Sledgehammer: Fast Shift Scheduling using Constraints

Michael Salib

October 28, 2002

Contents

List of Figures

1 Introduction

Sledgehammer is a program used to assign shifts to workers in order to fill a schedule. The resulting schedule is guaranteed to meet user specified constraints and represents a "best effort" attempt to meet user specified preferences. Sledgehammer is a constraint satisfaction program. It uses scheduled time slots as variables and worker names as values. Valid solutions consist of a set of worker names such that each time slot has exactly one worker assigned to it. In order to solve the associated constraint satisfaction problem, Sledgehammer performs a depth first search in the solution space using dynamic ordering to speed up the process.

This paper describes Sledgehammer's design and operation as applied to the problem of scheduling Random Hall's desk. We briefly review the internal representation for input constraints and assertions, what constraints the system makes use of, the search techniques used, implementation goals, and the methods used to measure success. Later, we examine how well Sledgehammer performs on sample data runs and some justifications for the particular design choices underlying Sledgehammer.

2 Knowledge Engineering

2.1 Data Representation

Sledgehammer relies on four input files, as described below. Note that the Rank Data input file is equivalent to the first part of the supplied sample data file while the Worker Schedule file is equivalent to the second half of the sample data file. In addition to the data conveniently supplied in the sample data file, Sledgehammer tracks whether each worker is conditional or not, whether they are unreliable (and if so, for what days are they unreliable), and if they work Nightwatch, for which days are they willing to work desk before or after their Nightwatch shift. Sledgehammer retains all of the original information supplied in the input files; there is no loss of information.

Rank Data The number of summers and terms each worker has worked desk.

Worker Schedule What free time each worker has along with information on their constraints.

Nightwatch Exceptions List of people willing to work before or after a Nightwatch shift on specific nights.

Unreliable People People who can't be relied upon on certain days and the days they are unreliable for.

2.2 Constraints

In order to guarantee correct output, Sledgehammer only considers candidate solutions that meet the following constraints.

One Shift Per Day Rule

Workers cannot work more than one shift per day. This constraint checks that all slots scheduled to a worker on any given day are contiguous.

Ten Hour Rest Rule

There must be at least ten hours separating any pair of shifts assigned to the same worker.

Maximum Hours Per Week Rule

No worker can work more than the maximum number of hours they specified in any week. Nightwatch workers have their maximum number of hours deducted by five to account for typical Nightwatch shift work.

Shift Length Rule

All shifts must have lengths between the minimum and maximum specified for that worker. Enforcing the minimum length constraint is problematic; see the Search Section for more information.

Conditional Worker Rule

Workers labeled as conditional may only be scheduled for slots that no unconditional worker can fill. Such workers have their open schedules trimmed so that they are only listed as available for slots that no unconditional worker can work.

Nightwatch Rule

Nightwatch workers cannot work desk for ten hours before their shift begins and for ten hours after their shift ends. Slots that conflict with Nightwatch requirements are cut from worker's open schedules before the search process even begins.

2.3 Search

While constraints and initial values provide the structure for this problem, they are not sufficient for rapidly building a solution. In order to quickly converge on a solution, Sledgehammer uses a search algorithm to assign one candidate worker to each empty slot. The initial design specifies depth first search with dynamic ordering based on the most constrained variable. After choosing which variable to work on, Sledgehammer selects appropriate values using a combination of least constraining value ordering and polynomial scoring.

There is one critical subtlety in the system described so far. Since Sledgehammer fills slots one at a time while never holding on to an inconsistent solution, it is impossible to schedule any worker that specifies a minimum shift length. When Sledgehammer tries to schedule such a worker, it does so for one single slot, but then immediately discards that solution since a single slot violates the worker's minimum shift length requirement. The solution we have adopted is to generalize the notion of selecting values for an empty variable. When a worker that specified a minimum shift length is a candidate for an empty slot, we instead consider all possible combinations of worker shifts that are at least the minimum length and that

contain the slot in question. In the process, we may overwrite previously selected slots, but Sledgehammer is able to back out almost all previous selections in this case. This technique could potentially be simplified by the adoption of dependency directed backtracking.

