Fast Discovery of Interesting Collections of Web Services

Zhou Zhu and James Bailey
NICTA VRL, University of Melbourne
jbailey@csse.unimelb.edu.au

Abstract

Web Services are beginning to play a major role in future Web architectures and software applications. One of the most important research directions in the area of Web services is the development of techniques for automatically discovering collections of services that satisfy a set of interestingness constraints. Discovery of interesting collections of Web services is a challenging problem, however, due to the high complexity involved. In this paper, we present a new method for discovery of interesting collections of Web services, according to user specified cost constraints. We show that this task is closely related to the well-known problem of computing the transversals of a hypergraph. We experimentally evaluate our approach and show that pruning and partitioning techniques can significantly improve running time.

1. Introduction

Web Services offer an important role in future Web architectures and software applications [17]. A Web Service is a remotely invoke-able application that aims to provide some functionality or a set of functionalities over a network. One well-known example is credit card verification and processing for online purchasing.

The World Wide Web is currently moving towards a service-oriented architecture, which goes beyond the standard user interaction of browsing and searching. Instead of visiting Web sites, carrying out searches and then manually downloading software, users can interact with a Web service. The Web service providers produce and publish their Web services. E.g. a cinema that provides a Web service for ticket bookings and inquiries for (possibly remote) users. The discovery entity provides discovery assistance both for providers to advertise their services and for consumers to obtain information about what services are available. Industry standards, such as UDDI, define the environment and the possible agreements between the various entities involved.

Before a user requests and invokes a Web Service, mutual agreement on shared standards is required between the user and the provider. Users can then send their request through the service interface to the provider side or discovery entity. Once the provider/discovery entity gets the request, it will determine matching services. Users may then invoke the Web Service(s) appropriate for their application. One example is online virus detection, Without a Web service facility, the user has to visit the virus provider's Website, choose the appropriate program, download it, install it and then finally run it. Using a remote Web service, they can send their request and accompanying parameters specifying what type of scan they require. The service can then perform a remote scan of the user's computer. Three industry-based building blocks form a foundation for Web Service applications. They are the Web Service Description Language, WSDL, specifying how to describe the capabilities of a Web Service; SOAP, a transport protocol; and UDDI, an XML-based registry schema for Web Service discovery that uses a classification catalogue characteristics, such as keywords, to indicate functionality.

Web Services Discovery

The Discovery entity plays an important role in the Web Service setting. Three components are an essential part of the discovery process: shared standards, request descriptions and discovery approaches. Queries containing a set of requested functionalities are the most popular method of specification from users. The precise form of these requests may vary dramatically across description languages [3,8]. Many discovery approaches have been proposed for dealing with a query that specifies just a single desired functionality. Much of this work focuses on techniques for semantic matching between the desired functionality and an index of Web service capabilities [13]. When a query specifies that a set of multiple functionalities are required, the discovery entity is required to return sets of relevant Web Services, since just a single Web service may not be able to provide all desired functionalities. This problem of discovery of combinations of Web services is in fact very similar to the data mining problem of finding interesting patterns in a dataset. Users specify certain constraints on the collections they are interested in (via functionality descriptions and possibly cost constraints) and the discovery engine must determine all sets which satisfy the query.

From the discovery engine's procedure point of view, the discovery process involves three steps. The first step is to parse the user query. The second step is to match the descriptions in the query against a catalogue of known Web services. The third step is to find the combinations of matched Web Services that satisfy the cost constraints of the query. The first and second steps have been researched extensively and implemented in various approaches, such as the index method. Other methods include semantic technology approaches built upon languages such as DAML-S [1]. The third step is principally concerned with determining collections of Web services that satisfy the cost constraints. This is what we focus on in this paper, assuming that the functionality matching has already been performed. We now give an example.

Web Service Discovery Example

A Professor intends to travel to several places overseas at Christmas. He needs to arrange an airline ticket, accommodation and determine high quality restaurants serving the local cuisine. He hopes that he can book these tickets and places as soon as possible and wishes to minimize any time spent searching on the Web. A Web Service discovery engine can assist with this task. To use an individual Web Service, he must register and pay a registration fee. He would like to find the cheapest collection of Web Services that can fulfill all his requirements.

