BF*: Web Services Discovery and Composition as Graph Search Problem

Seog-Chan Oh
IE / Penn State
sxo160@psu.edu

Byung-Won On
CSE / Penn State
on@cse.psu.edu

Eric J. Larson
IST / Penn State
ejl175@psu.edu

Dongwon Lee
IST / Penn State
dongwon@psu.edu

Abstract

When there are a large number of web services available (e.g., in the range of 1,000 - 10,000), it is non-trivial to quickly find web services satisfying the given request. Furthermore, when no single web service satisfies the given request fully, one needs to "compose" multiple web services to fulfill the goal. Since the search space for such a composition problem is in general exponentially increasing, it is important to have wise decision on underlying data structures and search algorithms. Toward this problem, in this paper, we present a novel solution, named as BF (BF-Star), that adopts the competitive A* as a search algorithm while utilizing the Bloom Filter as a succinct data structure.*

1 Introduction

A web service, w, has typically two sets of parameters: $w_{in} = \{I^1, ..., I^p\}$ for SOAP request (as input) and $w_{out} = \{O^1, ..., O^q\}$ for SOAP response (as output). When w is invoked with all input parameters, w_{in}, it returns the output parameters, w_{out}. In general, all input parameters in w_{in} must be provided (i.e., mandatory). For instance, consider the following web service `findRestaurant`:

```
<message name='findRestaurant_Request'>
  <part name='zip' type='xs:string'>
  <part name='foodPref' type='xs:string'>
</message>
<message name='findRestaurant_Response'>
  <part name='name' type='xs:string'>
  <part name='phone' type='xs:string'>
</message>
```

Now, suppose a user, u, currently has zip code, city name, and food preference information, but is looking for the phone number of a restaurant. Then, u can invoke w with `zip` and `foodPref` and receive `name` and `phone` in return, $\{\text{name}, \text{phone}\} \leftarrow w(\text{zip}, \text{foodPref})$, which contains the desired phone number in it. In general, when a request (by users or s/w agents), r, has initial input parameters r_{in} and desired output parameters, r_{out}, the following holds:

Proposition 1 (Full-Matching) *A web service w can "fully" match r iff (1) $r_{in} \supseteq w_{in}$, and (2) $r_{out} \subseteq w_{out}$.* □

That is, one can invoke w by using r_{in} since the required input parameters of w_{in} are subset of what is given in r_{in}. Similarly, since the output parameters of w_{out} are superset of what is desired in r_{out}, the goal is achieved.

In practice, however, it is often impossible that one web service can fully satisfy the given request. Then, one has to combine multiple web services that only partially satisfy the request. Given a request r and two web services x and y, for instance, suppose one can invoke x using inputs in r_{in}, but the output of x does not have what we look for in r_{out} (i.e., $r_{in} \supseteq x_{in} \wedge r_{out} \not\subseteq x_{out}$). Symmetrically, the output of y generates what we look for in r_{out}, but one cannot invoke y directly since it expects inputs not in r_{in} (i.e., $r_{in} \not\supseteq y_{in} \wedge r_{out} \subseteq y_{out}$). Furthermore, using initial inputs of r_{in} and the outputs of x, one can invoke y (i.e., $(r_{in} \cup x_{out}) \supseteq y_{in}$). Then, the request r can be satisfied by the flow of: $x_{out} \leftarrow x(r_{in}); y_{out} \leftarrow y(r_{in} \cup x_{out}); r_{out} \leftarrow y_{out}$. Based on this observation follows the following Lemma (proof is omitted):

Lemma 1 (Joint-Matching). *A chain of web services, $w^1 \Rightarrow ... \Rightarrow w^n$, can "jointly" match r, iff (1) $r_{in} \supseteq w^1_{in}$, (2) $(r_{in} \cup w^1_{out} \cup ... \cup w^{i-1}_{out}) \supseteq w^i_{in} (1 \leq i \leq n-1)$, and (3) $(r_{in} \cup w^1_{out} \cup ... \cup w^n_{out}) \supseteq r_{out}$.*

2 Our Approaches

In handling large number of web services, two issues are critical: (1) the frequently-occurring "membership" checking (e.g., $r_{in} \supseteq w_{in}$) needs to be handled efficiently; and (2) an efficient search algorithm to support joint-matching case is needed. To solve these problems, we propose two techniques below.

