Allen Rabinovich
Sebastian Ortiz

Building a Desk Worker Allocation System For Random Hall Dormitory

October 28th, 2002
6.034 Intensive
TA: Jake Beal

Abstract

The issue of resource allocation is a popular problem that arises in many areas of human life. The difficulty of the problem can range from very easy to extremely challenging, and depends primarily on the number and complexity of requirements and preferences of variables and values involved. To solve the problem as applied to the schedule allocation of desk workers in Random Hall dormitory, we have developed an application in Java. The application takes a set of workers, each with individual preferences and requirements, and attempts to produce a set of assignments that would maximally satisfy first the requirements, and then the preferences of the assignees. The sample executions demonstrate that the system is capable of creating a working set of assignments with a fair measure of preference satisfaction.

Overview

Resource allocation is a realistic problem, frequently encountered in systems that require scheduling or unique assignments. In this particular application, a group of n (between 10 and 20) workers is to be scheduled for 133 hours of desk shifts (7 days of 19 hours each), each shift can last for multiple hours. Each worker has a set of requirements and preferences, defined by their existing schedule, personal preference, seniority and reliability. Our application takes all of these factors into account, and attempts to produce a set of assignments that maximally satisfy the requirements, while also maximizing the number of satisfied preferences. The utilized method involved performing "intelligent" constraint propagation both among students and time slots, and sequentially eliminating candidates, until a set of assignments with satisfied requirements and maximally satisfied preferences is produced.

Design

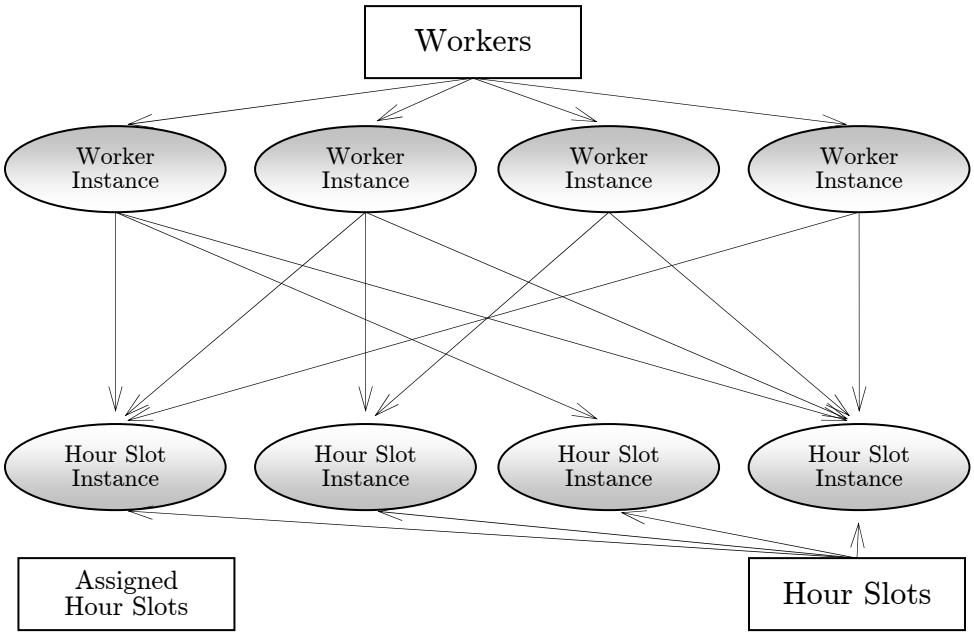
The first logical step in developing a solution to the problem was to design a convenient representation for the two main components of the system and the interaction between them. We have decided to regard workers and hourslots as individual objects, each with a set of specific properties. For the purposes of constraint propagation, we made a choice of considering HourSlots to be variables, and Workers to be values. The reason for this choice was the fact that by the end of the program's run, each HourSlot must contain a unique Worker value; however, each Worker can be assigned to multiple HourSlots.

