A Family of Bloom Filter Based A* Algorithms for Large-Scale Web Services Composition Problem

Seog-Chan Oh
Penn State / IE
sxol60@psu.edu

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

Dongwon Lee
Penn State / IST
dongwon@psu.edu

Soundar Kumara
Penn State / IE
skumara@psu.edu

Abstract
As web services become more popular, the number of available web services proliferates. As such, 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. Finding an optimal solution in such a setting is generally known as NP-complete, and thus can be doable only for a small number of web services. However, in dealing with large-scale web services composition, since the search space exponentially increases as the number of web services grows, it is important to make a wise decision on the underlying data structures and approximation algorithms. Toward this problem, in this paper, we present a family of solutions, named as BF* (BF-Star), that adopts the competitive A* search algorithm of AI community while utilizing the Bloom Filter as a succinct data structure. Experimental results verify the efficiency of our BF* algorithms and their suggested heuristic functions.

Introduction
Web Services (WS) is a piece of XML-based software interface that can be invoked over the Internet, and can be roughly viewed as a next-generation successor of CORBA or RPC technique. As such, web services are often considered as one of the most important and vital building blocks to fully achieve the vision of the “Semantic Web” (Berners-Lee, Hendler, & Lassila 2001).

Typically, a client program, s/w agent, first finds a server, web services, that can satisfy certain needs from a yellow page (UDDI), and obtain a detailed specification (WSDL) about the service. Then, using the known API and data types in a specification, the client sends a request to the server via a standard message protocol (SOAP), and in return receives a response from the server. Unlike conventional programming interface, web services is self-explanatory. That is, by interpreting XML tags, applications can interpret the meanings of operations and data in an easier way than before. The problem of interest in this paper concerns the first step

Copyright © 2005, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.
More formally, these three ways to satisfy both requests can

$$w = w_w, \text{all input parameters in } A_w$$

findDirection('100 Atherton to get the driving direction.

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., $$r_{in} \supseteq x_{in} \land r_{out} \not\subseteq x_{out}$$). 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., $$r_{in} \not\supseteq y_{in} \land r_{out} \subseteq y_{out}$$). 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: $$x_{out} \leftarrow x(Fin); y_{out} \leftarrow y(Fin \cup x_{out}); r_{out} \leftarrow y_{out}$$ Based on this observation follows the following Lemma:

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

PROOF. Proof by induction (details omitted). (q.e.d)

Problem & Challenge

In this paper, we aim at solving the following problem.

Given a large number of web services, $$W$$, quickly find:
(1) all fully-matching web services, $$X(\subseteq W)$$; or (2) a chain of joint-matching web services, $$Y(\subseteq W)$$, if there are no fully-matching ones.

In particular, the second problem of finding a chain of joint-matching web services is what we refer to as the Web Service Composition problem, and can be essentially formulated as graph planning/search problem. (Kautz & Selman 1992) has shown that graph planning problem can be modeled as the satisfiability (SAT) problem, which is known to be NP-complete (Cook 1971). Therefore, a brute-force exhaustive search algorithm with an exponential runtime complexity may be able to find an optimal web service composition for only a small size input, but not for large-scale input. For instance, one of the state-of-the-art solutions (Vossen et al. 1999) solves the planning problem by exploiting the integer linear programming (ILP) problem, and reportedly takes about 1,400 seconds for a small size of 285 nodes input.

Since we aim at solving the web services in the range of 1,000 to 10,000, such existing solutions are prohibitively expensive and are not feasible. Therefore, in the following, we investigate alternative non-exhaustive but approximate solutions (based on A* that can find sub-optimal web service compositions. In addition, for practical concern, we also

Table 1: Example web services.

16801, PA’, ‘410 S. Allen St. 16802, PA’) to get the driving direction.

Second, invoke findRestaurant('16801

Thai') to get the address of the closest restaurant, say “410 S. Allen St. 16802, PA’. Then, invoke the web service findDirection('100 Atherton Ave, 16801, PA’, ‘410 S. Allen St. 16802, PA’) to get the driving direction.