There are six desired functionalities in the request. These are shown in the left part of Figure 2. Web Services that offer the desired functionality are shown on the right side. There are many possible combinations that satisfy all requested functionalities. E.g. {FlightCentre, HotelService, SmartTraveller, BookPlaces} or {BargainHunter, HotelService, Goumer, Travel&Stay, BookPlaces}. But which sets of Web Services are good choices ? Registration cost is an important factor that should be considered, since each collection of Web services will have an overall registration cost (which is the sum of the individual registration fees).

Desired functionalities	**Matching Web services**
find flight ticket price	BargainHunter SmartTraveller FlightCenter
find accommodation	HotelService Travel&Stay AccomNet
find restaurant information	SmartTraveller Goumer
book and pay tickets	Travel&Stay TicketWatcher FlightCentre
book accommodation	AccomNet BookPlaces
book restaurant	BookPlaces

Figure 2: An Example of Web Service Discovery

Determining the collections of Web services that offer all the desired functionalities and satisfy a maximum cost constraint is similar to a constrained set cover problem that aims to find all collections of Web service candidates (the cover) that can satisfy all the desired functionalities in the input. The set cover problem is equivalent to the hypergraph transversal problem [5].

Contributions

In this paper, we make the following contributions

- Show how the problem of computing cost constrained collections of Web services can be modeled as a constrained set cover/hypergraph transversal problem.
- Provide an efficient algorithm for computing constrained hypergraph transversals.
- Experimentally evaluate our approach and identify pruning and partitioning techniques which significantly affect running time.

2. Background and Preliminaries

We now provide the necessary background on hypergraphs and then give a formal description of the problem

Definition 1 A **hypergraph** $\mathcal{H}$ is a pair

$$\mathcal{H} = (\mathcal{E}, \mathcal{V})$$

where $\mathcal{V}$ is a finite set of vertices $\mathcal{V} = \{ v_1, v_2,\ldots,v_m \}$ and $\mathcal{E}$ is a family of (hyper) edges, $\mathcal{E} = \{ e_1, e_2,\ldots, e_t \}$ where each e_i is a subset of $\mathcal{V}$, $1 \le i \le t$. We assume $\mathcal{V} = \{\cup e_i \mid e_i \in \mathcal{E}, 1 \le i \le t \}$.

Definition 2 A set $\mathcal{T} \subseteq \mathcal{V}$ is a **transversal** of H if for each $e \in \mathcal{E}$, $\mathcal{T} \cap e \neq \phi$. A transversal $\mathcal{T}$ is minimal if no proper subset of it is also a transversal. The set of all transversals of a hypergraph $\mathcal{H}$ is represented as $\mathcal{Tr\_H}$.

We now describe the reduction of the Web service discovery problem to one involving hypergraphs. Each Web Service is modeled as a vertex and $\mathcal{W}$ represents the set of all Web services. We will use w to represent a single vertex. Each edge corresponds to a set of vertices (Web services) which offer a particular functionality requested by the user. We use $\mathcal{F}$ to represent the set of all possible functionalities and f to represent a single functionality. Suppose for a given user query with t desired functionalities, there are a total of k Web services that can provide at least one of these functionalities.

So we have $|\mathcal{F}|=|\mathcal{E}| = t$. The hypergraph can be rewritten as $\mathcal{H} = (\mathcal{W}, \mathcal{F})$ where $\mathcal{W} = \{ w_1, w_2,\ldots, w_k\}$ and $\mathcal{F} = \{ f_1, f_2,\ldots, f_t \}$. Each $f_i \in \mathcal{F}$ and $f_i \subseteq \mathcal{W}$.

A transversal of the Web service hypergraph now corresponds to a set of Web services that cover all the functionalities requested by the user. The correspondence between the Web service discovery process and hypergraphs is summarized in Figure 3.

one functionality	$\rightarrow$	one edge
one Web Service	$\rightarrow$	one vertex
cost of a set of Web Services	$\rightarrow$	cost of a set of vertices
one output set of Web Services	$\rightarrow$	one transversal satisfying cost

functionality	**The matching Web Services**
find flight ticket price	BarginHunter SmartTraveller FlightCentre
maps to:	edge1 = {BarginHunter, SmartTraveller, FlightCentre}
find accommodation	HotelService Travel&Stay AccomNet
maps to:	edge2 = {HotelService, Travel&Stay, AccomNet}

Figure 3 Mapping Between Web Service Discovery and Hypergraphs

As mentioned earlier, the process to find sets of Web Services satisfying the user query can be reduced to finding transversals of the Web service hypergraph. However, all transversals computed must satisfy certain cost constraints.