2.1 Fast Membership Checking w. Bloom Filters

A *Bloom Filter* [1] is a simple space-efficient randomized data structure for representing a set in order to support membership queries efficiently. Since it is based on a myriad of hash functions, it takes $O(1)$ to check the membership, and its space efficiency is achieved at the small cost of

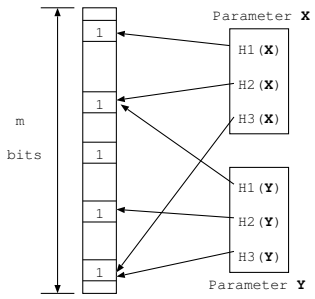

Figure 1. Bloom filter with 2 parameters, X and Y, stored through 3 hash functions.

errors. In the context of our problem, the idea is the following. For a web service, w, with $w_{in} = \{I^1, ..., I^p\}$ and $w_{out} = \{O^1, ..., O^q\}$, two corresponding bloom filters are prepared, BF^w_{in} and BF^w_{out}, respectively, where the bloom filter is a vector of m bits initialized to 0. Further, k independent hash functions, $H_1, ..., H_k$, are given, each with the output range {1, ..., m}, in sync with m bits of bloom filters. Then, each parameter $I^i \in w_{in}$ (resp. $O^j \in w_{out}$) is fed to k hash functions, and each bit at positions $H_1(I^i), ..., H_k(I^i)$ in BF^w_{in} (resp. $H_1(O^j), ..., H_k(O^j)$ in BF^w_{out}) is set to 1. If the bits at some positions were set to 1 by previous hash functions (due to hash collision), do nothing. Once all parameters in w_{in} and w_{out} are processed this way, the bloom filters, BF^w_{in} and BF^w_{out}, become a succinct representation of potentially long list of input and output parameters of a web service. The idea is illustrated in Figure 1.

To check the membership if $X \in w_{in}$, one checks the bits at positions $H_1(X), ..., H_k(X)$ in BF^w_{in}. If any of them is 0, then one concludes that $X \notin w_{in}$ for sure. Otherwise, one concludes that $X \in w_{in}$ with a small probability of being false. To merge two bloom filters, simple OR of two is sufficient. The salient feature of bloom filter is that one can control the probability of false positive by adjusting m, k, p, and q — the precise probability is known as [2]: $\left(1 - \left(1 - \frac{1}{m}\right)^{k(p+q)}\right)^k \approx \left(1 - e^{k(p+q)/m}\right)^k$. For instance, with 5 hash functions (i.e., $k = 5$), 10 bits in bloom filter, and upto 100 input/output parameters (i.e., $\frac{(p+q)}{m} = 10$), the probability becomes mere 0.00943.

2.2 BF*: A* Based Graph Search Algorithm

2.2.1 Stratified Flooding Algorithm

Lemma 1 readily suggests a naive fixed-point algorithm for solving joint-matching case as shown in Algorithm 1. Ω (resp. Σ) is a set of web services that have been visited so far (resp. a set of parameters gathered so far). At each iteration,

let $W \leftarrow$ all web services, $\Omega \leftarrow \emptyset$ and $\Sigma \leftarrow r_{in}$;
print r_{in}, "$\Rightarrow$";
while $\Sigma \not\supseteq r_{out}$ **do**
  $\delta \leftarrow \{w | w \in W, w \notin \Omega, w_{in} \subseteq \Sigma\}$;
  $\Omega \leftarrow \Omega \cup \delta$;
  $\Sigma \leftarrow \Sigma \cup (\bigcup_{w \in \delta} w_{out})$;
  print δ, "$\Rightarrow$";
print r_{out};

