A Binary Integer Programming Model for Optimal Object Distribution

M. C. Bastarrica, A. A. Shvartsman, and S. A. Demurjian, Sr.
Computer Science & Engineering Dept.
University of Connecticut, Storrs, Connecticut 06269-3155

October 9, 1998

Abstract

Whether deploying new distributed software, redeploying existing distributed application components or distributing a standalone legacy application, a key decision that needs to be made is the location of each software component in the target distributed environment. This decision depends on the usage and interaction patterns of the distributable components. For a given pattern of interactions among the components, choosing the component locations involves a trade-off between the resulting local and remote communication of the deployed components. To assist in the process of deploying distributed software, we defined an abstract model of distributed applications and we have developed a binary integer programming (BIP) model that takes into account an application’s communication patterns and the target network characteristics. The goal is to minimize the overall remote communication bandwidth for the application, while achieving a meaningful distribution of components in the target network setting. The BIP model yields solutions (when they exist) that are optimal for systems that can be described in terms of our application model. Object-oriented software is especially well-suited for this methodology since it is inherently structured in terms of distributable components (e.g., classes, objects) and the major communication patterns can be determined by identifying the foreign method invocations. We include examples of optimal object deployment solutions using our methodology and sample object-oriented applications.

Keywords: Distributed objects, communication bandwidth, component distribution, integer programming.

Contact author: M. C. Bastarrica. Computer Science & Engineering Dept, 191 Auditorium Road, U-155, University of Connecticut, Storrs, Connecticut 06269-3155, USA. Email cecilia@cse.uconn.edu. Fax 860-486-4817.
1 Introduction

Performance of distributed systems depends on many factors that range from the efficiency and scalability of the abstract algorithms used in the systems to the capabilities of the underlying communication networks and the processing nodes in the networks. The performance of a particular system that is designed as a collection of interacting components also depends on how the components are deployed in a specific network. By varying the mapping of the system components to the nodes of the network, we can also vary the nature of interactions among the components. Components that are mapped to the same node interact by communicating locally, while components mapped to different nodes interact by communicating remotely using the underlying network. Co-locating some components on a network node has the advantage of faster and cheaper interaction among the components, which is not affected by the network characteristics. Distributing components enables the application to take advantage of available computation and storage resources in the network, but requires remote communication, which is typically slower and more expensive, and which critically depends on the underlying network characteristics. Thus, for a given pattern of interactions among the components, choosing the component locations involves a trade-off between the resulting local and remote communication of the components. Thus, minimizing the remote communication in a distributed system while maintaining the necessary distribution of its components is an attractive goal.

Development of distributed software is made easier by middleware such as CORBA [12] that allows the designers to concentrate on system functionality and to defer the decisions about the distribution of system components until the actual deployment of the system. When a system consists of a few components, ad hoc techniques may be reasonable in determining how to distribute the components. Here experimentation, simulation, or random choices may be sufficient to guide the deployment decision. For example, Colored Petri Nets [4, 13] and static coupling evaluation [5] have been used to simulate distributed software execution. However, such techniques, while checking for constraint violations and for the absence of bottlenecks, do not provide means to determine an initial deployment of components to be tested. For more complicated systems with many distributable components it is important to have tools that help us establish a viable mapping of abstract software components to network nodes that satisfy the required communication needs without exceeding the available network resources.

In this paper, we present a binary integer programming (BIP) model for generating an optimal deployment of a software application over a given network. Our criterion of optimality is the minimization of the overall remote communication bandwidth. Our model includes a number of parameters that have to be either measured, estimated or deduced, namely: the storage requirements for each software component, the storage available in each node of the network, the communication frequency between components, and the communication capacity across the connections between components.

For the model to be relevant, the parameters used in the analysis must be carefully assembled. There are several sources of information that yield the necessary knowledge. The users of the system can provide information on access patterns to the presentation and service components of the system, and on whether certain system resources need to be bound to specific network nodes. The target network can be analyzed to determine its communication capacity and the
capabilities of the nodes on the network. The designers of the system can provide information on internal interaction patterns within the system. The system can be instrumented to gather additional information about the component interactions in simulated settings. Specific quantitative information about the system can be obtained through the analysis of the source code and formal specifications. In the case of sophisticated object-oriented (OO) applications, the system documentation may include, in addition to the specifications of service interfaces and type inheritance, the specifications of instantiation of objects and implementations, and of inheritance of implementations. Such specifications may be given informally or stated formally, as in the OSF's proposed I4DL [7] set of specifications.

