Efficient anytime algorithm for large-scale QoS-aware web service composition

Hyunyoung Kil

The Research Institute of Computer Information and Communication,
Korea University,
Seoul 136-701, Korea
Email: hykil@korea.ac.kr

Wonhong Nam*

Department of Internet & Multimedia Engineering,
Konkuk University,
Seoul 143-701, Korea
Email: wnam@konkuk.ac.kr
*Corresponding author

Abstract: The *QoS-aware web service composition* (*WSC*) problem aims at the fully automatic construction of a composite web service with the optimal accumulated QoS value. It is, however, intractable to solve the QoS-aware WSC problem for large scale instances since the problem corresponds to a global optimisation problem. That is, in the real world, traditional algorithms can require significant amount of time to finally find the optimal solution, and such an unexpected long delay is unfavourable to users. In this paper, we propose a novel *anytime algorithm* using *dynamic beam widths* for the QoS-aware WSC problem. Our algorithm generates early sub-optimal solutions and keeps improving the quality of the solution along with the execution time, up to the optimal solution if a client allows enough time. We empirically validate that our algorithm can identify composite web services with high quality much earlier than an optimal algorithm and the *beam stack search*.

Keywords: anytime algorithm; web service composition; QoS; quality of services; optimisation.

Reference to this paper should be made as follows: Kil, H. and Nam, W. (2013) 'Efficient anytime algorithm for large-scale QoS-aware web service composition', *Int. J. Web and Grid Services*, Vol. 9, No. 1, pp.82–106.

Biographical notes: Hyunyoung Kil is currently a Postdoctoral Researcher at The Research Institute of Computer Information and Communication in Korea University. She received the BS and MSc degrees from Korea University, Seoul, Korea, in 1998 and 2001, respectively. She received the MSE degree from the University of Pennsylvania, Philadelphia, PA, USA in 2003, and the PhD degree from the Pennsylvania State University, State College, PA, USA in 2010. Her research interests include automated planning, web services composition, SOA and web science.

Wonhong Nam received the BS and MSc degrees from Korea University, Seoul, Korea, in 1998 and in 2001, respectively, and the PhD degree from the University of Pennsylvania, Philadelphia, PA, USA in 2007. From 2007 to 2009, he was a Postdoctoral Researcher with the College of Information

Sciences and Technology, Pennsylvania State University, University Park, PA, USA. He is currently an Assistant Professor of the Department of Internet & Multimedia Engineering, Konkuk University, Seoul, Korea. His research interests include formal methods, formal verification, model checking, automated planning, and web services composition.

1 Introduction

Web services are software systems designed to support machine to machine interoperation over the Internet. Many researches (Web services activity, 2002; WSLA, 2004; WS-Policy, 2006; WS-Agreement, 2007) have been carried out for the web service standards based on semantic web techniques, significantly improving the flexible and dynamic functionalities of the *service oriented architectures* (*SOA*). However, abundant research challenges still remain (Hull and Su, 2005); e.g. automatic web service discovery (Benatallah et al., 2005), web service composition (Aggarwal et al., 2004; Ardagna and Pernici, 2007; Kil et al., 2009; Wang et al., 2011), or formal verification of composed web services (Narayanan and McIlraith, 2002; Fu et al., 2004).

In general, the *web service composition* (*WSC*) problem is, given a set of web services and a user request, to find the shortest sequence of web services satisfying the request. Recently, the WSC problem requires discovering service providers that satisfy not only functional requirements but also nonfunctional ones, including *Quality of Services* (*QoS*) constraints. In this case, one indeed desires *not* the shortest sequence of web services as a solution *but* the composite web service with the optimal accumulated QoS value. This problem is called the *QoS-aware WSC* problem. It is, however, computationally hard to identify such a composite web service since the problem is a global optimisation problem. That is, if the number of web services is large, as real web services on the Web, the problem is intractable. If we try to solve the problem by using traditional optimal algorithms which generate a single optimal solution after a fixed number of computations, it may take a long time until completion. Sometimes, even after a long waiting, optimal algorithms may not provide a solution due to a lack of system resource. Needless to say, such an unexpected long delay is not allowable in the real e-business scenario.

For the scalability, selection-based web service composition techniques have been suggested (Alrifai and Risse, 2009; Alrifai et al., 2010). They predefine (or require for a client to create) a task-flow which consists of subtasks to achieve a whole request and then locally select the best web service among the candidate web services for each subtask. Although they can generate a near-optimal solution successfully in a large sized problem instance, they need a manual or semi-automatic task-flow construction process first. In addition, due to their local search mechanism based on the fixed task-flow, they cannot guarantee the global optimal solution. However, in general, a client wants to choose the quality of solution based on his/her preference and situation. If the QoS of the composite service is critical, clients may be willing to endure even a lengthy wait for the most optimised QoS value of the composite service. On the other hand, if enough time is not allowed or the QoS of the composite web service is not the most decisive factor, the client would rather a reasonable sub-optimal solution within a given time-out than the optimal solution requiring a long time. Therefore, to guarantee both a fully automatic composition and client's choice for the quality of the solution, we propose a novel anytime algorithm to the QoS-aware WSC problem.

Anytime algorithms are a pragmatic method to trade the quality of solutions against the cost of computation. They provide available best-so-far answers (i.e. approximations) as soon as each one is found. In most of cases, anytime algorithms can produce early sub-optimal solutions, and moreover the quality of results improves gradually along with the execution time. Finally, if enough time is allowed, the answer converges to the optimal solution eventually. During the execution of anytime algorithms, the user can examine at anytime the quality of the answer returned, and decide whether to terminate with satisfaction or to continue its journey to better answers. Due to these capabilities, anytime algorithms have been successfully employed in AI and real-time system literatures. However, to the best of our knowledge, there is no work to employ an anytime algorithm to the WSC problems.

In this paper, we propose two *anytime algorithms* for the QoS-aware WSC problem: *the basic anytime algorithm* and *the dynamic anytime algorithm*. By using our anytime algorithms for the QoS-aware WSC problem, we can identify composite web services with high quality earlier than optimal algorithm. Our basic anytime algorithm proposed in this paper is based on *beam stack search* (Zhou and Hansen, 2005), which explores a fixed number of candidate states in each level. In the searching mechanism of the beam stack search, a bad decision in an earlier phase pays more penalties than a bad selection in a later phase does. To reduce this cost, given a problem instance, our dynamic anytime algorithm assigns a larger beam width to early phases so that we can consider more qualified candidates at the beginning of the composition. In addition, we propose several heuristics to improve the quality of current approximate solutions and overall execution time. We validate our proposal with experiment on a number of examples that are generated by the web service test set generator employed in Web Services Challenge 2009 (Kona et al., 2009).

The rest of this paper is organised as follows. In Section 2, we discuss related work with our research. Next, in Section 3 we lay out the notions and the problem we study in this paper. We then present our basic anytime algorithm and dynamic anytime algorithm for the QoS-aware WSC problem in Section 4 and Section 5, respectively. In Section 6, we present the experimental result to validate our algorithms proposed in this paper. Finally, we give conclusions in Section 7.

2 Related work

Recently, there have been a number of standardisation efforts on QoS specification of web services. The *WS-Policy* (2006) is a proposed framework to express the capability, requirements, and general QoS characteristics for each service. The *Web Service Level Agreement* (*WSLA*) *language* (2004) that IBM has proposed defines assertions of a service provider according to agreed guarantees for QoS parameters and methods to handle deviation and failure in order to meet the asserted service guarantees. In addition, the *grid resource allocation agreement protocol working group* is developing standards such as *WS-Agreement* (2007) to negotiate and manage services based on QoS attributes.

Based on these standards for the QoS of web services, the QoS-aware web service composition (WSC) also has gained the attention. To find the best composition of web services satisfying user's constraints, various optimisation algorithms such as linear programming, genetic algorithm and database query techniques are applied (Aggarwal et al., 2004; Cardoso et al., 2004; Zeng et al., 2004; Canfora et al., 2005; Srivastava et al.,

2006; Yu and Bouguettaya, 2012). The METEOR-S project (Aggarwal et al., 2004; Cardoso et al., 2004) deals with the QoS-aware WSC problem. It presents an integer linear programming technique to generate an optimal execution plan, but emphasises on dynamic configuration and exception handling. Zeng et al. (2004) also use a linear programming approach to calculate a global optimised QoS value of a composite web service. Ardagna and and B. Pernici (2007) extend the linear programming model to include local constraints. Canfora et al. (2005) present a method using genetic algorithms for a web service composition problem, which results in a better performance and scalability than linear integer programming. Srivastava et al. (2006) apply a traditional database query optimisation technique to find a web service composition which has the minimum value in the total response time by ordering or pipelining the operations to participating web services. Yu and Bouguettaya (2012) propose a multi-attribute optimisation technique, called service skyline computation, that returns a set of most interesting service providers.