3 Code Architecture

As shown in Section B, Sledgehammer consists of seven files. The files `dataInput.py` and `deskWorker.py` contain code to parse the input data files. The file `logger.py` implements the system logger used to generate the graphs in Figures 1 and 2. The file `fakekjset.py` provides a slower version of the kjbuckets [5] set and graph manipulation library if the native version is unavailable. The file `solver.py` implements much of the actual machinery needed for solving the CSP. Finally, the file `sledgehammer.py` brings everything together by loading the input data files, solving the CSP, producing a GraphViz log for analysis and printing the schedule output. This is the file that users run to solve the CSP.

3.1 Input and Output Processing

We begin by looking at `dataInput.py` in Section B.3. This modules contains code to parse text files containing the raw preferences data into a list of "blocks" such that each block corresponds to the preferences for exactly one worker. Each block is then feed into a `workerPrefs` class which parses the preference data and performs some basic consistency checks. This module also includes code to load the supplementary data files that indicate which workers are unreliable for certain days and which workers are willing to tolerate exceptions to the Nightwatch rules. Finally, this module also includes code to read and parse worker rank data as well as to calculate individual worker ranks.

Once basic data processing is complete, the `workerPrefs` objects are used to generate a list of `deskWorker` objects. The `deskWorker` class is listed in Section B.4. These objects are designed to derive a number of constraint attributes from the corresponding `workerPrefs` object. For example, `deskWorker` objects synthesize all the data about a worker regarding their Nightwatch duty and any possible exceptions to the standard Nightwatch rules into a worker mask. A worker mask is an bit array where rows represent days of the week and columns represent the hours that desk is open. A 0 entry in the worker mask indicates that that worker is unable or unwilling to work during that day and hour. By combining the Nightwatch data with the basic schedule preference data into a worker mask, the `deskWorker` object eliminates the need to explicitly track Nightwatch data during the rest of the program. We have chosen to isolate data parsing and loading in `workerPrefs` objects from data mangling in `deskWorker` objects so as to make code maintenance easier.

When Sledgehammer finds an optimal solution, it prints the solution and some subsidiary results to the screen and terminates, as shown in the bottom of `sledgehammer.py` (see Section B.1). In particular, it prints out the optimal schedule and then prints a table listing for each worker, how many hours they were actually scheduled for, how many hours they were supposed to be scheduled for (based on rank preferences), their maximum hours per week, a list of their shift lengths per day, and their minimum and maximum shift length constraints. Sledgehammer also details how many workers were assigned more hours than

their target amount and how many were assigned less, along with a histogram. Finally, Sledgehammer prints out the names of each worker that was unscheduled. Normally, this list should only include conditional workers that were not needed.

To aid in debugging, Sledgehammer logs the paths it traverses to the search tree. After solving the problem, this log is used to generate a graph which indicates what each slot was being assigned to and the worker being assigned at each node in the search tree. The `logger` class is defined in `logger.py` listed in Section B.6. The code to actually build and dump the GraphViz source file to disk are located at the very end of `sledgehammer.py` in Section B.1.

3.2 Solving the CSP

The heart of Sledgehammer lies in `solver.py` (see Section B.2). This modules implements a `candidateSolution` class, a `change` class, and a `solve` function. Given a candidate solution, the `solve` will always return either false if it cannot construct a valid solution starting from the candidate or the optimal solution it can find starting from the candidate. It first checks to see all possible variables have been assigned; if so, the solution is complete and `solve` simply returns it. Otherwise, `solve` asks the candidate to pick the most constrained slot remaining and then to provide an ordered list of possible changes to that slot. For each choice, `solve` creates a completely isolated copy of the candidate solution. If the new candidate satisfies all the constraints, solve recursively calls itself on the new candidate.

