The Thirty-Second AAAI Conference
on Artificial Intelligence (AAAI-18)

Accelerated Best-First Search with Upper-Bound Computation for Submodular Function Maximization

Shinsaku Sakaue
NTT Communication Science Laboratories
sakaue.shinsaku@lab.ntt.co.jp

Masakazu Ishihata
Hokkaido University
ishihata.masakazu@ist.hokudai.ac.jp

Abstract

Submodular maximization continues to be an attractive subject of study thanks to its applicability to many real-world problems. Although greedy-based methods are guaranteed to find $(1-1/e)$-approximate solutions for monotone submodular maximization, many applications require solutions with better approximation guarantees; moreover, it is desirable to be able to control the trade-off between the computation time and approximation guarantee. Given this background, the *best-first search* (BFS) has been recently studied as a promising approach. However, existing BFS-based methods for submodular maximization sometimes suffer excessive computation cost since their *heuristic functions* are not well designed. In this paper, we propose an accelerated BFS for monotone submodular maximization with a knapsack constraint. The acceleration is attained by introducing a new termination condition and developing a novel method for computing an upper-bound of the optimal value for submodular maximization, which enables us to use a better heuristic function. Experiments show that our accelerated BFS is far more efficient in terms of both time and space complexities than existing methods.

Introduction

Submodular maximization has attracted much attention thanks to its applicability to various problems: document summarization (Lin and Bilmes 2010), sensor placement (Krause, Singh, and Guestrin 2008), influence maximization (Alon, Gamzu, and Tennenholtz 2012), and so on. We let $U := [n]$ be a finite set, where $[n] := \{1, \dots, n\}$. Set function $g : 2^U \to \mathbb{R}$ is said to be *submodular* if $g(X) + g(Y) \geq g(X \cup Y) + g(X \cap Y)$ for any $X, Y \subseteq U$ and *monotone* if $g(Y) \geq g(X)$ for any $X \subseteq Y \subseteq U$.

In this paper, we consider monotone submodular maximization problems with a knapsack constraint, which we call submodular knapsack problems (SKPs). Let $c_v \geq 0$ be cost values for all $v \in U$; we define $c(X) := \sum_{v \in X} c_v$ and $c_X := \{c_v : v \in X\}$ for any $X \subseteq U$. Given monotone submodular function g that satisfies $g(\emptyset) = 0$ and a budget $B > 0$, we address the following SKP:

$$\text{(1)} \qquad \underset{X \subseteq U}{\text{maximize}}\ g(X) \quad \text{subject to } c(X) \leq B.$$

In what follows, X^* denotes an optimal solution for SKP (1).

For SKPs, the greedy algorithm is known to achieve $(1-1/\sqrt{e})$-approximation (Lin and Bilmes 2010); we say solution X (or an algorithm that returns X) is α-optimal if X is an α-approximate solution, i.e., $g(X) \geq \alpha g(X^*)$. Furthermore, $(1-1/e)$-optimal solutions can be obtained by executing the greedy algorithm $O(|U|^3)$ times (Sviridenko 2004). On the other hand, it is known that $(1-1/e+\varepsilon)$-approximation guarantees cannot be obtained in polynomial time unless P = NP (Feige 1998).

Although the aforementioned greedy algorithms are often effective, their approximation guarantees are sometimes unsatisfiable. In many important decision problems, we need to find solutions whose approximation guarantee is better than $1-1/\sqrt{e} \approx 0.39$ or $1-1/e \approx 0.63$. The *best-first search* (BFS), a general framework that includes the well-known A* search, is one of the most promising approaches that can achieve better approximation guarantees. The BFS-based approach for submodular maximization was established in (Chen, Chen, and Weinberger 2015); their original paper call the algorithm *filtered search* (FS). Although the time and space complexities of BFS are generally exponential in $|U|$, α-approximate solutions can be found for any $\alpha \in [0, 1]$, and we can control the trade-off between the computation cost and optimality by tuning the hyper-parameter, α. As in the standard A* search, BFS uses *heuristic function* $h(X)$ to measure how promising the current solution X is, and the performance of BFS strongly depends on how we design the heuristic function $h(\cdot)$. Unfortunately, as we will see in the experiments, BFSs with existing heuristic functions sometimes work poorly in important SKP instances.

Our Contribution

In this paper, we propose an accelerated BFS for SKPs. The acceleration is attained by introducing a new termination condition to BFS and using a novel heuristic function. As we will see later, computing heuristic function values reduces to computing upper-bounds for optimal values of SKPs that appear as subproblems in BFS. Therefore, we propose a novel technique to compute such upper-bounds. The obtained upper-bound is empirically more accurate than those obtained by existing techniques, thus enabling us to use a better heuristic function in BFS. Combining the upper-bound with the proposed termination condition substantially reduces the computation cost of BFS; our accelerated BFS is partic-

ularly effective when computing α-approximate solutions for $\alpha < 1$ (e.g., $\alpha = 0.7$ or 0.8). Experiments show that our algorithm outperforms existing methods in terms of both time and space complexities.

We note that obtaining accurate upper-bounds for SKPs is also beneficial in the empirical performance analysis of existing approximation algorithms for SKPs. Since SKP is NP-hard in general, computing optimal values for large SKPs is prohibitively expensive, making it almost impossible to evaluate the quality of obtained solutions for large SKPs. As shown later, our upper-bound computation method is as fast as the standard greedy algorithm, and it finds an upper-bound that is close to the optimal value. Thus the proposed upper-bound enables us to obtain empirical approximation guarantees that are typically better than $1 - 1/e$, which is helpful when evaluating the quality of given solutions for large SKPs.

Related Work

For size-constrained monotone submodular maximization (i.e., $c_v = 1$ for all $v \in U$), the greedy algorithm has been proved to be $(1 - 1/e)$-optimal (Nemhauser, Wolsey, and Fisher 1978). However, no polynomial-time algorithm is $(1-1/e+\varepsilon)$-optimal in the worst case unless P = NP (Feige 1998). If the *curvature* value $\kappa \in [0, 1]$ of a submodular function is small, which means the submodular function is close to a modular function, then improved approximation guarantees that depend on κ can be obtained (Conforti and Cornuéjols 1984; Sviridenko, Vondrák, and Ward 2015).

SKP is a more general problem and includes the size-constrained monotone submodular maximization. For a special case of SKP, the greedy algorithm has been proved to be $(1-1/\sqrt{e})$-optimal (Khuller, Moss, and Naor 1999), and this result has been extended to the general SKP (Lin and Bilmes 2010). If we are allowed to execute the greedy algorithm $O(|U|^3)$ times, then $(1 - 1/e)$-approximate solutions can be obtained for SKP (Sviridenko 2004); unfortunately, this is too computationally expensive in most applications.