However, these approaches still show limitations in real-time e-business scenario with increasing numbers of candidate web services. For the better efficiency in a large-scale environment, Yu et al. (2007) propose heuristic algorithms to find a near-to-optimal solution faster. Alrifai et al. (2009, 2012) present an algorithm to combine global optimisation with local selection techniques; it first decomposes a given global QoS constraint into local constraints by using mixed integer programming (MIP), and then selects the best web services that satisfy these local constraints by a distributed local selection method. Alrifai et al. (2010) propose a skyline notion-based approach for efficient service selection. These researches are based on QoS-aware selection models where the top-level work flow is a *manually* or *semi-automatically* predefined arrangement of sub-functionalities required to achieve the whole request and the best web service is selected among each sub-functionality group. After all, these methods cannot guarantee that the answer is optimal. Moreover, any work does not consider anytime algorithms.

Since the term, *anytime algorithm*, was coined by Boddy and Dean (1989) in 1980's, a number of anytime algorithms have been proposed and successfully applied to the applications in the AI areas considering resource bounded systems. Recently, some interesting work about complete anytime algorithms (Zhou and Hansen, 2005; Aine et al., 2007; Richter et al., 2010) has been introduced. Zhou and Hansen (2005) transform the beam search into a complete search algorithm by integrating with a backtracking mechanism. Aine et al. (2007) present an iterative anytime heuristic search algorithm called anytime window A* (AWA*), where node expansion is localised within a sliding window comprising of levels of the search graph. Richter et al. (2010) suggest to restart the search from the initial state whenever a new solution is found by a heuristic strategy. Thayer and Rumi (2010) present an empirical evaluation of anytime algorithms including sub-optimal based anytime searches and state-of-the-art anytime search algorithms such as beam stack search and AWA* algorithm, across a set of common benchmark domains.

3 QoS-aware web service composition

The *Quality of Services* (*QoS*) is considered as various non-functional properties of a service, such as response time, throughput, capacity and accuracy. As a natural price of a

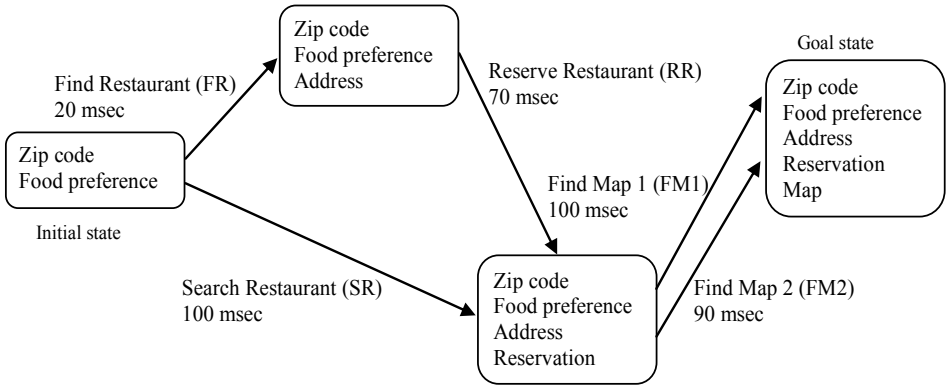
web service, QoS helps clients (applications as well as human being) select a service provider with good quality among many candidates with similar functionality. Moreover, enhanced QoS of web services will bring a highly competitive advantage for web service providers. This implies that users and providers need to be able to engage in QoS negotiation. In this section, we formally define the QoS-aware web service composition problem.

3.1 Example: restaurant reservation system

Consider that a client wants to find and reserve a restaurant, and to get a map to the restaurant. She wants to search a restaurant by a zip code (e.g. '305701') and a food preference (e.g. 'Chinese'), and her QoS requirement is fastest response time. Assume that there does not exist such a single web service to find/reserve a restaurant and provide a map but there are a number of various services for restaurants and maps with different response times. In this case, needless to say, the client wants to compose a set of web services to achieve her whole request, which has the minimum aggregated response time.

Figure 1 illustrates this example. Initially, she has information on a *zip code* which represents the area she wants and a *food preference* which she wants to eat. The Find Restaurant (FR) service returns a restaurant address for a given zip code and food preference. When the Reserve Restaurant (RR) service receives a restaurant address, it makes a reservation for the restaurant. On the other hand, the Search Restaurant (SR) service can find *and* reserve a restaurant for a given zip code and food preference. Once the Find Map 1 (FM1) and Find Map 2 (FM2) receive an address, they both provide a map for the address. The response time for each web service is as follows: R_{FR} = 20 *msec*, R_{RR} = 70 *msec*, R_{SR} = 100 *msec*, R_{FM1} = 100 *msec*, and R_{FM2} = 90 *msec*, respectively. In this example, we have four sequences to find/reserve a restaurant and provide a map: i.e. FR-RRFM1, FR-RR-FM2, SR-FM1, and SR-FM2. If considering the length of composite web services as our aim, SR-FM1, and SR-FM2 would be the best compositions. However, since the minimum response time is our goal, FR-RR-FM2 is the optimal composition (i.e. $R_{FR–RR–FM2}$ = 180 *msec*).

Figure 1 Restaurant reservation system



3.2 QoS-aware web service composition problem

Now, we formalise the notion of web services with QoS criteria and their QoS-aware composition that we consider in this paper. A *web service* is a tuple $w(I;O;Q)$ with the following components:

- I is a set of *input parameters* for w.
- O is a set of *output parameters* for w; each input/output parameter $p \in I \cup O$ has a type t_p.
- Q is a set of *quality criteria* for w.

When a web service $w(I;O;Q)$ is invoked with all the input parameters $i \in I$ with the type t_i, it returns all the output parameters $o \in O$ with the type t_o. The invocation of the web service w corresponds to each service quality criterion $q \in Q$, for instance, a response time or a throughput. Given two types t_1 and t_2, t_1 is a *subtype* of t_2 (denoted by $t_1 <: t_2$) if t_1 is more informative than or equal to t_2 so that t_1 can substitute for t_2 everywhere. In this case, t_2 is a *supertype* of t_1. This relation is reflexive (i.e. $t <: t$ for any type t) and transitive (i.e. if $t_1 <: t_2$ and $t_2 <: t_3$ then $t_1 <: t_3$). Given two web services $w_1(I_1;O_1;Q_1)$ and $w_2(I_2;O_2;Q_2)$, we denote $w_1 \sqsupseteq_I w_2$ if w_2 requires less informative inputs than or the same inputs with w_1; i.e. for every $i_2 \in I_2$, there exists $i_1 \in I_1$ such that $t_{i1} <: t_{i2}$. Given two web services $w_1(I_1;O_1;Q_1)$ and $w_2(I_2;O_2;Q_2)$, we denote $w_1 \sqsubseteq_O w_2$ if w_2 provides more informative outputs than or the same outputs with w_1; i.e. for every $o_1 \in O_1$, there exists $o_2 \in O_2$ such that $t_{o2} <: t_{o1}$ Note that $\sqsupseteq_I$ and $\sqsubseteq_O$ are both reflexive. A *web service discovery problem* is, given a set W of available web services and a request web service w_r, to find a web service $w \in W$ such that $w_r \sqsupseteq_I w$ and $w_r \sqsubseteq_O w$. Intuitively, the solution web service w requires less inputs than or the same inputs with w_r, and provides more outputs than or the same outputs with w_r.

However, it might happen that there is no single web service satisfying the requirement. In this case, we want to find a sequence $w_1 \cdots w_n$ of web services such that we can invoke the next web service in each step and achieve the desired requirement eventually. Formally, we extend the relations, $\sqsupseteq_I$ and $\sqsubseteq_O$, to a sequence of web services as follows.

- $w \sqsupseteq_I w_1 \cdots w_n$ (where $w = (I; O;Q)$ and each $w_j = (I_j ;O_j ;Q_j)$) if $\forall 1 \leqslant j \leqslant n$: for every $i_2 \in I_j$ there exists $i_1 \in I \cup \bigcup_{k<_j} O_k$ such that $t_{i1} <: t_{i2}$ .Intuitively, we need less input parameters to invoke $w_1 \cdots w_n$ than w.

- $w \sqsubseteq_O w_1 \cdots w_n$ (where $w = (I;O;Q)$ and each $w_j = (I_j ;O_j ;Q_j)$) if for every$o_1 \in O$ there exists $o_2 \in \bigcup_{1 \leqslant j \leqslant n} O_j$ such that $t_{o2} <: t_{o1}$. In other words, we acquire more outputs by calling $w_1 \cdots w_n$ than w.

