A Graphical Approach to Providing Infrastructure Recommendations for IT

Ashwin Lall
University of Rochester

Anca Sailer
IBM Research

Mark Brodie
IBM Research

Abstract

*We present SPIRIT, a **S**ervice for **P**roviding **I**nfrastructure **R**ecommendations for **I**nformation **T**echnology. SPIRIT allows maintenance support providers for Small-to-Medium Businesses (SMBs) to recommend solutions which are standardized (SMBs usually cannot afford customized IT solutions), flexible (accommodating as much as possible the customer's existing IT environment), and cost-effective (minimizing the cost of upgrading the customer's environment). SPIRIT works by first aligning the customer's IT infrastructure with a "template" describing the best practices recommended by the maintenance support provider. This step is done using an efficient graph algorithm that finds the most cost-effective transformations of the customer's environment that are consistent with the template. Then the aligned environment can be upgraded by choosing from a standard set of well understood, highly automated (and therefore economical) options. We show that the algorithm performs well on both real and synthetic data.*

1 Introduction

In previous work [14] the SPIRIT framework was developed to address the issue of delivering a ***S***ervice for ***P***roviding ***I***nfrastructure ***R***ecommendations for ***I***nformation ***T***echnology for Small-to-Medium Businesses (SMB). An SMB customer, defined as having less than 1000 employees, usually cannot afford the elevated cost of highly customized IT applications and infrastructure. When an SMB customer needs to modify their IT infrastructure in order to correct a problem or to maintain or improve over-all service availability and quality, their IT maintenance service provider needs to recommend solutions that (1) minimize the risk of unavailability by minimizing the number of changes needed in the customer's environment and by increasing automation, (2) maximize the benefit of applying the resolution, and (3) can be shown to be cost-effective for both the customer and the maintenance service provider.

The SPIRIT framework is explained further in Section 2. It attempts to both (1) optimize for the customer the cost-benefit ratio of the resolutions suggested by the maintenance provider and (2) minimize the service cost for the maintenance provider by providing standardized, automated (and hence cost-effective) solutions. At the heart of the SPIRIT methodology is the need to optimally align the SMB customer's existing infrastructure with a "best practices" template. This alignment problem was formulated in [14] as an optimization problem but was not addressed there. In this paper we describe and analyze a graph-theoretic algorithm to solve the optimization problem efficiently and present experimental results on both real and simulated data.

The rest of this paper is organized as follows. In Section 2 we give an overview of SPIRIT; Section 3 introduces the formulation; we detail the graphical optimization algorithm in Section 4; the experimental environment and results are shown in Section 5; Section 6 presents related work, and we conclude in Section 7.

2 The SPIRIT Framework

In the past, IT maintenance service providers' efforts were mostly related to fixing their customers' hardware and software problems in isolation. Modern enterprise environments increasingly demand more sophisticated support that considers the whole IT infrastructure and its interdependencies. For instance, in a multi-tier e-commerce system (see Figure 1), upgrading the application server may benefit the application business logic and fix its issues at the risk of introducing end-to-end performance degradation due to database overload or incompatibility. Existing solutions that enable maintenance support to provide more elaborate resolutions to the customer are primarily directed at specific resolution niches. For instance, performance problem resolutions focus mainly on run time provisioning [5], capacity planning [10, 11], or limiting traffic access [7, 8] in order to satisfy the service level agreements for the incoming traffic. Solutions related to cost-effective resolutions focus on optimization through server and storage consolidation [6, 13]. There are many approaches to problem reso-

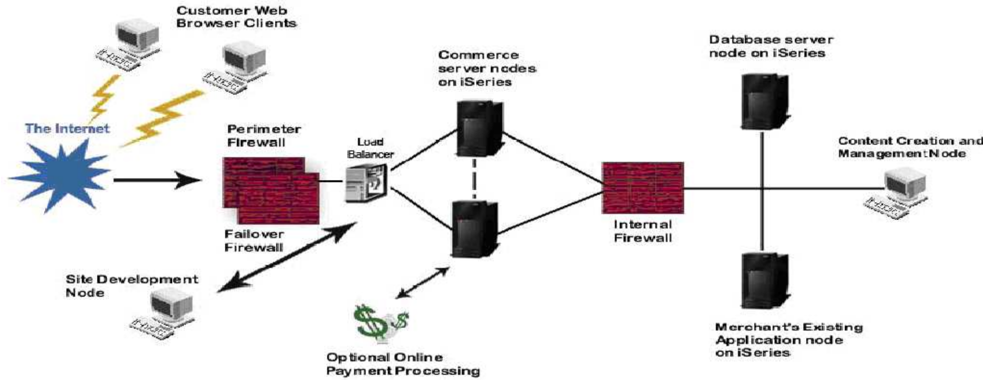


Figure 1. Complex e-commerce IT environment.