When it comes to obtaining approximation guarantees that are better than $1 - 1/e$, there are two major approaches: integer programming (IP) and BFS. The IP-based approach was first studied by (Nemhauser and Wolsey 1981), who proposed a *branch-and-bound* algorithm. A more efficient IP-based algorithm using the *submodularity cut* is proposed in (Kawahara et al. 2009); although this method is applicable to non-monotone submodular maximization, it can only deal with the size-constrained case. Unfortunately, IP-based methods are sometimes inefficient since they must solve subproblems too many times. The BFS-based methods are typically more efficient than the IP-based ones by virtue of their search strategy based on certain priorities, and so are attracting much attention recently. For submodular maximization with a specific objective function and an s-t path constraint, an A* search algorithm is proposed in (Zeng et al. 2015); their algorithm employs the greedy algorithm for computing heuristic function values. For some submodular maximization problems including SKP, a BFS-based algorithm, called filtered search (FS), was proposed in (Chen, Chen, and Weinberger 2015); in the FS for SKP, heuristic function values are computed by solving relaxed knapsack problems (KPs). FS has a hyper-parameter, $\alpha \in [0, 1]$, that a user can control, and FS is guaranteed to find α-approximate solutions; this technique is studied as the weighted A* search in the field of search problems (Pohl 1970; Ebendt and Drechsler 2009).

BFS for SKPs and Acceleration Techniques

We explain some basics of BFS for SKP (1) and additional techniques for its acceleration: *under estimation* and *termination condition*. The detail of BFS with these techniques is shown in Algorithm 1. We also provide a theoretical analysis on the proposed algorithm.

In what follows, we define $X+Y := X \cup Y$ and $X-Y := X \backslash Y$ for any $X, Y \subseteq U$. We abuse notation and sometimes regard $v \in U$ as a subset of U; for instance, we let $X + v = X \cup \{v\}$. We also define $g(X \mid Y) := g(X + Y) - g(Y)$.

BFS for SKPs

We briefly review the procedures of BFS for SKPs, as detailed in (Chen, Chen, and Weinberger 2015).

Let $\mathcal{F} := \{X \subseteq U : c(X) \leq B\}$ be the collection of all feasible solutions, and remember that elements in U are numbered by $1, \ldots, n$. We define a *state-space tree* $G = (\mathcal{F}, E)$, which is a directed tree whose root is $\emptyset \in \mathcal{F}$.[1] Each node corresponds to a feasible solution $X \in \mathcal{F}$, and is called a *state*; the pair $X, Y \in \mathcal{F}$ has a directed edge $(X, Y) \in E$ if and only if $X = Y - \max Y$ holds, where $\max Y$ is an element in Y with the largest number. For example, $X = \{2\}$ and $Y = \{2, 4\}$ have an edge (X, Y) since $Y - \max Y = \{2, 4\} - \{4\} = \{2\} = X$. We abuse the notation and suppose that the set U has the linear order $<$; for example, we use $\{2\} < \{4\}$. Although $|\mathcal{F}|$ can be exponential in $|U|$, BFS expands states in $\mathcal{F}$ on-demand, typically requiring the storage of a very small fraction of $\mathcal{F}$.

As in Algorithm 1, BFS employs a max heap (i.e., a priority queue) to manages $\langle$key, value$\rangle$ pairs. A key corresponds to state $X \in \mathcal{F}$ and the priority value of X is defined as $f(X) := g(X) + h(X)$, where $h(\cdot)$ is a heuristic function. In each step of BFS, we pop the state with the biggest priority value from the heap and push its child states onto the heap. Let $T \in \mathcal{F}$ be a state popped from the heap. Then all child states $S \in \mathcal{F}$ such that $(T, S) \in E$ (i.e., $S = T + v$ for all $v \in U$ such that $v > \max T$) are pushed onto the heap. These procedures are repeated until solution T such that $h(T) = 0$ is obtained. BFS is guaranteed to be optimal if the heuristic function is *admissible*, i.e., $h(X^*) = 0$ and $h(X) \geq g(X^* \mid X)$ for any optimal solution X^* and $X \subseteq U$.

Under Estimation

While computing an optimal solution with BFS is sometimes too computationally expensive, an approximate solution can often be obtained efficiently by employing the under estimation technique as in (Chen, Chen, and Weinberger 2015);

[1] The original paper of FS uses a *state-space graph* and *closed list* so that each state is examined at most once. In practice, however, it is inefficient to use the closed list explicitly. Thus we employ here the state-space tree often used in *reverse search* (Avis and Fukuda 1996).

Algorithm 1 BFSTC$(U, g(\cdot), c_U, B, \alpha)$

1: $S_{\max} \leftarrow$ GetFeasible$(\emptyset)$
2: $f(\emptyset) \leftarrow \alpha h(\emptyset)$
3: $g_{\text{upper}} \leftarrow f(\emptyset)/\alpha$
4: MaxHeap.push$(\langle \emptyset, f(\emptyset) \rangle)$
5: **while** MaxHeap is not empty **do**
6: $T \leftarrow$ MaxHeap.pop()
7: **if** $h(T) = 0$ **then**
8: **return** T
9: **end if**
10: $g_{\text{upper}} \leftarrow \min\{g_{\text{upper}}, f(T)/\alpha\}$
11: **for** each S such that $(T, S) \in E$ **do**
12: $\hat{S} \leftarrow$ GetFeasible(S)
13: $S_{\max} \leftarrow \text{argmax}_{X \in \{S+\hat{S}, S_{\max}\}} g(X)$
14: **if** $g(S_{\max})/g_{\text{upper}} \geq \alpha$ **then**
15: **return** $S_{\max}$
16: **end if**
17: $f(S) \leftarrow g(S) + \alpha h(S)$
18: MaxHeap.push$(\langle S, f(S) \rangle)$
19: **end for**
20: **end while**

this approach is analogous to that of the weighted A* algorithm (Pohl 1970; Ebendt and Drechsler 2009). Specifically, instead of using $f(T) = g(T) + h(T)$, we employ the under-estimated priority $f(T) = g(T) + \alpha h(T)$, where $\alpha \in [0, 1]$ is a controllable hyper parameter. From the non-negativity of $g(\cdot)$ and the admissibility of $h(\cdot)$, we have $f(T) \geq \alpha(g(T) + h(T)) \geq \alpha g(X^*)$, which guarantees the α-optimality of BFS.

Proposed Termination Condition

We now explain the proposed termination condition (Steps 12–16), which detects that an α-approximate solution has been already obtained; we refer to our algorithm as BFS with termination condition (BFSTC).

As we will show later, evaluating $h(S)$ requires to obtain subset $\hat{S} \subseteq U$ such that $S + \hat{S} \in \mathcal{F}$. If we can detect $g(S + \hat{S})/g(X^*) \geq \alpha$ without knowing the true value of $g(X^*)$, we can immediately stop the search and return $S + \hat{S}$ as an α-approximate solution. We use this idea to create BFSTC. In Steps 1 and 12, GetFeasible(S) computes $\hat{S}$; for example, we may use the standard greedy algorithm to obtain $\hat{S}$. How to compute $\hat{S}$ depends on the heuristic function used, and thus we show the details of GetFeasible later for each heuristic function. $S_{\max}$ maintained in Algorithm 1 always gives a feasible solution, and it is updated in Step 13 so that $S_{\max}$ is the current best solution. Furthermore, since we have $f(T) \geq \alpha g(X^*)$ thanks to the admissibility of $h(\cdot)$, g_{upper} maintained in Algorithm 1 always gives an upper-bound of $g(X^*)$; $g(S_{\max})/g(X^*) \geq g(S_{\max})/g_{\text{upper}}$ always holds. Namely, $g(S_{\max})/g_{\text{upper}} \geq \alpha$ implies $S_{\max}$ is an α-approximate solution, i.e., $g(S_{\max})/g(X^*) \geq \alpha$. As in the experiments, the termination condition reduces the search effort of our algorithm, particularly when computing α-approximate solutions for $\alpha \leq 0.7$.

