Vol.2017-MPS-113 No.38
Vol.2017-BIO-50 No.38
2017/6/25

フラグメント伸長型化合物ドッキング計算のための重み付きオフラインキャッシュ問題の厳密解アルゴリズム

柳澤 渓甫[1,2,a)]　小峰 駿汰[1,2]　久保田 陸人[1]　大上 雅史[1,3]　秋山 泰[1,2,3]

概要：バーチャルスクリーニングにおける大規模なタンパク質-化合物ドッキング計算の高速化のために，化合物の部分構造であるフラグメントのドッキング計算結果を保存し，他の化合物の評価時に計算結果を再利用する方法が提案されている．しかし，従来提案されてきた手法はディスクアクセスを大量に発生させ，高速化率が十分ではなかった．さらなる高速化のためには，メモリ上に計算結果を保持してディスクアクセスを減らすことが考えらえるが，メモリ容量には上限があるため効率的な計算結果の保持を実現することが重要になる．本研究では，最適な計算結果の保持順番の導出を重み付きオフラインキャッシュ問題として定式化し，これを最小費用流問題に帰着させ，さらにこの帰着させたグラフの特徴を利用した高速な厳密解アルゴリズムを提案した．従来提案されていた非巡回有向グラフに対する最小費用流問題の厳密解アルゴリズムに比べて約 7 倍の高速化を達成した．

キーワード：バーチャルスクリーニング, タンパク質-化合物ドッキング, 重み付きオフラインキャッシュ問題, 最小費用流問題

An exact algorithm for the weighted offline cache problem in protein-ligand docking based on fragment extension

Keisuke Yanagisawa[1,2,a)]　Shunta Komine[1,2]　Rikuto Kubota[1]　Masahito Ohue[1,3]　Yutaka Akiyama[1,2,3]

Abstract: The need to accelerate large-scale protein-ligand docking in virtual screening against a huge compound database led researchers to propose a strategy that entails memorizing the evaluation result of the partial structure of a compound and reusing it to evaluate other compounds. However, the previous method required frequent disk accesses, resulting in insufficient acceleration. Thus, more efficient memory usage can be expected to lead to further acceleration, and optimal memory usage could be achieved by solving the weighted offline cache problem. In this research, we propose an exact algorithm for the weighted offline cache problem, which we reduce to the minimum cost flow problem, and utilize the characteristics of the graph generated for this problem as constraints. The proposed algorithm was shown to be approximately seven times faster compared to an existing exact algorithm specified for directed acyclic graphs.

Keywords: virtual screening, protein-ligand docking, weighted offline cache problem, minimum cost flow problem

1. はじめに

創薬研究では，まず大規模な化合物データベースから新薬の候補となる化合物を探索するが，これは「干草の山か

[1] 東京工業大学 情報理工学院 情報工学系
Department of Computer Science, School of Computing, Tokyo Institute of Technology
[2] 東京工業大学 情報生命博士教育院
Education Academy of Computational Life Sciences (ACLS), Tokyo Institute of Technology
[3] 東京工業大学 科学技術創成研究院 スマート創薬研究ユニット
Advanced Computational Drug Discovery Unit (ACDD), Institute of Innovative Research, Tokyo Institute of Technology
[a)] yanagisawa@bi.c.titech.ac.jp

© 2017 Information Processing Society of Japan

図 1　本研究の流れ

ら針を探す」("Finding a needle in a haystack") ようなものであるとたとえられており [1]，大量の化合物すべてについて *in vitro* 実験を行う前に計算機を用いてタンパク質との結合しやすさを評価して候補を絞り込むバーチャルスクリーニング (virtual screening, VS) が広く行われている [2]．特に，タンパク質や化合物の 3 次元構造情報に基づいた手法 (structure-based virtual screening, SBVS) は，タンパク質と化合物の物理化学的な相互作用を考慮することができ，既知の薬剤などを必要としないため注目されている．SBVS ではタンパク質や化合物の 3 次元構造情報が必要であるが，手法の広がりに従って立体構造データベースも年々拡充されている．例えば，タンパク質の 3 次元構造情報のデータベースである Protein Data Bank (PDB) には 2016 年末の時点で 12.5 万件余りの構造が登録されており，1 年あたり約 10%増加している [3]．化合物のデータベースについても，ZINC データベース [4] には約 3,400 万件の化合物が登録されているなど，大量のデータが取得可能である．