lution and cost optimization, however these do not provide a holistic maintenance service approach involving multiple resolution types for a given problem, nor do they optimize across different resolutions. Maintenance deals with a wide range of problem resolutions and what SPIRIT provides is a way of comparing different possible resolutions in terms of their cost, benefit and risk, for both the customer and the maintenance support provider.

The SPIRIT methodology [14] for providing the customer with optimal recommendations includes two *off-line* procedures and two *on-line* procedures, described next.

2.1 Off-line procedures and golden templates

The two off-line procedures are *collect* and *pre-process* (see Figure 2). The data gathered during the *collect* phase include (i) remedies recommended to solve known problems — these can be obtained from manuals [3, 4], web sites [1], or forums [12], (ii) costs of the products supported by the maintenance service provider (these costs, including product cost, operations cost, administration cost, downtime cost, etc, are typically available for TCO calculation [18]), and (iii) IT constraints and dependencies known to exist between products supported by the maintenance service provider.

The *collect* phase is also used to build an IT configuration template describing a specific aggregation of best practices IT configurations recommended by the maintenance service provider, which we refer to as a "Golden Template" (GT). A maintenance provider may use different GTs, individualized per industry or per customer type. Figure 3 illustrates the GT used in our experiments.

A GT includes IT dependencies and constraints that reflect the best practices configuration templates supported by the maintenance provider. The bold nodes in Figure 3 identify the products supported by the maintenance provider considered in our experiments. An example of GT products and their dependencies is: "Web application server M version a.b.c works with database server N version x.y.z". The solid arrows in Figure 3 indicate such configuration dependencies. Lack of an arrow indicates either an unfeasible configuration or a constraint on a potential dependency that is unsupported.

A special type of configuration constraints are the classes of equivalence rules that indicate which products provide similar functionality. Examples of classes of equivalence rules are: "Web application servers are: WebLogic, JBoss, JRun, Tomcat," "Database servers are: DB2, Access, dBase, MySQL, Oracle, SyBase." In Figure 3, the nodes clustered together belong to the same class of equivalence.

Configuration conflicts are another potential type of constraint. Examples of software conflict constraints are: "Windows Defender has issues on Windows Vista," "Websphere AS v5 on AIX conflicts with Oracle Web Services Manager." Finally, dependencies like amount of RAM, hard drive or CPU required by particular products, which need to be summed up on each machine, are indicated by the dashed arrows in Figure 3. Note that not all valid dependencies and constraints are shown in Figure 3 for visibility reasons. Additionally, a typical IT infrastructure scenario would have many more configuration constraints than the smaller examples presented in this paper. Our goal is to capture a wide selection of relevant constraint types; a larger number of instantiations of a given constraint type does not change the results.

The second off-line procedure is *pre-process*. Here the operations for performing common tasks such as software products installation, configuration, upgrade, migration and troubleshooting, are standardized and automated. This is the advantage of a GT over customized IT solutions - the automated operations can be made cost-effective and their benefit evaluated. It is only because the GT reduces the number of possibilities so drastically that it is even feasible for us to consider automation of all these tasks. Doing this in the *pre-process* off-line procedure reduces the run-time