Theoretical Analysis

We here show a sufficient condition for BFSTC to achieve α-approximation; more specifically, we show a weaker version of the admissibility that suffices to obtain the α-optimality of BFSTC. The condition is a key to obtaining the proposed heuristic function. As shown by Chen, Chen, and Weinberger, if heuristic function $h(\cdot)$ satisfies the following *critical admissibility* (CA),[2] then BFS is α-optimal. In what follows, we define $V_S := \{v \in U : v > \max S\}$ for any given $S \subseteq U$; namely, V_S consists of the elements that are considered to be added to S in BFSTC.

Definition 1 (critical admissibility)**.** *We say $h(\cdot)$ satisfies CA if the following conditions hold for any $S \subseteq U$:*

1. *$h(S) = 0$ if $g(v \mid S) \leq 0$ or $S + v \notin \mathcal{F}$ for all $v \in V_S$,*
2. *$h(S) \geq \max_{X \subseteq V_S : X + S \in \mathcal{F}} \sum_{v \in X} g(v \mid S)$ otherwise.*

We introduce here the *weak critical admissibility* (WCA). With WCA, we can design a novel heuristic function as shown later, which substantially enhances BFS performance.

Definition 2 (weak critical admissibility)**.** *We say $h(\cdot)$ satisfies WCA if the following conditions hold for any $S \subseteq U$:*

1. *$h(S) = 0$ if $g(v \mid S) \leq 0$ or $S + v \notin \mathcal{F}$ for all $v \in V_S$,*
2. *$h(S) \geq \max_{X \subseteq V_S : X + S \in \mathcal{F}} g(X \mid S)$ otherwise.*

We name this condition WCA since heuristic function $h(\cdot)$ satisfying CA always satisfies WCA, while the reverse is not true; this can be confirmed using the submodularity of g. The following theorem guarantees that BFSTC is α-optimal if its heuristic function satisfies WCA. For proof, see the appendix.

Theorem 1. *If $h(\cdot)$ satisfies WCA, then solution $R \subseteq U$ obtained by BFSTC satisfies $g(R) \geq \alpha g(X^*)$.*

In the next section we discuss how to design $h(\cdot)$. The first condition of WCA (or CA) is easily satisfied by examining all $v \in V_S$, and thus we focus on the second condition. Given current solution S, heuristic function value $h(S)$ satisfying the second condition of WCA is obtained as an upper-bound of the optimal value of the following SKP:

$$(2) \qquad \underset{Y \subseteq V}{\text{maximize}} \; g_S(Y) \quad \text{subject to } c(Y) \leq B_S,$$

where $B_S := B - c(S)$, $V := \{v \in V_S : c(v) \leq B_S\}$ and $g_S(\cdot) := g(\cdot \mid S)$ is a monotone submodular function defined on $U - S$; $g_S(\cdot)$ is sometimes called the *contraction* of g on S (see, e.g., (Bach 2013)) and satisfies $g_S(\emptyset) = 0$. Therefore, in the next section, we discuss how to obtain upper-bounds of the optimal value of SKP (2).

Upper-bound Computation for SKP

As we have seen above, given solution S, heuristic function value $h(S)$ is obtained by computing an upper-bound of SKP (2). In this section, we let $Y^* \subseteq V$ denote an optimal solution for SKP (2), and we discuss how to compute an upper-bound of $g_S(Y^*)$, which we use as heuristic function value $h(S)$. In what follows, for a given ordered subset $X \subseteq$

[2] The above definition of CA is slightly different from the original one presented in (Chen, Chen, and Weinberger 2015) since we employed the state-space tree instead of the state-space graph.

Algorithm 2 GreedySK$(V, g_S(\cdot), c_V, B_S)$

$X \leftarrow$ GreedyAdd$(V, g_S(\cdot), c_V, B_S)$
$v_{\max} \leftarrow \max_{v \in V} g_S(v)$
$Z \leftarrow \operatorname{argmax}_{Y \in \{v_{\max}, X\}} g_S(Y)$
return Z

Algorithm 3 GreedyAdd$(V, g_S(\cdot), c_V, B_S)$

$X \leftarrow \emptyset$
while $V \neq \emptyset$ **do**
  $\hat{v} \leftarrow \operatorname{argmax}_{v \in V} g_S(v \mid X)/c_v$
  if $c(X + \hat{v}) \leq B_S$ **then**
    $X \leftarrow X + \hat{v}$
  end if
  $V \leftarrow V - \hat{v}$
end while
return X

V, we let $X_{1:i}$ denote the set of the first i elements ($X_{1:i} = \emptyset$ if $i < 1$) and X_i denotes its i-th element.

For later use, we show the ordinary greedy algorithm for SKP (2) in Algorithm 2, which we call GreedySK. The algorithm computes two candidates, X and $v_{\max}$, as its output: X is obtained by adding the elements with the largest cost-benefit ratio sequentially, and $v_{\max}$ is a singleton with the maximum objective value. We refer to the algorithm for computing the first candidate X as GreedyAdd, and we describe it separately from GreedySK for later use.

In the rest of this section, we show three upper-bounds of $g_S(Y^*)$, which we name u_{app}, u_{mod}, and u_{dom}. The first two bounds are obtained with existing techniques. The third one is the proposed upper-bound, and is used as heuristic function value $h(S)$ in our algorithm.

Upper-bound with Approximation Ratios (u_{app})

Given γ-approximate solution Z for SKP (2), the simplest way to obtain an upper-bound of $g_S(Y^*)$ is to compute $u_{\text{app}} := g_S(Z)/\gamma$. As in (Lin and Bilmes 2010), GreedySK for SKPs achieves at least $(1 - 1/\sqrt{e})$-approximation; more strictly, if X is the output of GreedyAdd, the approximation ratio achieved by GreedySK is at least $\gamma = 1 - \left(1 - \frac{1}{2|X|}\right)^{|X|}$. In the size-constrained case, we can improve the upper-bound using the results in (Conforti and Cornuéjols 1984) as follows: we compute the curvature $\kappa \in [0, 1]$ of submodular function g_S and we let $\gamma = \frac{1}{\kappa}\left(1 - (1 - \kappa/|X|)^{|X|}\right)$. The A* algorithm, whose heuristic function is a variant of the above u_{app}, is studied in (Zeng et al. 2015) for a specific objective function.

We now consider the computational aspect of BFSTC that uses u_{app} as its heuristic function value. For current S, we can compute $h(S) = u_{\text{app}}$ efficiently once we obtain $g_S(Z)$, where Z is the output of GreedySK$(V, g_S(\cdot), c_V, B_S)$. Therefore, in Steps 1 and 12 of BFSTC, we use GreedySK$(V, g_S(\cdot), c_V, B_S)$ as GetFeasible(S), with which we compute $g_S(Z)$ and let $h(S) = g_S(Z)/\gamma$.