SBVS では，タンパク質-化合物ドッキング計算（以下，ドッキング計算）を用いてタンパク質と化合物の結合親和性を予測することが多い [5] が，最も利用されているドッキングツールの 1 つである AutoDock Vina[6] で 500 CPU コア秒/タンパク質-化合物ペアと，計算量がかなり大きい．この計算量では 1 つのタンパク質と ZINC 全体の約 3,400 万化合物とのドッキングには 500 CPU コア年以上もの計算時間を要するため [7]，ドッキング計算の高速化が不可欠である．高速化の方策として，化合物は多数の共通な部分構造を持つため [8]，化合物群を評価する上で発生する共通な計算をまとめることが考えられる．例えば，化合物を部分構造であるフラグメントに分割する eHiTS[9] や FlexX[10] などの手法では，フラグメントごとにドッキング計算を行った上で，その計算結果を利用して化合物を再構成する．この場合，複数の化合物が同一のフラグメントを持つのであれば，そのフラグメントのドッキング計算結果を再利用することが可能である．eHiTS ではフラグメントのドッキング計算結果を SQL に保存する機能が実装されており，数百化合物から数千化合物のドッキングについて 2–4 倍の高速化が達成されているが，1 万化合物程度に達すると高速化率が低下することが報告されている [9]．この速度低下は，化合物数が増えるに従い SQL データベースのサイズが肥大し，ディスクアクセスのコストが増大していることが原因として考えられる．SQL を利用する代わりにメモリ中にフラグメントのドッキング計算結果を保存することでさらなる高速化が見込まれるが，フラグメントのドッキング計算結果は 1 フラグメントあたり最大数百 MB の容量を必要とするため，有限のメモリ空間を効率的に利用するようなフラグメントのドッキング計算結果の保存を行う必要がある．本研究では，あらかじめ化合物やフラグメントの評価順番が決定されている仮定のもとで，フラグメントのドッキング計算結果の保存の最適化を行う．これは重み付きオフラインキャッシュ問題 (weighted offline cache problem) [13] として問題を定式化することができる．

重み付きオフラインキャッシュ問題とは，

- k 個までのページを保持できるキャッシュが存在し，
- ページの要求される順番があらかじめ判明しており，
- キャッシュに保持するためのコストがページによって異なる

という制約の中で，合計コストを最小化する問題である．ここで，フラグメントのドッキング計算結果がページに，メモリがキャッシュに対応する．重み付きオフラインキャッシュ問題は最小費用流問題に帰着できることが Lopez-Ortiz らによって証明されている [14]．最小費用流問題はグラフに編集を加えながら最短経路を導出し続ける Successive Shortest Path アルゴリズムが 1958 年に提案 [15], [16], [17], [18] されて以来，Push-relabel アルゴリズムの一般化でグラフのコストの最大値が計算量に影響する Cost-Scaling アルゴリズム [19] やシンプレックス法を用いた Network-Simplex アルゴリズム [20], [21], [22]，有向非巡回グラフに特化することで高速に厳密解を導出する Pirsiavash らの手法 [23] などが存在している．

重み付きオフラインキャッシュ問題から帰着された最小費用流問題のグラフは特殊な性質を有する．本原稿では，まずフラグメントのドッキング結果の必要な順序から最小費用流問題への帰着方法を述べた上でグラフの特殊性について論じ，この性質を利用した高速な最小費用流問題アルゴリズムを提案する．