The Worker class is essentially a list of properties with an identifier property (the name of the student). The properties are: availability hours (recorded as a two-dimensional array with days as columns, hours as rows, and binary values representing occupation during a certain day-hour), minimum hours, maximum hours, rank, worker's fill-in property, hours available (a count of hours that chooses the smaller of maximum hours and total availability hours), and assigned hours. An Hourslot class, in turn, is identified by properties Day and Hour, and contains a list of all possibleWorkers for the particular hour. Both classes have a set of built-in methods that allow us to easily modify and retrieve information from each instance.

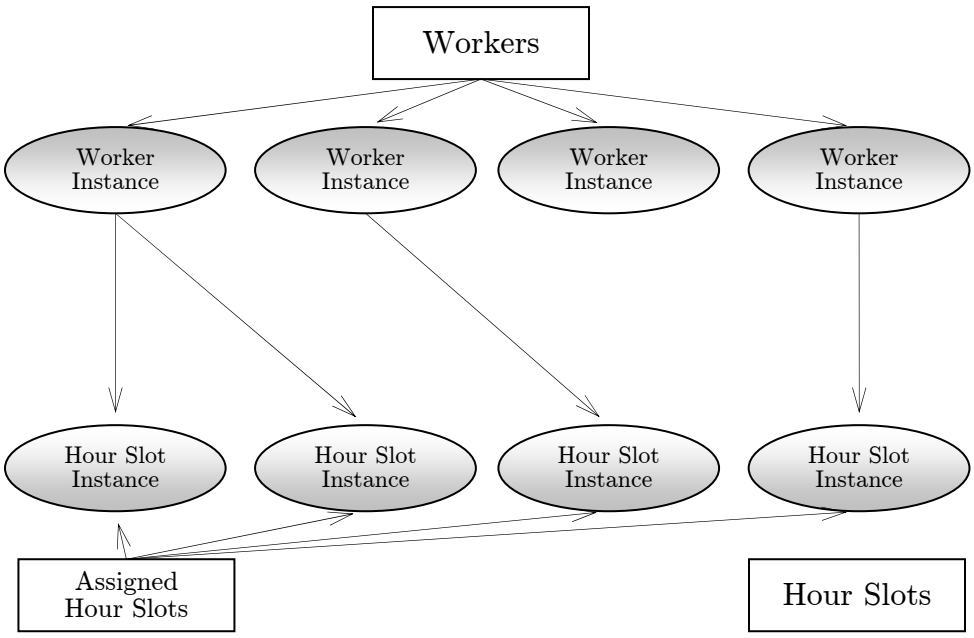
The input to the system is a set of definitions of instances of Worker class; the output is an assignedSlots vector of all assigned Hourslot instances, each containing 0 or 1 possibleWorkers as an assignment. An additional workerList class is used for bookkeeping purposes.

Here is a graphical representation of the system structure prior to and post execution:

Prior to execution:



After execution:



From the very beginning, we considered the maximal schedule filling to be a priority over the worker preference satisfaction. However, the requirements for each worker were a top priority as well.

Implementation

The constraint propagation in our design is conducted on multiple levels -- after every "run around the circle", the uniquely determined assignments are accounted for, and the propagation is continued, until no possibility for making unique assignments is left. We exercise constraint propagation both among workers and hour slots. Here's an algorithmic step-by-step representation of the run:

Pre-processing: Parse the string assignment lists of workers into their availability hours matrices; at parse-time, immediately introduce the basic constraints (i.e., cannot work Saturday & Sunday nights if nightwatch, unless otherwise specified). Add the workers to jgurch

1. If there exist hourslots with unique worker assignments, move the hourslot to the list of assigned slots list; update the respective worker's properties to reflect the assignment. Repeat until no unique worker assignments are left.
2. Choose the next hour slot to consider based on a set of properties of workers who can possibly be assigned to it. If no such slots are left, consider "fill-in"workers for these slots, assign them if possible, and then exit the program returning the current assignedSlots vector. Else, proceed to next step.
3. For the chosen hour slot, choose a worker based on a set of properties of each candidate worker.
4. Proceed to step 1.