Finally, given a set W of available web services and a service request w_r, the *QoS-aware WSC problem* $\langle W;w_r \rangle$ which we focus on in this paper is to find a sequence $w_1 \cdots w_n$ (every $w_j \in W$) of web services such that $w_r \sqsupseteq_I w_1 \cdots w_n$ and $w_r \sqsubseteq_O w_1 \cdots w_n$. Intuitively, the sequence $w_1 \cdots w_n$ of web services requires less inputs than or the same inputs with w_r, and provides more outputs than or the same outputs with w_r. The optimal solution for this problem is such a sequence $\sigma = w_1 \cdots w_n$ to minimise the aggregate QoS value $Q(\sigma)$. Given a sequence $\sigma = w_1 \cdots w_n$, the aggregate QoS value $Q(\sigma)$ is computed as follows:

- $Q(\sigma)$ $c_1 \cdot Q_1(\sigma) + \cdots + c_m \cdot Q_m(\sigma)$ where each c_i is a given weight for the i-th quality criterion.
- Each function Q_i depends on the corresponding quality criterion. For instance, consider response time as the quality criterion. If $\sigma = w_1$ (i.e. $|\sigma| = 1$), then $Q_i(\sigma) = rt_{w1}$ where rt_{w1} is the response time of w_1. Otherwise (i.e. $\sigma = w_1 \cdots w_n$, $n > 1$), $Q_i(\sigma) = rt_{w1} + Q_i(w_2 \cdots w_n)$. On the other hand, consider throughput. If $\sigma = w_1$, then $Q_i(\sigma) = th_{w1}$ where th_{w1} is the throughput of w_1. Otherwise (i.e. $|\sigma| > 1$), $Q_i(\sigma) = \text{Min}(th_{w1}, Q_i(w_2 \cdots w_n))$.

4 Basic anytime algorithm for QoS-WSC problem

Our proposed algorithms are based on the *beam stack search algorithm* (Zhou and Hansen, 2005) which is a state-of-the-art anytime search algorithm grounded on the *beam search* (Russell and Norvig, 2003). In this section, we first introduce the beam search and the beam stack search algorithm. We then propose our basic anytime algorithm for the QoS-aware WSC problem.

4.1 Beam search

The *beam search* (Russell and Norvig, 2003) is a well-known approximate search algorithm which explores *not* all the nodes *but* only a set of candidate nodes in each search level. We denote the set of nodes to be explored in each level as a *beam* and the size of the set as the *beam width*. Since the beam search uses a fixed beam width, the space and time complexities are linear in the depth of the search, instead of exponential in the depth of the search. Hence, the beam search is widely applied to problems requiring huge search spaces. However, its space-restricted search induces a serious drawback–incompleteness. By pruning nodes inadmissibly without any rationale, it may miss the optimal solution.

Algorithm 1 illustrates the beam search. For each search level, while any node to be searched exists (line 5), the algorithm repeats to select the most promising node at the current level, which has the minimum cost among the open nodes (line 6). It then checks whether it is a goal. If the node is a goal, the algorithm returns the path from the initial node to the node as a solution (lines 8–9). Otherwise, the algorithm expands the node with its successors (line 11). If the expansion induces the number of candidate nodes in the next level to exceed a given beam width, the algorithm prunes the next layer (lines 12–15). Once the beam search completes the search for each level, it checks whether there exist some nodes to be explored in the next level (line 18). If there is no node to be searched, then the algorithm returns *null* (line 19), which reports that it has not been able to find any path from the initial node to the goal node. Otherwise (i.e. we still has some nodes in the next level), it increases the current level index and repeats the loop.

4.2 Beam stack search

Since the beam search explores only some prominent nodes in each level, it is able to identify a solution early but may miss the optimal solution. To overcome the drawback upholding efficiency of the beam search, the *beam stack search algorithm* (Zhou and Hansen, 2005) repeats iteratively the beam search until it identifies the optimal solution. Once the beam stack search completes one iteration of the beam search, it records the progress of the search and continues the next iteration to find a better solution. To keep track of which nodes have been visited so far, the beam stack search manages the *beam stack* which contains an element for each level of the search graph. The element, $[c_{\min}; c_{\max})$, of the beam stack represents that for corresponding level only successor nodes with a cost in the range are expanded in the next level; we call $c_{\min}$ the *lower bound* and $c_{\max}$ the *upper bound*. That is, when generating successor nodes in a layer associated with a beam stack element $[c_{\min}; c_{\max})$, the beam stack search prunes any successor nodes with a cost less than $c_{\min}$, or greater than or equal to $c_{\max}$. When the algorithm expands successors in a layer for the first time, it pushes the beam stack element, $(0, U)$, for the corresponding layer where U is an allowed upper bound of the cost a user provides. If there is no information on the upper bound of the cost, we may use the infinity as U. During an iteration of the beam search, if the number of successors expanded is greater than a given beam width, the beam stack search prunes inadmissibly them within the beam width. To record this inadmissible pruning, we modify the upper bound of the corresponding beam stack element with the best cost among discarded nodes. This modification means that we have explored only the nodes of which cost is less than the new upper bound. In this case, even when the algorithm completes an iteration of the beam search, we cannot guarantee that the current solution is optimal due to inadmissible pruning. Hence, it backtracks to the last level where inadmissible pruning has occurred. Whenever the beam stack search backtracks to a layer, it shifts the range of the stack element from $[c_1; c_2)$ to $[c_2; U)$ in order to admit the next set of successors. A layer is denoted as *backtracking-complete* if $c_{\max}$ is greater than or equal to the cost of the current solution. It means that in the corresponding level all the successors of which cost is less than the current solution are already visited. The beam stack search terminates when all the layers are backtracking-complete. Finally, we can declare that the so-far-best solution is optimal since for every level all the nodes of which cost is less than the current solution are explored.

Algorithm 1: Beam Search

Input : an initial node n_0 and a set G of goal nodes.
Output: a sequence of nodes.

```
1 level := 0;
2 Open[0] :={n0};
3 Open[1] := ∅;
4 repeat
5    while Open[level] ≠ ∅ do
6       node := argmin{cost(n) | n ∈ Open[level]};
7       Open[level] := Open[level]\{node};
8       if (node ∈ G) then
9          return solutionReconstruction(node);
10      else
11         node:generateSuccessors(level + 1);
12         if w < layerSize(level + 1) then /* w is a given beam width. */
13            Keep := the best w nodes ∈ Open[level + 1];
14            Open[level + 1] := Keep;
15      end if
16   end if
17 end while
18 if Open[level + 1] = ∅ then
19    return null;
20 else
21    level := level + 1;
22    Open[level + 1] := ∅;
23 end if
24 until;
```

Algorithm 2: Beam Stack Search

Input : an initial node n_0, a set G of goal nodes, and an allowed upper cost U.
Output: a sequence of nodes.

```
1 beamStack:push([0;U));
2 bestPath := null;
3 while beamStack.top() ≠ null do
4   path := Search(n0, G, U);
5   if path ≠ null then
6      bestPath := path;
7      U := cost(path);
8      print path;
9   end if
```

```
10  while beamStack:top().c_max ⩾ U do
11      beamStack.pop();
12  end while
13  if beamStack:isEmpty () then
14    print "bestPath is optimal.";
15    return bestPath;
16  end if
17  beamStack:top():c_min := beamStack:top(q:cmax ;
18  beamStack:top():c_max := U;
19 end while
```

Algorithm 2 describes the beam stack search which iteratively invokes a search function in Algorithm 3. In this paper, we employ a divide-and-conquer version of the beam stack search proposed explicitly in Zhou and Hansen (2005). The beam stack search has three inputs, i.e. an initial node n_0, a set G of goal nodes, and an allowed upper bound U of the cost. It first pushes a beam stack element, $[0;U)$, for the initial level (line 1). While any entry of the beam stack exists, the algorithm tries to find a better answer of which cost is lower than the current U by calling Search() (lines 3–19). Since the search function always tries to find a better solution, once it returns any path, we can consider the path as a best-so-far answer (lines 5–9). After that, we clear all the backtracking-complete layers in the beam stack, by popping stack elements of which c_{max} is greater than or equal to U (lines 10–12). Then, if the beam stack is empty, the algorithm terminates with the claim that the best-so-far answer is an optimal solution (lines 13-16). Otherwise, it updates the lower and upper bounds at the top item of the beam stack as $[c_{max};U)$ (lines 17–18). By this shifting, in the next iteration the algorithm will search nodes of which cost is in the range between the best cost among previously discarded nodes and the cost of the current solution.

Algorithm 3: Search

Input : an initial node n_0, a set G of goal nodes, and an allowed upper cost U.