More formally, these three ways to satisfy both requests can be captured in Datalog notation as in Table 2.

A web service, $$w$$, has typically two sets of parameters: $$w_{in} = \{I^1, I^2,...\}$$ for SOAP request (as input) and $$w_{out} = \{O^1, O^2,...\}$$ for SOAP response (as output). When $$w$$ is invoked with all input parameters, $$w_{in}$$, it returns the output parameters, $$w_{out}$$. In general, to be able to invoke $$w$$, all input parameters in $$w_{in}$$ must be provided (i.e., $$w_{in}$$ are mandatory). 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 $(1) r_{in} \supseteq w_{in}, \text{and} (2) r_{out} \subseteq w_{out}$. ⊢

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.

Problem & Challenge

In this paper, we aim at solving the following problem.

Given a large number of web services, $$W$$, quickly find:
(1) all fully-matching web services, $$X(\subseteq W)$$; or (2) a chain of joint-matching web services, $$Y(\subseteq W)$$, if there are no fully-matching ones.

In particular, the second problem of finding a chain of joint-matching web services is what we refer to as the Web Service Composition problem, and can be essentially formulated as graph planning/search problem. (Kautz & Selman 1992) has shown that graph planning problem can be modeled as the satisfiability (SAT) problem, which is known to be NP-complete (Cook 1971). Therefore, a brute-force exhaustive search algorithm with an exponential runtime complexity may be able to find an optimal web service composition for only a small size input, but not for large-scale input.

For instance, one of the state-of-the-art solutions (Vossen et al. 1999) solves the planning problem by exploiting the integer linear programming (ILP) problem, and reportedly takes about 1,400 seconds for a small size of 285 nodes input. Since we aim at solving the web services in the range of 1,000 to 10,000, such existing solutions are prohibitively expensive and are not feasible. Therefore, in the following, we investigate alternative non-exhaustive but approximate solutions (based on A* that can find sub-optimal web service compositions. In addition, for practical concern, we also
investigate an efficient data structure to expedite the search process.

As the dynamic discovery and composition of web services become more important in recent years, there appear a few relevant works (McDermott 2002; Thakkar et al. 2002). For instance, METEOR-S (Verma et al. 2003) allows designers to build web services based on business/process constraints, but its composition is still manual. On the other hand, Proteus (Ghandeharizadeh et al. 2003) is a framework that composes and executes a plan incorporating available web services. Although their execution is automatic, web service plan is composed rather manually. Such a manual composition is viable for a small number of carefully selected services. However, we believe that for a large-scale web service composition, automatic composition is a must. (Thakkar et al. 2002) is also similar in the sense that automated service composition is only achieved by semi-automatically modeling services as web information sources via wrappers.

**Main Ideas**

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 – fast membership checking using bloom filters and A*-based graph search algorithms.

**Fast Membership Checking with Bloom Filters**

The Bloom Filter (Bloom 1970) 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 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, \( BF_{in} \) and \( BF_{out} \), 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 \( I^p \) in \( w_{in} \) (resp. \( O^1 \) in \( w_{out} \)) is fed to \( k \) hash functions, and each bit at positions \( H_1(I^p), ..., H_k(I^p) \) in \( BF_{in} \) (resp. \( H_1(O^1), ..., H_k(O^1) \) in \( BF_{out} \)) 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, \( BF_{in} \) and \( BF_{out} \), become a succinct representation of potentially long list of input and output parameters of a web service.

To check the membership if \( X \in w_{in} \), one checks the bits at positions \( H_1(X), ..., H_k(X) \) in \( BF_{in} \). 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 XOR 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 (Fan et al. 2000):

\[
\left(1 - \left(1 - \frac{1}{m}\right)^k\right)^q \approx \left(1 - e^{k(p+q)/m}\right)^k.
\]

For instance, with 5 hash functions (i.e., \( k = 5 \)), 10 bits in bloom filter, and up to 100 input/output parameters (i.e., \( q = 100 \)), the probability becomes mere 0.00943.