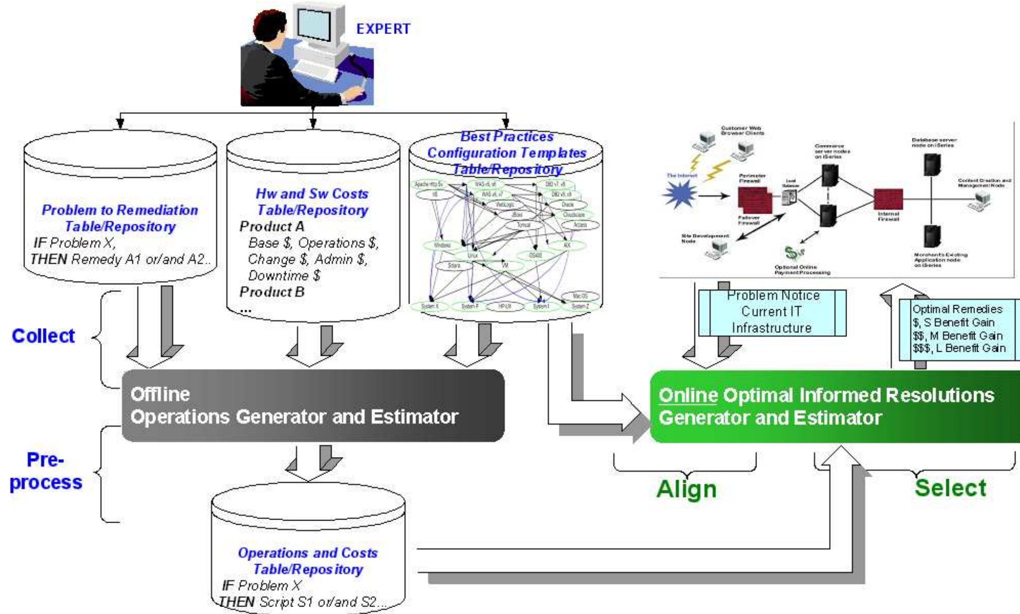


Figure 2. SPIRIT includes two off-line procedures and two on-line procedures. Our focus is on the on-line alignment procedure.

labor cost of the maintenance service and the time to repair the problem.

2.2 On-line procedures and cost-benefit optimization

The core of SPIRIT consists of the two on-line steps, *align* and *select* (see Figure 2). The on-line procedures take place after a problem reported by a customer (or, proactively, a potential failure) has been identified and the root cause determined. In the first step SPIRIT *aligns* the customer's current IT infrastructure with the maintenance service's supported products by making the customer's infrastructure consistent with the provider's GT, using the minimum number of changes and taking into account any special customer restrictions. Aligning the customer's IT infrastructure with the GT is a key facet of our solution. It helps to match the SMB customer's cost expectations and minimize the maintenance provider service cost. This optimized migration is the major challenge for the maintenance service provider because of the multiple potential customer restrictions, e.g., minimum costs of changes, minimum or no changes of the products directly related to their applications, software restrictions, etc. Moreover, this optimization is the key enabler of the optimized remedy recommendations service since it makes possible the use of the automated remediation operations built off-line for the GT.

Note that most IT infrastructures rely on redundancy at any of the stack layers, e.g., application, middleware, hardware. In such cases, we consider for the migration optimization only the base pattern of the IT infrastructure, with unique products, rather than the whole IT environment with duplicates. The results of the alignment are applied seamlessly to all the infrastructure products, duplicates or not. Our focus in the rest of the paper is on the solution for this alignment of the base pattern of the IT infrastructure to the maintenance provider GT and the generation of the optimized target customer infrastructure(s).

The second on-line step consists of *selecting* from the multiple identified resolutions for all target infrastructures the optimal ones, from a cost-benefit tradeoff perspective. The advantages of our solution for the customer are that for each resolution option the recommended resolution is optimized with respect to the cost and benefit of the resolution; also the customer can compare and choose between a range of resolution options based on their cost and benefit. The advantage of our solution for the maintenance support service lies in reducing the maintenance cost by first aligning the customer's IT infrastructure to one of a limited number of best practices templates and then applying the chosen resolution using a standardized, highly-automated, and inexpensive process.

3 Formulation

In this section we formally introduce the notation and formulate the problem of aligning the customer's IT environment with a GT.

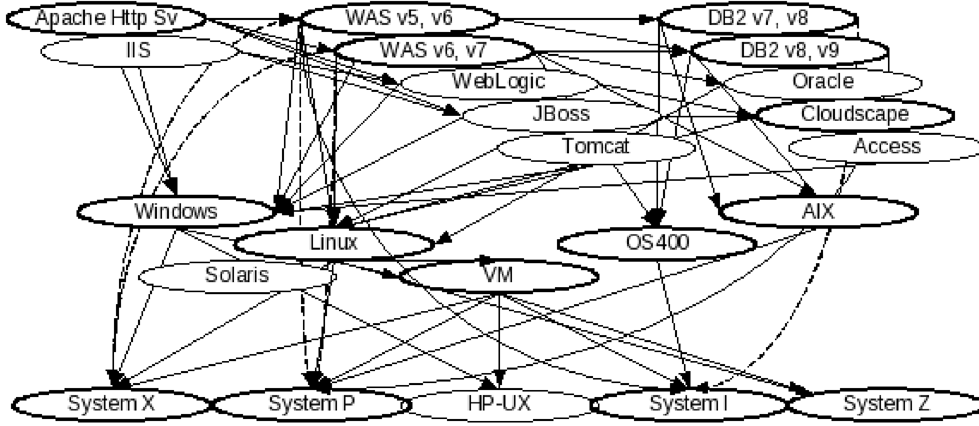


Figure 3. Example of Golden Template as a specific instantiation of best practice IT infrastructure.