Output: a sequence of nodes.

```
1 level := 0;
2 Open[0] := {n_0};
3 Open[1] := ∅;
4 solution := null;
5 repeat
6   while Open[level] ≠ ∅ do
7     node := argmin{cost}(n) | n ∈ Open[level]};
8     Open[level] := Open[level]\{node};
9     if (node ∈ G) then
10    if cost(node) < cost(solution) then
11      solution := node;
12      U := cost(node);
13    end if
```

```
14    else
15       node.generateAdmittedSuccessors(beamStack:top());
16       if w < layerSize(level + 1) then /* w is a given beam width. */
17          Keep := the best w nodes ∈ Open[level + 1];
18          Prune := Open[level + 1]\Keep;
19          beamStack:top().c_max :=min{cost(n) | n ∈ Prune};
20          Open[level + 1] := Keep;
21       end if
22     end if
23   end while
24   level := level + 1;
25   Open[level + 1] := ∅;
26   beamStack:push([0;U));
27 until;
28 if solution ≠ null then
29    return solutionReconstruction(solution);
30 else
31    return null;
32 end if
```

Algorithm 3 illustrates the search function Search() that explores only w nodes in each level of a search graph as the beam search, where w is the beam width. One of important differences with the beam search is that Search() expands successors according to the corresponding beam stack element. It always produces successors of which cost is in the range of the beam stack element, and once it prunes successors, it records which nodes are inadmissibly cut off by modifying the upper bound of the beam stack element. Besides, while the beam search terminates once it identifies any solution, Search() keeps exploring a search graph until it reaches leaf nodes. For each search level, while any node to be searched exists (line 6), Search() repeats to select the most promising node at the current level (line 7) and checks whether it is a goal. If the node is a goal and better than the current solution, Search() records the node and its cost (lines 9–14). Otherwise, the algorithm expands the node with its successors according to the corresponding beam stack element (line 15). If the expansion induces the number of successors to exceed a given beam width, the algorithm prunes the next layer (lines 16–21). Once Search() completes exploring each level, it increases the current level index and pushes a beam stack element for the next level, and repeats the loop. After the loop, it returns a solution path if any (line 29). For memory efficiency, the divide-and-conquer version (Zhou and Hansen, 2005) of the beam stack search we use removes all previously visited nodes from memory, and reconstructs them whenever they are needed again.

4.3 Basic anytime algorithm for QoS-WSC problem using beam stack search

Our basic anytime algorithm uses the beam stack search to solve the QoS-aware web service composition (WSC) problem. To do this, we first reduce a QoS-aware WSC

problem into a graph search problem; given a QoS-aware WSC problem instance $\langle W;w_r\rangle$, we can reduce it into a search problem instance on a *weighted state-transition system* $S = (S, s_0, G, A, T,C)$ with the following components:

- S is a set of *states*, where a state corresponds to a node in search graphs.
- $s_0 \in S$ is the *initial state*.
- $G \subseteq S$ is a set of *goal states*.
- A is a set of *actions*.
- $T : S \times A \to S$ is a *transition function*.
- For each state s and each action a, $C(s, a)$ is the action cost.

The graph search problem on the weighted state-transition system is to find a sequence of actions from the initial state to a goal state of which total cost is minimal.

Given a set $W = \{w_1, \cdots, w_n\}$ of web services where for each j, $w_j = (I_j;O_j, Q_j)$, we denote as TP a set of all types, i.e. $TP = \{t \mid \text{for each } p \in \bigcup (I_j \cup O_j), t = type\ (p)\}$. Then, given a set $W = \{w_1, \cdots, w_n\}$ of web services and a requirement web service $wr(I_{wr}, O_{wr}, Q_{wr})$, we can construct a weighted statetransition system $S = (S, s_0, G, A, T, C)$ as follows:

- $S = \{(x_1, \cdots, x_m) \mid x_j = true$ or $false, m = |TP|\}$. Each boolean variable x_j represents whether we have an instance with the type t_j at a state s.
- $s_0 = (x_1, \cdots, x_m)$, where each x_j is *true* if there exists an input parameter $i \in I_{wr}$ such that its corresponding type t_{xj} is a supertype of t_i (i.e. $t_i <: t_{xj}$); otherwise x_j is *false*. Note that any type t is its supertype as well as subtype; the supertype and subtype relations are reflexive, i.e. for all type t, $t <: t$.
- $G = \{(x_1, \cdots, x_m) \mid$ each x_j is *true* iff there exists an output parameter $o \in O_{wr}$ such that its corresponding type t_{xj} is a subtype of *to* (i.e. $t_{xj} <: t_o$) $\}$.
- $A = W$.
- For $s = (x_1 \cdots, x_m)$, $s' = (x'_1, \cdots, x'_m)$, and $w = (I;O;Q)$, $T(s,w) = s'$ iff (1) for every $i \in I$, there exists x_j in s such that x_j is *true* and its corresponding type t_{x_j} is a subtype of the type of i (i.e. $t_{x_j} <: t_i$), (2) if x_k is *true*, x'_k is also *true*, and (3) $\forall o \in O_j$: for every variable x'_j in s', if its corresponding type t_{x_j} is a supertype of t_o, x'_j is *true*. Intuitively, a web service w can be invoked at a state s if we have data instances that are more informative than inputs of w at the state s. If w is invoked at such a state s, then we proceed to a state s' where we retain all the data instances from s and acquire outputs of w as well as their supertypes.
- For every state s, $C(s, w) = c_1 \cdot q_1 + \cdots + c_k \cdot q_k$, where each q_j is j-th quality criterion value of w.

Intuitively, we have an initial state where we possess all the data instances corresponding to the input of w_r as well as ones corresponding to their supertypes. If a state is more informative than the outputs of w_r, it is a goal state. Moreover, the optimal solution of the graph search problem on the weighted state-transition system corresponds to the optimal composition of web services of the QoS-aware WSC problem. Finally, as our basic anytime algorithm, we recast the beam stack search in Algorithm 2 to be suitable for the QoS-aware WSC problem. Our implementation first reduces a given QoS-aware WSC problem instance into a graph search problem instance, and then solve the search problem using the beam stack search. Whenever the beam stack search produces a best-so-far path, we construct a corresponding sequence of web services and return it.

5 Dynamic anytime algorithm for QoS-WSC problem

In this section, we propose a dynamic anytime algorithm for the QoS-aware WSC problem to identify better solutions earlier than our basic anytime algorithm described in Section 4. In the searching mechanism of the beam stack search algorithm, a bad decision in early phases requires more searching space than a bad selection in late phases does. To reduce this cost, our dynamic anytime algorithm assigns larger beam widths to early phases to consider more qualified candidates. In addition, we propose two more heuristics in its implementation to improve the quality of current approximate solutions and overall execution time.

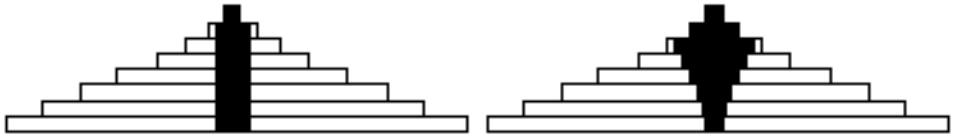
5.1 Carrot-shaped dynamic beam width

The original beam stack search employs the fixed size of the beam width at all levels of a search graph regardless of a given problem instance. However, the size of the beam width plays a significant role in determining the efficiency. A small sized beam width causes excessive pruning of qualified successor nodes, and therefore it requires more extensive backtracking. On the other hand, with a large sized beam width, many candidate nodes will be under consideration, and it makes a searching space increased significantly. In fact, with an extremely small or large size of the beam width, the beam stack search is no better than the depth-first search and the breadth-first search. Therefore, we need an efficient method to control the size of beam width.

To resolve this issue, we propose to dynamically adjust the beam width, i.e. to assign larger beam widths to early stages in order to consider more candidate nodes. This idea is influenced by a following natural intuition; when finding out an unknown path from a source to a destination, we do not know whether we should go north, south, east or west, especially at the beginning. A good choice at early stages leads to a good solution path more quickly. However, a bad decision in early stages requires longer backtracking and more retrials to other paths while a wrong decision in late stages induces just short backtracking. Hence, the choices at early stages are more critical than ones at late stages. In addition, in the beam stack search, the profit and loss at earlier levels are more significant due to its backtracking mechanism based on the beam stack. Since, to identify the better solution, it traces back a node selected from the deepest level of the previous search iteration, once candidate nodes are determined at a level, the other nodes excluded cannot be visited until we complete the whole search for the subgraphs of the selected candidates.

Based on this observation, we propose to apply different sized beam widths during searching. Our technique uses a large sized beam width first and decreases its size gradually as going down to a search graph. As a result, a constructed search graph for one search iteration appears like a carrot in shape (see Figure 2).