Expressing Cost Constraints

We allow the user to define a conjunction of two constraints for restricting collections of Web services, one a price constraint and one a cardinality constraint. The cost constraint has the form:

$$\textbf{size}(S) < k1 \text{ and } \textbf{totalprice}(S,F) < k2$$

Where S is a set of Web services that will be returned, F is the set of functionalities requested, k1 and k2 are integers. The function **size(S)** returns the number of Web services in the set S and **totalprice** is:

$$\textbf{totalprice}(S,F)=\sum_{f\in F} \min(\{P_f^s \mid s \in S \cap f\})$$

where P_f^s is the price charged by Web service s for using functionality f. i.e. the **totalprice** of a set of Web services is the sum of the minimum cost required for each functionality. To ease presentation, our examples assume that each Web service charges the same price for each functionality it provides.

Our problem can thus be stated as follows. Given a Web service hypergraph $\mathcal{H} = (\mathcal{W}, \mathcal{F})$ and a cost constraint (k1, k2), find all sets of Web Services $\mathcal{R}$, $\mathcal{R} \subseteq \mathcal{W}$, such that $\mathcal{R}$ is a transversal of $\mathcal{H}$ and $\textbf{size}(\mathcal{R}) < k1$ and $\textbf{totalprice}(\mathcal{R},\mathcal{F}) < k2$. It is a challenging problem, since the size of the output may in the worst case be exponential in the number of Web services.

3. Finding Minimum Cost Transversals

We now describe how to determine the set of transversals satisfying the cost constraints. We use the Web service hypergraph shown in Figure 5 as a running example. Prices of individual functionalities are shown using superscripts. Any transversals found must satisfy the constraints: **size**(S) <5 and **totalprice**(S)<14. The core of our method for finding transversals is to perform a depth first enumeration of transversals of the hypergraph satisfying the cost constraint. We also use three different optimisation strategies, namely – edge pruning, horizontal partitioning and vertical partitioning, to help reduce the size of the search space.

```
Input hypergraph H and user cost constraints;
Output all Tr_$\mathcal{H}$ satisfying constraints

1  $H_{EP}$ = Edge Prune (H)
2  $S_{HV}$ = Horizontal_Partition($H_{EP}$);
3  for each  $H_i \in S_{HV}, H_i = (V_i, E_i)$
4    for each  $x \in V_i$
5      $H_i^x$ = Vertical_Partition($H_i$, x);
6      Tr_$H_i^x$ = Depth_first_enumerate($H_i^x$);
7      Tr_$H_i$ = Tr_$H_i^x \cup$ Tr_$H_i$;
8    end for
9    If i=1
10     then Tr_$H_{EP}$=Tr_$H_1$;
11   else
12     Tr_$H_{EP}$= Tr_$H_i \times$ Tr_$H_{EP}$;
13 end for
14 Tr_H = Tr_$H_{EP}$;
15 Return Tr_H
```

Figure 4 Horizontal and Vertical Partitioning

Figure 4 illustrates the basic outline of the algorithm. The hypergraph first has edge pruning applied, it is then decomposed into several connected subhypergraphs using horizontal partitioning. Vertical partitioning is applied to each subhypergraph and transversals are computed depth first within each vertical partition. The results of the vertical partitions are unioned together. Finally, once all horizontal partitions have had their transversals computed, cross products between all horizontal partitions are calculated to yield the final output. The algorithm is guaranteed to compute all transversals satisfying the cost constraints (proof omitted).