3.1 Notation

We represent our universe of objects (IT products) as

$$V = \bigcup_i C_i,$$

where the C_i are pair-wise disjoint classes. Each class represents a set of objects that can be substituted for one another and designates a class of equivalence as described in Section 2.1.

We are given the Golden Template (GT) graph $G = (V, E)$, which expresses the relationships between objects in V. An example of such a graph is given in Figure 3.

We are also given the customer IT environment as a graph $S = (V', E')$, where V' is the set V with some objects appearing zero or more times. That is, for any node $v \in V$, V' may have k copies of this node represented as $v^1, \ldots, v^k$, for some k. We will refer to these as separate *instances* of the same object. Since a particular product may appear several times in the customer environment (e.g., there may be multiple copies of WAS v6 running on different machines), we use this concept of instances to distinguish them. The dependencies in the customer environment can be obtained using an automatic dependency discovery tool [2].

Objects within a class can be substituted for one another, and for this we define a *substitution cost function* called

$$c : V \times V \to \mathbb{R},$$

where $c(v_1, v_2) = \infty$ for v_1 and v_2 in separate classes. In practice, this function will have the property that, for all x, $c(x, x) = 0$ (since there is no migration cost when there is no product change). The substitution cost is computed by taking into account all relevant costs, such as product costs, administration costs, downtime costs, and weighting them appropriately to reflect the customer's preferences or criteria of optimization; e.g., if the customer requires that particular products not be changed, their substitution cost can be made very large. We extend the cost function to the domain $V' \times V'$, treating each instance identically to the original.

Lastly, we have an *edge cost function*

$$e : V' \times V' \times V \times V \to \mathbb{R}$$

which represents the cost of replacing a link between two objects in the customer's current environment by a link between two objects in the GT, i.e., products supported by the maintenance provider. It has the property that if x' is an instance of x and y' is an instance of y, then $e(x', y', x, y) = 0$. That is, if the pair remains unchanged, there is no edge cost.

Note that the substitution and edge costs are among the data collected in the off-line phase of SPIRIT described above.

3.2 Problem Statement

Our goal is to transform the current customer environment graph S to a target graph T by replacing some vertices in S by objects in the corresponding class. An example of such an environment transformation is given in Section 5 and illustrated in Figure 5. The constraint that we place upon T is that every edge in it must also be an edge in the GT graph G. Our goal will be to find the minimum cost transformation (assuming that one exists).

More formally, we want to find a function $f : V' \to V$ that re-labels the vertices of S in such a way that, for every edge $(v_1, v_2) \in E'$, we have that $(f(v_1), f(v_2)) \in E$, and such that the total cost

$$\sum_{v \in V'} c(v, f(v)) + \sum_{(u,v) \in E'} e(u, v, f(u), f(v))$$

is minimized.

4 Algorithm

This problem can be shown to be NP-complete (we omit the proof here due to lack of space). Since it is unlikely that a polynomial-time solution will be possible for it, we shift our focus to methods that efficiently prune the search space. We propose a heuristic algorithm for the problem that in practice gives the optimal solution quickly (assuming that one exists), but for which we cannot guarantee a running time any shorter than the brute-force algorithm.

We need some terminology from graph theory. A *cut vertex* (or *articulation vertex*) is one whose removal, along with all its incident edges, results in a disconnected graph (see Figure 4). This type of vertex has been studied extensively in the theory of building robust networks. Of course, an arbitrary graph need not have a cut vertex — any graph that does not contain a cut vertex is called *bi-connected*.

