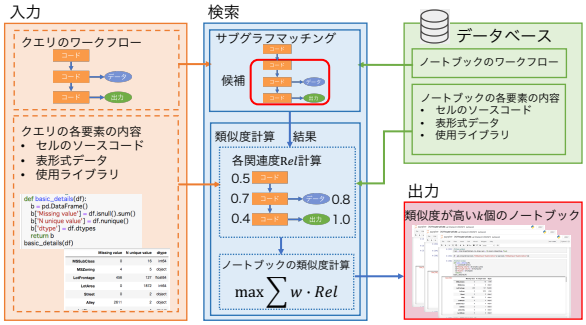

図 2 類似検索フレームワーク

$e(v_{i-1}, v_i) \in E_w$, $i \geqq 2$ を満たすように辺を張る．ラベル l_D のノード $v_D \in V_w$ $(L_w(v_D) = l_D)$ は，そのデータが v_i に対応するセルのソースコードによって変数に格納されるならば，辺 $e(v_i, v_D)$ を張る．さらに，v_D に対応する変数を利用するセルのコードが存在するならば，そのコードに対応するノード $v_j \in V_w$ $(L_w(v_j) = l_S, i < j)$ に対して，辺 $e(v_D, v_j)$ を張る．ラベル O のノード $v_O \in V_w$ $(L_w(v_O) = l_O)$ は，その出力を行ったセルが i 番目に実行されているならば，辺 $e(v_i, v_O)$ を張る．

5.2 素朴な手法

類似検索の単純な手法は以下の流れで行う．

（1） データベースの計算ノートブックのワークフローグラフ W に対して Q のサブグラフマッチングを行い，その結果全てを列挙する．

（2） Q と W のライブラリの関連度 $Rel_l(Q, W)$ を計算する．それぞれの $M \in \mathbb{M}$ に対してノードの関連度 $Rel_{L_q(v)}(v, M(v))$, $\forall v \in V_q$ を計算し，さらに類似度 $Sim(Q, W, M)$ を求める．全ての類似度を計算したら，ワークフローグラフの類似度 $Sim(Q, W)$ を求める．

（3） 全てのワークフローグラフの類似度を求めるまで上記を繰り返す．その後，$Sim(Q, W)$ で降順ソートし，$Sim(Q, W)$ が高い上位 k 個のワークフローグラフに対応する計算ノートブックを検索結果として返す．

素朴な手法は，明らかに Top-k に入らない場合でも計算を行い，計算コストの大小にかかわらず全ての関連度計算するため非効率である．さらに，同じ内容の類似度計算を複数回行い，明らかにマッチしないワークフローグラフに対してもサブグラフマッチングを実行するという無駄な計算を含んでいる．

6 検索の高速化

本章では，5.2 項を踏まえ，より高速に検索を行う手法について説明する．素朴な手法では，検索結果を求めるために，全てのワークフローグラフに対して，全ての要素の関連度計算を行っている．そのため，ワークフローグラフの一部の関連度しか計算していなくても，Top-k に入らないことが明らかである場合に，計算の枝刈りをする．次に，表形式データの関連度計

算は他の関連度計算に比べて計算コストが大きい．そのため，表形式データの関連度計算以外の関連度を先に計算し，表形式データの関連度計算の前に枝刈りが実行できるように計算順序を最適化する．さらに，マッピングの一部が同一である $M \in \mathbb{M}$ が複数存在する場合に，関連度計算結果をキャッシュしておくことにより重複した計算を削減する．最後に，サブグラフマッチングにおいて，W は Q よりノード数が少ない，あるいは辺の最大入出次数が小さい場合，明らかに W は Q にマッチしない．そのため，ワークフローグラフの構造情報をインデックス化し，サブグラフマッチングの枝刈りを行う．

6.1 節に検索時に行われる計算に関する高速化および 6.2 節にインデックス作成を示す．

6.1 検索の実行

素朴な手法では，サブグラフマッチングの結果を全て列挙した後，列挙した結果ごとに関連度・類似度を求め，類似度に基づいて検索結果となる計算ノートブックを特定する．ここでは，関連度計算・類似度計算のステップで，候補空間の枝刈りを行うことで，5.2 節で示した計算の非効率性を解消する．