Algorithm 1: Stratified Flooding Algorithm

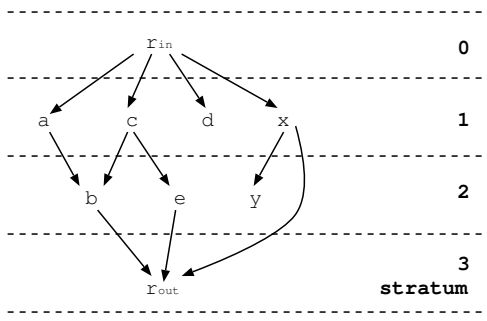

Figure 2. Example lattice.

new set of web services, δ, are found that can be invoked using Σ. Since there are only finite number of web services, W, and each iteration adds only "new" set of web services (i.e., $w \notin \Omega$), the iterations must end. At some point, if $\Sigma \supseteq r_{out}$, then it means that using the parameters gathered so far (i.e., Σ), one can get the desired output parameters in r_{out}, thus solving the joint-matching problem.

In general, our problem can be naturally casted into a partially-ordered set (i.e., lattice), where "$x \Rightarrow y$" means that one can invoke a web service y using the parameters gathered in $\Sigma \cup x_{out}$, and the *least upper bound (lub)* is r_{in} and the *greatest lower bound (glb)* is r_{out}. Note that the function "$\Rightarrow$" in the lattice is "monotonic" (i.e., always downward), and therefore, as Knaster-Tarski Theorem [4] implies, there always exists a fixed point, ensuring the correctness of Algorithm 1. Figure 2 is an example lattice. Using parameters in r_{in} at stratum 0, one can invoke web services a, c, d, and x at stratum 1. Then, $a \Rightarrow b$ means that using parameters in $\Sigma \cup a_{out}$, one can invoke b (i.e., $r_{in} \cup a_{out} \supseteq b_{in}$). Similarly, $e \Rightarrow r_{out}$ means that using parameters in $\Sigma \cup e_{out}$, one can invoke r_{out} (i.e., $r_{in} \cup c_{out} \cup e_{out} \supseteq r_{out}$), reaching the goal.

The naive stratified flooding algorithm in Algorithm 1 is simple but inefficient since at each stratum, it finds "all" web services (i.e., flooding) that can be invoked, and accumulate them into Ω. For instance, for the example of Figure 2, the naive algorithm would generate: (1) Stratum 0: $\Omega \leftarrow \emptyset$, $\Sigma \leftarrow r_{in}$; (2) Stratum 1: $\Omega \leftarrow \{a, b, d, x\}$, $\Sigma \leftarrow r_{in} \cup a_{out} \cup c_{out} \cup d_{out} \cup x_{out}$; (3) Stratum 2:

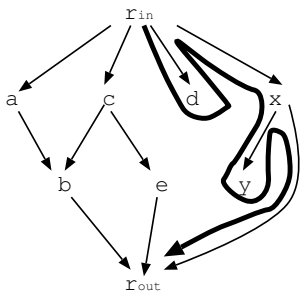

Figure 3. Example of search. Web services d and y are visited unnecessarily, but eventually the path $r_{in} \Rightarrow x \Rightarrow r_{out}$ is found.

$\Omega \leftarrow \{a, b, d, x, b, e, y\}$, $\Sigma \leftarrow r_{in} \cup a_{out} \cup c_{out} \cup d_{out} \cup x_{out} \cup b_{out} \cup e_{out} \cup y_{out}$; and (4) Stratum 3: The goal is reached since $\Sigma \supseteq r_{out}$. Note that any one of the "four" correct solutions could have been sufficient: (1) $r_{in} \Rightarrow a \Rightarrow b \Rightarrow r_{out}$; (2) $r_{in} \Rightarrow c \Rightarrow b \Rightarrow r_{out}$; (3) $r_{in} \Rightarrow c \Rightarrow e \Rightarrow r_{out}$; and (4) $r_{in} \Rightarrow x \Rightarrow r_{out}$. However, Algorithm 1 finds all four solutions unnecessarily, making it less than optimal.