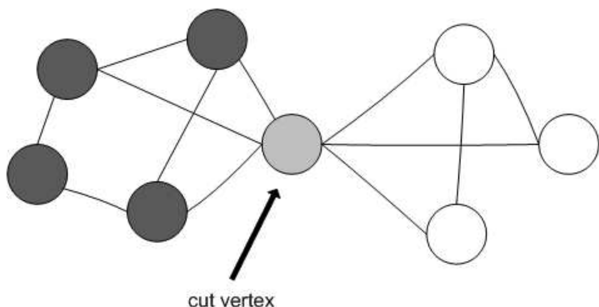


Figure 4. The cut vertex (in light gray) disconnects the dark nodes from the white ones

Our algorithm exploits the fact that the customer graphs we look at are very far from bi-connected, meaning that they tend to have many cut vertices. Note that any graph that has a vertex of degree one must have a cut vertex. In our setting such vertices often occur since the individual machines are linked with their operating system only. More generally, it is usually possible to partition the functionality of different clusters of machines along one (or a few) vertices. The empirical results section confirms the common presence of cut vertices in our target customer environments.

Although cut vertices are undesirable in networking (they represent a single point of failure, disconnecting the network), they are useful to us because they allow the problem for large graphs to be split into smaller sub-problems. Our algorithm iterates through the vertices of the customer environment graph S, identifying each cut vertex and the relative size of the sub-graphs it cuts the original graph into. We want to find a cut vertex that splits the original graph into two sub-graphs as close in size as possible. By doing this, we can reduce the larger problem to two smaller problems of roughly even size and solve them recursively. A naive way to identify cut vertices in a graph is to iterate over the vertices and test, for each vertex, whether removing it disconnects the graph, which can be done in $O(n(n+m))$ time, where n is the number of vertices and m the number of edges in the graph. However, there are more efficient algorithms known that can identify cut vertices in $O(n+m)$ time [9].

Once the best cut vertex is found (with ties broken randomly), we split the graph into two sub-graphs, each including a copy of the cut vertex and its edges to that sub-graph. Then, we map the vertex to each object in its equivalence class. We recursively solve the problem on the two induced sub-graphs. If there is a feasible solution for both the sub-graphs, then we stitch the graphs back together, compute the cost of the solution using the c and e cost functions as shown in Section 3.2, and compare the cost of this solution to the best so far.

The search problem is further simplified by the fact that every edge in the target graph T must also be an edge in the GT G, constraining the possibilities for each vertex. As we try different possibilities for a cut vertex, we appropriately prune the list of possible objects at each of its neighbors. If the set of possible objects for a neighbor of the cut vertex gets pruned down to a single element, we recursively prune its neighbors. This, too, reduces the search space considerably.

For graphs that do not have a cut vertex (i.e., bi-connected graphs) and graphs that are smaller than a certain size (e.g., 4 vertices), we resort to the naive depth-first search to exhaustively find the solution. In the worst case, our algorithm does not find a cut vertex for the original graph and is identical to brute-force depth-first search. The exact algorithm is given in Algorithm 1.

Our recursive solution always returns the optimal solution since it is guaranteed to give the optimal solution for the sub-problems. (A formal proof can be given by induction — we omit it here.)

To give an idea of the improvement of our algorithm over the simple brute-force alternative, we analyze it assuming that we are always able to find a cut vertex in each graph that splits the graph into equal-sized sub-graphs. Of course, we can give no guarantee that such a cut vertex exists, but empirically we find cut vertices to be common.

Let n be the number of vertices in S, let k be the maximum size of the equivalence classes C_i, and d the maximum degree of any vertex in S. Then the upper bound on the running time of a naive depth-first search is $O(k^n)$.

Let T denote the running time of our algorithm. Then we get the recursive relationship

$$T(n) = 2kT(n/2) + O(dn),$$

since we split the graph into two equal-size sub-graphs (taking time $O(dn)$) and try k possibilities for the cut vertex in both the sub-graphs. Solving this recursion, we get that

Algorithm 1 The Cut Vertex Algorithm

CutSearch(G, c, e)

1: **if** G is small or has no cut vertex **then**
2: Run the brute-force algorithm and return its result.
3: **else**
4: Search for the cut vertex v that minimizes the difference in size of the two induced sub-graphs.
5: Let C_v be the class of vertex v.
6: **for** each $v' \in C_v$ **do**
7: Split G into two sub-graphs G_1 and G_2 along v and re-label vertex v with v'.
8: Make recursive calls CutSearch(G_1, c, e) and CutSearch(G_2, c, e).
9: If both graphs have solutions, put both solutions together to get a solution for G, and compare the cost to that of the best solution so far.
10: **end for**
11: Return the best solution found.
12: **end if**

the running time of our algorithm is $O(kdn^{1+\log_2 k})$, which is polynomial in n, assuming that the maximum number of possibilities for any vertex is constant. This is a huge potential improvement over the naive exponential-time solution.