Candidate solutions include both a state table and a domains table. The state table is indexed by slot day and hour; each entry specifies which worker (or none if no one has been assigned) for that day and hour has been assigned within this candidate solution. The domain table is similar, but instead of listing the single worker that has been assigned, it specifies a set of workers that could be assigned to that slot. When a candidate needs to pick it's most constrained variable, it does so by choosing the slot whose domain length is smallest. `candidateSolution` objects have methods corresponding to different types of constraints. In particular, they have a `checkWeeklyHoursConstraint` that ensures workers never exceed their weekly hours maximum and that those that do are eliminated from all other unchosen slot domains. They also have a `checkPerDayConstraints` method that verifies that each worker works at most one shift per day and that that shift is between the worker's specified minimum and maximum lengths. Finally, they have a `checkTenHourConstraint` method that ensures workers never have shifts less than ten hours apart.

All of the machinery described so far ensures that we build solutions that satisfy the constraints but that don't necessarily meet worker preferences. Preference satisfaction is addressed in the `newValues` method which for a given empty slot, produces an ordered list of possible changes that would fill it. Changes are sorted based on scoring function that takes into account both the hour ratio and the vitality of the worker being added. For a particular worker, the hour ratio is the ratio between how many hours they have been scheduled for so far divided by the average number of hours they are supposed to be scheduled for. Vitality, as implemented by the `vitality` method, is an esoteric measure of how badly a particular worker needs a new slot. It attempts to integrate both the number of other potential slots the worker could fill and the hour ratios of all of those other slots' potential workers. The

use of vitality does improve the quality of Sledgehammer's preference matching.

While originally Sledgehammer would simply assign workers to the slot under consideration, we quickly discovered that such a program would never schedule any worker who had requested a minimum shift length constraint. Sledgehammer solves this problem by generalizing the notion of choosing a new value for a given slot: instead of picking a single worker to fill that slot, it picks a `change` object. `change` objects specify a set of additions and deletions to the candidate solution. For workers without minimum shift length constraints, `change` objects simply specify that the worker should be added to the slot in question. For workers with minimum shift lengths, `change` objects specify both that they should be added and that other workers in adjacent slots should be removed. Each `change` object will always lead to a solution that satisfies that worker's minimum shift length constraint. In order to assign a minimum shift length worker to a given slot, we will need to generate several `change` objects, each of which will include the slot in question but vary in what other slots they include.

3.3 Requirements

Sledgehammer is written in Python, an interactive, dynamic, object oriented language [4]. It relies on the Numerical Python extensions [1] to provide efficient array data types and the kjbuckets extension [5] to provide efficient set and graph data types. Sledgehammer should run on any system that has Python 2 or greater installed and both the kjbuckets and Numerical Python extensions. Sledgehammer includes a slower version of kjbuckets written in Python (see Section B.5 that is automatically used if the compiled version is unavailable; however, this substitution reduces performance by about 20%. All of the above software packages are available on the Internet and can be built from source.

The GraphViz software package [3] is required in order to generate the graphs shown in Figures 1 and 2. Sledgehammer will produce an appropriate GraphViz input file (named log.dot) which must be processed with GraphViz to generate an appropriate postscript file.

3.4 Enhancements

We have experimented with several different techniques to enhance the quality of Sledgehammer's search. These enhancements are listed in Section B.7. Specifically, we attempted to use a metric we have named the cross vitality to guide both most constrained variable selection and least constrained value selection. Cross vitality is a recursively defined version of vitality. For each worker, it is a measure of how badly that worker needs to be assigned to a new slot. A worker's cross vitality is thus the sum over all possible slots they can be assigned to of the weighted cross vitalities of the other candidates for that slot. We hypothesized that at any point in the search, the best slot to select next would be the slot that had the smallest sum of its candidate workers cross vitality. We also hypothesized that once Sledgehammer had chosen a slot to examine, the best change to assign would be the addition of the worker with the minimum cross vitality.

We further hypothesized that cross vitality could be calculated in practice by constructing an n-by-n weight matrix w where n is the number of workers and the entries of w. The value of w at indices i and j specify the number of unchosen slots for which both worker i and

worker j are candidates. After normalizing this matrix, we then construct a matrix h is a length n vector. The value of h at index i is simply the hour ratio of worker i. After normalizing h, we calculate a new value of h using $h = w' \times (w \times h)$. This technique was inspired by [2], which details a method for automatic resource compilation on the world wide web. It was also inspired by the PageRank algorithm used by Google.