2.2.2 BF* Graph Search Algorithm

Suppose, at i-th stratum, there are N web services that one can invoke. Then, (1) (*sequential mode*) if one can invoke one web service at a time, then there are N choices; and (2) (*parallel mode*) if one can invoke multiple web services together at a time, then there are $2^N - 1$ choices. For instance, in Figure 3, starting from r_{in}, (1) in sequential mode, there are 4 ways to invoke subsequent web services: {a}, {c}, {d}, and {x}; and (2) in parallel mode, there are 15 ways to invoke subsequent web services: {a}, {c}, {d}, {x}, {a,c}, {a,d}, {a,x}, {c,d}, {c,x}, {d,x}, {a,c,d},{a,c,x}, {c,d,x}, and {a,c,d,x}. Algorithm 1 is equivalent to invoking "all" of these choices "always" – making it a correct, but inefficient. Since there are large number of subsequent choices available at each stratum, one needs to pick next choice carefully. For this selection strategy, we propose to use A* algorithm [3].

A* algorithm is a heuristics-based competitive search algorithm. At each state, it considers some heuristics-based cost to pick the next state with the lowest cost. For instance, in Figure 3, starting from r_{in}, one has 4 choices to make in sequential mode. Then, A* will suggest only 1 out of 4 as a next web service to visit based on heuristics. Suppose d was visited. Then, from d, again all possible next choices are computed and one of them is suggested. However, in this case, there is no available next choice to make, thus one has to backtrack to the previous state. Then, another one, say x, out of 4 is suggested as a next move, and so on. When next

let $W \leftarrow$ all web services, $\Omega \leftarrow \emptyset$ and $\Sigma \leftarrow r_{in}$;
print r_{in}, "$\Rightarrow$";
while $\Sigma \not\supseteq r_{out}$ **do**
  $\delta \leftarrow \{w | w \in W, w \notin \Omega, w_{in} \subseteq \Sigma\}$;
  $w^{min} \leftarrow w(\in \delta)$ with $\mathbf{MIN}(f(w))$;
  $\Omega \leftarrow \Omega \cup w^{min}$;
  $\Sigma \leftarrow \Sigma \cup w^{min}_{out}$;
  print w^{min}, "$\Rightarrow$";
print r_{out};

Algorithm 2: BF* Algorithm

move is the goal state, i.e., r_{out}, the search is successful. Since the performance of A* algorithm heavily depends on the quality of the heuristics, it is important to use the right heuristics to strike a good balance between accuracy and speed. In our context, A* algorithm can be captured as follows. Given a set of candidate web services to visit next, N ($\not\subseteq \Omega$), one chooses $n(\in N)$ with the "smallest" **f(n) (= h(n) + g(n))** such that:

$$\mathbf{h}(n) = \frac{1}{|(r_{out} \setminus \Sigma) \cap n_{out}|} \quad (1)$$

$$\mathbf{g}(n) = |\Omega| \quad (2)$$

That is, the remaining parameters of r_{out} that are yet to be found are $r_{out} \setminus \Sigma$. Then, the intersection of this and n_{out} is a set of parameters that n helps to find. The more parameters n finds, the bigger contribution n makes to reach to the goal. Therefore, A* favors the n whose contribution to find remaining parameters is the max (i.e., $h(n)$ is the smallest). In order words, our heuristics is based on the hypothesis that "*visiting a web service with bigger contribution would find the goal faster than otherwise.*" Combining this idea with Bloom Filter of Section 2.1, our main proposal, BF* algorithm, is illustrated in Algorithm 2. The missing operational semantics of BF* algorithm is similar to that of A* algorithm (e.g., using OPEN and CLOSED priority queues or normalizing $h(n)$ and $g(n)$ properly before addition), and omitted here. Note that if we set $h(n) = 0$, then BF* algorithm degenerates to Dijkstra's shortest path algorithm.

References

[1] B. Bloom. "Space/Time Tradeoffs in Hash Coding with Allowable Errors". *Comm. ACM*, 13(7):422–426, 1970.

[2] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. "Summary Cache: A Scalable Wide-Area Web Cache Charing Protocol". *IEEE/ACM Trans. on Networking*, 8(3):281–293, 2000.

[3] S. J. Russell and P. Norvig. "*Artificial Intelligence: A Modern Approach (2nd Ed.)*". Prentice-Hall, 2002.

[4] A. Tarski. "A Lattice-Theoretical Fixpoint Theorem and its Applications". *Pacific J. Math.*, 5:285–309, 1955.