Finally, note that our algorithm can be generalized from a single cut vertex to sets of vertices (e.g., identifying *pairs* of vertices the removal of which disconnects the graph). However, we decided not to pursue this direction because of the additional overhead in the resulting algorithm together with the efficiency of the simpler single cut vertex algorithm.

5 Empirical Results

We implemented our optimization algorithm, along with the naive depth-first search solution, and tested the performance of both algorithms. All the experiments were run on a 1.59 GHz Intel Pentium processor with 1 GB of RAM running Windows XP.

The environments that we considered were those of a typical SMB eCommerce infrastructure: several machines running HTTP Servers connected to Application Servers, which in turn were connected to multiple databases. Both real as well as synthetically-generated IT infrastructures were used to test SPIRIT.

5.1 Test Data

To run our experiments we need two types of data. First, we need the constraint and cost data for the specific environment, describing which software products are compatible with one another, what the cost of moving from one software/OS/machine to another is, and the cost of configuring softwares with one another (e.g., installing WAS v6 on Windows, or configuring WAS v5 with DB2 v7). We manually collected this data from various websites and IBM Redbooks, using plausible estimates for missing values.

We also need examples of IT infrastructures that are not compatible with the GT, so that we can apply SPIRIT to compute how to optimally migrate them to fit the GT. To this end, we obtained two real environments from labs in IBM (one of which is shown in Figure 5).

Next, we generated input customer environments synthetically. This allows us to test the algorithm on many different examples, and to study its performance with variation in problem size. For a fixed graph size, some of the nodes were designated machines, and an operating system and one or more middleware or applications were placed on them randomly. Then, valid dependencies between the middleware and applications were added randomly, ensuring that the dependencies were realistically plausible. The resulting graph in each case was an environment in an arbitrary state (perhaps with incompatible software) that we could now transform to fit the GT.

5.2 Evaluation Metrics

To compare our optimization algorithm against the simple brute-force approach, we make use of two metrics: *program execution time* and *number of configurations* searched.

The *program execution time* is simply the wall-clock time that the algorithm took to find the optimal solution. The *number of configurations* is the total number of subgraphs solved in each case. These two metrics give different perspectives on the comparison of the algorithms. The first gives the actual time that the algorithms take to run, taking into account all the processing done to set up the recursions. The second counts only the actual feasible configurations encountered, and hence is more independent of the implementation of the algorithms.

5.3 Experimental Results

We first examine the performance of our optimization algorithm on a real environment. In Figure 5 we show how our algorithm transforms the existing environment into one that is consistent with the GT. Although the costs of making individual changes are not shown here, this transformation is the optimal migration possible. Next, we use synthetically generated environments to study what happens as the number of nodes in the customer environment varies. We generate synthetic environments with different numbers

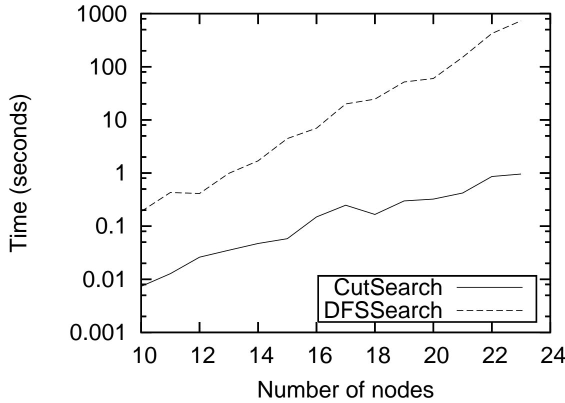


Figure 6. Running times (y-axis in log scale)