Upper-bound with Modular Functions (u_{mod})

The second upper-bound u_{mod} is the one used in (Chen, Chen, and Weinberger 2015), which gives a heuristic function satisfying CA. Thanks to the submodularity of g_S, we have $\sum_{v \in X} g_S(v) \geq g_S(X)$ for any $X \subseteq V$, and thus we obtain

$$\max_{X \subseteq V : c(X) \leq B_S} \sum_{v \in X} g_S(v) \geq \sum_{v \in Y^*} g_S(v) \geq g_S(Y^*) = g_S(Y^*). \tag{3}$$

Therefore, in order to bound $g_S(Y^*)$ from above, we compute an upper-bound of the left-hand side of eq. (3), which is an optimal value of KP. Such an upper-bound is easily obtained by considering the *fractional relaxation* of KP as follows. We sort V in the non-increasing order of $r_v := g_S(v)/c_v$; $V_{1:i}$ denotes the first i elements of the sorted set. If we have $r_v = \infty$ (i.e., $g_S(v) > 0$ and $c_v = 0$) for some $v \in V$, we place them at the beginning of V in arbitrary order. Furthermore, we define $l := \max\{i \in [|V|] : c(V_{1:l}) \leq B_S\}$. If $l < |V|$, then the following value bounds the left-hand side of eq. (3) from above (see, (Dantzig 1957)):

$$u_{\text{mod}} := \sum_{v \in V_{1:l}} g_S(v) + (B_S - c(V_{1:l}))\, r_{V_{l+1}}.$$

If $l = |V|$, then $g_S(Y^*) = g_S(V)$ thanks to the monotonicity of g_S, and thus we let $u_{\text{mod}} := g_S(V)$. Consequently, u_{mod} always bounds $g_S(Y^*)$ from above.

Given current solution S, we consider computing $h(S) = u_{\text{mod}}$ efficiently in BFSTC. As discussed above, u_{mod} is obtained by solving the relaxed KP. Thus in Steps 1 and 12 of BFSTC, we let GetFeasible(S) compute $V_{1:l+1}$ and output $V_{1:l}$ if $l < |V|$; note that we have $V_{1:l} + S \in \mathcal{F}$. We then set $h(S)$ to the objective value that is obtained by solving the relaxed KP; namely $h(S) = \sum_{v \in V_{1:l}} g_S(v) + (B_S - c(V_{1:l}))\, g_S(V_{l+1})/c_{V_{l+1}}$. If $l = |V|$, we let GetFeasible(S) output V and set $h(S) = g_S(V)$.

For later use, given any $Y \subseteq V$, we define $u_{\text{mod}}(Y)$ as an upper-bound of $\max_{X \subseteq V \setminus Y : c(X) \leq B_S} g_S(X \mid Y)$ that is obtained by solving its relaxed KP as shown above.

Upper-bound with Dominant Elements (u_{dom})

We here propose the new upper-bound u_{dom}. To compute u_{dom}, we use an output of GreedyAdd$(V, g_S(\cdot), c_V, B_S)$, which we denote $X_{1:k}$; $X_{1:i}$ is the set of the first i elements added by GreedyAdd for $i \in [k]$. We define

$$\beta_{1:k} := \begin{cases} 0 & \text{if } u_{\text{mod}}(X_{1:i}) = 0 \text{ for some } i \in [k], \\ \prod_{i=1}^{k} \beta_i & \text{otherwise,} \end{cases}$$

$$\beta_i := 1 - \frac{g_S(X_i \mid X_{1:i-1})}{u_{\text{mod}}(X_{1:i-1})} \quad \text{for } i \in [k].$$

Then we let $u_{\text{dom}} := g_S(X_{1:k})/(1 - \beta_{1:k})$.

Theorem 2. *$u_{dom} \geq g_S(Y^*)$ holds for any given $S \subseteq U$.*

The proof, presented in the appendix, is analogous to the well-known proof of $(1 - 1/e)$-approximation for size-constrained submodular maximization. In the size-constrained case, we have $\beta_i = 1 - 1/k$ in the worst case,

which leads to the $(1-1/e)$-approximation guarantee; from this fact, the simple upper-bound $g_S(X_{1:k})/(1-1/e) \geq g_S(Y^*)$ can be obtained. In most cases, however, we have $\beta_i < 1-1/k$, and hence u_{dom} is expected to be closer to $g_S(Y^*)$ than $g_S(X_{1:k})/(1-1/e)$.

We discuss when u_{dom} gives an upper-bound that is close to $g_S(Y^*)$. From the definition of u_{dom}, we see that u_{dom} is likely to overestimate $g_S(Y^*)$ if $\beta_{1:k} \in [0,1]$ is close to 1; namely $\beta_{1:k} \approx 0$ is desirable, which holds if we have $\beta_i \approx 0$ for some $i \in [k]$. Such small β_i can be obtained if we have $g_S(X_i \mid X_{1:i-1})/u_{\text{mod}}(X_{1:i-1}) \approx 1$, which occurs if X_i is has a dominant marginal gain compared to $v \in V - X_{1:i}$. In other words, if $g_S(X_i \mid X_{1:i-1}) \gg g_S(v \mid X_{1:i-1})$ holds for all $v \in V - X_{1:i}$, then we have $\beta_i \approx 0$, and consequently we obtain $\beta_{1:k} \approx 0$. Namely, the accuracy of u_{dom} depends on the dominance of the elements selected by GreedyAdd.

We consider computing $h(S) = u_{\text{dom}}$ efficiently in BFSTC for current solution S. As shown above, u_{dom} can be computed easily if we have $g_S(X_{1:k})$ and the marginal gains $g_S(v \mid X_{1:i-1})$ for all $v \in V - X_{1:i-1}$ and $i \in [k]$, which can be obtained once GreedyAdd$(V, g_S(\cdot), c_V, B_S)$ is executed. Therefore, in Steps 1 and 12 of BFSTC, we get $\hat{S}$ by using GreedySK$(V, g_S(\cdot), c_V, B_S)$, which includes GreedyAdd$(V, g_S(\cdot), c_V, B_S)$ as its building block. Then we can compute u_{dom} without additional function evaluation.

Experiments

All experiments were conducted on a 64-bit Cent7.3.1611 machine with Xeon 5E-2697 v3 2.6 GHz CPUs and 256 GB of RAM. Our algorithm is BFSTC with the heuristic function given by u_{dom}, which we call DOM. To benchmark our algorithm, we use BFSTC with the heuristic functions given by u_{app} and u_{mod}, which we call APP and MOD, respectively. Note that APP and MOD are improved variants of those in (Zeng et al. 2015) and (Chen, Chen, and Weinberger 2015), respectively. All instances considered are formulated as SKPs. For the objective functions, we use the following *weighted coverage* (COV) function, *facility location* (LOC) function, and *bipartite influence* (INF) function.

Weighted Coverage (COV) The first one is the well-known weighted coverage function (see, e.g., (Krause and Golovin 2013)). Let $[m]$ be a set of m items. We define $w_i \geq 0$ as the weight of i-th item for each $i \in [m]$. Each $v \in U$ covers some items, and we let $I_v \subseteq [m]$ indicate the set of items covered by v. The following weighted coverage function is monotone and submodular:

$$g(X) := \sum_{i \in \bigcup_{v \in X} I_v} w_i.$$

This function often appears in the context of itemset mining (e.g., (Kumar et al. 2015)). Here c_v is the cost of choosing v.

Facility Location (LOC) The second one is the *facility location* function (see, e.g., (Krause and Golovin 2013)). We regard U as a set of locations and consider selecting some locations at which to build certain facilities. Let $[m]$ be a set of m clients. We define $q_{i,v} \geq 0$ as the benefit i-th client gains from the facility built at v. Given $X \subseteq U$, which represents the set of locations where the facilities are built, each client gains benefit from the most beneficial facility, and thus the total benefit for the clients is defined as

$$g(X) = \sum_{i \in [m]} \max_{v \in X} q_{i,v};$$

this is known to be a monotone submodular function. Here c_v represents the cost of building a facility at v.