6.1.1 計算済み類似度との比較による枝刈り

計算済みの類似度と比較し，検索結果に含まれない場合を早期に特定することによって，枝刈りを可能にする．ある $M \in \mathbb{M}$ に対し，$Sim\,(Q,W,M)$ がとりうる値の最大値を $MaxSim\,(Q,W,M)$ とする．この値と上位 k 番目の計算済み類似度，および同一ワークフローグラフの計算済み類似度の比較を行うことで，検索結果に含まれるか判定する．そのために，まず $MaxSim\,(Q,W,M)$ を求める．

すでにいくつかのノードの関連度 $Rel_{L_q(v)}\,(v, M(v))$, $\forall v \in V'_q \subseteq V_q$ およびライブラリ関連度 $Rel_l\,(Q,W)$ が計算されているとする．また，関連度未計算の $u \in V_q \setminus V'_q$ に対しラベル $L_q(u)$ の関連度がとりうる値の最大値を $MaxRel_{L_q(u)}$ とする．4 章で示すように，関連度の計算は各ノードの種類の平均と重み和であるので，

$$\begin{aligned} &MaxSim\,(Q,W,M) \\ &\quad = w_l Rel_l\,(Q,W) + \sum_{v \in V'_q} w_{L_q(v)} Rel_{L_q(v)}\,(v, M(v)) \\ &\qquad\qquad + \sum_{u \in V_q \setminus V'_q} w_{L_q(u)} MaxRel_{L_q(u)} \end{aligned} \tag{3}$$

となる．

次に，上位 k 番目の計算済類似度との比較を行い，枝刈りをする．すでに k 個以上のワークフローグラフの類似度が計算されているとき，k 番目に高い類似度を Sim_k とすると，もし

$$Sim_k > MaxSim\,(Q,W,M) \tag{4}$$

が成り立つならば，明らかにこの類似度 $Sim\,(Q,W,M)$ は Top-k の検索結果に入らない．

さらに，同一ワークフローグラフの計算済み類似度との比較による枝刈りをする．類似度計算済みのマッピング集合を $\mathbb{M}' \subseteq \mathbb{M}$ とすると，

$$\max_{M' \in \mathbb{M}'} Sim\,(Q,W,M') > MaxSim\,(Q,W,M) \tag{5}$$

が成り立つならば，式 (2) より，明らかにこの類似度 $Sim\,(Q,W,M)$ はワークフローグラフ W の類似度 $Sim(Q,W)$ になり得ない．

これらにより，構成する全てのノードに関して関連度を求めなくても，明らかに Top-k に入らない，またはそのワークフローグラフにおける最大の値にならない類似度計算を枝刈りできる．

6.1.2 関連度計算順序の最適化

関連度計算順序を最適化することによって，6.1.1 項の枝刈りの効果を向上させる．データの関連度は計算コストが大きいことを踏まえ，できるだけデータ関連度を計算せずに候補空間や計算の枝刈りをするために，検索の実行時に関連度計算の優先順位によって計算順序の最適化を行う．ここで，ワークフローグラフにおけるサブグラフマッチングの計算コストは，検索全体に対しては非常に小さいことが明らかとなっている（7 章を参照）．したがって，先にサブグラフマッチングを行うことによって計算対象のノードを枝刈りする．その後の各ノードの関連度およびライブラリの関連度の計算順序は，以下のようになる．まず，最も計算コストが大きいもの以外の関連度を先に計算し，ワークフローグラフ W とマッピング $M \in \mathbb{M}$ の組み合わせ (W,M) ごとに暫定の類似度を求める．次に類似度に基づいて (W,M) を降順ソートする．最後に，ソートの順に残りの関連度を計算しながら，逐次 6.1.1 項の枝刈りを行う．