We also considered, but did not implement the idea of using maximum flow algorithms to help determine optimum selection of the most constrained variable and least constrained value.

4 Core Design Decisions

4.1 Language Choice

Sledgehammer is designed around two primary goals: to be simple, and to be reasonably fast, with a premium placed on bypassing large areas of the search tree early in the search process. The need for speed, particularly improvements associated with non constant factor optimizations derives directly from the problem's vast search space. Simplicity is simply a requirement for finishing the project on time and ensuring a reasonably low defect count. These requirements compelled us to develop using Python, a language that while slow at runtime, makes development much faster.

4.2 CSP versus Rules

We decided to build Sledgehammer as a Constraint Satisfaction problem rather than a rule based system because our experience suggested that rule based systems provided a flexibility that the problem didn't require at the expense of being excessively slow and fragile. Our prospective rule based design had too few rules to justify the costs of a rule based system.

4.3 Specificity

When designing Sledgehammer, we initially considered building a very generic constraint satisfaction problem solver where the specific features describing the Random Hall desk scheduling problem were small and well isolated from the generic solver. Instead, we opted to build a highly specific solver where Random Hall problem features were tightly integrated into the solver. We did so because we felt that there were already many generic scheduling systems available; by building an integrated system, we could develop a better feeling for how the particular features of the Random Hall problem changed the solution process. This was especially helpful when we attempted to solve the pathological interactions associated with the minimum shift length constraints.

4.4 Separate versus Integrated Input Data Files

Sledgehammer opts to process several similar input data files rather than one large data file containing everything it needs. We made this decision because keeping data files isolated

allows for finer grained control: we don't necessarily want workers specifying whether they are reliable or how many terms they have worked. Such information should probably be entered by a trusted party, not the individual workers.

5 Results

In order to measure Sledgehammer's success, we ran it against the sample data from Random Hall and verified that all solutions it produced met all of the constraints described in the Constraints section above. Of course, we have to be careful to ensure that Sledgehammer actually works on a broad array of inputs and is not simply being tuned and tweaked to perform well on the sample data set. In addition to checking that Sledgehammer produces solutions that satisfy all constraints, we also evaluated how well Sledgehammer's solutions satisfy preferences. Specifically, we've looked at whether or not conditional workers ever get scheduled, how many unconditional workers actually get scheduled, how many workers get scheduled for days which they list as unreliable, and how closely we're able to approach worker's assigned hours preferences.

The results of this sample run are listed in Section A and in Figures 1 and 2. For the sample data, Sledgehammer was able to produce a solution that did not schedule any unconditional workers and assigned 11 out of 22 workers the exact number of hours their rank required while assigning 9 other workers within one hour of their preferred number of hours per week.

6 Contributions

We have provided the world with a well designed worker shift scheduling program that uses a variety of techniques to meet conflicting constraints while incorporating user preferences. Our major contributions include the development of vitality as search guiding metric which has proven very effective at improving the quality of search. We have also developed the notion of cross vitality, and while that has not proven nearly as useful so far, we expect that continued research in that area will in time bear fruit. Finally, we have demonstrated how a highly specific system that aggressively isolates constraints and obsessively reduces available options can produce excellent results with good performance, despite slower technology and very short development time.

References

[1] David Ascher, Paul F. Dubois, Konrad Hinsen, Jim Hugunin, and Travis Oliphant. *Numerical Python*. Lawrence Livermore National Laboratory, Livermore, CA 94566, September 2001. Available at `http://www.pfdubois.com/numpy/numpy.pdf`.

[2] Soumen Chakrabartia, Byron Doma, Prabhakar Raghavana, Sridhar Rajagopalana, David Gibsonb, and Jon Kleinbergc. Automatic resource compilation by analyzing hyperlink structure and associated text. In *Seventh International World Wide Web Conference*,

Figure 1: The search tree traversed by Sledgehammer while scheduling with the sample data.

