RC24309 (W0707-105) July 17, 2007
Mathematics

IBM Research Report

Approver Selection Optimization for Access Management

Walter C. Dietrich, J. P. Fasano, Jon Lee
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

APPROVER SELECTION OPTIMIZATION FOR ACCESS MANAGEMENT

WALTER C. DIETRICH, J.P. FASANO & JON LEE

Abstract. We describe the problem of matching approvers to access requests in a dynamic environment at IBM. We cast the problem as a minimum-weight set-covering formulation and develop an integer-programming methodology to efficiently solve instances in a real-time environment. Our solution, which has been successfully deployed at IBM, uses state-of-the-art optimization tools from COIN-OR.

Introduction

In many businesses, accesses to applications and data must be controlled so that only authorized people are allowed to see and manipulate the data. This is a well-understood problem and can be solved with existing tools such as Database Management Systems and Application Servers. Managing authorization is a more subtle problem. In general, before someone is authorized, someone else must decide whether the person should be authorized. In large organizations, the identity of the person who makes a particular authorization decision may be based on the application, the data, and the operations to be performed on the data. For example, if an application contains data for two departments, the manager of each department might make authorization decisions for their own department's data, and the managers' manager might be able to make authorization decisions for both departments' data.

Some applications require fine-grained control over both access and authorization. Imagine a system that contains financial data for a multinational company that contains multiple divisions that span many countries. Some people would only have access to one department's data. Some people would only have access to all of the data that pertains to a certain country. Some people would only have access to data that pertains to a certain division. Some people would only have access to data that pertains to certain divisions in certain countries. In this situation, the identities of the people who can make authorization decisions depend on the set of data that will be accessed. The situation can become more complicated because of hierarchies: data might be grouped by geographic region and by organizational constructs such as division, subdivision and department. While data access is important, operations on the data are also important: one person might be allowed to update data, while another person might only be allowed to view data.

One way to control data operations is to group operations by role. For example, in a payroll application, an employee might be able to change their paycheck's destination (bank, address, etc.), a manager might be able to change the employee's salary, and a payroll clerk might be able to change both.

As can be seen from the above examples, many factors must be taken into account when authorizing access, and these factors can come into play when deciding who can authorize access. When a large number of applications or users are involved, automation can make the authorization process more efficient, and it can improve accountability and auditability.

At IBM, we designed and implemented a new tool for entitlement and access management in 2005. Although there were multiple existing applications for authorization management, we needed a system that had more flexibility and functionality. The system needed to support a wide variety of applications, and it needed to be able to add new applications without code changes. The system needed to support complex and dynamic sets of authorization rules without code changes. The system also needed to support sequences of authorization steps in an approval workflow, in which each step can have a different set of authorization rules, with optimization being invoked at each step. The tool has

Date: July 28, 2007.

been in production for a year. It now manages authorizations for more than 70 applications. It has been used to manage application access for more than 40,000 different users.

In what follows, we formalize the problem, demonstrate how it can be cast as a set cover problem, and describe the algorithm and implementation that we put in place at IBM.

In §1, we set our notation. In §2, we cast our problem as a minimum-weight set cover formulation. In §3, we describe our ILP (integer linear programming) based solution. In §4, we describe solver enhancements. In §5, we describe a further use of our solution to reduce sets of approver rules. In §6, we propose a method for updating approver weights to achieve target utilizations. In §7, we describe software integration business results.

1. Notation

We have a finite set of *attributes* D. Some commonly used attributes are Country and Job Role. The dimension d of D is the number of attributes. The "width" in dimension k $(1 \leq k \leq d)$ is the number n_k of allowed values for attribute k. A *point* is a specification of an allowed value for each attribute. Each point represents a set of attribute values containing one value for each attribute. The *attribute space* H is the set of points. Let $N := \prod_{k=1}^{d} n_k$ be the number of points in the attribute space. A *slice* of the attribute space is a specification, for each attribute, of an allowed value or a " * " (i.e., wildcard). An example of a slice is a single point. Another example is a *hyperplane* (in which case all but one attribute has the value " * ").