どの関連度計算コストが大きいかにより最適化後の順序は異なるが，例として今回の実験設定と同様の計算コストが条件である場合を考える．この場合，サブグラフマッチング，ライブラリ関連度，出力関連度の計算コストが非常に小さく，コード関連度の計算コストが次いで小さく，表形式データ関連度の計算コストが非常に大きい．計算順序を最適化すると，まずサブグラフマッチングを行なった後，全ての $M \in \mathbb{M}$ に対し関連度およびその重み和 $w_{l_O} Rel_{l_O}(v, M(v)) + w_l Rel_l(Q,W) + w_{l_S} Rel_{l_S}(v, M(v))$ を計算し，その値で降順ソートし，組み合わせ (W,M) を並べる．そして，それぞれの (W,M) に対してソートした順序で $Rel_{l_D}(v, M(v))$ を計算する．ここで，1 つの v に対して $Rel_{l_D}(v, M(v))$ を計算するたびに，6.1.1 項の枝刈りを行う．また，もし表形式データ関連度の重み $w_{l_D} = 0$ である場合は，残りのうち最も計算時間がかかるのはコード関連度 $Rel_{l_S}(v, M(v))$ であるので，コード関連度以外を計算し，その後コード関連度を計算する順序にする．

6.1.3 計算済み類似度のキャッシング

一度計算したノード間の関連度の値を保持しておく．そして，その後の計算中に同一ノード間の関連度の値が必要となった場合に活用する．これにより，同一ノード間の関連度計算を複数回計算してしまう無駄な計算を削減する．

6.2 インデックスの構築

ワークフローグラフの構造情報を利用することで候補空間の枝刈りを行う．W のノード数を $size(V)$，辺の最大入次数を $d_{in}(W)$，最大出次数を $d_{out}(W)$ とする．W に対し Q のサブ

Algorithm 1 高速な手法のアルゴリズム

Input: Query graph Q, set of workflow graphs $\mathbb{W}$, index $\mathbb{I}$
Output: k workflow graphs with the k highest $Sim(Q, W)$

1: **foreach** $W \in \mathbb{W}$ **do**
2: **if** Q has no matching subgraph in M based on $\mathbb{I}$ **then**
3: $\mathbb{M}[W] \leftarrow \phi$
4: **else**
5: $\mathbb{M}[W] \leftarrow$ Subgraph Matching(Q, M)
6: **end if**
7: $Sim(Q, W) \leftarrow 0$
8: $Rel_l(Q, W) \leftarrow$calculating $Rel_l(Q, W)$
9: **foreach** $M \in \mathbb{M}[W]$ **do**
10: $Sim(Q, W, M) \leftarrow w_l Rel_l(Q, W)$
11: **foreach** $v \in V_q$ **do**
12: **if** $L_g(v)! = l_D$ **then**
13: $Rel_{L_q(v)}(v, M(v)) \leftarrow$ calculating $Rel_{L_q(v)}(v, M(v))$
14: $Sim(Q, W, M) \leftarrow Sim(Q, W, M) + w_{L_q(v)} Rel_{L_q(v)}(v, M(v))$
15: **end if**
16: **end for**
17: **end for**
18: **end for**
19: **foreach** $W \in \mathbb{W}$ **do**
20: **foreach** $M \in \mathbb{M}[W]$ **do**
21: **foreach** $v \in V_q$ **do**
22: **if** $L_g(v) == l_D$ **then**
23: $Rel_{l_D}(v, M(v)) \leftarrow$ calculating $Rel_{l_D}(v, M(v))$
24: $Sim(Q, W, M) \leftarrow Sim(Q, W, M) + w_{L_D} Rel_{l_D}(v, M(v))$
25: $MaxSim\,(Q, W, M) \leftarrow$ calculating $MaxSim\,(Q, W, M)$
26: $Sim_{cf} \leftarrow \min\,(Sim_k, Sim(Q, W))$
27: **if** $MaxSim\,(Q, W, M) < Sim_{cf}$ **then**
28: break
29: **end if**
30: **end if**
31: **end for**
32: **if** $Sim(Q, W) < Sim(Q, W, M)$ **then**
33: $Sim(Q, W) \leftarrow Sim(Q, W, M)$
34: **end if**
35: $\mathbb{W} \leftarrow$ Sorting $\mathbb{W}$ by $Sim(Q, W)$
36: $Sim_k \leftarrow Sim(Q, \mathbb{W}[k-1])$
37: **end for**
38: **end for**
39: Return $\mathbb{W}[0...k-1]$