Once the necessary parameters are marshaled for a given system, we instantiate the BIP model that can be solved using available integer programming software. The solution, when it exists, optimizes remote communication (throughput), while satisfying the necessary component distribution requirements. Our application model is similar to that in [14], which utilizes Petri nets. Their effort uses similar parameters, but it does not give specific decision criteria and instead suggests the use of AI techniques or attempt to reuse an existing similar deployment. In contrast, our approach provides a specific algorithm for optimizing the deployment. We also estimate an upper-bound time complexity for finding a solution using our model. We use the Z specification language [16] to formally define the model of application systems and their deployment.

Three common situations can benefit from our approach: designing a new distributed software, distributing a standalone legacy application, and redeploying a distributed legacy application. In the paper we illustrate our technique by giving example solutions for distribution of two object-oriented legacy applications that employ a public domain class library [1].

In Section 2 we define our application model. In Section 3 we present the BIP model for optimal deployment providing a formal definition of its parameters in Z. In Section 4 we show an example of how our model is used.

2 Distributed Application Model

We now describe the model for distributed applications for which we formulate our BIP solution for component deployment. The model abstracts the following attributes.

Distribution unit granularity: A unit is a component that can be mapped to a network node. In general, a unit can be a class, a group of objects, an inheritance hierarchy, a database, a GUI, etc. For simplicity, we deal with classes, allowing the class extent to be partitioned over several nodes. A class is auxiliary whenever its objects are not deployed independently. Typical auxiliary classes are those whose instances are always a part of a container structure.

Network: The network is fixed and is provided as input. The network is a complete graph, whose vertices are nodes (computers) and edges are connectors.

Nodes: All nodes (computers) are identical, except for the storage capacity. Local computation is assumed to be negligible compared to remote communication. In particular, method execution time is negligible.
Connectors: A connector exists between each pair of nodes. Each connector has a fixed communication bandwidth or capacity. Different connectors may have different capacity. There exists a (possibly abstract) connector between every pair of nodes. We also assume that the connections with each other node are independent, which allows the communication between two nodes to not affect other nodes in the network.

Messages: We consider only the messages that are sent between distribution units. When a message is delivered to a unit that responds to it, we consider the response to be an overhead on the original message as in [6]. Thus we coerce many possible types of interactions among units (cf. [10, 9, 5, 17]) to our simple message concept. For each type of message that can be sent from one unit to another, we know the (average) size of the message and the frequency with which the message is sent. (Estimating these frequencies can be the most difficult part of the distributed system modeling.) We disregard any communication due to inheritance because we assume that inherited methods are stored locally.

Local communication: Communication between units on the same node is considered to be equivalent to local computation, i.e., it is negligible compared to remote communication.

These model characteristics are formally stated when we define the BIP model.

In instantiating the model for an application, many decisions can be made by analyzing the I4DL-like [7] framework of the application. I4DL defines four aspects of distributed systems, namely, Interface, Inheritance, Implementation, and Instantiation, which are discussed below.

Component interfaces are defined using the usual exports and uses. For OO components, the exports can be built directly form from the union of the public interface of all classes contained in the component. The uses has to be calculated by parsing the code and finding all method invocations that are not part of the same component.

When dealing with inheritance in a distributed OO system, it may be the case that the parent class is located on a node different from the node of its children classes. In this case, there are three possibilities for locating the inherited methods at runtime: in the child's node, in the parent node, or to be delivered to the child on demand.

Instantiation is used to determine whether attributes of one class defined in another class are instantiated where the container class is or in some place else. This decision affects the amount of storage each distributable class requires. Implementation specification determines whether certain components are bound to certain nodes for portability or known performance reasons.

3 A BIP Model for Application Deployment

The deployment of an application over the given network must satisfy the following requirements:

- Each application unit is mapped to one and only one node
- For each node, the total storage required by the deployed units does not exceed the node's storage capacity
- Units communicate with remote units via connectors
- For each connector, the total communication bandwidth does not exceed the bandwidth of the connector
Among the possible deployments that satisfy all of these requirements we are interested in the one that minimizes the remote communication bandwidth.