Bipartite Influence (INF) The third one represents an influence model on bipartite graphs, which is a special case of the problem studied in (Alon, Gamzu, and Tennenholtz 2012). Let U be a set of items. We regard $[m]$ as a set of m targets. Given bipartite graph $G = (U, [m]; A)$, where $A \subseteq U \times [m]$ is a set of directed edges, we consider an influence maximization problem on G. The probability that i-th target gets activated by items $X \subseteq U$ is $1 - \prod_{v \in X:(v,i) \in A}(1-p_v)$, where $p_v \in [0,1]$ is the activation probability of item v. Objective function $g(X)$ is the expected number of targets that are activated by items in X, which can be written as

$$g(X) = \sum_{i \in [m]} \left(1 - \prod_{v \in X:(v,i) \in A} (1-p_v)\right).$$

In this setting, c_v represents the cost of choosing v.

Artificial Instances

We randomly generated 100 instances for COV, LOC, and INF, and we compared the performance of the three algorithms. In all instances, we set $|U| = 100$ and $B = 1$. The costs c_v $(v \in U)$ are drawn randomly from a uniform distribution on $[0,1]$; we denote a uniform distribution over $[a,b]$ by $u_{[a,b]}$. We let $m = 1000$ for each m that appears in the definitions of COV, LOC, and INF. In COV instances, w_i are drawn from $u_{[0,1]}$, and each $v \in U$ randomly covers each $w \in W$ with probability 0.3. In LOC instances, all benefits, $q_{i,v}$, are drawn from $u_{[0,1]}$. In INF instances, all p_v are drawn from $u_{[0,1]}$, and each pair $(v,i) \in U \times [m]$ is connected randomly with probability 0.3. In all instances, we set the time limit to one hour.

The results are summarized in Table 1 for various approximation ratios $\alpha = 0.4, \ldots, 1.0$, which can be controlled by the user; since the greedy algorithm for SKP achieves $1 - 1/\sqrt{e} \approx 0.39$ approximation, we consider only the case $\alpha > 0.39$. For each method, we observed and compared the number of instances solved without exceeding the time limit, number of nodes pushed onto the heap (averaged over all 100 instances), and running time. In Table 2, which is provided in the appendix, we also present some detailed results on the number of pushed nodes averaged over solved instances. The running times in Table 1 is compared for two pairs (DOM, MOD) and (DOM, APP), both of which are averaged over all instances solved by the both methods. Our method DOM substantially outperforms the other methods in all aspects. Notably, in the case $\alpha \leq 0.7$, DOM pushes only one node onto the heap, thus requiring dramatically less time and space complexities than the other methods. This result is thanks to the high accuracy of u_{dom} and the proposed termination condition; our upper-bound u_{dom} is so accurate that DOM detected

Table 1: Numerical results for artificial instances. '# solved' is the number of instances solved without exceeding the time limit. '# nodes' is the number of nodes pushed onto the heap averaged over all instances including those unsolved. 'A–B time' compares the running times for the two methods $(\mathrm{A}, \mathrm{B}) = (\mathrm{DOM}, \mathrm{MOD})$ and $(\mathrm{DOM}, \mathrm{APP})$; the running times are averaged over instances solved by both A and B. For each value of α the best results are given in bold.

	Approximation ratio α		0.4	0.5	0.6	0.7	0.8	0.9	1.0
COV	# solved	DOM	**100**	**100**	**100**	**100**	**100**	**96**	**84**
		MOD	**100**	97	91	84	73	64	62
		APP	**100**	83	59	47	27	24	20
	# nodes	DOM	**1.00e+0**	**1.00e+0**	**1.00e+0**	**1.00e+0**	**3.98e+3**	**5.96e+5**	**1.55e+6**
		MOD	1.13e+5	4.66e+5	9.05e+5	1.39e+6	1.98e+6	2.56e+6	2.92e+6
		APP	1.35e+4	1.43e+6	2.88e+6	4.05e+6	7.29e+6	8.16e+6	9.67e+6
	DOM–MOD time (s)	DOM	**2.47e-2**	**2.61e-2**	**2.60e-2**	**2.35e-2**	**2.92e+0**	**6.32e+1**	**1.95e+2**
		MOD	2.72e+1	1.24e+2	2.79e+2	4.13e+2	4.83e+2	4.80e+2	8.75e+2
	DOM–APP time (s)	DOM	**2.47e-2**	**2.49e-2**	**2.49e-2**	**2.19e-2**	**2.07e+0**	**2.30e+1**	**4.75e+1**
		APP	3.42e+0	4.28e+2	7.19e+2	1.36e+3	1.09e+3	1.05e+3	8.04e+2
LOC	# solved	DOM	**100**	**100**	**100**	**100**	**100**	**100**	**100**
		MOD	**100**	**100**	**100**	98	93	87	76
		APP	**100**	98	77	60	45	38	32
	# nodes	DOM	**1.00e+0**	**1.00e+0**	**1.00e+0**	**1.00e+0**	**2.08e+3**	**1.03e+5**	**5.38e+5**
		MOD	2.27e+4	8.83e+4	2.53e+5	5.40e+5	9.25e+5	1.37e+6	1.91e+6
		APP	4.99e+2	5.13e+5	1.79e+6	2.88e+6	4.12e+6	5.10e+6	6.18e+6
	DOM–MOD time (s)	DOM	**1.72e-2**	**1.70e-2**	**1.74e-2**	**1.70e-2**	**1.55e+0**	**3.17e+1**	**7.74e+1**
		MOD	3.67e+0	1.38e+1	6.74e+1	1.50e+2	2.49e+2	4.48e+2	4.33e+2
	DOM–APP time (s)	DOM	**1.72e-2**	**1.68e-2**	**1.64e-2**	**1.57e-2**	**1.25e+0**	**1.27e+1**	**2.62e+1**
		APP	1.90e-1	1.20e+2	4.96e+2	7.74e+2	6.84e+2	9.41e+2	8.35e+2
INF	# solved	DOM	**100**	**100**	**100**	**100**	**100**	**100**	**100**
		MOD	**100**	**100**	**100**	**100**	97	93	89
		APP	**100**	99	87	69	48	38	29
	# nodes	DOM	**1.00e+0**	**1.00e+0**	**1.00e+0**	**1.00e+0**	**8.76e+2**	**6.02e+4**	**3.52e+5**
		MOD	5.59e+2	9.62e+3	5.81e+4	2.11e+5	5.05e+5	7.99e+5	1.11e+6
		APP	2.85e+2	4.25e+5	1.24e+6	2.23e+6	3.71e+6	5.04e+6	6.78e+6
	DOM–MOD time (s)	DOM	**1.59e-3**	**1.65e-3**	**1.66e-3**	**1.82e-3**	**8.14e-2**	**2.97e+0**	**1.61e+1**
		MOD	2.25e-2	2.83e-1	1.93e+0	2.86e+1	1.26e+2	1.88e+2	3.01e+2
	DOM–APP time (s)	DOM	**1.59e-3**	**1.65e-3**	**1.60e-3**	**1.71e-3**	**6.70e-2**	**1.20e+0**	**3.20e+0**
		APP	1.45e-2	1.43e+2	4.12e+2	8.00e+2	6.99e+2	8.49e+2	8.04e+2

the α-optimality of the solution obtained by GreedySK at the beginning, and thus it terminated in Step 15 of Algorithm 1 without pushing additional nodes onto the heap.