グラフマッチングを行う際，以下のいずれかが成り立つとき，明らかにマッチしない．

- $V_1 \subseteq V_q, V_2 \subseteq V_w\ s.t.\ \forall v \in V_1, \forall u \in V_2, L_q(v) = L_w(u)$
- $d_{in}(W) < d_{in}(Q)$
- $d_{out}(W) < d_{out}(Q)$

これに対し，ワークフローグラフのラベルごとのノード数，ワークフローグラフごとの辺の最大入出次数をあらかじめ保持しておくことによって，明らかにマッチしないワークフローグラフの枝刈りを行う．

6.3 アルゴリズム

アルゴリズム 1 に高速なアルゴリズムの擬似コードを示す．全体の構成は 6.1.2 項に基づき，1-18 行目でサブグラフマッチングと表形式データ以外の関連度を計算し，19-38 行目で枝刈りをしながら表形式データの関連度計算とワークフローグラフの類似度計算をする．最後の 39 行目で上位 k 個の計算ノートブックを返す．$\mathbb{M}[W]$ は，注目しているワークフローグラフ W とクエリ Q のマッピングを表す．2 行目で 6.2 節のインデックスを利用して，マッチするサブグラフが存在しない場合を枝刈りする．7, 10 行目で類似度の初期化をする．13, 23 行目の $Rel(v, M(v))$ の計算では，6.1.3 項に基づき，すでに計算したことがある関連度の場合は保持してある値を利用する．25-29 行目で 6.1.1 項の枝刈りを行い，比較対象の類似度は 32, 33 行目で式 (2) を逐次行うことで $Sim(Q, W)$ を，35, 36 行目で Sim_k を求める．

7 実　　験

本章では，評価実験について説明する．7.1 節で実験環境を示し，7.2 節で実データから作成したデータセットについて説明する．7.3 節に実験に利用した各関連度の計算方法を示し，7.4 節に比較手法を示す．7.5 節で検索精度と検索効率の実験結果を示す．

7.1 実 験 環 境

実験は 2.30GHz Intel Core i5，16GB メモリを搭載した Mac OS Mojave 10.14.6 サーバ上で実行し，全てのアルゴリズムは Python を利用して実装・実行した．ワークフローグラフの保存には Neo4J を利用し，セルのソースコードや表形式データ，ライブラリの保存には PostgreSQL を利用した．

7.2 データセット

データセットとして， Kaggle で共有されている 111 個の計算ノートブックを使用する．計算ノートブックの目的は機械学習やデータ分析，データの視覚化など多岐に渡る．Kaggle で共有されている計算ノートブックは，上から順に実行すると同一の結果を得られるものが多いため，セルの実行順は上から順であるとしてワークフローグラフへの変換を行なった．さらに，実際に全てのセルを上から順に実行することで，各セルで取り扱われた表形式データの内容を取得し，データベースに保存した．データセットの計算ノートブックに関する各種統計については，表 1，2，および 3 にそれぞれライブラリ数を含むワークフローグラフの統計，セルのソースコードの統計，および表形式データの統計を示す．

7.3 各類似度計算

実験における各要素の関連度計算方法について説明する．

ソースコード: セルのソースコードの文字列を区切って単語の集合にする．区切りは，スペース・改行・ピリオド・イコールとする．ある 2 つのセルの類似度は，その変換で得られた単語の集合に対する Jaccard 係数をコードの類似度とする．**表形式データ**: 比較する 2 つの表形式データ T_A と T_B を列に分解し，列毎のユニークな値の集合をそれぞれ col_A と col_B とする．T_A の列数を s とし，T_B の列数以下とする．col_A と col_B の類似度は Jaccard 係数とし，$Jacc(col_A, col_B)$ と表記する．$Jacc(col_A, col_B)$ が高い順から，同一の列が含まれないように s 個を選択する．組み合わせの対応関係を単射 $M : col_A \rightarrow col_B$ とし，表の類似度を $\frac{1}{s}\sum_{col_A \in T_A} Jacc(col_A, M(col_A))$ とする．**ライブラリ**: 計算ノートブックで使用しているライブラリ名の単語集合に対する Jaccard 係数を類似度とした．ただし，ライブラリのインポートに文字列 from がある場合は， from の直後の部分がライブラリ名であるとした．**出力形式**: {DataFrame, text, png} の一致で類似度を定義する．一致していれば類似度は 1.0，一致していなければ類似度は 0.0 とする．