Hypergraph H					
edge1:	$a^{0.5}$	$c^{1.5}$	g^{4}	h^{7}	z^{10}
edge2:	$b^{0.6}$	$d^{2.5}$	i^{6}	$w^{8.1}$	
edge3:	$c^{1.5}$	$p^{1.6}$	$m^{7.1}$	$n^{8.4}$	k^{9}
edge4:	$d^{2.5}$	$q^{2.6}$	$m^{7.1}$	$j^{8.2}$	$l^{8.5}$
edge5:	$e^{0.1}$	f^{3}	r^{6}		
edge6:	f^{3}	$o^{3.1}$	$s^{7.9}$	$t^{8.9}$	z^{10}

cardinality(S)<5 and totalprice(S)<14

Figure 5 Hypergraph and Cost Parameters

Edge Pruning using Cost Constraints

Edge pruning aims to reduce the length of each edge of the input hypergraph, using the specified price constraint. Vertices which can't be contained in any answer satisfying the price constraint may be deleted from the hypergraph.

First observe that the minimum cost of any transversal of the hypergraph is given by the following:

$$\mathcal{M}inCost_{\mathcal{F}}^{\mathcal{W}} = \sum_{f \in F} min(\{ P_f^{\,w} \mid w \in W \cap f \})$$

where the symbol $P_f^{\,w}$ represents the price of Web Service w on functionality f. Furthermore, the minimum cost of a transversal containing a vertex w_i from edge e_i ($e_i = f$) is given by

$$\mathcal{M}inCost_f^{w^i} = P_f^{\,w} + \mathcal{M}inCost_{\mathcal{F}-f}^{\mathcal{W}}$$

Thus, if $\mathcal{M}inCost_f^{wi}$ is greater than the maximum value allowed for the price of a set of Web services, then we can conclude that w may be deleted from e_i, since it cannot participate in any transversal whose cost is less than the maximum, no matter what other vertices are placed in the transversal.

For the example in Figure 5, vertices 'h' and 'z' can be deleted from edge1, since the minimum cost of a transversal with 'h' would be 14.7 and the minimal cost of transversal with 'z' would be 17.7. These are both higher than 14. More pruning could be performed to remove vertices in other edges. Thus, the edge pruning process generates a smaller sized hypergraph H_{EP} (shown in Figure 6 b) from the original input hypergraph H (Figure 6a). This is beneficial since the time taken to generate transversals is highly dependent on the hypergraph size.

Hypergraph H	Hypergraph H_{EP}
edge 1: a c g h z	a c g
edge 2: b d i w	b d i
edge 3: c p m n k	c p m
edge 4: d q m j l	d q m j
edge 5: e f r	e f
edge 6: f o s t z	f o s
Original Hypergraph (a)	HAfter edge pruning (b)

Hypergraph H1 a c g b d i c p m d q m j	Hypergraph $H1_i$ b d a c g c d
HypergraphH2 e f f o s	Hypergraph $H1_j$ d q m a c g b d i c p m
After horizontal partitioning (c)	After vertical paritioning (d)

Figure 6. Example of pruning strategies

Horizontal Partitioning

Horizontal partitioning decomposes an input hypergraph into its maximally connected subhypergraphs. First we give some definitions that are needed to describe the process.

Definition 3 Connected Edges

An edge $e_i \in \mathcal{E}$ is **connected** to an edge $e_j \in \mathcal{E}$ if either $e_i \cap e_j \neq \phi$, or there exists an $e_k \in \mathcal{E}$, such that e_i is connected to e_k and e_k is connected to e_j

Definition 4 Maximally Connected Subhypergraph

A hypergraph $\mathcal{H}_S = (\mathcal{V}_S, \mathcal{E}_S)$, is a maximally **connected subhypergraph** of hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ if $\mathcal{E}_S$ is a subset of $\mathcal{E}$, $\mathcal{V}_S$ is a correponding appropriate subset of $\mathcal{V}$, all pairs of edges in $\mathcal{E}_S$ are connected and no edge in $\mathcal{E}_S$ is connected to an edge in $\mathcal{E}$-$\mathcal{E}_S$.

Observe that the transversals of any two connected subhypergraphs have an empty intersection, since they cannot have any vertices in common. In figure 6 (c), horizontal partitioning finds edge 1 intersects with edge 3, edge 2 intersects with edge 4, edge 3 intersects edge 4, etc. These four edges form a maximally connected subhypergraph H1. Similarly, H2 is another maximally connected subhypergraph generated (also in Figure 6 (c)). Obviously H1 and H2 have no vertices in common.