2.　手法

2.1　ドッキング計算結果保存最適化の重み付きオフラインキャッシュ問題への定式化（図 2-(B)）

FlexX[10] のようなフラグメント伸長型の手法においては，最も大きなフラグメントを最初に配置し，配置したフ

© 2017 Information Processing Society of Japan

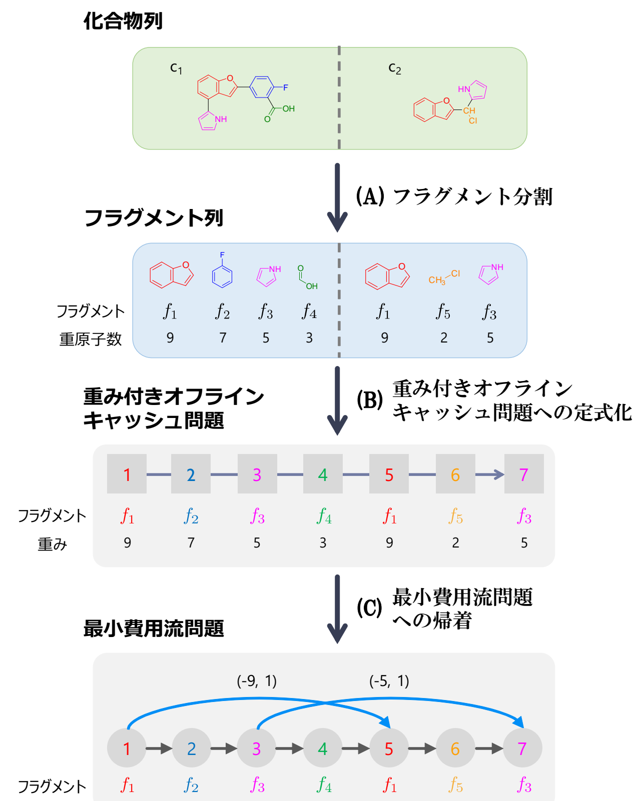


図 2　化合物のフラグメント分割から重み付きオフラインキャッシュ問題を経て，最小費用流問題へ帰着するまでの流れ

ラグメントに接続している最も大きなフラグメントを次に配置する，というように，化合物の構造からフラグメントのドッキング計算結果の必要な順序を一意に定めることができる．このため，フラグメントのドッキング計算結果をキャッシュ問題における 1 つのページに，また，メモリ中に保持できるドッキング計算結果の個数をキャッシュ幅 k に対応させ，各フラグメントのドッキング計算結果の作成コストを重みとすることで，重み付きオフラインキャッシュ問題として表現することができる．各フラグメントのドッキング計算結果の作成コストはフラグメントの持つ原子数に比例するためこれを適用した．

2.2　重み付きオフラインキャッシュ問題の最小費用流問題への帰着（図 2-(C)）

重み付きオフラインキャッシュ問題のクエリ列に対して以下の操作を行うことで，最適なドッキング結果の保存順序を導出できる最小費用流問題に帰着させることができる [14]．

(1) すべてのページクエリを節点とする．これらの節点はページクエリ順と同様に一列にならんでいる．
(2) すべてのページクエリについて，直後のページクエリに容量 ∞，コスト 0 の辺を作成する（図 3 の灰色の矢印に相当する）．
(3) あるページクエリについて，1 つ後の同じフラグメントのドッキング計算結果を要求するページクエリに辺を張る．このとき，容量は 1，辺のコストはページクエリのコストの正負を逆転させた値とする（図 3 の青色の矢印に相当する）．
(4) キャッシュ幅 k を最小費用流問題の総流量とする．

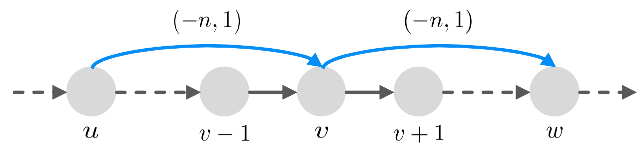


図 3　帰着によって生成されたグラフ中のノード．u, w はそれぞれ v と同じフラグメントのドッキング計算結果を要求するノードであり，これは存在しないこともある．

2.3　作成されるグラフの特徴

重み付きオフラインキャッシュ問題から生成された最小費用流問題のグラフ $G(V, E)$ はノード V が 1 列に並んでおり，1 つのノード v は，図 3 のように高々 4 つのエッジが関係する．したがって，作成されるグラフは以下のような特徴が存在する．

- 有向非巡回グラフである（巡回路が存在しない）
- 一意にトポロジカルソートが可能である（ノードを一意に，1 列に並べることが可能である）
- 疎グラフである（1 つのノードは最大 4 つのエッジしか持たないため，$O(|E|) = O(|V|)$ である）
- すべてのエッジは流量が 1 か ∞ である．

以上のグラフの特徴を考慮し，これまで提案されてきたアルゴリズムより高速なアルゴリズムの提案を行う．

2.4　提案手法：最小費用流問題の厳密解アルゴリズム

提案手法を Algorithm 1 に示す．提案手法は Successive Shortest Path（以下 SSP）に基づいている．SSP は総流量 k を流しきるまで，ShortestPath 関数によってグラフ（残余ネットワークと呼ぶ）の始点 v_1 から終点 v_n への最小コスト路を見つけ，その経路の容量の最大の流れ（フロー）を追加し，残余ネットワークを更新する．

始点から終点への最小コスト路の探索を実際に行っているのは CalcCosts 関数であり，これは v_1 から各ノードへの最小コストを動的計画法で算出する．CalcCosts 関数の動作例を図 4 に示す．

2.5　最適解が得られることの証明

SSP の枠組みに基づくアルゴリズムが厳密解を得ることは，ShortestPath 関数内の CalcCosts 関数が最小コスト路を発見できることを証明することで保証される．したがって，本節では CalcCosts 関数が最小コスト路を発見できることを証明する．

© 2017 Information Processing Society of Japan

Algorithm 1 提案手法

1: **function** SUCCESSIVESHORTESTPATH(G, k)
　　▷ G：グラフ，k：総流量
2: 　$f_0 \leftarrow \phi$ 　▷ f：フロー
3: 　$G_{f_0} \leftarrow G$ 　▷ G_f：フローが f のときの残余ネットワーク
4: 　$i \leftarrow 0$
5: 　**while** $|f_i| \neq d$ **do**
6: 　　$i \leftarrow i+1$
7: 　　$P_i \leftarrow$ SHORTESTPATH($G_{f_{i-1}}$)
8: 　　$f_i, G_{f_i} \leftarrow$ AUGMENTEDFLOW($f_{i-1}, G_{f_{i-1}}, P_i$)
　　▷ f_{i-1} に P_i を追加し，残余ネットワークを更新する関数
9: 　**end while**
10: 　**return** f_i
11: **end function**