The decision-making details in the above algorithm are of particular importance and deserve elaboration.

In step 1, the "unique worker assignments" could either be naturally existing in the initial set of hour-worker assignment, i.e. in a case where an hour only has one possible worker, or could be produced by a preceding run of the algorithm. Since hour-filling was a top priority, we first assigned these cases, to make sure that hours with the great restrictions upon them are filled.

In step 2, after elimination of all hour/single worker pairs, we proceed to determine the most appropriate hour slot to pick the worker for. The decision algorithm uses the number of possible workers for each hour slot as the heuristic, using the sum of worker's possible shift lengths (shift length is the longest shift each worker can work that would include the particular slot) as a secondary heuristic in case of a tie between the numbers of workers. We fill the slots that have the minimum number of workers, but resolve ties by picking the slots with greatest sums of shift lengths; the motivation is as follows: it is important to assign slots with fewest possible chances to be filled; however, it is also essential that the slots that have a potential to introduce the maximum number of assignments in the neighboring slots are prioritized as well.

If we cannot find any slots with the number of possible workers greater than 1, we leave the program and output the latest value of the assignment vector. If a slot is produced, we continue to choosing a worker for it.

In step 3, we look at the set of workers who can work the particular slot and decide in favor of one among them. The decision is made by considering if a worker is allowed to work that day (the assumption is made, as stated by the problem, that a worker is only allowed to work one shift during the day -- we check for the presence of another shift and possibility of "connecting" to that shift -- i.e. extending it -- and if the shift is presence, and connection is not possible, we remove the worker), and by comparing the difference of each worker's available hours and minimum hours. The produces a heuristic that can be described as "number of hours the worker will have left after satisfying his minimum hours". This heuristic is simply the number of available hours as soon as the worker satisfies his minimum hours (i.e., no negative numbers) Since satisfying minimum hour constraint is a requirement, it may at first appear strange that we are using a difference of a requirement and a preference; however, we found that simply considering minimum hours produces inferior results. It is more important to consider workers who have the least *ability* to satisfy their minimum hours, rather than workers with the greatest minimum hours (for example, a worker with 10 minimum hours but 20 available hours is in better position than a worker with 2 minimum hours, but only 2 available hours.) The

secondary heuristic we use in this step is the product of the worker rank with the maximal shift length of the workers who tied; i.e., how long of a shift a worker can have around the particular hour slot. That allows us to assign the worker who has the greatest potential of filling the slots around the current slot, while balancing it with his or her seniority; if the worker's seniority greatly exceeds the difference between two workers' maximal shift lengths, that worker will prevail; this produces longer continuous shifts, while allowing us to satisfy workers' preferences.

Conclusion

When we were choosing the particular elements for the design of our implementation, we considered the suggestions made Amanda Wozniak; the provided samples gave us a good idea of the importance of certain factors and the priority of some aspects of the problem over the other. On the basis of these samples, we constructed a system that takes in a plausible number of constraints (a number that does not make the system unnecessarily complex), and produces a reasonable output, where the workers are assigned in a more "human" way (i.e., continuous shifts, priority of filling the time slots over the worker preferences.). There is plenty of space for improvement, however: we did not consider the worker reliability factor (the sample sets had "Unreliable on weekends" comment fields), and we did not consider the worker preference as much as we would have wanted. There's also much to be desired in terms of interface: we have not developed a GUI, and though that is not a difficult problem, it's an inconvenience. However, above the shortcomings, the system does produce a plausible result and can be used for building weekly desk schedules.

Contributions

Jake Beal and Amanda Wozniak have provided the specifications for the assignment; Allen Rabinovich and Sebastian Ortiz have implemented the application in Java.