**BF*: A* Based Graph Search Algorithm**

1. **Stratified Flooding Algorithm**

   Lemma 1 readily suggests a naive fixed-point algorithm for solving joint-matching case as shown in Algorithm 1. \( \Omega \) (resp. \( \Sigma \)) is a set of web services that have been visited so far (resp. a set of parameters gathered so far). At each iteration, new set of web services, \( \delta \), are found that can be invoked using \( \Sigma \). 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., \( \Sigma \)), one can get the desired output parameters in \( r_{out} \), thus solving the joint-matching problem.

<table>
<thead>
<tr>
<th>Request</th>
<th>Invocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r_1(\text{HotelAddr}) )</td>
<td>( X(\text{HotelAddr}, \text{HotelZip}) \leftarrow \text{findHotel}(‘\text{Atherton’}, ‘\text{State College’}, ‘\text{PA}’) )</td>
</tr>
<tr>
<td>( r_2(\text{Direction}) )</td>
<td>( Y(\text{Rating}, \text{RestAddr}) \leftarrow \text{guideRestaurant}(‘\text{Thai’}, \text{HotelAddr}) )</td>
</tr>
<tr>
<td>( r_2(\text{Direction}) )</td>
<td>( Z(\text{Map}, \text{Direction}) \leftarrow \text{findDirection}(\text{HotelAddr}, \text{RestAddr}) )</td>
</tr>
</tbody>
</table>

Table 2: Three ways to satisfy two requests in Datalog notation.
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 (Tarski 1955) implies, there always exists a fixed point, ensuring the correctness of Algorithm 1. Figure 1 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., \( r_{in} \cup e_{out} \cap e_{out} \supseteq r_{out} \)), 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 \( \Omega \). For instance, for the example of Figure 1, the naive algorithm would generate:

- **Stratum 0**: \( \Omega \leftarrow \emptyset \), \( \Sigma \leftarrow r_{in} \);
- **Stratum 1**: \( \Omega \leftarrow \{a, b, d, x\} \), \( \Sigma \leftarrow r_{in} \cup a_{out} \cup e_{out} \cup d_{out} \cup x_{out} \);
- **Stratum 2**: \( \Omega \leftarrow \{a, b, d, x, b, c, y\} \), \( \Sigma \leftarrow r_{in} \cup a_{out} \cup c_{out} \cup d_{out} \cup x_{out} \cup b_{out} \cup e_{out} \cup y_{out} \); and
- **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. BF* 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 1, 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 (Russell & Norvig 2002).

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 1, 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 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” \( h(n) = h(n) + g(n) \) such that:

\[
h(n) = \frac{1}{|\left( r_{out} \cup \Sigma \right) \cap n_{out}|} \quad (1)
\]

\[
g(n) = |\Omega| \quad (2)
\]

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 following hypothesis:

**Hypothesis**: Visiting a web service with bigger contribution earlier would find the goal faster than otherwise.

Combining this idea with the aforementioned Bloom Filter, 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.

3. Lookahead BF* Algorithm A popular way to improve the heuristic algorithm is to make it more “informed.” Since BF* chooses one as the next web service to invoke among \( N \not\subseteq \Omega \) candidates, one way to get it more informed is to “lookahead” further. That is, in addition to considering one’s \( h(n) \) value, one may look at the \( h(n) \) value of their children so that BF* chooses the web service whose combined (i.e., itself and its child) contribution is the maximum. This can be captured as the following \( h(n) \):