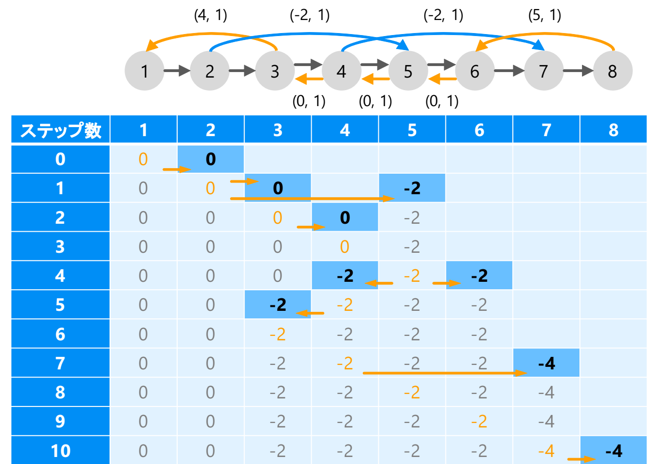
12: **function** SHORTESTPATH(G)
　　▷ 始点，終点はそれぞれ v_1, v_n とする
13: 　$prev \leftarrow$ CALCCOSTS(G)
14: 　$path \leftarrow$ GENERATEPATH($prev$)
　　▷ v_1 から v_n への最小コスト路を生成する関数
15: 　**return** $path$
16: **end function**

17: **function** CALCCOSTS(G)
18: 　$cost \leftarrow \{cost(i) = 0 | 1 \leq i \leq n\}$
19: 　$prev \leftarrow \{prev(i) = \phi | 1 \leq i \leq n\}$
20: 　$now \leftarrow 1$
21: 　**while** $now < n$ **do**
22: 　　$updated \leftarrow \phi$
23: 　　**for all** $\{e | e \in G.E, e.\mathrm{from} = now\}$ **do**
　　▷ e.from：エッジ e の始点
　　▷ e.to：エッジ e の終点
　　▷ e.cost：エッジ e のコスト
24: 　　　**if** $cost(e.\mathrm{to}) > cost(now) + e.\mathrm{cost}$ **then**
25: 　　　　$cost(e.\mathrm{to}) \leftarrow cost(now) + e.\mathrm{cost}$
26: 　　　　$prev(e.\mathrm{to}) \leftarrow now$
27: 　　　　$updated.\mathrm{add}(e.\mathrm{to})$
28: 　　　**end if**
29: 　　**end for**
30: 　　$next \leftarrow \min(updated, now + 1)$
31: 　**end while**
32: 　**return** $prev$
33: **end function**

補題 1. CALCCOSTS 関数において，最後に $now = i$ の処理が完了した時刻を t_i，時間 t の時の $cost(v)$ の値を $cost[t][v]$ と定義する．エッジの集合 E にエッジ (i, j) が含まれるならば，以下の不等式が成り立つ．

$$cost[t][j] \leq cost[t][i] + (i, j).\mathrm{cost} \ \textbf{if} \ t \geq t_i$$

証明． Algorithm 1 の 23–28 行目の操作により，ノード v_i から接続されているすべてのノードは $now = i$ の計算中に必ず更新される．従って，$cost[t_i][j] \leq cost[t_i][i] + (i, j).\mathrm{cost}$ が成立する．時刻 $t \geq t_i$ のとき，コストは必ず小さくなるように更新されるため $cost[t][i] = cost[t_i][i], cost[t][j] \leq cost[t_i][j]$ が成立する．以上より，補題 1 の不等式が成立する． □



ステップ数	1	2	3	4	5	6	7	8
0	0	0						
1	0	0	0		-2			
2	0	0	0	0	-2			
3	0	0	0	0	-2			
4	0	0	0	-2	-2	-2		
5	0	0	-2	-2	-2	-2		
6	0	0	-2	-2	-2	-2		
7	0	0	-2	-2	-2	-2	-4	
8	0	0	-2	-2	-2	-2	-4	
9	0	0	-2	-2	-2	-2	-4	
10	0	0	-2	-2	-2	-2	-4	-4

図 4 CALCCOSTS の実行例．辺にはそれぞれ (コスト, 容量) が記載されており，特に記載の無い辺は $(0,\infty)$ であり，表の i 行 j 列目は i 回目の更新時における $cost(j)$ の値を表している．各ステップで更新されたセルを水色の背景で示す．

補題 2. CALCCOSTS 関数は最小コスト路を発見できる．

証明． 背理法によって証明する．すなわち，$cost[t_n][n] > \hat{d}_n$ を仮定する．ここで $\hat{d}_n$ は v_1 から v_n までの真の最小コストを示す．