Horizontal partitioning is useful since the transversals of the maximally connected hypergraphs can be computed independently and then the transversals of the original hypergraph are just the cross product of the transversals of all the subhypergraphs. The more horizontal partitions that can be formed, the more effective this optimization can be, since the expensive depth first enumeration (described shortly) will be limited to the smaller subhypergraphs. Observe that in the original hypergraph, horizontal partitioning may be initially impossible, but it may become possible after edge pruning. Also, as we discuss later, for real applications, it is quite likely there may be many different subhypergraphs, due to edges having very small intersection sizes.

Vertical Partitioning

As its name suggests, instead of taking horizontal slices through the hypergraph, vertical partitioning partitions the hypergraph into a number of vertical slices, the union of which is equivalent to the input hypergraph. Given a hypergraph H, we begin by forming an ordering on the vertices in the hypergraph: $w_1<w_2<\ldots<w_n$, such that cost $(w_1)<$ cost$(w_2) < .. <$ cost (w_n), where cost(w_i) is the average cost of Web service w_i across all functionalities it offers. The first vertex in the ordering is associated with the lowest average cost and the final vertex in the ordering is the one with the highest average cost.

We then vertically partition the hypergraph as follows:

H_{w1} = H with all vertices >= w_1 deleted
…
H_{wi} = H with all vertices >= w_i deleted
…
H_{wn} = H all vertices >= w_n deleted.

We require that any transversal found for H_{wi} must have the web service w_i appended. In Figure 6c, hypergraph H_1, it has the ordered set of {a, b, c, d, g, i, p, q, m, j} vertices. For the partition induced by vertex i, all vertices ordered after and including i are deleted. The resulting hypergraph $H1_i$ is shown in Figure 6(d).

To understand why vertical partitioning is useful, consider the first and last partitions. The first partition H_{w1} has the fewest vertices (only one) and thus this hypergraph is small. The last partition H_{wn} has all the original vertices, but any transversal discovered in it is required to also contain the highest average cost service w_n. We therefore expect this partition to contain few transversals, since it will be harder to satisfy the price constraint.

Depth First Enumeration

After the pruning and partitioning steps have been performed, we need to enumerate all transversals satisfying the constraints from the resulting hypergraph. Vertices within each edge are now ordered from lowest price to highest price (left to right). Candidates are grown in a depth first manner, moving top down, left to right. Each candidate is tested against the cost constraints at each step. If it satisfies, then the candidate set is grown further (if possible) by moving downwards and if the last edge has been reached then the candidate set must be a transversal and the search moves to the right. If at any stage the candidate set does not satisfy the constraints, then the search moves to the next vertex to the right. Once no more vertices exist to the right, the search backtracks one level. With computer transversals of each hypergraph pruned through edge pruning, horizontal partitioning and vertical partitioning, partial results are then unioned together and combined using a cross product across subhypergraphs. Space constraints prevent a full explanation. For more details see [18].

4. Experimental Results

We now give an experimental evaluation of our Web service discovery algorithm. We expect Web service hypergraphs may have the following characteristics: i) Relatively few edges (since the user will not ask for too many functionalities), ii) A much larger number of vertices than edges (since the Web contains many Web services), iii) The frequency of each vertex is low, because it is expensive for providers to offer many functionalities and hence the size of intersections between edges is small. We make the prices of the Web services follow a normal distribution and use the IBM Quest Data Generator to generate the hypergraphs having these characteristics [9].

Table 1 lists the characteristics of the datasets, giving number of edges (#edge), average length of edges (avL), number of vertices (#vert), cardinality constraint value (CC), price constraint value (OC), size of the output (number of transversals satisfying constraints, #output) and number of subhypergraphs (#sub) after edge pruning and horizontal partitioning.

In Table 2, we compare the running times of 5 different algorithms, composed from the techniques discussed. Suppose E=edge pruning, V=vertical partitioning, H=horizontal partitioning, D=depth first enumeration and B=breadth first enumeration. Then versions compared are EHVD, EVD, ED, D and B (a naive breadth first transversal enumeration). All algorithms were benchmarked on a 1GHz Intel PIII, with 2GB of memory. All running times are in seconds.

Looking at Table 2, we see that EHVD is always the most effective, though the differences in running time only become apparent when the input hypergraph is large. The number of subhypergraphs has a strong impact on the algorithm running time, as can be seen from the vastly superior performance of EHVD over

EVD. Pure depth first and pure breadth first are by far the slowest methods and edge pruning significantly improves pure depth first (ED versus D). Vertical partitioning also has a significant improving effect (EVD versus ED). As expected, the running time of all algorithms increases for situations where the constraints are not selective and as the input gets larger.