Figure 2 Search with static beam width vs. dynamic beam width



There are various considerable features to decide the beam width values. Among them, the number of total nodes in each level of a search graph is a key factor to determine the size of the searching space. However, to exactly identify the number of nodes in each level, it requires a significant amount of computations, i.e. the whole search graph construction. Therefore, by assuming that every node has a similar number of successors, we approximate these values. In the level 0, we have only one node which is the initial node (as the root node of the search graph). We count the number of successors of the initial node called numSucc, and then assume that each node has numSucc successors. Now, in the level 1, we have exactly numSucc nodes, and there are approximately numSucc^2 nodes, numSucc^3 nodes, numSucc^4 nodes, $\cdots$ in the level 2, the level 3, the level 4, $\cdots$, respectively. Based on the above approximation, we decide the beam width $beamWidth(i)$ for each level i as following:

- $beamWidth(1)$ = numSucc. We consider all nodes for the most careful choice at the first level.
- $beamWidth(2) = R_{inc} \cdot beamWidth(1)$, where R_{inc} is the increasing rate for the level 2 ($1 < R_{inc} <$ numSucc); in our experiment, we take 1.5 as the value of R_{inc}. In fact, the approximate number of nodes in the second level is numSucc^2, which may be too large to consider their descendants. Moreover, for a linear complexity, we need to restrict the beam width. At this level, however, we do not decrease the beam width but increase it linearly by a user provided constant R_{inc} since this strategy is helpful to find a short-length composition.
- For $i \geqslant 3$, $beamWidth(i) = R_{dec} \cdot beamWidth(I - 1)$, where R_{dec} is the decreasing rate ($0 < R_{dec} < 1$); in our experiment, we take 0.9 as the decreasing rate. Since a small sized beam width can make a search iteration quickly complete and the cost of bad choices at deeper levels is relatively not significant, we gradually decrease the beam width by using a user provided constant R_{dec}.
- $width_{min}$ is the lower bound of the beam width where $0 < width_{min} <$ numSucc; in our experiment, we take $0.3 \cdot$ numSucc as the minimum value. As we decrease the size of the beam width along the level, it may be down to 0. Besides, if a beam width is too small, it may generate excessive backtracking and retrial. Finally, for every i, if $beamWidth(i) < width_{min}$, then $beamWidth(i) = width_{min}$.

Algorithm 4 and Algorithm 5 present our dynamic anytime algorithm and its search function, respectively, which are based on our basic anytime algorithm in Section 4. The dynamic allocation of the beam width above is implemented in the function determineBeamWidth() (line 7 in Algorithm 5).

5.2 Additional heuristics

In addition to the dynamic beam width, we propose two more heuristics to improve the basic anytime algorithm: *short backtracking* and *upper bound propagation*. By avoiding the duplicated state expansion and unnecessary state inclusion, these techniques help to efficiently find approximate solutions with higher quality.

5.2.1 Short backtracking

The divide-and-conquer version of the beam stack search (Zhou and Hansen, 2004) mainly aims to save the memory consumption by keeping in memory only four levels of a search graph, which are most recently considered. By this technique, the level to which the algorithm has to backtrack may be removed from memory, and in this case the algorithm must recover this missing level. Unfortunately, the construction of the target level requires us to construct all the upper levels of the target level. In the other word, the algorithm must return to the initial node and construct again successor nodes at each level until the target level is rebuilt. This node reconstruction may occur many times during the whole algorithm execution.

To avoid this problem, we propose a short backtracking technique. Our search function, DBWSearch, stores in memory a set of the closed states for every visited level during a search iteration, unlike the basic search function where all closed sets are removed before exploring the next level. On the next invocation of DBWSearch, our algorithm passes the level corresponding to beamStack[top-1] to DBWSearch as an input parameter (line 26 and line 10 in Algorithm 4). Note that the level corresponds to the parent level of the deepest level in which inadmissible pruning occurs. Then, our heuristic search function DBWSearch can easily build the open set of the target level by simply moving the nodes in the closed set to the open set (lines 2–5 in Algorithm 5), and resumes searching at the level corresponding to beamStack[top] by generating new successors according to the updated lower bound and upper bound of beamStack[top].

Algorithm 4: DAA QoSWSC: Dynamic Anytime Algorithm for the QoS aware WSC

Input : a set W of web services, a requirement web service w_r, and an allowed upper cost U.

Output: a sequence of web services.

1 Reduce $\langle W, w_r \rangle$ to $S(S, s_0, G, A, T, C)$;

2 *beamStack:push*(0,U);

3 *bestPath* := *null*;

4 *level* := 0;

5 *Open* [0] := $\{n_0\}$;

6 *Open* [1] := $\varnothing$;

7 *Closed* [0] := $\varnothing$;

```
8 bestPath := null;
9 while beamStack:top() ≠ null do
10    path := DBWSearch(level, G, U);
11    if path ≠ null then
12       bestPath := path;
13       U := cost(path);
14       print ReduceToWebService(path);
15    end if
16    while beamStack:top():c_max ⩾ U do
17       beamStack.pop();
18    end while
19    if beamStack:isEmpty() then
20       print "bestPath is optimal.";
21       return ReduceToWebService(bestPath);
22    end if
23    beamStack.top().c_min := beamStack.top().c_max ;
24    beamStack.top():c_max := U;
25    Propagate_c_max (U);
26    level := beamStack:size() – 1;
27 end while
```

5.2.2 Upper bound propagation

Each entry of the beam stack contains c_{min} and c_{max} which represent a set of nodes under consideration at the corresponding level; i.e. a qualified node at the level should have an aggregated QoS value in between c_{min} and c_{max}. The upper bound c_{max} has a significant influence to the size of the search space. As the value is closer to the optimal value, the algorithm can ignore more nodes of which the aggregated QoS value is greater than the upper bound c_{max} .

Algorithm 5: DBWSearch: Dynamic Beam Width Search with heuristics

Input : a start level *l*, a set *G* of goal nodes, and an allowed upper cost *U*.
Output: a sequence of nodes.

```
1 solution := null;
2 if Closed[l] ≠ ∅ then
3    Open[l] := Open[l] ∪ Closed[l];
4    Closed[l] := ∅;
5 end if
6 repeat
7    w := determineBeamWidth(l);
8    while Open[l] ≠ ∅ do
9       node := argmin{cost(n) | n ∈ Open[l]};
```

```
10      Open[l] := Open[l]\{node};
11      Closed[l] := Closed[l]∪ {node};
12      if (node ∈ G) then
13          if cost(node) < cost(solution) then
14              solution := node;
15              U := cost(node);
16          end if
17      else
18          node.generateAdmittedSuccessors(beamStack:top());
19          if w < layerSize(l+1) then
20              Keep := the best w nodes ∈ Open[l + 1];
21              Prune := {n | n ∈ Open[l + 1] ^ n ∉ Keep};
22              beamStack:top():c_max :=min{cost (n) | n ∈ Prune};
23              Open[l + 1] := Keep;
24          end if
25      end if
26  end while
27  l := l + 1;
28  Open[l + 1] := ∅;
29  beamStack:push([0;U]);
30 until;
31 if solution ≠ null then
32  return solutionReconstruction (solution);
33 else
34      return null;
35 end if
```

However, whenever the beam stack search finds a new better solution, it updates only the aggregated QoS value of the so-far-best solution U and retains the value for c_{max} at every level as the same. Without updating a tight low value for c_{max}, we may explore unnecessary nodes in the next search iteration. Such states more likely exist at higher levels since the aggregated QoS value of a node in high levels is relatively small. Its effect, however, can be significant in a whole searching space since the nodes in higher levels include a number of descendants in their subgraphs. Therefore, once we identify a new better solution in a search iteration, our new method updates the value of c_{max} for every level with the aggregated QoS value of the solution we have found (line 25 in Algorithm 4).

6 Experiment

We have implemented two automatic tools implementing the algorithms in Section 4 and Section 5, i.e. a QoS-aware WSC tool with the beam stack search and a QoSaware WSC tool with the dynamic beam width method. Given a WSDL file for a set of web services, an OWL file for the type information, a WSLA file for the QoS information, and a WSDL file for a requirement web service, our tools reduce the QoS-aware WSC problem into a graph search problem, and then compute an optimal strategy. Finally, our tools translate the solution strategy into a BPEL file for the optimal composition.

To demonstrate that our tools efficiently identify an optimal solution, we have experimented on six WSC problems (i.e. P1, P2, …, P6) produced by the TestSetGenerator which the Web Service Challenge 2009 competition (Kona, 2009) provides. Table 1 presents, for each problem instance, the number of total web services and total parameters, and the length of the optimal solution. All experiments have been performed on a PC using a 2.33GHz Core2 Duo processor, 1GB memory and a Linux operating system.