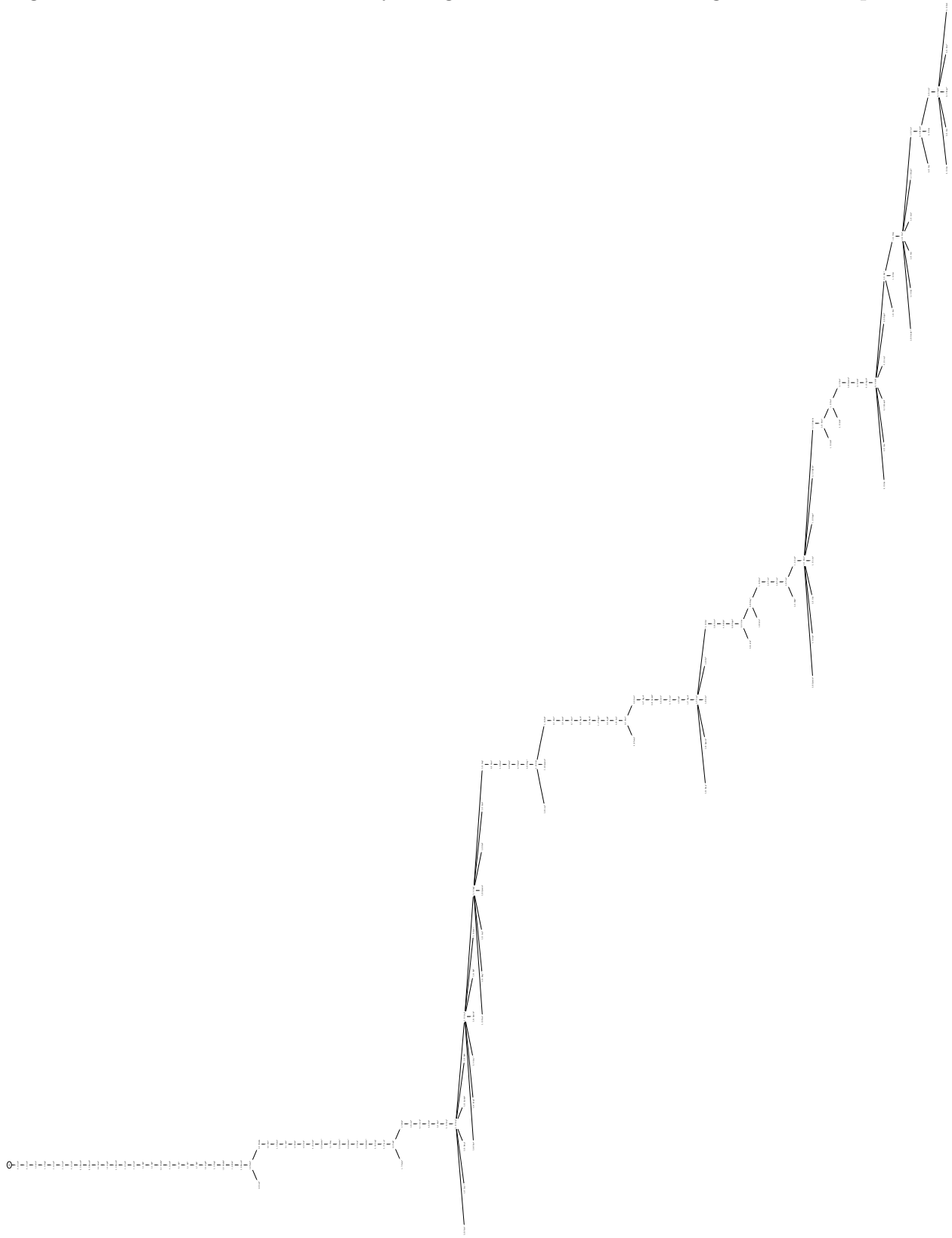