$C(i) \equiv cost[t_n][i] - \hat{d}_i$ と定義すると，$C(n) > 0, C(1) = 0$ であることから，真の最小コスト路を含む任意の v_1 から v_n に至る経路中に最低 1 つ以上 $(p, q) \in E, C(p) = 0, C(q) > 0$ なるノード p, q が存在する．ここで，真の最小コスト路に含まれるエッジ (p', q') について

$$\begin{aligned} cost[t_n][q'] &> cost[t_n][p'] + \hat{d}'_q - \hat{d}'_p \\ &= cost[t_n][p'] + (p', q').\mathrm{cost} \end{aligned}$$

となるはずだが，これは補題 1 と矛盾する．

$\hat{d}_n \leq cost[t_n][n]$ は自明に成立することから，$cost[t_n][n] = \hat{d}_n$ が導出される．すなわち，CALCCOSTS 関数は最小コスト路を見つけることができる． □

3. 評価実験

3.1 データセット

本研究では 2 つのデータセットを用いて手法の性能を評価した．

- 実データ：ドッキング計算に基づいたデータ
FlexX[10] のような木探索に基づくフラグメント伸長型ドッキング計算でのフラグメントのドッキング計算結果の再利用を想定し，ZINC データベース [4] に登録されている化合物を用いてクエリ列を生成した．
  (1) ZINC Purchasable サブセット（2014-11-28 版，22,724,825 化合物）から，化合物数 $= \{10000, 100000, 1000000\}$ をランダムサンプリング

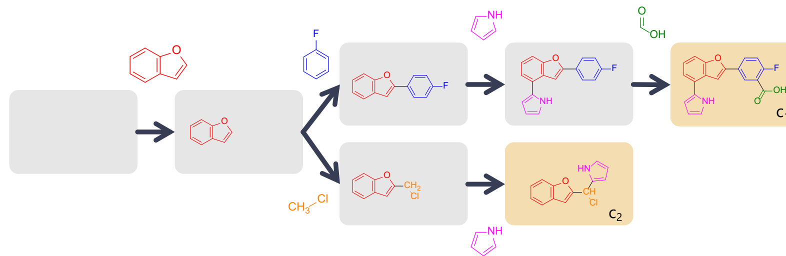


図 5　フラグメント伸長計算の途中結果の再利用．赤色で示したフラグメントは化合物 c_1, c_2 に共通であるため，フラグメント伸長計算の途中結果を再利用することが可能である．

表 1　LEMON に実装された既存手法の実データに対する計算時間．フローの総流量は $k = 100$ である．

化合物数	ページ数 (n)	Cost-Scaling	Capacity-Scaling	Network-Simplex	Cycle-Canceling
	272,851	25,263 ms	67,219 ms	404,497 ms	3,047,511 ms
10,000	273,095	26,853 ms	67,251 ms	402,680 ms	3,118,584 ms
	274,127	26,650 ms	65,219 ms	404,778 ms	2,823,840 ms
	2,327,638	455,921 ms	1,741,264 ms	$>$ 3 h	$>$ 3 h
100,000	2,332,426	452,442 ms	1,727,584 ms	$>$ 3 h	$>$ 3 h
	2,343,338	445,881 ms	1,991,428 ms	$>$ 3 h	$>$ 3 h
	18,663,564	$>$ 3 h	$>$ 3 h	$>$ 3 h	$>$ 3 h
1,000,000	18,663,905	$>$ 3 h	$>$ 3 h	$>$ 3 h	$>$ 3 h
	18,698,105	$>$ 3 h	$>$ 3 h	$>$ 3 h	$>$ 3 h

(2) それぞれの化合物を回転可能な結合で分割し [8]，フラグメントにする

(3) 木探索の再利用を想定し，化合物の部分構造でソートを行い，直前の化合物と重複している部分構造までを構成するフラグメントのデータ取得クエリを削除する（図 5）

(4) 残ったフラグメント列をクエリ列とする

化合物数ごとに 3 回ずつのランダムサンプリングを行い，データセットを作成した．化合物によってフラグメント数が異なりクエリ列長が変化するため，計算結果を示す際にはクエリ列長も記載する．

- 人工データ：様々なパラメータを考慮したランダム生成データ
  重み付きオフラインキャッシュ問題にはクエリ数，ページの重み，ページの種類数の 3 つの要素があるが，化合物分割によって生成されたグラフではこれらを独立に変化させることはできない．提案手法の性能をさらに詳細に見るため，以下のパラメータに沿ってデータを生成した．
  - クエリ数：$\{10000, 100000, 1000000\}$
  - ページの最大重み：100，それぞれのページの重み：一様分布で決定
  - ページの種類数：$\{1000, 10000, \ldots,$ クエリ数 $\}$