Table 1 Experimental problem instances

Problem	*Web services*	*Parameters*	*Solution length*
P1	50	1000	7
P2	100	10,000	5
P3	100	10,000	7
P4	500	10,000	4
P5	1000	10,000	5
P6	1500	10,000	6

First, we explain how our anytime algorithm works on the QoS-aware WSC problem with P4, comparing an optimal algorithm. As an optimal algorithm, we employ the uniform cost search algorithm (Russell and Norvig, 2003) which is a general optimal algorithm for the weighted-graph search problem. We compare it with our dynamic anytime algorithm which includes the dynamic beam width mechanism and heuristics described in Section 5. Figure 3 illustrates this comparison, where X-axis represents execution time in seconds when an algorithm returns a solution, and Y-axis means the solution quality ratio of each returned solution (i.e. the QoS value of the optimal solution over the QoS value of a solution returned by an anytime algorithm). The uniform cost algorithm finds only one optimal solution with the QoS value of 510 at 61.4 seconds (see ◆ in Figure 3). On the other hand, our anytime algorithm returns a set of solutions: 850 at 2.6 sec (the quality ratio is 510/850=0.6), 680 at 2.7 sec (the quality ratio is 510/680=0.75), 650 at 3.5 sec (the quality ratio is 510/650=0.78), 560 at 10.1 sec (the quality ratio is 510/560=0.91), 530 at 25.0 sec (the quality ratio is 510/530=0.96), and 510 at 57.7 sec (the quality ratio is 510/510=1). Then, our algorithm finally concludes that the solution is the optimal at 90.0 sec. Remark that although our algorithm has already found the optimal solution at 57.7 sec, it needs more time to confirm that it is indeed optimal. Although our anytime algorithm takes longer to complete with the optimal solution, it provides several solutions with high quality (i.e. over the quality ratio of 0.9) at 10.1, 25.0, and 57.7 seconds much earlier than the uniform cost algorithm which returns the solution at 61.4 sec. Furthermore, solutions with moderate quality (i.e. between the quality ratio of 0.6 and 0.9) are identified very quickly (i.e. at 2.6, 2.7, and 3.5 seconds).

Figure 3 Optimal algorithm vs. anytime algorithm

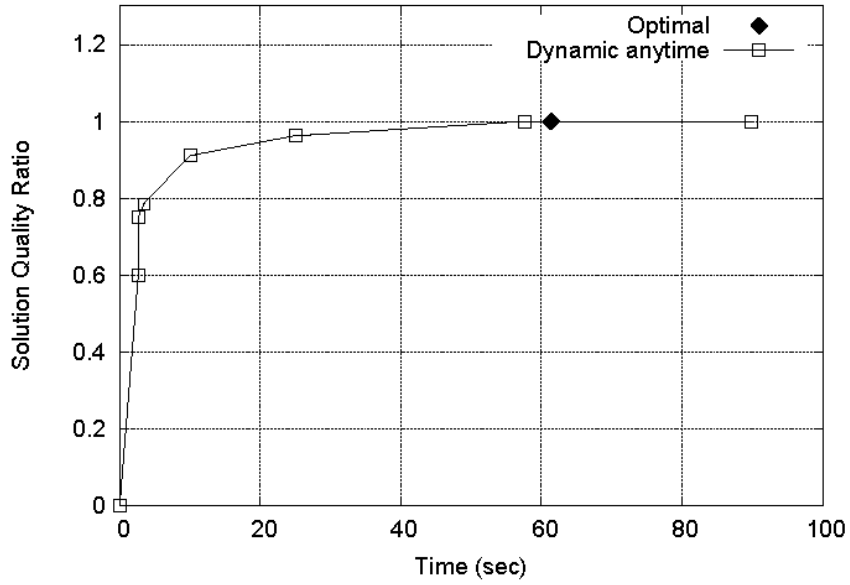


Table 2 presents how three search algorithms (i.e. the uniform cost search, the basic anytime algorithm, and the dynamic anytime algorithm) solve the six problems in Table 1 within specific time thresholds (i.e. 60, 120, 300, and 600 seconds, respectively). It shows whether each algorithm solves problems within each timeout and how good so-far-best solutions are. In the case of the uniform cost algorithm, we report only whether a given problem is solved or not. For basic and dynamic anytime algorithms, even after they has found the optimal solution, they need to visit some part of search space to prove that the solution is indeed optimal. *Complete* and *Incomplete* indicate if the algorithms accomplish the confirmation step. Within 60 seconds, all algorithms do not complete solving all problems except P1. However, while the optimal algorithm does not provide any result, anytime algorithms generate approximate solutions of which QoS quality ratios are from 0.68 to 1. As a given timeout get longer, the uniform cost algorithm solves more problems, i.e. it solves 1, 4, 4, and 5 problems with 60, 120, 300, and 600 seconds, respectively. Likewise, anytime algorithms complete more problems or return solutions with higher quality. The basic anytime algorithm completes 1, 1, 4, and 4 problems with 60, 120, 300, and 600 seconds, respectively, and for incomplete problems it improves qualities of the solutions, e.g. for P5, the quality ratios of solutions returned are 0.68, 0.68, 0.81, and 0.87 with 60, 120, 300, and 600 seconds, respectively. The dynamic anytime algorithm completes 1, 3, 4, 4 problems with 60, 120, 300, 600 seconds, respectively, and for incomplete problems it also improves qualities of the solutions, e.g. for P6, the quality ratios of solutions returned are 0.85, 0.96, 0.96, and 1 with 60, 120, 300, and 600 seconds, respectively. For the problem P5, the optimal algorithm cannot solve it even in 600 seconds, which means that the algorithm does not return anything to users after a long waiting. On the other hand, even within 60 seconds, the basic and dynamic anytime algorithms provide approximate solutions of which quality ratio are 0.68 and 0.81, respectively, and their qualities get improved with longer

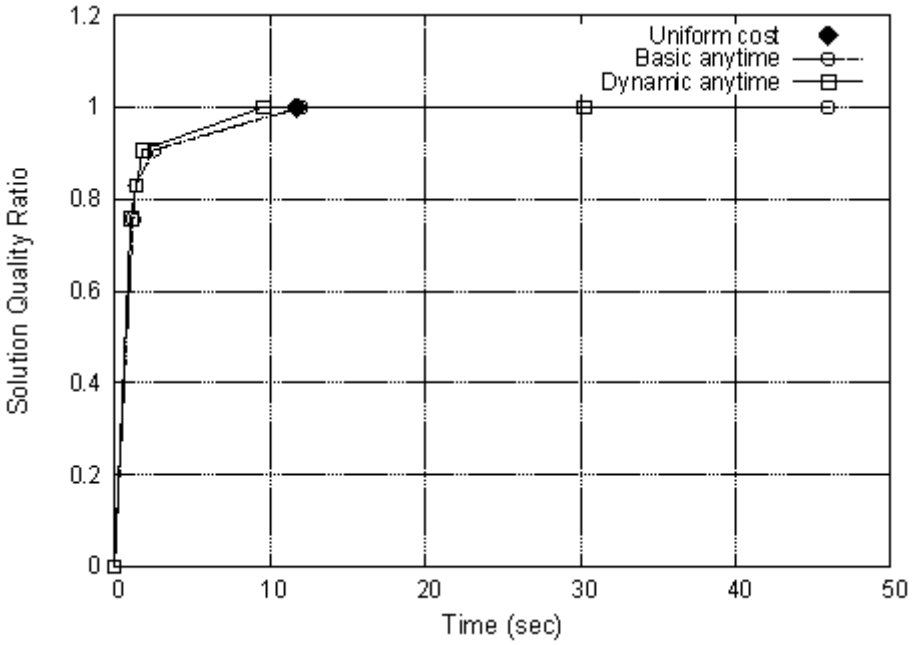
timeouts (up to 0.87 within 600 seconds). In all the cases, our dynamic anytime algorithm outperforms the basic anytime algorithm; it finds higher quality solutions earlier and/or accomplishes the whole search faster than the basic algorithm.

Table 2 Experimental results with time thresholds

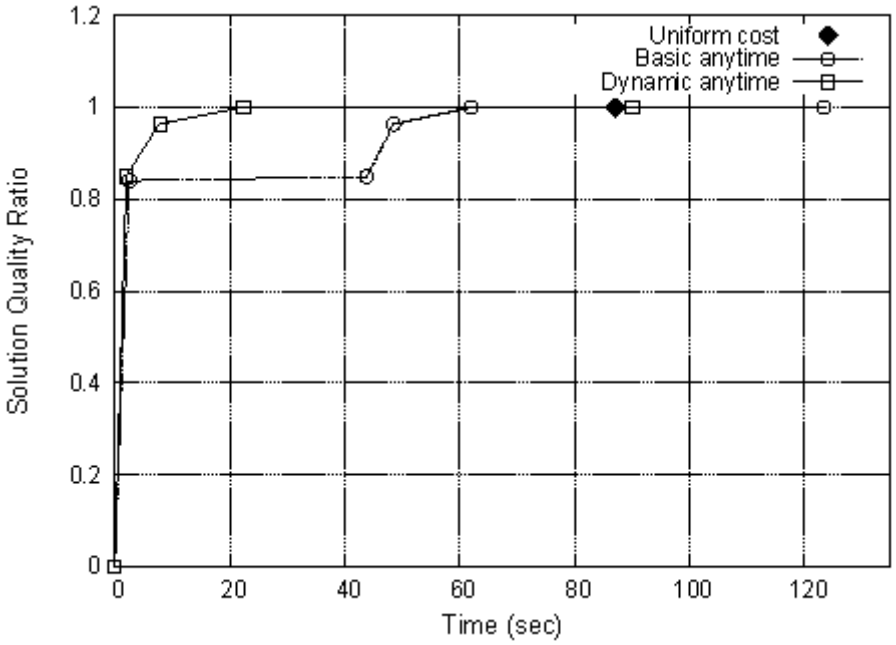
Problem	*Timeout (sec)*	*Uniform cost*	*Basic anytime*	*Quality ratio*	*Dynamic anytime*	*Quality ratio*
P1	60	Solved	Complete	1	Complete	1
P2	60	Not solved	Incomplete	0.84	Incomplete	1
P3	60	Not solved	Incomplete	0.81	Incomplete	1
P4	60	Not solved	Incomplete	0.96	Incomplete	1
P5	60	Not solved	Incomplete	0.68	Incomplete	0.81
P6	60	Not solved	Incomplete	0.85	Incomplete	0.85
P1	120	Solved	Complete	1	Complete	1
P2	120	Solved	Incomplete	1	Complete	1
P3	120	Solved	Incomplete	0.94	Incomplete	1
P4	120	Solved	Incomplete	0.96	Complete	1
P5	120	Not solved	Incomplete	0.68	Incomplete	0.81
P6	120	Not solved	Incomplete	0.85	Incomplete	0.96
P1	300	Solved	Complete	1	Complete	1
P2	300	Solved	Complete	1	Complete	1
P3	300	Solved	Complete	1	Complete	1
P4	300	Solved	Complete	1	Complete	1
P5	300	Not solved	Incomplete	0.81	Incomplete	0.87
P6	300	Not solved	Incomplete	0.96	Incomplete	0.96
P1	600	Solved	Complete	1	Complete	1
P2	600	Solved	Complete	1	Complete	1
P3	600	Solved	Complete	1	Complete	1
P4	600	Solved	Complete	1	Complete	1
P5	600	Not solved	Incomplete	0.87	Incomplete	0.87
P6	600	Solved	Incomplete	0.96	Incomplete	1