表 1　計算ノートブックのワークフローグラフの統計

要素	最大数	最小数	平均	合計
l_S ノード数	123	5	29.92	3321
l_D ノード数	44	0	8.86	983
l_O ノード数	107	0	26.59	2952
辺数	282	12	73.10	8114
最大入次数	8	1	2.71	-
最大出次数	21	2	4.97	-
ライブラリ数	22	2	6.86	762
DataFrame 出力数	43	1	4.28	475
png 出力数	46	1	8.79	976
text 出力数	47	1	13.52	1501

表 2　計算ノートブックのセルのソースコードの行数の統計

最大	最小	セルあたり	計算ノートブックあたり	合計
9043	1	232.52	48261.63	772186

表 3　計算ノートブックの表形式データの統計

最大 (MB)	最小 (Byte)	平均 (MB)	合計 (MB)
248.7	144	85.9	9538.9

7.4 比較手法

比較手法としては，ワークフローグラフを利用する検索手法として，5.2 節で示した素朴な手法 (naive)，素朴な手法に対して 6 章で示した高速化の手法を加えた提案手法 (proposal) とする．また，ワークフローグラフを利用しない検索手法として，表形式データのみの類似度に基づき検索する手法 (D)，計算ノートブック内のセルのソースコードを全結合した文字列に対して Jaccard 係数に基づくソースコード類似度のみを検索に利用した手法 (C) とする．さらに，サブグラフマッチングを行わず単に関連度の和による類似度計算によって検索する手法 (C+D+L+O) とする．これは，サブグラフマッチングによるノード間順序の考慮を行わず，クエリグラフのノードそれぞれと関連度が高いノードの組み合わせをマッピングとし，ソースコードの関連度は C と同様の計算を行う検索方法である．さらに，その手法に 6.1.1 項に示した計算済み類似度との比較による枝刈り，6.1.2 項で示した類似度計算順の最適化を同様に適用したもの (C+D+L+O (fast)) とする．これらの手法で検索時間および出力結果を比較する．

7.5 実験結果

本節で提案手法の検索精度と検索効率の評価実験結果を示す．

7.5.1 検索精度

本項でユーザ実験による検索精度の実験結果について示す．評価対象は proposal, C+D+L+O, D，および C とする．各手法の検索結果に対し，どの程度クエリと類似しているかを人手評価することで，検索精度を評価する．ラベル $*$ を含む 3 つのクエリに対し，検索対象をそれぞれ人手評価された 15 個の計算ノートブックに限定してクエリを検索し，結果の順位の比較を行った．人手評価では 7 人のユーザによって，検索対象の計算ノートブックを，クエリとコード，データ，出力，それらの実行順，ライブラリの類似性，および総合的な類似性についてそれぞれ 1-5 点で評価した．評価には人手評価の各点数の平均値を計算ノートブックのスコアとし，$nDCG$ を利用する [11]．

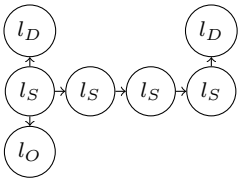

図 3　クエリグラフ

図 4 に結果を示す．proposal は， C+D+L+O と同等かつ，D および C よりも高い精度である．この結果は，計算ノートブックの複数の要素を組み合わせることで，目的に沿った計算ノートブックが検索できることを示している．さらに，proposal は C+D+L+O と同等の精度であり，計算ノートブックの各要素の順序や関係性の類似性も人手評価の基準の一つであるため，提案手法はそれらの情報を保持できていることを示している．