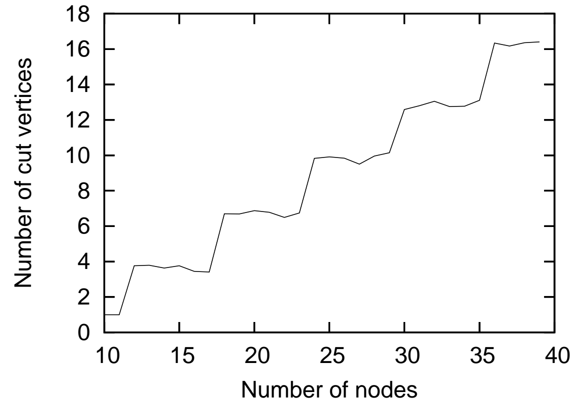


Figure 8. Number of cut vertices

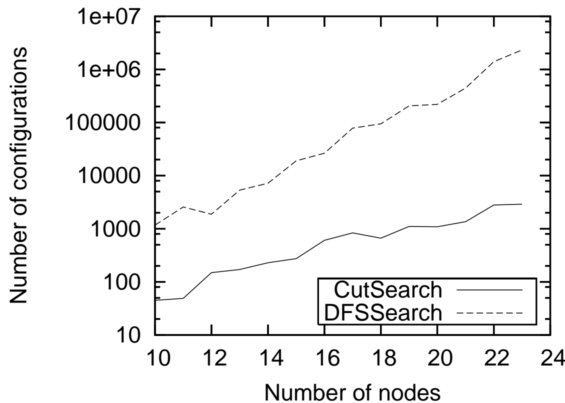


Figure 7. Configurations tried (y-axis in log scale)

of nodes, i.e., varying numbers of machines, operating systems, and middleware. For each problem size that we studied, we generated one hundred environments and computed the metrics' average across them.

In Figure 6 we see the variation in the amount of time both algorithms take. Note that the y-axis is in log scale. When the synthetic environments reach about two dozen nodes, the brute-force approach takes close to an hour to complete. Our algorithm, on the other hand, consistently finishes in under a second.

Similarly, if we compare the number of configurations considered, our algorithm performs several orders of magnitudes better, as illustrated in Figure 7.

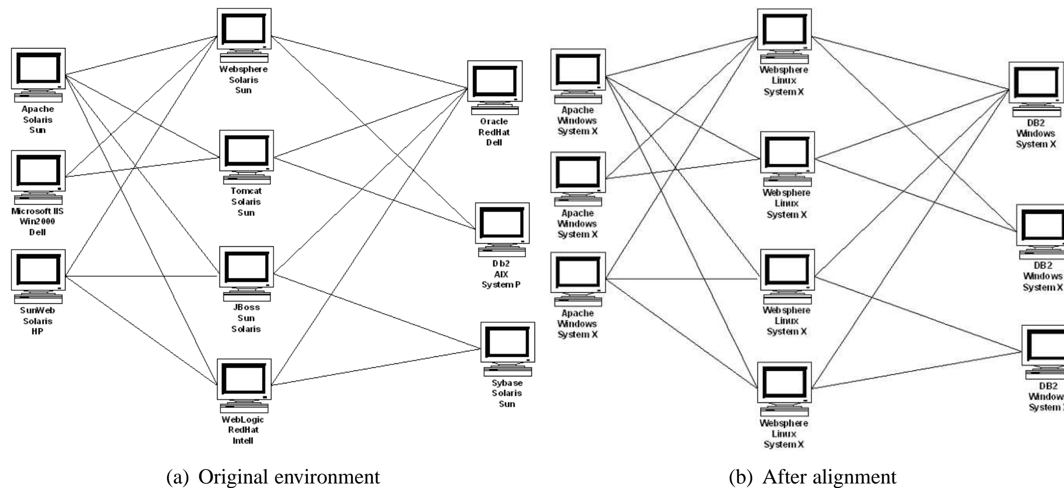
Recall that we had earlier claimed that the environments that we are interested in have many cut vertices. The average number of cut vertices is shown in Figure 8. The graph looks like a step function because the number of cut vertices is proportional to the number of machines in the environment, which was made to increase at regular intervals in the number of nodes. Additionally, we found that less than one percent of the graphs that we generated did not have a cut vertex. Hence, for the type of graphs that are likely to arise in customer environments, there are typically many cut vertices.

6 Related Work

As discussed earlier, most existing solutions have narrow applicability, such as run time provisioning [5], capacity planning [10, 11], restricting traffic access to maintain service level agreements [7, 8], and server and storage consolidation [6, 13]. SPIRIT, on the other hand, has broad applicability to the entire customer environment. Additionally, SPIRIT provides multiple resolution options, each with cost/benefit tradeoffs.