5. Related Work

We now briefly survey related work in the areas of Web service discovery and computation of hypergraph transversals.

Depending on the description language capability, the service discovery process can be categorized as being either semantic level discovery or non-semantic (syntactic) level discovery. UDDI is an example of non-semantic level discovery that is based on keyword matching. A number of ontology languages have been developed for describing properties and capabilities of Web services [15] and DAML-S [1] is one well-known example. Many semantic level discovery approaches have been developed, e.g. [13, 16] for matching between the provided capabilities of services and service requestors' needs. The discovery space of available candidates (Web Services) can also be categorised as being either a centralized registry approach like UDDI or a distributed approach such as a grid environment or peer to peer. The tradeoffs between the two lie in the Web Service Architecture. In this paper we have followed the centralized approach. A major difference of this work from ours is that we aim to discover combinations of services that can satisfy multiple functionalities, rather than discovering just a single Web service providing a single functionality.

As we have shown, the Web service discovery problem can be viewed as a generalized case of hypergraph transversal computation, where the output must satisfy a collection of cost constraints. There is a considerable amount of work on hypergraph transversals, which principally concentrates on the case of generating either the set of all minimal transversals or generating only a single transversal (or a single covering set) having minimal cardinality [5,12]. Although the precise complexity of the former problem is still an open problem, there exists an algorithm taking quasi-exponential time in the combined size of the input and output [6]. It is NP-complete to compute just a single transversal [5,12]. In contrast, in our work we are not focusing on computing either the minimal transversals or a single transversal, but rather the set of transversals satisfying the specified cost constraints.

Nevertheless, the techniques our approach uses do have similarities with other previous work on computing minimal transversals. Vertical partitioning based on frequency is discussed in [2], as a means of speeding up the computation of minimal transversals. Work in [10] describes a depth first algorithm for computing the minimal transversals. Horizontal partitioning for general hypergraphs is discussed in [14]. Hacid et al [7] also make a connection between hypergraphs and Web services. The difference from our work is i) that they only aim to find a single transversal having minimal cost, ii) the query is specified using Description Logic and the cost measure depends on the number of extra concepts contained in the transversal and not in the query; our cost constraints consider prices of Web services, iii) Their algorithm is based on breadth first enumeration, an improved version of the classical minimal transversal method.

6. Conclusion and Future Work

In conclusion, we have shown how the problem of computing all cost constrained collections of Web services that provide a given set of functionalities can be modeled as a hypergraph transversal problem. We presented a number of partitioning and pruning techniques for discovering cost constrained transversals and showed they have a significant effect on running time. An interesting direction for future work is to consider the incorporation of relationships between groups of Web services in the cost model [11] or the facility to specify other kinds of constraints such as degrees of interoperability.

***Acknowledgments**:* This work is partially sponsored by National ICT Australia. National ICT Australia is funded by the Australian Government's Backing Australia's Ability Initiative, in part through the Australian Research Council.

No.	#edges	avL	#vert	CC	OC	#output	#sub
1	6	10	50	6	6790	9887	1
2	5	10	35	5	106028	10214	1
3	6	12	60	5	12621	13449	2
4	7	11	70	5	16473	8975	3
5	8	10	70	7	20483	15343	3
6	10	8	70	9	24562	16523	3
7	9	9	75	8	24332	45739	3
8	11	14	143	8	27057	12786	4
9	11	15	130	11	27057	13534	4
10	13	10	120	10	27900	29687	4
11	14	18	237	13	32500	39730	6
12	12	10	109	10	33500	55780	4

13	15	20	278	6	34837	44576	6

Table 1: Character of Data Sets

No.	**EHVD**	**EVD**	**ED**	**D**	**B**
1	0.71	0.74	0.82	0.82	0.95
2	*0.74*	0.76	0.82	0.85	1.01
3	0.8	0.83	2.11	2.15	1.97
4	0.64	6.0	10.9	13.90	24.30
5	0.67	6.21	49.76	67.340	129.37
6	0.7	12.51	97.59	155.47	278.56
7	0.73	37.22	149.73	208.6	412.01
8	0.82	4.24	9.4	22.40	57.49
9	0.69	4.55	9.82	20.10	60.55
10	0.79	17.08	21.87	40.01	88.27
11	0.71	27.1	43.43	81.55	159.11
12	0.76	29.09	119.93	328.86	897.54
13	0.76	83.45	160.8	840.33	2239.45