7.5.2 検索時間

本項で検索時間の実験結果について示す．入力には，ワークフローグラフ図 3 に示す同一の形であるが，各ノードの属性が異なるクエリを 6 つ使用する．クエリ 1, 2, 3 は，データセット内の 3 つの計算ノートブックの一部を切り出してワークフローグラフに変換したものとする．クエリ 4 はクエリ 1 の，クエリ 5 はクエリ 2 の，クエリ 6 はクエリ 3 の一部のノードの内容ををを入れ替えたものとした．

Top-10 検索を実行した検索時間の結果を図 5 に示す．naive に対する proposal および C+D+L+O に対する C+D+L+O (fast) は，高速化を適用することで検索時間を短縮している．ここで，いずれのクエリにおいても，ワークフローグラフを利用する naive は，利用しない C+D+L+O より計算時間が短縮している．これは，ワークフローグラフを利用する場合はサブグラフマッチングの結果に含まれるノードしか関連度の計算対象でないので，計算コストの大きい表形式データの関連度計算の回数が削減されたためである．

また，k の値を変えた検索時間の結果を図 6 に示す．ここで，高速化の工夫を入れた場合，すなわち naive に対する proposal, C+D+L+O に対する C+D+L+O (fast) を比較すると，ワークフローグラフを利用する検索の場合の方がより高速であり，特に k の値が大きくなっても高速性を保っている．これは，ワークフローグラフを利用する検索の場合の方が，枝刈りの可能性が大きいためである．6.1.1 項に示したように，ワークフローグラフ利用の場合の高速なアルゴリズムではデータ関連度を一つ計算するたびに， 上位 k 番目の類似度との比較だけでなく，同一ワークフローグラフの計算済み類似度と比較して枝刈りを行う．その一方で，ワークフローグラフ利用なしの場合は，関連度が最大が高いノードをマッピングとするため，その計算ノートブック内に含まれる表形式データ全てと関連度計算をしなければ計算ノートブックの類似度が計算できず，データ関連度計算中に枝刈りが発生しない．

マイクロベンチマークとして， proposal に対し 6.1.1 項で示した計算済み類似度との比較による枝刈り (PRuning;

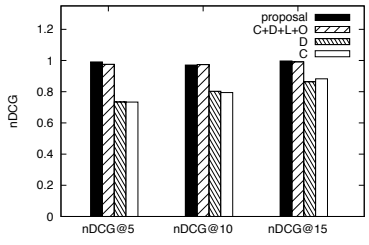

図 4　検 索 精 度

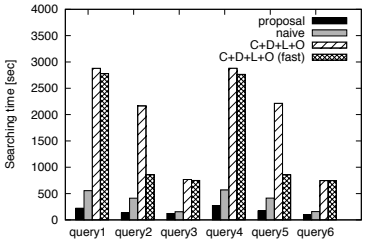

図 5　Top-10 検索の検索時間

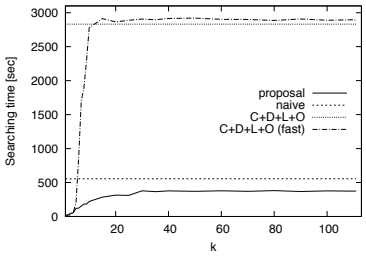

図 6　各 k での検索時間

表 4　各高速化技術の否適用による検索時間

Method	Searching time (sec)
PR + OPT + CA + IND (proposal)	208.66
OPT + CA + IND	506.62
PR + CA + IND	333.47
PR + OPT + IND	251.28
PR + OPT + CA	209.77
CA + IND	508.96
naive	554.81

表 5　検索時間に対する各関連度計算時間の割合 (%)

Method	Subgraph matching	Code	Table data	Output	Library
proposal	0.0230	0.5408	99.4287	0.0003	0.0022
naive	0.0088	0.2503	99.7372	0.0002	0.0014
C+D+L+O	-	0.0040	99.9325	0.0000	0.0001
C+D+L+O (fast)	-	0.0039	99.9371	0.0000	0.0001

PR)，6. 1. 2 項で示した関連度計算順序の最適化 (OPTimization; OPT)，6. 1. 3 項で示した計算済み関連度のキャッシング (CAche; CA)，6. 2 項で示したインデックスの構築 (INDexing; IND) の適用の有無で検索速度を比較する．