3.2　比較手法

ここでは，グラフライブラリ LEMON に実装された 4 種類の汎用的な厳密解アルゴリズム，および非巡回有向グラフに特化した厳密アルゴリズム，合わせて 5 つの既存アルゴリズムを示す．また，LEMON に実装された 4 種類のアルゴリズムについての比較を行い，4 章のベースラインとする手法を決定する．

- LEMON[24], [25]
  LEMON (Library for Efficient Modeling and Optimization in Networks) は，C++で記述された，グラフやネットワーク構造に関するデータ構造およびアルゴリズムの実装がまとめられたライブラリである．最小費用流問題の最適解導出については Cost-Scaling, Capacity-Scaling, Network-Simplex, Cycle-Canceling の 4 つのアルゴリズムを選択することができる．これら 4 つのアルゴリズムで実データの厳密解を導出するまでに要した計算時間は表 1 の通りであり，Cost-scaling が最も高速であった．このため，Cost-scaling をベースラインの手法として 4 章の実験結果に示す．
- Pirsiavash らの手法 [23]
  最小費用流を求めるグラフについて非巡回有向グラフであるという条件を付けることで，より高速な計算を可能にした手法である．このアルゴリズムも SSP の枠組みに基づいた手法であり，各ステップでは Dijkstra 法を用いて最小コスト路を求めている．

3.3　計算機環境

本研究では，東京工業大学の TSUBAME 2.5 の Thin

© 2017 Information Processing Society of Japan

表 2　実データに対する計算時間．フローの総流量は $k = 100$ である．また，3 時間経過しても終了しなかったケースについては > 3 h と記載しており，括弧内は Pirsiavash らの手法に対する提案手法の高速化率を示す．

化合物数	ページ数 (n)	提案手法	Pirsiavash+, 2012	Cost-Scaling
10,000	272,851	723 ms (×7.97)	5,761 ms	25,263 ms
	273,095	681 ms (×8.42)	5,731 ms	26,853 ms
	274,127	688 ms (×8.36)	5,755 ms	26,650 ms
100,000	2,327,638	6,702 ms (×7.80)	52,281 ms	455,921 ms
	2,332,426	6,670 ms (×7.69)	51,314 ms	452,442 ms
	2,343,338	6,732 ms (×7.83)	52,685 ms	445,881 ms
1,000,000	18,663,564	55,354 ms (×7.81)	432,360 ms	> 3 h
	18,663,905	55,151 ms (×7.75)	427,696 ms	> 3 h
	18,698,105	62,283 ms (×7.48)	466,169 ms	> 3 h

表 3　人工データに対する計算時間．フローの総流量は $k = 100$ である．括弧内は Pirsiavash らの手法に対する提案手法の高速化率を示す．

ページ数 (n)	ページ種類数	提案手法	Pirsiavash+, 2012	Cost-Scaling
10,000	1,000	60 ms (×2.97)	178 ms	361 ms
100,000	1,000	632 ms (×3.90)	2,467 ms	6,576 ms
	10,000	462 ms (×6.07)	2,804 ms	9,195 ms
1,000,000	1,000	6,521 ms (×4.24)	27,645 ms	293,918 ms
	10,000	4,913 ms (×7.08)	34,767 ms	167,089 ms
	100,000	5,640 ms (×6.70)	37,763 ms	178,613 ms

ノードを利用した．Thin ノードには 6 コアの Intel Xeon X5670 CPU が 2 個，メモリが 54 GB 搭載されている．すべての手法は並列化されておらず，1 CPU コアを用いた場合の速度計測を行う．

4. 実験結果

総流量 $k = 100$ についての実験結果を表 2 および表 3 に示す．フラグメントのドッキング計算結果は 1 フラグメントあたり数百 MB のメモリを消費するため，$k = 100$ は数十 GB のメモリ空間を計算結果の保存のために用意することに対応する．

4.1 実データにおける提案手法と既存手法の比較

実データに対して提案手法，Pirsiavash らの手法，および Cost-scaling 法をそれぞれ用いた計算時間を表 2 に示す．この結果より，提案手法は Pirsiavash らの手法に比べて 7.5–7.8 倍，Cost-Scaling 法に比べて 200 倍以上の高速化を達成していることがわかる．また，この高速化率は化合物数やランダムサンプリングの試行に依存しておらず，安定して提案手法が高速であることがわかる．

4.2 人工データにおける比較

提案手法の性能をより詳細に評価するためにパラメータを変化させて生成した人工データに対する手法の計算時間の比較を表 3 に示す．提案手法は依然として既存手法に比べて高速である一方，高速化率が $n = 1{,}000{,}000$ のケースで 4.2–7.1 倍程度と多少低下していることがわかる．