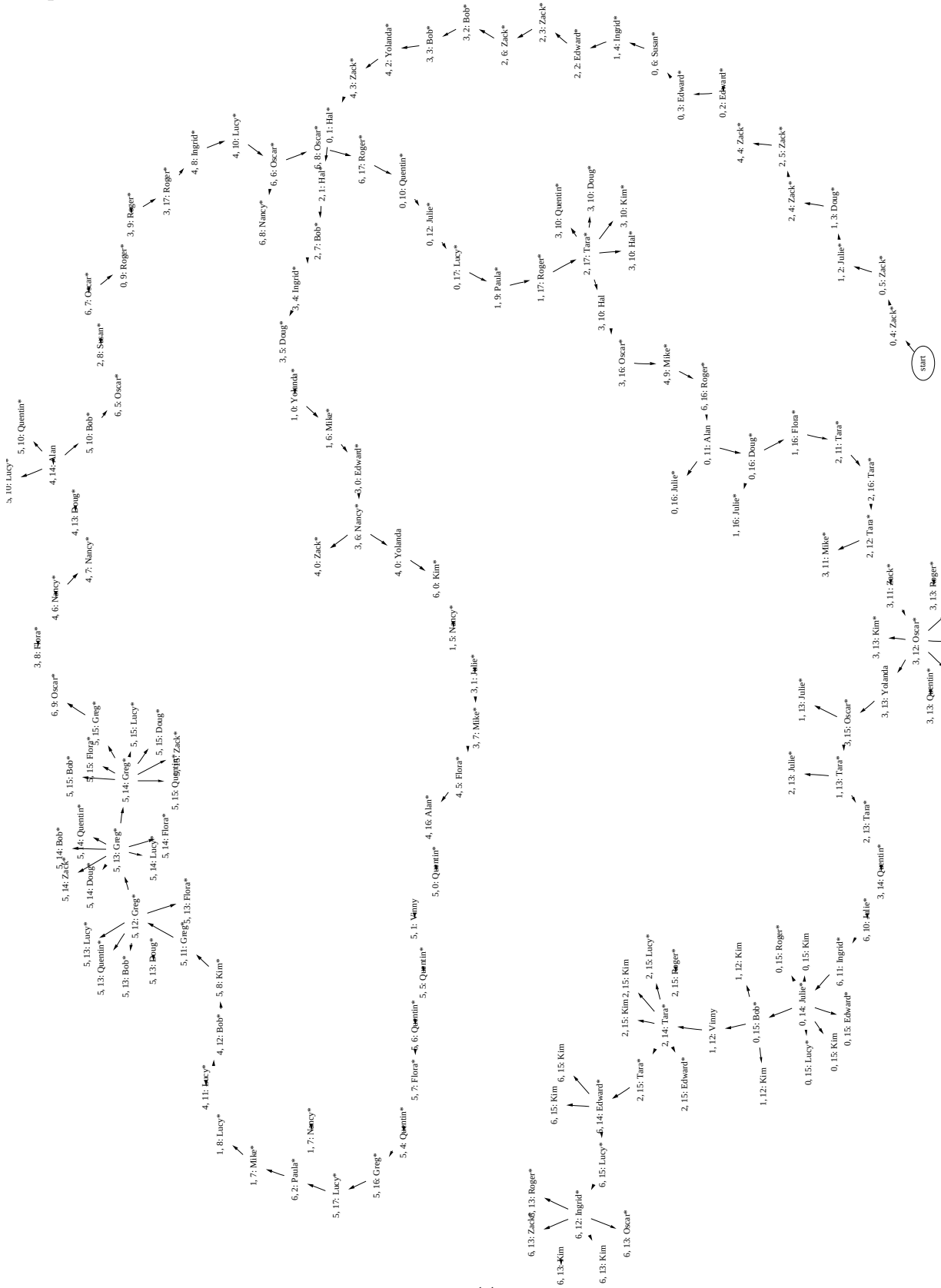