\[
h(n) = \frac{1}{|\left( r_{out} \cup \Sigma \right) \cap (n_{out} \cup c(n)_{out})|} \quad (3)
\]
with 650, 1,300, 3,250, 4,550 and 6,500 web services, respectively. Since we need to know the optimal solution for each case to compare our solution against, we generated those data sets as follows. We prepared ten jars with each having five parameters. Each parameter consists of one of the alphabets “a” to “j” as prefix, followed by one of the numbers 1 to 5 as suffix (as shown in Figure 2). Parameters belonging to same jar have the common prefix with different suffixes. Then, we generated thirteen web services clusters (i.e., links \( W \) to \( W_{13} \) in Figure 2) that connect from a jar \( A \) to another jar \( B \). All web services in the cluster are generated using a random number parameters in \( A \) as “input” and those in \( B \) as “output”. For instance, a web service, \( w_{112} \) belongs to the cluster \( W_1 \) in Figure 2, and thus has input parameters of \( \{ a_1, a_4 \} \) and output parameters of \( \{ b_2, b_1 \} \).

As test queries, we prepared the five composition goals: \( a_1 \rightarrow j_5, \{ d_4, e_1 \} \rightarrow j_5, a_3 \rightarrow \{ h_2, h_5 \}, \{ e_4, b_4 \} \rightarrow \{ h_1, h_2, j_2, j_5 \}, \) and \( \{ c_5, f_2 \} \rightarrow \{ d_4, j_5 \} \). For instance, for the goal \( a_1 \rightarrow j_5 \), initially parameter \( a_1 \) is given, and one needs to find a composed web service that can return \( j_5 \) as output.

Finally, to see the effect of distributions of parameters among web services, we used three distribution models in generating test cases: skewed, uniform, and normal distributions. In the skewed distribution, 90% of web services have only one input and output parameters, and 4%, 3%, 2% and 1% of web services have 2, 3, 4 and 5 input and output parameters, respectively. In the normal distribution, 50% of web services have 3 input and output parameters, and 20%, 20%, 10% and 10% of web services have 1, 2, 4 and 5 input and output parameters, respectively. Finally, in the uniform distribution, the same ratio of web services have the same number of parameters in a uniform way.

At the end, overall, we experimented a combination of 5 data set sizes \( \times \) 5 test queries \( \times \) 3 distributions \( \times \) 4 algorithms = 300 combinations.

### Results

Figure 3(a)-(c) summarizes the search time of our proposals on three distribution models, where all algorithms perform reasonably fast, in less than 50 seconds even for 6,500 web services. The only exception to this is the case of lookahead for normal distribution models. Nevertheless, it takes about 330 seconds to finish all. Since lookahead BF* algorithm needs to search all children’s estimated costs (i.e., one more level) for each node, its running time rapidly increases as the data sizes increases. Interestingly, this phenomena does not occur for the other two distribution models. This is somewhat intuitive since in the skewed and uniform models, the degree of overlaps between web services are low, and per each node, thus there are substantially fewer number of children to estimate \( h(n) \). Figure 3(d)-(f) illustrates the path length of the solution that our algorithms find, compared to the optimal solution that we pre-determined. Consistently, flooding algorithm performs worse than others, and in particular adaptive algorithm shows “almost” as good solution as optimal one. Also, for all cases, lookahead and adaptive slightly outperforms the regular BF* since they are in a sense more “informed.”

### Experimental Validation

#### Set-up

To validate the efficacy of our proposals, we conducted extensive experiments. First, we build five data sets...
Figure 3: Comparison of search time and path length for three web services distribution models.

**Conclusion**

In this paper, we have considered the problem of locating an efficient web service composition, and suggested a solution using (1) a succinct data structure known as Bloom Filter, and (2) A* based heuristic algorithms. Despite its exponential time and space complexity for worst case, in practice, our proposals show good performance and accuracy.

Much direction is ahead for further research. For instance, other QoS factors than the path length of solution that we used can be considered (e.g., response time, cost). As it is, we have considered the web service composition only at the “syntactic” level. More difficult and rewarding scenario is to be able to invoke web services if their “semantic” input/output parameters are matched. Finally, more complex constraints (e.g., AND, OR, XOR) than the simple sequential invocation that we considered can be enforced in the web service invocation.

**References**