5. 考察

5.1 実データと人工データにおける，提案手法の高速化率の違い

4.1 節および 4.2 節で述べたように，提案手法の高速化率は実データで 7.5–7.8 倍，人工データで 4.2–7.1 倍と差が発生した．この差が発生した原因として，ページクエリ列を構成するページの種類の偏りが考えられる．人工データの生成では各クエリのページの種類に一様分布を仮定しているが，実データではごく少数のフラグメントが多くの化合物に出現する一方で，大部分のフラグメントはまれにしか出現しない，などといったフラグメントの出現頻度の偏りが存在している．頻出するフラグメントを長期間保持すると最小費用流問題におけるコストを大幅に削減することができるため，そのエッジが選択されやすく，後退距離の短いエッジが残余ネットワークに生成される傾向にある．人工データではページの存在率に一様分布を仮定しているため，実データよりも後退距離の長いエッジが残余ネットワークに生成されており，計算量が比較的大きくなりやすいと考えられる．

5.2 計算量の推定

前節で述べたように，提案手法の計算量は CalcCosts 関数中の while ループ回数に依存し，先に進むときは常に 1 つずつ進むことからノード列を後退するような更新の「総後退距離」に依存することが Algorithm 1 からわかる．後退するようなエッジは残余ネットワークが更新されるにつ

表 4 比較手法の最悪計算量および提案手法の推定される最悪計算量．本研究で対象としているグラフの特性に合わせ，$O(|E|) = O(|V|) = O(n)$, 辺の最大コスト $= C$, 頂点の最大流量 $= U$ として記述している．

アルゴリズム	適用可能なグラフ	最悪計算量
Cost-Scaling[19]	すべてのグラフ	$O(n^3 \log(nC))$
Network-Simplex[21]	すべてのグラフ	$O(n^3 CU)$
Capacity-Scaling[26]	すべてのグラフ	$O(n^2 \log U \cdot \log n)$
Cycle-Canceling[27]	すべてのグラフ	$O(n^3 CU)$
Pirsiavash+, 2011[23]	非巡回有向グラフ	$O(Un \log n)$
提案手法	トポロジカルソート可能，疎グラフ	未導出, $O(Un \log n)$?

れて増えるため，フローの総流量 k とページ数 n に依存するはずである．最小費用流の厳密解の導出全体にわたっての最悪ケースとなるようなグラフを作ることができれば最悪計算量を求めることができるが，これまでのところ最悪ケースのグラフを導出するに至っていない．これからの課題として，最悪計算量および平均計算量を求めることが必要であると考えられる．

5 つの既存手法および提案手法の最悪計算量を表 4 にまとめる．表 2 の結果より，提案手法と Pirsiavash らによる手法の計算速度比はほぼ一定であり，このことから Pirsiavash らの手法と同等の最悪計算量である可能性が示唆される．

5.3 ステップ数の削減の検討

提案したアルゴリズムは Algorithm 1 の 30 行目で示したように，現在注目しているノード i の次のノード $(i+1)$ に関して，最後にノード $(i+1)$ の更新が行われてからの変化がある/ないに関わらず *cost* の更新を行うようになっているが，変化がないノードについて再度更新を行うのは無駄な計算である．これについて，更新すべきノードを優先度付きキューで管理し，番号の小さなノードから更新していくことで無駄な計算を行わないようにすることが可能である．しかし，優先度付きキューは enqueue や dequeue に $O(\log n)$ の時間を必要とし，かえって計算時間が増大してしまうという結果が得られたため，本提案手法は 1 つずつ節点を進むという方策をとった．

6. 結論

本研究では，重み付きオフラインキャッシュ問題に対する高速なアルゴリズムの提案を行い，従来提案されていた手法よりも約 7 倍高速に最適解を導出することができた．応用の一例として，eHiTS や FlexX のようなフラグメント伸長型ドッキング計算におけるフラグメントのドッキング計算結果の再利用を示した．しかしこれは，計算に必要なデータが一度にはメモリ上に乗らない場合に共通して発生する問題であり，ゲノムアセンブリや，GWAS 解析，メタゲノム解析など，保持するデータサイズの大きなバイオインフォマティクスの分野において幅広く利用することが可能な手法である．

今後の課題としては，計算量の詳細な解析および，この手法を用いて計算結果の再利用を行ったタンパク質-化合物ドッキング計算の実装・評価があげられる．

謝辞 本研究の一部は文部科学省 博士課程教育リーディングプログラム 東京工業大学「情報生命博士教育院」，JST CREST 「EBD: 次世代の年ヨッタバイト処理に向けたエクストリームビッグデータの基盤技術」(JPMJCR1303)，JST 研究成果展開事業 世界に誇る地域発研究開発・実証拠点（リサーチコンプレックス）推進プログラム「世界に誇る社会システムと技術の革新で新産業を創る Wellbeing Research Campus “Tonomachi”」，および JSPS 科研費 基盤研究 (B) (17H01814, 15H02776) の支援を受けて行われた．