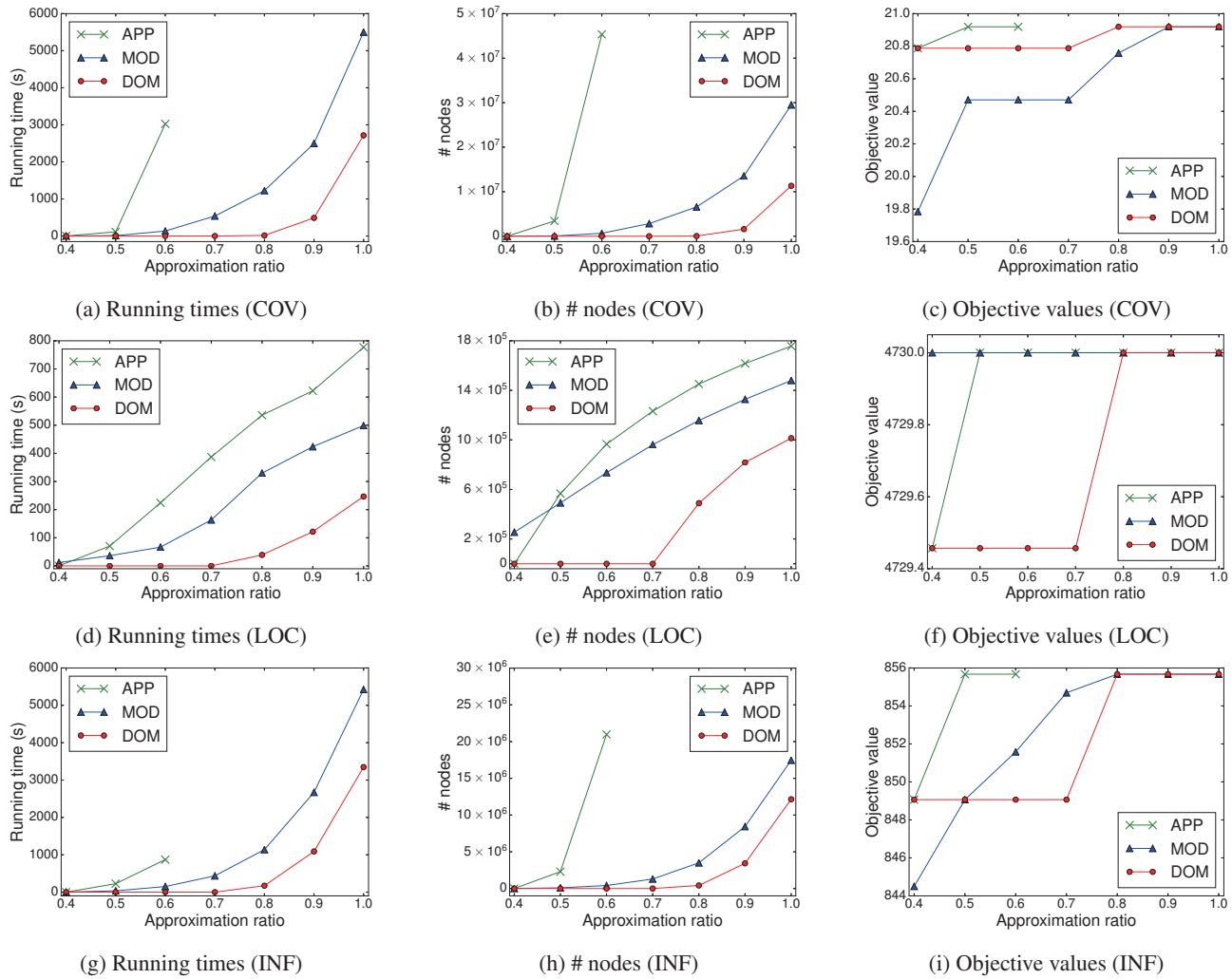
Real-world Instances

We applied the three algorithms to COV, LOC, and INF instances that are generated with real-world data. Since the data did not include information on cost values c_v, we drew them from $u_{[0.1,1]}$ in all instances; in practice it is rare for any item v to have cost of $c_v \approx 0$, and thus we set the lower-bound of cost to 0.1. The budget value, B, is set for each instance so that the resulting instance does not become computationally too demanding; we set B to a small value if $|U|$ is large. In all instances, the time limit is one day (86,400 seconds). Below we detail the experimental settings:

COV: The COV instance uses a dataset on messages exchanged by 899 users on 522 topics (Opsahl 2013). Each user posts messages on some topics with which he/she is familiar; we regard a user covers a topic if he/she posts messages to the topic. Each topic is weighted based on the number of posted messages; namely, the importance of topics is measured by the number of posted messages. We consider selecting some users so that the total weights of covered topics is maximized. Here we let $B = 1.25$.

LOC: The LOC instance uses a dataset on 473 subway stations in New York City (NYC) (Roest and Mashariki 2015). We regard the stations as clients, and consider building several facilities so that people at any station can easily reach one of the facilities. The candidate locations, on which facilities can be built, are given by a 10×10 grid that discretizes the NYC map, and we select some locations from the 100 candidate locations. The benefit a client (station) gains from a facility is computed based on the distance between them. In this instance we let $B = 1.5$.

INF: The INF instance uses the MovieLens 100K dataset (Harper and Konstan 2015). The dataset contains



(a) Running times (COV) (b) # nodes (COV) (c) Objective values (COV)

(d) Running times (LOC) (e) # nodes (LOC) (f) Objective values (LOC)

(g) Running times (INF) (h) # nodes (INF) (i) Objective values (INF)

Figure 1: Numerical results on real-world instances. Figures (a)–(c), (d)–(f), and (g)–(i) show results for COV, LOC, and INF instances, respectively. (a),(d),(g) Running times of the three algorithms: DOM, MOD, and APP. (b),(e),(h) Number of nodes pushed onto the heap. (c),(f),(i) Achieved objective values.

$100,000$ ratings (1-5) by 943 users on 1682 movies, and we consider selecting some influential movies. The activation probability of each movie is computed from the ratings, and a movie can activate a user if he/she has rated the movie. Here we let $B = 1$.

Figures 1 (a)–(i) summarize the results; some results of APP for COV and INF instances are omitted since they exceeded the time limit for COV instances, and memory shortages occurred with in INF instances. Similar to the results on artificial instances, DOM outperforms the other methods both in time and space complexities. MOD and APP tend to require too much computational effort even for small α, achieving slightly higher objective values than DOM in some instances. This implies that MOD and APP are poor at using the hyper-parameter α to control the trade-off between the complexity and optimality. On the other hand, DOM successfully controlled the trade-off, and reduced the complexities when computing α-approximate solutions for small α; in particular DOM is efficient when $\alpha \leq 0.7$.

Conclusion and Discussion

We proposed an accelerated BFS for SKP. The acceleration is achieved by introducing a new termination condition and developing a novel method for computing an upper-bound of the optimal value of the SKP. For any given $\alpha \in [0, 1]$, our algorithm is proved to find an α-optimal solution. Experiments showed that our algorithm finds approximate solutions with substantially less time and space complexities than the existing methods.