Once a model is instantiated and quantified, we can state the problem of satisfying the deployment requirements as a binary integer programming problem. In the next sections we introduce the main problem parameters and then define our BIP model, i.e., the constraints and the objective function, for the application deployment problem. We also discuss the time complexity for solving the BIP model using the branch and bound method.

### 3.1 Parameters

The parameters of the model quantify distributable units, node capabilities, and communication requirements. The parameters of our model are: the storage requirements for the application's components, the storage available at each node of the network, the average communication frequency between the application components, and the capacities of the connectors between the various nodes. The parameters are defined based on the application model formal specification included in the Appendix.

We define formally a distributable *Unit* as follows:

```
Unit
  u_id : string
  state : \mathbb{N}
  exports_list : \mathbb{P} Message
  uses_list : \mathbb{P} Message
  instances : \mathbb{N}

\forall m \in exports_list \bullet m.target = self
```

The *state* is the total amount of storage each instance of the *Unit*—or class—requires. The *state* is estimated as an average value every time there are lists or any variable size structure included in the unit. The *exports_list* is the list of methods other units can invoke on this unit, those messages targeted to “this” unit, as stated in the condition. The *uses_list* represents the methods the unit calls on other units. Notice that the *exports_list* can be derived from the public, protected and inherited methods defined for the class, but the *uses_list* should be deduced by analyzing all methods implementations. For our example in Section 4, we use a tool that generates the list of couplings between the classes by parsing the C++ code [15].

We calculate the required storage for a unit \( u \) as the product of the storage (*state*) required for each instance of a unit and the number of instances:

\[
US(u) = u.state \times u.instances
\]  

(1)

Similarly, we define a *Node* as:

```
Node
  n_id : string
  storage : \mathbb{N}
```

The available storage for each node \( n \) in the network is given as the simple parameter:

\[
NS(n) = n.storage
\]

(2)
We can formally define a Message as follows:

\[
\begin{array}{l}
\text{Message} \\
m\_id : \text{string} \\
target : \text{Unit} \\
size : \mathbb{N}_1 \\
frequency : \mathbb{N} \\
\text{self} \in \text{target\_exports\_list}
\end{array}
\]

A message has a defined target, that is the unit to which it is sent. The size of the message is the amount of bytes one of these messages has; this value depends mainly on the number and type of the parameters sent. In Section 4, we assume that methods without parameters have size 1. Finally, the frequency is the number of times the message is sent per time unit (from each instance of the unit to which it is related).

We define the total communication from unit \(a\) to unit \(b\) as the summation of the communication due to each message sent from \(a\) to \(b\):

\[
F(a, b) = \sum_{(m \in a\_uses\_list) \land (m\_target = b) \land (m \in b\_exports\_list)} m\_frequency \times m\_size \times a\_instances
\]  

\(F\) describes the total number of bits transmitted per unit of time. Even though this communication may be bursty, in our model this value represents a constant bandwidth.

We can define the network connectors as follows:

\[
\begin{array}{l}
\text{Connector} \\
c\_id : \text{string} \\
end1, end2 : \text{Node} \\
capacity : \mathbb{N} \\
end1 = end2 \Leftrightarrow capacity = \infty
\end{array}
\]

Nodes \(end1\) and \(end2\) are the vertices the connector. Its capacity is its maximum throughput. We state that local communication is unconstrained by assigning infinite capacity to the (abstract) connectors between a node and itself.

The capacity (bandwidth) of the connector from node \(i\) to node \(j\) is:

\[
C(i, j) = c\_capacity \mid c\_end1 = i \land c\_end2 = j
\]

Based on the former definitions we can define an Application and a Network as follows:

\[
\begin{array}{l}
\text{Application} \\
Units : \mathbb{N} \text{ Unit} \\
\forall u \in Units \bullet \\
\forall m \in u\_uses\_list \bullet \exists u' \in Units \mid m\_target = u' \land m \in u'\_exports\_list
\end{array}
\]
An application is a set of Units such that all messages defined in the uses_list of a unit of the application must be part of the exports_list of another application unit that is also the target of the message.