参考文献

[1] Klon A.E. *et al.*: “Finding more needles in the haystack: a simple and efficient method for improving high-throughput docking results”, *J. Med. Chem.* 47, 2743–2749, 2004.
[2] Sliwoski G. *et al.*: “Computational methods in drug discovery”, *Pharmacol. Rev.* 66, 334–395, 2014.
[3] Rose P.W. *et al.*: “The RCSB Protein Data Bank: views of structural biology for basic and applied research and education”, *Nucleic Acids Res.* 43, D345–D356, 2015.
[4] Irwin J.J. *et al.*: “ZINC: a free tool to discover chemistry for biology”, *J. Chem. Inf. Model.* 52, 1757–1768, 2012.
[5] Meng X. *et al.*: “Molecular docking: a powerful approach for structure-based drug discovery”, *Curr. Comput. Aided Drug Des.* 7, 146–157, 2011.
[6] Trott O. *et al.*: “AutoDock Vina: improving the speed and accuracy of docking with a new scoring function, efficient optimization, and multithreading”, *J. Comput. Chem.* 31, 455–461, 2010.
[7] Drwal M.N. *et al.*: “Combination of ligand- and structure-based methods in virtual screening”, *Drug Discov. Today Technol.* 10, e395–e401, 2013.
[8] Yanagisawa K. *et al.*: “Spresso: An ultrafast compound pre-screening method based on compound decomposition”, *Bioinformatics*, doi: 10.1093/bioinformatics/btx178, 2017. (Epub ahead of print)
[9] Zsoldos Z. *et al.*: “eHiTS: A new fast, exhaustive flexible ligand docking system”, *J. Mol. Graph. Model.* 26, 198–212, 2007.

© 2017 Information Processing Society of Japan

[10] Rarey M. *et al.*: "A Fast Flexible Docking Method using an Incremental Construction Algorithm", *J. Mol. Biol.* 261, 470–489, 1996.

[11] Morris G.M. *et al.*: "AutoDock4 and AutoDockTools4: Automated docking with selective receptor flexibility", *J. Comput. Chem.* 30, 2785–2791, 2009.

[12] Friesner R.A. *et al.*: "Glide: A New Approach for Rapid, Accurate Docking and Scoring. 1. Method and Assessment of Docking Accuracy", *J. Med. Chem.* 47, 1739–1749, 2004.

[13] Buchbinder N. & Naor J.S.: "The Design of Competitive Online Algorithms via a Primal Dual Approach", *Found. Trends Theor. Comput. Sci.* 3, 93–263, 2009.

[14] Lopez-Ortiz A. & Salinger A.: "Minimizing Cache Usage in Paging", In *Proc. of WAOA2012* 145–158, 2013.

[15] Jewell W.S.: "Optimal flow through networks" *Interim Tech. Rep. No.8*, Operations Research Center, MIT, Cambridge, MA, 1958.

[16] Jewell W.S.: "Optimal flow through networks with gains", *Oper. Res.* 10, 476–499, 1962.

[17] Busacker R.G. & Gowen P.: "A procedure for determining a family of minimum-cost network flow patterns" *Interim Tech. Rep. No.15*, Operations Research Center, The Johns Hopkins University, Bethesda, MD, 1960.

[18] Iri M.: "A new method for solving transportation-network problems", *J. Oper. Res. Soc. Jpn.* 3, 27–87, 1960.

[19] Gabow H.N. & Tarjan R.E.: "Faster Scaling Algorithms for Network", *SIAM J.Comput.* 18, 1013–1036, 1989.

[20] Dantzig G.B.: "Linear Programming and Extensions" Princeton Univ. Press, West Sussex, England, 1963.

[21] Kelly D.J. & O'Neill G.M.: "The minimum cost flow problem and the Network-Simplex method", Master's thesis, University College, Dublin, Ireland, 1991.

[22] Orlin J.B.: "A polynomical time primal Network-Simplex algorithm for minimum cost flows", *Math. Program.* 78, 109–129, 1997.

[23] Pirsiavash H. *et al.*: "Globally-optimal greedy algorithms for tracking a variable number of objects", In *Proc. of 2011 IEEE Conf. CVPR* 1201–1208, 2011.

[24] Dezso B. *et al.*: "LEMON - an Open Source C++ Graph Template Library", *Electron. Notes Theor. Comput. Sci.* 264, 23–45, 2011.

[25] Kiraly Z. & Kovacs P.: "Efficient implementations of minimum-cost flow algorithms", *Acta Univ. Sapientiae Informatica* 4, 67–118, 2012.

[26] Edmonds J. & Karp R.M.: "Theoretical improvements in algorithmic efficiency for network flow problems", *J. ACM* 19, 248–264, 1972.

[27] Klein M.: "A primal method for minimal cost flows with applications to the assignment and transportation problems", *Manag. Sci.* 14, 205–220, 1967.

© 2017 Information Processing Society of Japan