Slices can be used in two different ways: they can be used to represent choices by end users who wish to receive access to an application, and they can be used to represent the authority of people who grant access. For example, if a user requests access to data for Japan and wants to access the data as an Accountant, that could be represented using a slice containing "Japan" and "Accountant." (In the example, the attributes are Country and Job Role.) Slices are also used in approver "rules," which specify the authority of people who grant access. Each rule, associated with an approver (or an approver code that maps to an approver name), is a slice. For example, if "John Doe" can approve (or reject) requests for the Japan data for any Job Role, then the slice ("Japan", "*") is associated with the approver "John Doe".

In most cases, the rules provide an approver for every valid request slice. However, in order to meet business requirements, the new system has configuration options that specify what happens if a request slice does not have an approver. For some applications, no approver is required if one has not been defined. For other applications, an approver is required, so the requested slice will be rejected.

A request R comprises a set of slices. Like a request, an *approver* A also comprises a set of slices in attribute space. For an approver, a slice is also called an *approver rule.* The set of available approvers is $\mathcal{A}$.

We assume that requests come in, one at a time, and that we must assign each request to a set of approvers in an online fashion.

An approver rule r *covers* a request slice a if for every attribute, whenever the specification of r is not " * ", the specification of that attribute in a matches the value of that attribute in r. An approver A *covers* a slice q of a request R, if some rule a of A covers q. A set of approvers $\hat{\mathcal{A}} \subset \mathcal{A}$ *covers* a request R if every slice of R is covered by some approver in $\hat{\mathcal{A}}$.

In our formulation, we are assuming that a slice is not "splittable" — it must be covered wholly by some approver. For example, if the specification of some attribute is " * ", then we can not cover this slice by a set of approver rules that cover, piecemeal, the individual allowable values for that attribute. This was given to us as a business requirement, though it is certainly possible to work with a more flexible formulation.

We assume that a weight $w(A)$ is assigned to each approver A that encapsulates the cost of assigning A to a request. Weights are specified at run time. Note that we could interpret an approver as a type of approver, of which there could be many. Then we might weight such a set of identical approvers according to the current workload of that set — and then treat the apportionment of the workload to that set as a postprocessing step.

When assigning weights to approvers we should pay careful attention to issues of this type: If approver A_1 has rules that are the disjoint union of the rules of approvers A_2 and A_3, and a request R includes all rules of A_1, then depending on which is the preferred assignment, we should consider which we want to be greater $w(A_1)$ or $w(A_2) + w(A_3)$.

One possible static strategy is to have weights that increase exponentially according to the power of the rules for an approver. Using a base of 10, a reasonable exponent for an approver is to count the number of different fields that are " * " in some rule for that approver. This should to some extent mimic the strategy of trying to push approvals down to the lower levels of the partial order of approver rules. In §6, we discuss another approach to weight setting.

In any case, when a request R comes in, what we desire is a minimum-weight set of approvers that covers all of R — or as many slices as can be feasibly covered by the given approvers.

2. A min-weight set-cover formulation

We cast our problem as a minimum-weight set-cover formulation. To learn a bit about set cover see [3] and [4], for example. We define an $|R| \times |\mathcal{A}|$ 0/1-valued matrix M. Each row r of M corresponds to a slice of the request R. Entry $M_{r,A}$ is 1 if approver A can approve slice r of the request. So column A of M just picks out (with 1's), the slices of R that approver A can approve. With this notation, letting x be a 0/1-valued $|\mathcal{A}|$-vector of variables, and w be the $|\mathcal{A}|$-vector of weights, our problem is just:

$$\begin{aligned} &\min\ w^T x \\ &\text{subject to} \\ &Mx \geq \mathbf{e} \\ &x \in \{0,1\}^{|\mathcal{A}|}, \end{aligned}$$

where $\mathbf{e}$ is a vector of all 1's.

Of course we can drop columns (i.e., approvers) that contribute no points to a request. Similarly, we drop request slices (and report this) when there is no approver with a rule that can cover the request slice.