We define the network as a set of Nodes and a bijection between Node × Node and Connectors, implying that all connectors have their ends in nodes of the network, and all pair of nodes in the network is connected.

\[
\begin{align*}
\text{Network} \\
\text{Nodes} & : \mathbb{P} \text{ Node} \\
\text{Net} & : \text{Node} \times \text{Node} \rightarrow \text{Connector} \\
\forall n, n' : \text{Node}; & \ c : \text{Connector} \mid (n, n') \mapsto c \in \text{Net} \bullet n \in \text{Nodes} \land n' \in \text{Nodes} \land \\
& c.\text{end}1 = n \land c.\text{end}2 = n'
\end{align*}
\]

The condition states that all tuples in the \text{Net} function are such that the two nodes are the ends for the connector that is their image in the function.

Any possible deployment of the application over the network could be defined as:

\[
\begin{align*}
\text{Deployment} \\
N & : \text{Network} \\
A & : \text{Application} \\
\text{Deploy} & : \text{Unit} \rightarrow \text{Node} \\
\text{Usage} & : \text{Connector} \rightarrow \mathbb{N} \\
\text{dom}(\text{Deploy}) & = A.\text{Units} \quad [1] \\
\text{ran}(\text{Deploy}) & \subseteq N.\text{Nodes} \quad [2] \\
\forall n \in N.\text{Nodes}; & \ u \in \text{Deploy} \ni n \bullet \sum_u US(u) \leq NS(n) \quad [3] \\
\text{dom}(\text{Usage}) & \subseteq \text{ran}(N.\text{Net}) \quad [4] \\
\forall c & \ni f \in \text{Usage}; \ u \in \text{Deploy} \ni c.\text{end}1; \ u' \in \text{Deploy} \ni c.\text{end}2 \bullet \\
& f = \sum_u \sum_{u'} F(u, u') \leq C(c.\text{end}1, c.\text{end}2) \quad [5]
\end{align*}
\]

A Deployment is a structure formed by a network \(N\), an application \(A\), a total function mapping each unit of the application to the nodes in the network (\(\text{Deploy}\)) and the resulting Usage of the network connectors. All units of the application should be deployed (condition [1]) only over nodes of the network (condition [2]). Condition [3] expresses that a node cannot store more units than its available storage enables it. Condition [4] says that communication takes place through connectors of the network. And finally, condition [5] says that the total communication that goes through each connector is lower than the connector’s capacity. Fixing some units’ location is to define a-priori the value of some tuples of the Deploy function.

3.2 Constraints

We define \(U \times N\) decision variables, where \(U\) is the number of distributable units in the application and \(N\) is the number of nodes in the network.

\[
z_{i,j} = \begin{cases} 
1 & \text{if unit } i \text{ is assigned to node } j \\
0 & \text{otherwise}
\end{cases}
\]
The **completeness** constraint states that an application unit is assigned to one and only one network node, and that all units in the application are assigned:

$$\sum_{j=1}^{N} x_{i,j} = 1$$  \hfill (5)

We can then state the **storage** constraint:

$$\sum_{i=1}^{U} x_{i,j} \times US(i) \leq NS(j)$$  \hfill (6)

The **communication** constraint states that for any connector, the bandwidth consumed is the total communication between units located in the incident nodes, and that this bandwidth does not exceed the connector:

$$\sum_{a=1}^{U} \sum_{b=1}^{U} F(a, b) \times x_{a,i} \times x_{b,j} \leq C(i,j)$$

The above inequality is not linear, and a non-linear system is unfeasible with binary decision variables. So we introduce **auxiliary decision variables** that yield a linear system.

$$y_{a,i,b,j} = \begin{cases} 
1 & \text{if unit } a \text{ is assigned to node } i \text{ and unit } b \text{ is assigned to node } j \\
0 & \text{otherwise} 
\end{cases}$$

That is, $$x_{a,i} \times x_{b,j} = y_{a,i,b,j}$$. This equation can be restated in terms of the following linear inequalities (recalling that they are all binary numbers):

$$y_{a,i,b,j} \leq x_{a,i}$$  \hfill (7)

$$y_{a,i,b,j} \leq x_{b,j}$$  \hfill (8)

$$1 + y_{a,i,b,j} \geq x_{a,i} + x_{b,j}$$  \hfill (9)

The communication constraint now becomes:

$$\sum_{a=1}^{U} \sum_{b=1}^{U} F(a, b) \times y_{a,i,b,j} \leq C(i,j)$$  \hfill (10)