表 4 にマイクロベンチマークの結果，および表 5 にそれぞれの関連度計算時間の割合を示す．表 4 より，PR を用いない場合に検索時間が増加していることがわかる．そのため，計算時間の削減に最も効果が高いのは PR である．さらに OPT，CA を加えることによって，より高速化している．IND は計算時間削減の効果が少ない．これは表 5 に示すように，サブグラフマッチングの計算時間は検索時間に占める割合が小さく，IND による時間削減効果が検索時間全体に対して小さいためである．

8　おわりに

本稿ではソースコード，表形式データ，使用ライブラリ，セル出力を総合的に考慮した計算ノートブックの検索に取り組み，新たなフレームワークを提案した．検索フレームワークは，計算ノートブックのソースコードだけでなく，表形式データや使用ライブラリ，セルの出力形式をクエリとしている．検索時にはサブグラフマッチングを利用することで，セルの処理順序や計算ノートブックを構成する要素間の依存関係を考慮した高速な検索が可能である．実験では，検索に計算ノートブックの複数の要素の類似度を考慮することで高精度に検索できることを示した．また，検索対象の候補空間を早期に枝刈りすることで高速な検索を実現し，フレームワークの有効性を示した．

今後の展望としては，計算ノートブックを構成するソースコード，表形式データ，使用ライブラリ，セル出力の類似度計算の精度をさらに向上させ，より検索目的に沿った計算ノートブック検索の実現を目指す．そのために，複数のクエリに基づく高速な OR 検索の実現や，類似性が高い要素だけでなく，類似性が低いことを条件とした検索の実装などが考えられる．

謝　　辞

本研究は JSPS 科研費 JP20H00583 の支援によって行われた．ここに記して謝意を表す．

文　　献

[1] Cong Yan and Yeye He. Auto-suggest: Learning-to-recommend data preparation steps using data science notebooks. In *Proceedings of the ACM SIGMOD*, pp. 1539–1554, 2020.

[2] Souti Chattopadhyay, Ishita Prasad, Austin Z Henley, Anita Sarma, and Titus Barik. What's wrong with computational notebooks? pain points, needs, and design opportunities. In *Proceedings of the ACM CHI*, pp. 1–12, 2020.

[3] Jens Krinke. Identifying similar code with program dependence graphs. In *Proceedings of the WCRE*, pp. 301–309, 2001.

[4] Rainer Koschke, Raimar Falke, and Pierre Frenzel. Clone detection using abstract syntax suffix trees. In *Proceedings of the WCRE*, pp. 253–262, 2006.

[5] Yi Zhang and Zachary G Ives. Finding related tables in data lakes for interactive data science. In *Proceedings of the ACM SIGMOD*, pp. 1951–1966, 2020.

[6] David Koop and Jay Patel. Dataflow notebooks: Encoding and tracking dependencies of cells. In *Proceedings of the TaPP*, pp. 17–17, 2017.

[7] Tomas Petricek, James Geddes, and Charles Sutton. Wrattler: Reproducible, live and polyglot notebooks. In *Proceedings of the TaPP*, pp. 6–6, 2018.

[8] Lucas A. M. C. Carvalho, Regina Wang, Yolanda Gil, and Daniel Garijo. Niw: Converting notebooks into workflows to capture dataflow and provenance. In *Proceedings of the K-CAP*, pp. 12–16, 2017.

[9] Yi Zhang and Zachary G Ives. Juneau: data lake management for jupyter. *Proceedings of the VLDB Endowment*, Vol. 12, No. 12, pp. 1902–1905, 2019.

[10] Libo Chen, Hrishikesh Dharam, and Doris Xin. Optimizing dataframes for interactive workloads. *Proceedings of ACM Conference*, 2019.

[11] Chris Burges, Tal Shaked, Erin Renshaw, Ari Lazier, Matt Deeds, Nicole Hamilton, and Greg Hullender. Learning to rank using gradient descent. In *Proceedings of the ICML*, pp. 89–96, 2005.