Figure 4 illustrates how three algorithms work on six problems by plots; the format is same as Figure 3. Anytime algorithms identify a set of approximate solutions, and the quality of their solutions improve as time passes. In Figure 4 (b), (c), and (f), the dynamic anytime algorithm notably outperforms the basic anytime algorithm, especially, at the beginning. We believe that the fact is caused by a large beam width at the early stage in the dynamic anytime algorithm. In the problem P3 (see Figure 4 (c)), while the basic anytime algorithm finds solutions with quality ratios, 0.77 and 0.80, at 3.0 seconds and 37.8 seconds, respectively, the dynamic anytime algorithm first produces a solution with 0.80 at 3.0 seconds. That means, the dynamic anytime algorithm bypasses the solution with 0.77 and finds a higher quality solution with 0.80 much earlier since it can do careful selections with more candidate states at the early stage than the basic algorithm. Furthermore, in the basic anytime algorithm, a long interval (34.8 seconds) between finding the solutions with 0.77 and 0.80 means that bad choices at early levels make a significant cost due to its backtracking mechanism. Therefore, the ability to bypass low-quality solutions is a valuable property of the dynamic anytime algorithm.

Figure 4 Experimental results

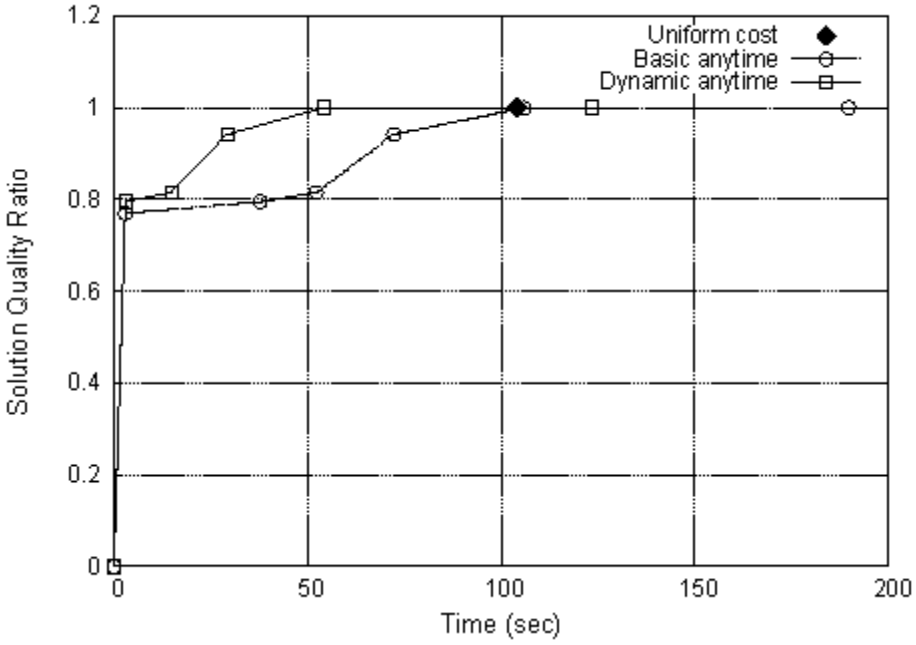


(a) P1

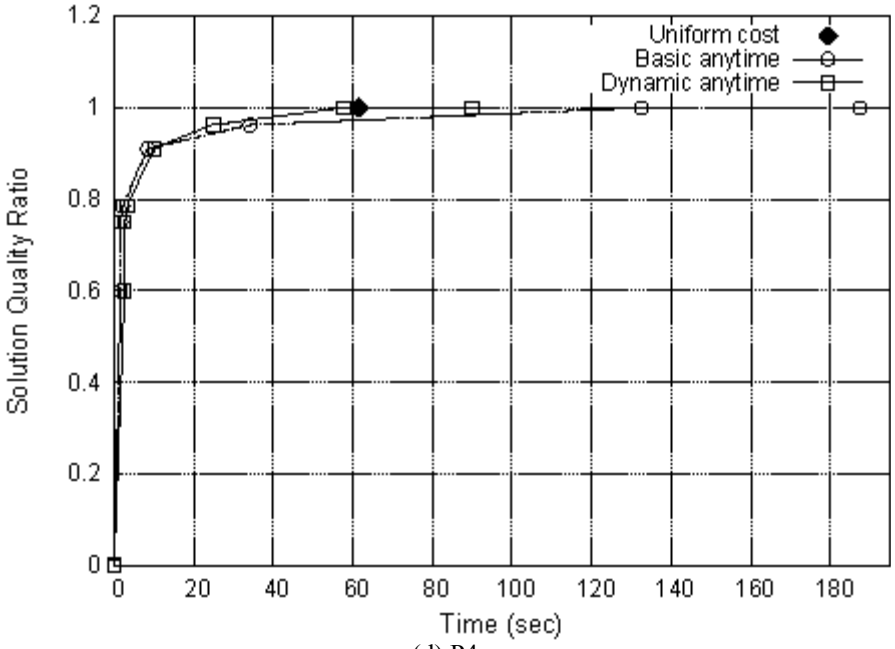


(b) P2

Figure 4 Experimental results (continued)