In this paper, as an application of the proposed upper-bound computation method, we focused on accelerating BFS. However, we believe that computing an upper-bound accurately is beneficial to many other algorithms that include submodular maximization as a subroutine. In future work we will try to develop more powerful variants of the upper-bound

Table 2: Results on the number of nodes pushed onto the heap. '# nodes (solved)' is the number of pushed nodes averaged over instances solved by each method. 'A–B # nodes' is the number of pushed nodes averaged over all instances solved by both methods $(\mathrm{A}, \mathrm{B}) = (\mathrm{DOM}, \mathrm{MOD})$ or $(\mathrm{DOM}, \mathrm{APP})$. For each value of α the best results given in bold.

	Approximation ratio α		0.4	0.5	0.6	0.7	0.8	0.9	1.0
COV	# nodes (solved)	DOM	**1.00e+0**	**1.00e+0**	**1.00e+0**	**1.00e+0**	**3.98e+3**	**4.29e+5**	**8.66e+5**
		MOD	1.13e+5	3.37e+5	5.40e+5	7.38e+5	7.86e+5	8.21e+5	1.02e+6
		APP	1.35e+4	7.47e+5	9.08e+5	1.06e+6	8.39e+5	7.66e+5	6.50e+5
	DOM–MOD # nodes	DOM	**1.00e+0**	**1.00e+0**	**1.00e+0**	**1.00e+0**	**4.35e+3**	**1.69e+5**	**4.71e+5**
		MOD	1.13e+5	3.37e+5	5.40e+5	7.38e+5	7.86e+5	8.21e+5	1.02e+6
	DOM–APP # nodes	DOM	**1.00e+0**	**1.00e+0**	**1.00e+0**	**1.00e+0**	**4.05e+3**	**6.77e+4**	**1.50e+5**
		APP	1.35e+4	7.47e+5	9.08e+5	1.06e+6	8.39e+5	7.66e+5	6.50e+5
LOC	# nodes (solved)	DOM	**1.00e+0**	**1.00e+0**	**1.00e+0**	**1.00e+0**	**2.08e+3**	**1.03e+5**	**5.38e+5**
		MOD	2.27e+4	8.83e+4	2.53e+5	4.64e+5	6.34e+5	8.24e+5	8.23e+5
		APP	4.99e+2	4.41e+5	8.13e+5	9.27e+5	7.73e+5	8.34e+5	7.43e+5
	DOM–MOD # nodes	DOM	**1.00e+0**	**1.00e+0**	**1.00e+0**	**1.00e+0**	**2.20e+3**	**7.32e+4**	**2.45e+5**
		MOD	2.27e+4	8.83e+4	2.53e+5	4.64e+5	6.34e+5	8.24e+5	8.23e+5
	DOM–APP # nodes	DOM	**1.00e+0**	**1.00e+0**	**1.00e+0**	**1.00e+0**	**1.99e+3**	**3.31e+4**	**9.85e+4**
		APP	4.99e+2	4.41e+5	8.13e+5	9.27e+5	7.73e+5	8.34e+5	7.43e+5
INF	# nodes (solved)	DOM	**1.00e+0**	**1.00e+0**	**1.00e+0**	**1.00e+0**	**8.76e+2**	**6.02e+4**	**3.52e+5**
		MOD	5.59e+2	9.62e+3	5.81e+4	2.11e+5	3.86e+5	5.18e+5	6.80e+5
		APP	2.85e+2	3.88e+5	7.59e+5	1.01e+6	9.51e+5	9.34e+5	8.45e+5
	DOM–MOD # nodes	DOM	**1.00e+0**	**1.00e+0**	**1.00e+0**	**1.00e+0**	**8.86e+2**	**4.94e+4**	**2.13e+5**
		MOD	5.59e+2	9.62e+3	5.81e+4	2.11e+5	3.86e+5	5.18e+5	6.80e+5
	DOM–APP # nodes	DOM	**1.00e+0**	**1.00e+0**	**1.00e+0**	**1.00e+0**	**7.94e+2**	**2.33e+4**	**7.48e+4**
		APP	2.85e+2	3.88e+5	7.59e+5	1.01e+6	9.51e+5	9.34e+5	8.45e+5

computation methods, which will improve the performance of various algorithms.

Acknowledgements This work was supported by JSPS KAKENHI Grant Numbers 15H05711 and 16K16011.

Appendix

Proof for Theorem 1

Let X^* be an optimal solution and assume $R \neq X^*$, otherwise the claim holds trivially.

We first show that the max heap always contains a subset $P \subseteq X^*$ such that $\max P < \min X^* \backslash P$ until R is popped; we regard $\max \emptyset = 0$ and $\min \emptyset = \infty$. This is apparently true at the beginning since $\emptyset \subseteq X^*$ is in the heap. Whenever such P is popped, we have either

(i) $P = X^*$, or

(ii) all $P' \subseteq X^*$ such that $(P, P') \in E$ are pushed onto the heap, one of which satisfies $\max P' < \min X^* \backslash P'$.

If case (i) occurs, then we have $h(P) = 0$ since $h(\cdot)$ satisfies the first condition of WCA and $P = X^*$ is optimal. As a result, BFSTC returns the optimal solution $P = X^*$, which contradicts the assumption. Thus case (ii) continues to hold until BFSTC returns R. Consequently, a subset $P \subseteq X^*$ such that $\max P < \min X^* \backslash P$ must be in the heap until R is popped.

We then prove $f(T) \geq \alpha g(X^*)$ for any T popped from the heap. As shown above, the heap always contains a subset $P \subseteq X^*$ with $\max P < \min X^* \backslash P$. For such P, we have

$$\begin{aligned} f(P) &= g(P) + \alpha h(P) \geq \alpha(g(P) + h(P)) \\ &\geq \alpha(g(P) + g(X^* \backslash P \mid P)) = \alpha g(X^*), \end{aligned}$$

where the second inequality comes from the fact that $h(\cdot)$ satisfies the second condition of WCA. Since the popped T has the largest f value in the max heap, we obtain $f(T) \geq f(P) \geq \alpha g(X^*)$. We note that, for some popped T, we always have $g_{\text{upper}} = f(T)/\alpha$. Hence g_{upper} always gives an upper-bound for $g(X^*)$, i.e., $g_{\text{upper}} \geq g(X^*)$.

We now consider two cases: R is obtained in Step 8 or in Step 15. If R is obtained in Step 15, we have $g(R) \geq \alpha g_{\text{upper}} \geq \alpha g(X^*)$. If R is obtained in Step 8, we have $h(R) = 0$, which leads to

$$g(R) = g(R) + \alpha h(R) = f(R) \geq \alpha g(X^*),$$

where $f(R) \geq \alpha g(X^*)$ comes from the fact that R is popped from the max heap. Therefore, $g(R) \geq \alpha g(X^*)$ holds in both cases, and thus the proof is completed.

Proof of Theorem 2

For $i = 0, \ldots, k$, we have

$$\begin{aligned} (4) \quad & g_S(Y^*) \\ & \leq g_S(Y^* + X_{1:i}) \\ & = g_S(X_{1:i}) + g_S(Y^* \mid X_{1:i}) \\ & \leq g_S(X_{1:i}) + \max_{X \subseteq V \backslash X_{1:i} : c(X) \leq B_S} g_S(X \mid X_{1:i}) \\ & \leq g_S(X_{1:i}) + u_{\text{mod}}(X_{1:i}), \end{aligned}$$

where the third inequality comes from the definition of $u_{\mathrm{mod}}(X_{1:i})$.