We solve instances of this formulation using open-source optimization code from COIN-OR (see [1]). In particular, we employ the C++ libraries `Clp` and `Cbc` for the solution of LPs (linear programs) and ILPs (integer linear programs), respectively. `Cbc` is a state-of-the-art library based on the *branch-and-cut* methodology. `Cbc` makes use of `Cgl` to generate cutting planes which speed the solution of the ILPs. These base solver libraries are freely available (under the Common Public License, see [2]) and have been very reliable for the problem instances encountered in this application.

3. Our Three-Phase Approach

We developed a three-phase approach to safeguard against the possibility of difficult instances arising.

(1) Solve the LP relaxation and round up the solution components, yielding a candidate feasible solution.
(2) Solve the ILP restricted to the variables indexed by the candidate solution in Phase 1, yielding a possibly improved candidate feasible solution.
(3) Solve the full ILP producing the optimal solution.

We provide options to control the effort spent in each of Phases 2 and 3. If either a time limit or branch-and-cut node limit is reached in Phases 2 or 3, the best solution determined in the partial search is returned.

Note that the Phase 1 solution obtained above may contain some redundant approvals. It is just used to quickly get a decent feasible solution. Phase 2 cleans this up and finds a good solution from which no approver can be deleted without uncovering some request slices. Phase 3 finally finds the optimal solution. Each phase helps the solver in the subsequent phase, and if resource limits are reached in the computation, good solutions are still available.

4. Solver Enhancements: Cuts and Heuristics

`Cbc` has a variety of heuristics and cutting-planes which, when judiciously used, can improve the performance in its solution of integer linear programs.

We expose control of the ones that may be useful for set-covering formulations through control parameters. In particular, we expose control of `Cbc` heuristics: `GreedyCover`, `Rounding`, `LocalSearch` and `FeasibilityPump`, as well as the `Cgl` cutting-plane generators `CglGomory`, `CglOddHole`, `CglRedSplit`, `CglMixedIntegerRounding` and `CglProbing`. By default, all of these are turned off and can be explicitly turned on via input parameters.

The following greedy scheme which is specific to set-cover formulations (and is implemented in `Cbc` as `GreedyCover`) has reasonably good properties.

(1) Start with $S := \emptyset$ ("selected"), and $U := \mathcal{A}$ ("unexamined").
(2) While the request R is not yet covered by the approvers in S:
  (a) Let $A_{\min}$ be the approver A in U that minimizes the ratio of $w(A)$ to the number of points of R that will be covered by A but were not covered by approvers in the current S. That is, $A_{\min} := \operatorname{argmin}\{w(A)/|A \cap (R \setminus \cup_{B \in S} B)|\}$.
  (b) Let $S \leftarrow S \cup \{A_{\min}\}$, and let $U \leftarrow U \setminus \{A_{\min}\}$.
(3) Return S as the set of approvers assigned to the request R.

Note, this heuristic may cause some unnecessary approvals (e.g., the approver chosen last may be costly and necessary, and assigning this approver may obviate the need for a previously chosen one). An *easy* cleanup at the end, is to re-instantiate and solve the set-covering ILP, but now only having columns for the approvers in the heuristically selected S — this will be a very small instance which will be easy to solve. However, we are using this heuristic within the ILP search carried out by `Cbc`, so this is not a substantial issue.

If the heuristic appears to be needed, its performance *might* be improved by doing a local search starting from its solution S. That is, repeatedly look for an approver $A \in S$ and an approver $A' \in \mathcal{A} \setminus S$ so that $w(A) > w(A')$ (improvement) and R is covered by $S \setminus \{A\} \cup \{A'\}$ (feasible), and then replace S by $S \setminus \{A\} \cup \{A'\}$. If such improving swaps exists, this may be worthwhile (and it can be followed by the cleanup mentioned above). This type of local search is realized by the `Cbc` module `LocalSearch`.