In [16], a similar problem is addressed in the context of migration in Service Hosting Environments. The solution proposed is to use a model, called the System Service Configuration Model, to describe the dependencies and configuration parameters. SPIRIT avoids dependence on a model (which may need to be updated frequently) by making use of automatic tools such as the Tivoli Application Dependency Discovery Manager (TADDM) [2] to obtain its data.

Cut vertices are commonly studied with the view of removing them from the network [15, 17]. A distributed mechanism is proposed in [15] to remove cut vertices from an overlay network. In [17], a network-design process is presented that builds networks with high connectivity by avoiding cut vertices. To the best of our knowledge, the disconnecting property of cut vertices has never been exploited to simplify a graph search problem by facilitating recursion on smaller sub-problems.



(a) Original environment (b) After alignment

Figure 5. Optimal alignment of a real environment (base pattern only, duplicates not shown)

7 Conclusion

We presented a maintenance service method for offering optimal resolution options for issues in the customer's IT environment. To do this, information such as the cost, benefit and complexity of change is evaluated by SPIRIT for each resolution option and the optimal cost-benefit resolutions are provided to the customer for selection. Additionally, we align the customer's IT infrastructure to a limited number of best practices templates, thus enhancing the reliability of the customer's environment and reducing the maintenance cost.

The advantages of SPIRIT for the customer are that for each resolution option the recommended resolution is optimized with respect to the cost and benefit of the resolution. The advantage of SPIRIT for the maintenance support service lies in reducing the maintenance cost by first aligning the customer's IT infrastructure to one of a limited number of best practices templates and then applying the chosen resolution using a standardized, highly-automated, and inexpensive process.

In this paper we focus on the alignment step of SPIRIT, presenting an algorithm that solves this hard problem efficiently with a heuristic graphical algorithm. We show that the algorithm performs well by performing experiments on both real and synthetically generated environments.

References

[1] IBM Support. http://www.ibm.com/support/troubleshooting/us/en/.

[2] TADDM. http://www.ibm.com/software/tivoli/products/taddm/.

[3] DB2 Warehouse Management: High Availability and Problem Determination Guide. SG24-6544-00, Redbook, 2002.

[4] Websphere application server v6 problem determination for distributed platforms. SG24-6798-00, Redbook, 2005.

[5] C. Adam, R. Stadler, C. Tang, M. Steinder, and M. Spreitzer. A service middleware that scales in system size and applications. In *Integrated Network Management*, 2007.

[6] M. Badaloo. An examination of server consolidation: trends that can drive efficiencies and help businesses gain a competitive edge. *White paper on IBM Global Services*, 2006.

[7] B. Callaway and A. Rodriguez. Enable XML Awareness in WebSphere Extended Deployment With WebSphere DataPower SOA Appliances. *White Paper on IBM developerWorks*, 2006.

[8] Y. Diao, J. Hellerstein, and S. Parekh. Stochastic modeling of Lotus Notes with a queueing model. In *Computer Measurement Group International Conference*, 2001.

[9] S. Even. *Graph Algorithms*. W. H. Freeman & Co., New York, NY, USA, 1979.

[10] M. Goldszmidt, D. Palma, and B. Sabata. On the quantification of e-electronic commerce. In *EC*, 2001.

[11] E. Hubbert and J.-P. Garbani. *Sustaining Application Performances: The Capacity Planning Software Market*. Forrester Research, 2007.

[12] IBM Website. Forums and community. http://www.ibm.com/developerworks/.

[13] A. Kochut, K. Beaty, and N. Bobroff. Dynamic placement of virtual machines for managing SLA violations. In *Integrated Network Management*, 2007.

[14] A. Lall, A. Sailer, and M. Brodie. Spirit: Service for providing infrastructure recommendations for it. In *IEEE NOMS*, 2008.

[15] X. Liu, L. Xiao, A. Kreling, and Y. Liu. Optimizing overlay topology by reducing cut vertices. In *ACM NOSSDAV*, 2006.

[16] Q. Ma, Y. Li, K. Sun, and L. Liu. Model-based dependency management for migrating service hosting environment. In *IEEE International Conference on Services Computing*, 2007.

[17] K. T. Newport and P. K. Varshney. Design of survivable communications networks under performance constraints. *IEEE Transactions of Reliability*, v. 40, 1991.

[18] T. Pissello. *The Business Value of HP-UX 11i: HP-UX 11i on Integrity Servers vs. IBM AIX 5L on eServer*. Alinean Inc., 2006.