We first consider the case where $u_{\mathrm{mod}}(X_{1:i}) = 0$ holds for some $i \in [k]$. In this case we have

$$g_S(Y^*) \leq g_S(X_{1:i}) + u_{\mathrm{mod}}(X_{1:i}) = g_S(X_{1:i})$$

from inequality (4). Thanks to the monotonicity of g_S, we have $g_S(X_{1:k}) \geq g_S(X_{1:i})$, and hence $u_{\mathrm{dom}} = g_S(X_{1:k}) \geq g_S(Y^*)$ holds; more precisely, $u_{\mathrm{dom}} = g_S(X_{1:k}) = g_S(Y^*)$ holds since Y^* is optimal.

We then consider the case where $u_{\mathrm{mod}}(X_{1:i}) > 0$ holds for all $i \in [k]$. For $i \in [k]$, we have

$$\begin{aligned} g_S(Y^*) &\leq g_S(X_{1:i-1}) + u_{\mathrm{mod}}(X_{1:i-1}) \\ &= g_S(X_{1:i-1}) \\ &\quad + \frac{u_{\mathrm{mod}}(X_{1:i-1})}{g_S(X_i \mid X_{1:i-1})}(g_S(X_{1:i}) - g_S(X_{1:i-1})) \\ &= g_S(X_{1:i-1}) \\ &\quad + \frac{1}{1-\beta_i}(g_S(X_{1:i}) - g_S(X_{1:i-1})) \end{aligned}$$

from inequality (4). Rearranging the terms yields

$$g_S(X_{1:i}) \geq \beta_i g_S(X_{1:i-1}) + (1-\beta_i) g_S(Y^*),$$

and thus the following inequality holds for $i \in [k]$:

$$g_S(Y^*) - g_S(X_{1:i}) \leq \beta_i (g_S(Y^*) - g_S(X_{1:i-1})).$$

Therefore, from $g_S(\emptyset) = 0$, we obtain

$$g_S(X_{1:k}) \geq \left(1 - \prod_{i=1}^{k} \beta_i\right) g_S(Y^*) = (1 - \beta_{1:k})\, g_S(Y^*).$$

Hence $u_{\mathrm{dom}} = g_S(X_{1:k})/(1-\beta_{1:k}) \geq g_S(Y^*)$ holds.

Results on the Number of Pushed Nodes

Table 2 summarizes the results on the number of nodes pushed onto the heap averaged over solved instances. Our algorithm, DOM, pushes fewer nodes onto the heap than other methods, thus requiring less space complexity.

References

Alon, N.; Gamzu, I.; and Tennenholtz, M. 2012. Optimizing budget allocation among channels and influencers. In *Proceedings of the 21st International Conference on World Wide Web*, WWW '12, 381–388. New York, NY, USA: ACM.

Avis, D., and Fukuda, K. 1996. Reverse search for enumeration. *Discrete Appl. Math.* 65(1):21 – 46. First International Colloquium on Graphs and Optimization.

Bach, F. 2013. Learning with submodular functions: A convex optimization perspective. *Foundations and Trends® in Machine Learning* 6(2-3):145–373.

Chen, W.; Chen, Y.; and Weinberger, K. 2015. Filtered search for submodular maximization with controllable approximation bounds. In *Proceedings of the 18th International Conference on Artificial Intelligence and Statistics*, volume 38, 156–164. PMLR.

Conforti, M., and Cornuéjols, G. 1984. Submodular set functions, matroids and the greedy algorithm: Tight worst-case bounds and some generalizations of the Rado-Edmonds theorem. *Discrete Appl. Math.* 7(3):251–274.

Dantzig, G. B. 1957. Discrete-variable extremum problems. *Oper. Res.* 5(2):266–288.

Ebendt, R., and Drechsler, R. 2009. Weighted A* search - unifying view and application. *Artificial Intelligence* 173(14):1310 – 1342.

Feige, U. 1998. A threshold of ln n for approximating set cover. *J. ACM* 45(4):634–652.

Harper, F. M., and Konstan, J. A. 2015. The MovieLens datasets: History and context. *ACM Trans. Interact. Intell. Syst.* 5(4):19:1–19:19. https://grouplens.org/datasets/movielens/. Last accessed 23 August 2017.

Kawahara, Y.; Nagano, K.; Tsuda, K.; and Bilmes, J. A. 2009. Submodularity cuts and applications. In *Proceedings of the 22nd International Conference on Neural Information Processing Systems*, 916–924. USA: Curran Associates Inc.

Khuller, S.; Moss, A.; and Naor, J. S. 1999. The budgeted maximum coverage problem. *Inform. Process. Lett.* 70(1):39–45.

Krause, A., and Golovin, D. 2013. Submodular function maximization. In *Tractability: Practical Approaches to Hard Problems*. Cambridge University Press.

Krause, A.; Singh, A.; and Guestrin, C. 2008. Near-optimal sensor placements in gaussian processes: Theory, efficient algorithms and empirical studies. *J. Mach. Learn. Res.* 9(Feb):235–284.

Kumar, R.; Moseley, B.; Vassilvitskii, S.; and Vattani, A. 2015. Fast greedy algorithms in mapreduce and streaming. *ACM Trans. Parallel Comput.* 2(3):14:1–14:22.

Lin, H., and Bilmes, J. 2010. Multi-document summarization via budgeted maximization of submodular functions. In *Proceedings of Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, 912–920. Association for Computational Linguistics.

Nemhauser, G., and Wolsey, L. 1981. Maximizing submodular set functions: Formulations and analysis of algorithms. *North-Holland Mathematics Studies* 59:279 – 301. Annals of Discrete Mathematics (11).

Nemhauser, G. L.; Wolsey, L. A.; and Fisher, M. L. 1978. An analysis of approximations for maximizing submodular set functions-I. *Math. Program.* 14(1):265–294.

Opsahl, T. 2013. Triadic closure in two-mode networks: Redefining the global and local clustering coefficients. *Social Networks* 35(6):159–167. https://toreopsahl.com/datasets/#online_forum_network. Last accessed 23 August 2017.

Pohl, I. 1970. Heuristic search viewed as path finding in a graph. *Artificial Intelligence* 1(3):193 – 204.

Roest, A., and Mashariki, A. R. 2015. NYC Open Data. https://data.cityofnewyork.us/Transportation/Subway-Stations/arq3-7z49/data. Last accessed 23 August 2017.

Sviridenko, M.; Vondrák, J.; and Ward, J. 2015. Optimal approximation for submodular and supermodular optimization with bounded curvature. In *Proceedings of the 26th Annual ACM-SIAM Symposium on Discrete Algorithms*, 1134–1148. SIAM.

Sviridenko, M. 2004. A note on maximizing a submodular set function subject to a knapsack constraint. *Oper. Res. Lett.* 32(1):41–43.

Zeng, Y.; Chen, X.; Cao, X.; Qin, S.; Cavazza, M.; and Xiang, Y. 2015. Optimal route search with the coverage of users' preferences. In *Proceedings of the 24th International Joint Conference on Artificial Intelligence*, 2118–2124.