Table 2 Running Time (seconds)

7. References

[1] Ankolekar, A., Burstein, M. and Hobbs, J.R., et al. (2002): DAML-S: Web Service Description for the Semantic Web. *Proc International Semantic Web Conference. (ISWC)*, Sardinia, Italy, LNCS 2342.

[2] Bailey, J. Manoukian, T. and Ramamohanarao, K. (2003): A Fast Algorithm for Computing Hypergraph Transversals and its Application in Mining Emerging Patterns. *3rd IEEE International Conference on Data Mining (ICDM) 2003.*

[3] Balke, W.-T. and Wagner, M. (2003): Cooperative Discovery for User-centered Web Service Provisioning. *Proc. First International Conference on Web Services.*

[4] Eiter, T. and Gottlob, G. (1995): Identifying the Minimal Trasnversals of a Hypergraph and Related Problems. *SIAM Journal on Computing*, 24(6): 1278-1304

[5] Eiter, T. and Gottlob, G. (2002): Hypergraph Transversal Computation and Related Problems in Logic and AI. *Proc. 8th European Conference on Logics in Artificial Intelligence (JELIA 2002)*, pages 549--564. Springer.

[6] Fredman, M.L. and Khachiyan, L. (1996): On the Complexity of Dualization of Monotone Disjuctive Normal Forms. *Journal of Algorithms*, 21(3): 618-628.

[7] Hacid, M.-S., Leger, A., Rey, C. and Toumani, F. (2002) Dynamic Discovery of E-Services in a Knowledge Representation and Reasoning Context. *18èmes Journées Bases de Données Avancées*, Evry, France.

[8] Hoschek, W. (2003): Peer-to-Peer Grid Databases for Web Service Discovery. *Grid Computing: Making the Global Infrastructure a Reality"*, Wiley Press.

[9] IBM Quest, http://www.almaden.ibm.com/software/quest/.

[10] Kavvadias, D and Stavropoulos, E.C. (1999): Evaluation of an Algorithm for the Transversal Hypergraph Problem. *Algorithm Engineering, Third International Workshop,* pages 72–84.

[11] Limthanmaphon, B. and Zhang. Y. (2003): Web Service Composition with Case-Based Reasoning. Database Technologies 2003, *Proc. 14th Australasian Database Conference (ADC),* Adelaide, Australia.

[12] Luigi Palopoli, F. Pirri, C. Pizzuti: Algorithms for Selective Enumeration of Prime Implicants. *Artif. Intell 111*(1-2): 41-72 (1999)

[13] Paolucci, M., Kawamura, T., Payne, T.R. and Sycara, K. (2002): Semantic Matching of Web Services Capabilities. *Proc. International Semantic Web Conference (ISWC02)*, Sardinia Italy.

[14] Rymon, R. (1994): An SE-tree-based Prime Implicant 720 Generation Algorithm. In *Annals of Mathematics and Artificial Intelligence, special issue on Model-Based Dagnosis*, vol. 11.

[15] Sheth, A. and Ramakrishnan, C. (2003): Semantic Web Technology In Action: Ontology Driven Information Systems for Search, Integration and Analysis. *IEEE Data Engineering Bulletin, Special issue on MAKING THE Semantic Web Real*.

[16] Sivashanmugam, K., Verma, K., Sheth, A. and Miller, J. (2003): Adding Semantics to Web Services Standards. *Proc. International Conference on Web Services (ICWS'03)*, page 395-401.

[17] Terziyan, V. and Kononenko, O. (2003): Semantic Web Enabled Web Services: State-of-Art an Industrial Challenges. In: Jeckle, M. and Zhang, L.J. (eds.): *Web Services. ICWS-Europe*, LNCS Vol. 2853, 183-197.

[18] Zhu, Z. (2005). Fast Computation of Interesting Collections of Web Services. Master of Computer Science Thesis, Department of Computer Science and Software Engineering, University of Melbourne.