### 3.3 Objective Function

Our goal is to minimize the total amount of communication bandwidth:

$$\text{Minimize} \quad Z = \sum_{i=1}^{N} \sum_{j=1}^{N} \sum_{a=1}^{U} \sum_{b=1}^{U} y_{a,i,b,j} \times F(a, b)$$  \hfill (11)

The BIP model is completely defined in terms of the parameters (1–4), four structural equations (5–10), the two storage and communication constraints, and the objective function (11).
3.4 On the Branch-and-Bound Complexity

Branch-and-bound is a standard procedure for solving BIP problems. In each step, one variable is considered (the decision variable) with each possible value, either 1 or 0, dividing the problem into two smaller subproblems. Applying the branch-and-bound [11] just to the \( x_{i,j} \) decision variables solves the entire problem since the values of \( y_{a,i,b,j} \) can be completely deduced from the values of the \( x_{i,j} \) (equations 7 through 9). We have to apply the branch-and-bound procedure at most \( N \times U \) times, once for each \( x_{i,j} \). This is not a problem since most mixed integer programming (MIP) software packages allow a user to give the order in which the variables are considered [8, 18].

Our problem’s structure is given by the fact that each unit has to be deployed to one and only one node in the network: \( x_{1,j} = 1 \Leftrightarrow \forall k \neq j \cdot x_{1,k} = 0 \). This condition reduces the search space from \( 2^{NU} \) to \( N^U \). The algorithm does not need to search more than \( N^U \) paths.

If \( x_{1,1} = 1 \), all the remaining \( x_{1,j} = 0 \), for \( j = 2 \ldots N \); the other branches containing \( x_{1,j} = 1 \) would be immediately fathomed because they violate the unique assignment constraint. The same occurs for all other \( x_{1,j} \)’s. Thus, after \( U \) iterations, we have a decision tree with \( N \) leaves, one for each of the \( N \) nodes where component 1 could be located.

Applying the same criterion for the remaining \( U - 1 \) components, we can build the search tree with \( N^U \) leaves. However, not all of the paths are generally evaluated because many of them are fathomed; so \( N^U \) is an upper bound to our problem. Most vendors of commercial integer programming software claim their products can run in linear time [8, 18] in the number of integer variables, and some of them also claim they can achieve superlinear speed-up using parallel computers [3].

4 Illustrating the Technique

In this section we give an example of our technique by showing how it is used to distribute a standalone legacy OO application. The parameters of the application (see Section 3.1) are estimated by examining the source code and measuring its runtime behavior, yielding \( US \) and \( F \) values. Using an example network, we determine its values for storage and communication capacities, yielding \( NS \) and \( C \). We compute the optimal object deployment using the GAMS optimization package [2]. We experiment with two scenarios that fix some units’ locations.

4.1 The Example Problem

Our test application consists of an AI class library [1] and two distributed programs. The library, as presented in Figure 1, includes searches: depth-first, breadth-first, and uniform-cost for trees and graphs, bidirectional depth-/breadth-first tree and graphs, and AND/OR depth-/breadth-first tree and graphs. There are three main inheritance hierarchies: SEARCH for uni-directional searches, BISEARCH for bi-directional searches, and VOBJECT for supporting structures. There are also classes for managing lists, i.e., LIST, LIST_NODE, and LIST_ITERATOR.

We use two programs:

1. PATH: Using a database of cities, find the shortest route from city \( x \) to city \( y \).
2. PUZZLE: Solve an $n$-square puzzle, i.e., a $n$ by $n$ frame of $n^2 - 1$ numbered and movable tiles and one empty cell.

The two programs are unrelated but they use the same AI library. We extend the library with four new classes: CITY, PATH, PNODE and PUZZLE, which are shown in Figure 1 in shaded boxes. (Note that hierarchies BISEARCH and AOSEARCH do not take part in the computation.)

Our main distributable units are: PUZZLE, PATH, PNODE and CITY. The class SLIST is instantiated as classes SLIST1 and SLIST2, because they are auxiliary. SLIST1 and SLIST2 are going to be located in the same nodes as PUZZLE and PATH, respectively. This is expressed as $X_{PUZZLE},i = X_{SLIST1},i$ and $X_{PATH},i = X_{SLIST2},i$. If classes in the NODE hierarchy are also considered auxiliary, there is no remote communication because they are replicated with PUZZLE and with PATH. However, if they are defined as independent, they could either (a) be placed with PUZZLE or with PATH (whichever minimizes remote communication), or (b) be placed in another location if PUZZLE and PATH have fixed locations but the storage is not sufficient. We do not consider NODEs as auxiliary classes in Section 4.2 because, even though we think it would be more natural, there would not be any remote communication and the problem would be trivial.