5. Shrinking bloated sets of approver rules

Prior to the development and use of the new tool, approver selection had been handled by a greedy algorithm in the old tool that it replaced. The greedy algorithm of the old tool encouraged extreme growth in the set of approver rules. With the old tool, the way to increase the priority of an approver who can cover a request, without changing his/her authority, is to give the approver a large set of very specific rules that are dominated by his/her more general rules. With our algorithm, we simply decrease the weight of the approver. An extreme case of this occurred in one application having 64 approvers. In that data set there were a total of 21,188 rules, but all of these rules were dominated by a subset of only 382 rules. Besides being hard to maintain, large sets of approver rules like this one cause the bottleneck in any algorithm to be the checking of request slices against approver rules. By using a minimal set of rules for each application, which we can compute in a single preprocessing phase, we make our production runs extremely fast.

We have provided some functionality to take care of this. We can obtain the minimal rules for each approver by treating all of their rules as slices of a request, and running this request against the very same set of rules treated now as approver rules. Using weights of 1, we use our basic algorithm to find the minimal set of rules for this approver. Repeating this for each approver, and aggregating the sets of minimal approver rules, we get a minimal set of approver rules for the application.

It is certainly possible to do this more efficiently from a running-time point of view, but this preprocessing does not have to be done during production runs, so it is preferable to re-use the existing code of our basic algorithm.

6. Weight setting

It may be useful to have a good mechanism for setting/updating weights for approvers. One possible approach to this is having target utilizations, on a per approver basis, to be achieved in the long run. Although we have not implemented this, we have worked out a reasonable scheme to achieve this.

For each approver A, we may have a target utilization $\tau(A) > 0$. This is the fraction of all requests that we would like to be assigned to A. Then, computed over a sequence of assignments (for example, the moving average computed over the prior few months), we can compute the actual utilization $\alpha(A)$. Then each approver A will have a *deviation*

$$\delta(A) := \alpha(A)/\tau(A) \ .$$

Whether we employ static or adaptive weights, we do need a way to set the weights initially. A reasonable choice is $w(A) := \delta(A)$. It is not so important how we initially set them, since we can simulate request arrivals and employ the updating process below, to get a good stable set of weights.

We need an easy mechanism to adaptively update the weights so that the long-run target utilizations can be achieved, and so that we can shift to new targets as desired. By simply periodically re-setting the weights according to the current deviations, we penalize approvers who are over utilized, and make under-utilized approvers look more attractive for assignment. Faster adjustment to the targets could be achieved by choosing an exponent $p > 1$, and letting $w(A) := (\delta(A))^p$. Our weight-update strategy keeps all weights positive, which seems to be a reasonable idea since some heuristics may rely on such an assumption.

7. Software Integration and Business Results

The approver selection engine in the new system is a component that is called as a service by the workflow component. The workflow component receives requests for authorization from a dynamic front-end component. Once requests have been approved, the workflow component passes the request to a provisioning component.

The provisioning component is responsible for making the changes that are needed to update a user's application access privileges. The provisioning component has interfaces with Lotus Notes (enterprise business email and collaboration middleware), IBM DB2 (a relational database management system), WebSphere Message Queuing (messaging and queueing middleware), IBM RACF (Resource Access Control Facility), and an IBM LDAP-based system for defining and maintaining groups of user IDs. If these interfaces are not sufficient, the provisioning component can send email to someone who can update the user's privileges in the application.

As of June 14, 2007, the new IBM tool for entitlement and access management was used to manage access for more than 70 applications. We expect that number to grow to 100 by the end of 2007. The new tool has been very successful and is now the strategic application for access management within IBM.

References

1. COIN-OR, Common Infrastructure for Operations Research, http://www.coin-or.org .
2. Common Public License, Common Public License, http://www.opensource.org/licenses/cpl1.0.php .
3. Gérard Cornuéjols, *Combinatorial Optimization: Packing and Covering*, CBMS-NSF Regional Conference Series in Applied Mathematics, vol. 74, Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 2001.
4. George L. Nemhauser and Laurence A. Wolsey, *Integer and Combinatorial Optimization*, Wiley-Interscience Series in Discrete Mathematics and Optimization, John Wiley & Sons Inc., New York, 1988.

IBM Corporation

E-mail address: `{wallyd,jpfasano,jonlee}@us.ibm.com`