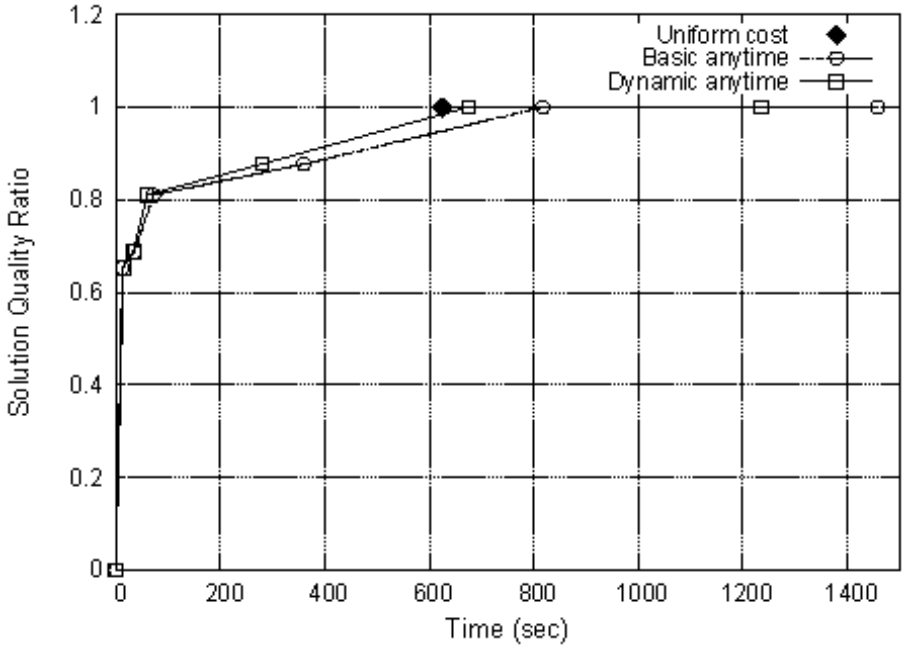


(c) P3

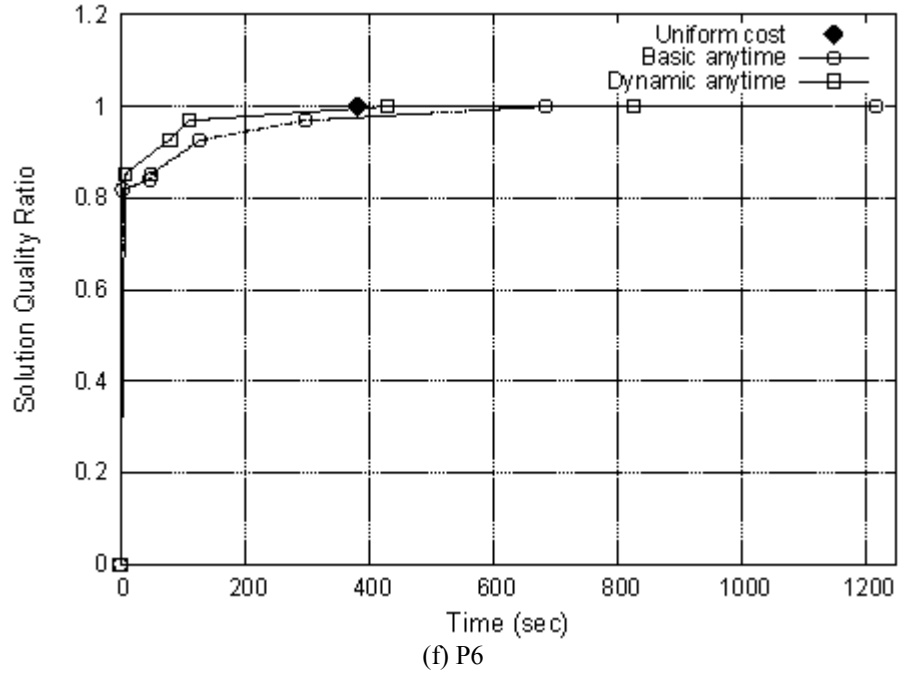


(d) P4

Figure 4 Experimental results (continued)



(e) P5



(f) P6

7 Conclusion

In this paper, we have proposed anytime algorithms for the QoS-aware WSC problem. The main contribution of this paper is to apply anytime algorithms to the QoS-aware WSC problem. To the best of our knowledge, this is the first work to employ an anytime algorithm to the WSC problem. Moreover, to improve the basic anytime algorithm, we have proposed a novel anytime algorithm to dynamically adjust the beam widths for a given problem instance. Our preliminary experiment has shown promising results.

There are several issues that are interesting for future study. First, while we have employed the beam stack search as the underlying engine for our anytime algorithm, there are various other anytime algorithms such AWA* (Aine, 2007), ARA* (Likhachev et al., 2003) and AD* (Likhachev et al., 2005). Consequently, it is worth comparing them in order to know which one is favorable to the QoS-aware WSC problem. Second, we want to study more efficient adaptive control of the beam width for a given instance. To do this, it is also required to observe distinctive properties of WSC problems. Finally, we plan ample experiment to validate our algorithms for various problem instances.

Acknowledgements

This research was supported by the MKE(Ministry of Knowledge Economy), Korea, under the ITRC(Information Technology Research Center) support program supervised by the NIPA(National IT Industry Promotion Agency): NIPA-2012- H0301-12-3006.

References

Aggarwal, R., Verma, K., Miller, J.A. and Milnor, W. (2004) 'Constraint driven web service composition in METEOR-S', *IEEE International Conference on Services Computing*, pp.23–30.

Aine, S., Chakrabarti, P.P. and Kumar, R. (2007) 'AWA* – a window constrained anytime heuristic search algorithm', *International Joint Conference on Artificial Intelligence*, pp.2250–2255.

Alrifai, M. and Risse, T. (2009) 'Combining global optimization with local selection for efficient QoS-aware service composition', *International World Wide Web Conference*, pp.881–890.

Alrifai, M., Risse, T. and Nejdl, W. (2012) 'A hybrid approach for efficient web service composition with end-to-end QoS constraints', *TWEB*, Vol. 6, No. 2, p.7.

Alrifai, M., Skoutas, D. and Risse, T. (2010) 'Selecting skyline services for QoS-based web service composition', *International World Wide Web Conference*, pp.11–20.

Ardagna, D. and Pernici, B. (2007) 'Adaptive service composition in flexible processes', *IEEE Transactions on Software Engineering*, Vol. 33, No. 6, pp.369–384.

Benatallah, B., Hacid, M., Léger, A., Rey, C. and Toumani, F. (2005) 'On automating web services discovery', *VLDB Journal*, Vol. 14, No. 1, pp.84–96.

Boddy, M.S. and Dean, T. (1989) 'Solving time-dependent planning problems', *International Joint Conference on Artificial Intelligence*, pp.979–984.

Canfora, G., Penta, M.D., Esposito, R. and Villani, M.L. (2005) 'An approach for QoS-aware service composition based on genetic algorithms', *Genetic and Evolutionary Computation Conference*, pp.1069–1075.

Cardoso, J., Sheth, A.P., Miller, J.A., Arnold, J. and Kochut, K. (2004) 'Quality of service for workflows and web service processes', *Journal of Web Semantics*, Vol. 1, No. 3, pp.281–308.

Fu, X., Bultan, T. and Su, J. (2004) 'Analysis of interacting BPEL web services', *Proceedings of the 13th International World Wide Web Conference*, pp.621–630.

Hull, R. and Su, J. (2005) 'Tools for composite web services: a short overview', *SIGMOD Record*, Vol. 34, No. 2, pp.86–95.

Kil, H., Nam, W. and Lee, D. (2009) 'Efficient abstraction and refinement for behavioral description based web service composition', *International Joint Conference on Artificial Intelligence (IJCAI-09)*, pp.1740–1745.

Kona, S., Bansal, A., Blake, M.B., Bleul, S. and Weise, T. (2009) 'WSC-2009: a quality of service-oriented web services challenge', *IEEE Conference on Commerce and Enterprise Computing*, pp.487–490.

Likhachev, M., Ferguson, D.I., Gordon, G.J., Stentz, A. and Thrun, S. (2005) 'Anytime dynamic A*: an anytime, replanning algorithm', *Proceedings of the 15th International Conference on Automated Planning and Scheduling (ICAPS 2005)*, pp.262–271.

Likhachev, M., Gordon, G.J. and Thrun, S. (2003) 'ARA*: anytime A* with provable bounds on sub-optimality', *Advances in Neural Information Processing Systems*, 2003.

Narayanan, S. and McIlraith, S.A. (2003) 'Simulation, verification and automated composition of web services', *Proceedings of the 11th International World Wide Web Conference*, pp.77–88.

WS-Agreement (2007) *Web services agreement specification*. Available online at: http://www.ogf.org/ documents/GFD.107.pdf

Richter, S., Thayer, J.T. and Ruml, W. (2010) 'The joy of forgetting: faster anytime search via restarting', *International Conference on Automated Planning and Scheduling*, pp.137–144.

Russell, S. and Norvig, P. (2003) *Artificial Intelligence: A Modern Approach*, 2nd ed., Prentice-Hall.

Srivastava, U., Munagala, K., Widom, J. and Motwani, R. (2006) 'Query optimization over web services', *International Conference on Very Large Data Bases*, pp.355–366.

Thayer, J. and Rumi, W. (2010) 'Anytime heuristic search: frameworks and algorithms', *International Symposium on Combinatorial Search*, pp.177–186.

Web services activity (2002) http://www.w3.org/2002/ws/

WS-Policy (2006) *Web services policy version 1.2*. http://www.w3.org/Submission/WS-Policy/

Wang, J., Wang, J., Chen, B. and Gu, N. (2011) 'Minimum cost service composition in service overlay networks', *World Wide Web*, Vol. 14, No. 1, pp.75–103.

WSLA (2004) *Web Service Level Agreements project*. http://www.research.ibm.com/wsla/documents.html

Yu, Q. and Bouguettaya, A. (2012) 'Multi-attribute optimization in service selection', *World Wide Web*, Vol. 15, No. 1, pp.1–31.

Yu, T., Zhang, Y. and Lin, K-J. (2007) 'Efficient algorithms for web services selection with end-to-end QoS constraints', *ACM Transactions on the Web*, Vol. 1, No. 1, p.6.

Zeng, L., Benatallah, B., Ngu, A.H.H., Dumas, M., Kalagnanam, J. and Chang, H. (2004) 'QoS-aware middleware for web services composition', *IEEE Transactions on Software Engineering*, Vol. 30, No. 5, pp.311–327.

Zhou, R. and Hansen, E.A. (2004) 'Breadth-first heuristic search', *International Conference on Automated Planning and Scheduling*, pp.92–100.

Zhou, R. and Hansen, E.A. (2005) 'Beam-stack search: integrating backtracking with beam search', *International Conference on Automated Planning and Scheduling*, pp.90–98.