Table 1 shows the runtime storage needed by each unit.

We estimate that PUZZLE will have up to 300 LIST_NODE (PNODE) instances in the SLISTs and PATH will have 100 (CITY). These values depend on the size of the puzzle and the number of cities and the initial settings of the puzzle and the pair of cities chosen as the end nodes of the path. Communication between the units is shown in Table 2.

We used an automated tool [15], to determine the list of messages (methods) sent between PUZZLE, SLIST1 and PNODE. We inserted monitoring code in the methods that were listed to count the number of times each method is called. Multiplying these numbers by the parameter-list length (storage in bytes), we determined the total number of bytes transmitted. Because
<table>
<thead>
<tr>
<th>Unit</th>
<th>State</th>
<th>Instances</th>
<th>US(Unit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUZZLE</td>
<td>10</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>SLIST1</td>
<td>12</td>
<td>2</td>
<td>24</td>
</tr>
<tr>
<td>PNODE</td>
<td>33</td>
<td>300</td>
<td>9600</td>
</tr>
<tr>
<td>PATH</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SLIST2</td>
<td>12</td>
<td>2</td>
<td>24</td>
</tr>
<tr>
<td>CITY</td>
<td>16</td>
<td>100</td>
<td>1600</td>
</tr>
</tbody>
</table>

Table 1: Units’ Storage ($US(a)$).

<table>
<thead>
<tr>
<th>Communication</th>
<th>PNODE</th>
<th>Communication</th>
<th>CITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUZZLE</td>
<td>3500</td>
<td>PATH</td>
<td>700</td>
</tr>
<tr>
<td>SLIST1</td>
<td>4500</td>
<td>SLIST2</td>
<td>500</td>
</tr>
</tbody>
</table>

Table 2: Communication between Application Units ($F(a,b)$).

PUZZLE and SLIST1 are stored together, their communication is local. We follow the same procedure to get the total communication between PATH, SLIST2 and CITY. These calculated values correspond to the total number of bytes transmitted during the execution. When the programs are repeatedly executed, these quantities are proportional to the remote communication bandwidth.

The example network has four nodes: USER1, USER2, BASE1 and BASE2. Figure 2 gives the network characteristics.

![Network Characterization](image)

Figure 2: Network characterization.

### 4.2 The Solution

We fixed the PUZZLE component in node USER2 ($X_{PUZZLE, USER2} = 1$). We solved two scenarios using the GAMS package:

1. $X_{PATH, USER2} = 1$. Table 3 shows the optimal distribution solution for this case.
2. PATH can be located in any node. Table 4 states the two optimal solutions.
<table>
<thead>
<tr>
<th></th>
<th>USER1</th>
<th>USER2</th>
<th>BASE1</th>
<th>BASE2</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUZZLE</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>SLIST1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>PNODE</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>PATH</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>SLIST2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>CITY</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 3: Optimal solution when $X_{PATH,USER2} = 1$. $Z = 1200 + 8000 = 9200$

<table>
<thead>
<tr>
<th></th>
<th>USER1</th>
<th>USER2</th>
<th>BASE1</th>
<th>BASE2</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUZZLE</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>SLIST1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>PNODE</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>PATH</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>SLIST2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>CITY</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 4: Optimal solutions when there is no constraint on PATH. $Z = 8000$ (for both solutions.)

As we have shown in Section 3.4, even when evaluating all possible distributions, there are no more than $4^6 = 4096$ different cases, and this number is not too big for automatic solving. Most notably, the GAMS package solved these scenarios in less than 5 seconds elapsed time and less than half second actual processing time, without specifying any variable evaluation order.

This BIP approach could not have been considered even a few years ago. Modern MIP software currently can handle up to 1000 binary variables. Even a small example as the one we showed here, with six units and four nodes, generates more than 500 variables and 2000 equations. With the availability of faster computers, new branch-and-cut and preprocessing techniques, and by continuing to exploit our problem’s structure, we anticipate that automatic BIP solutions will become worthwhile for distributed systems of considerable size.

References