Figure 2: A closeup of the search tree traversed by Sledgehammer while scheduling with the sample data.



Brisbane, Australia, April 1998. Available at http://www7.scu.edu.au/programme/fullpapers/1898/com1898.html.

[3] Eleftherios Koutsofios and Stephen North. *Drawing graphs with dot.* AT&T Research Labs – Shannon Laboratory, 180 Park Avenue, Florham Park, NJ 07932, February 2002. Available at http://www.research.att.com/sw/tools/graphviz/dotguide.pdf.

[4] G. van Rossum and F.L. Drake. *Python Reference Manual.* PythonLabs, Virginia, USA, 2001. Available at http://www.python.org/.

[5] Aaron Watters. *Set and Graph Datatypes for Python: kjbuckets Release 2.2.* New Jersey Institute of Technology, University Heights, NJ 07102, 1994. Available at http://starship.python.net/crew/aaron_watters/kjbuckets/kjbuckets.html.

A Sample Run Output Listing

```
[[Hal  Yolanda  Hal  Edward  Yolanda  Vinny  Kim ]
 [Hal  Yolanda  Hal  Julie  Yolanda  Vinny  Kim ]
 [Edward  Julie  Edward  Bob  Yolanda  Vinny  Paula ]
 [Edward  Doug  Zack  Bob  Zack  Vinny  Paula ]
 [Zack  Ingrid  Zack  Ingrid  Zack  Quentin  Paula ]
 [Zack  Nancy  Zack  Doug  Flora  Quentin  Oscar ]
 [Susan  Mike  Zack  Nancy  Nancy  Quentin  Oscar ]
 [Susan  Mike  Bob  Mike  Nancy  Flora  Oscar ]
 [Susan  Lucy  Susan  Flora  Ingrid  Kim  Oscar ]
 [Roger  Paula  Susan  Hal  Mike  Kim  Oscar ]
 [Quentin  Paula  Susan  Hal  Lucy  Bob  Julie ]
 [Alan  Paula  Tara  Zack  Lucy  Greg  Ingrid ]
 [Alan  Vinny  Tara  Yolanda  Bob  Greg  Ingrid ]
 [Alan  Vinny  Tara  Yolanda  Doug  Greg  Zack ]
 [Julie  Vinny  Tara  Quentin  Alan  Greg  Edward ]
 [Bob  Vinny  Tara  Oscar  Alan  Greg  Lucy ]
 [Doug  Flora  Tara  Oscar  Alan  Greg  Roger ]
 [Lucy  Roger  Tara  Roger  Alan  Lucy  Roger ]]

above = 9     below = 5
[(-2, 1), (-1, 3), (0, 11), (1, 6), (3, 1)]
(Alan, 1, 7, 6, 42, array([3, 0, 0, 0, 4, 0, 0]), 3, 6)
(Bob, 0, 6, 6, 65, array([1, 0, 1, 2, 1, 1, 0]), 1, 18)
(Doug, 0.0, 4, 4.0, 90, array([1, 1, 0, 1, 1, 0, 0]), 1, 5)
(Edward, 0, 5, 5, 44, array([2, 0, 1, 1, 0, 0, 1]), 1, 18)
(Flora, 0.0, 4, 4.0, 71, array([0, 1, 0, 1, 1, 1, 0]), 1, 5)
(Greg, 1, 6, 5, 15, array([0, 0, 0, 0, 0, 6, 0]), 1, 15)
(Hal, 1, 6, 5, 50, array([2, 0, 2, 2, 0, 0, 0]), 2, 18)
(Ingrid, 0, 5, 5, 8, array([0, 1, 0, 1, 1, 0, 2]), 1, 4)
(Julie, 0.0, 4, 4.0, 29, array([1, 1, 0, 1, 0, 0, 1]), 1, 18)
(Kim, -1, 4, 5, 5, array([0, 0, 0, 0, 0, 2, 2]), 2, 4)
(Lucy, 0, 6, 6, 61, array([1, 1, 0, 0, 2, 1, 1]), 1, 18)
(Mike, 0.0, 4, 4.0, 10, array([0, 2, 0, 1, 1, 0, 0]), 1, 5)
(Nancy, -2, 4, 6, 10, array([0, 1, 0, 1, 2, 0, 0]), 1, 10)
(Oscar, -1, 7, 8, 30, array([0, 0, 0, 2, 0, 0, 5]), 1, 18)
(Paula, 1, 6, 5, 19, array([0, 3, 0, 0, 0, 0, 3]), 3, 18)
(Quentin, 0, 5, 5, 81, array([1, 0, 0, 1, 0, 3, 0]), 1, 18)
(Roger, -1, 5, 6, 39, array([1, 1, 0, 1, 0, 0, 2]), 1, 18)
(Susan, 1, 6, 5, 32, array([3, 0, 3, 0, 0, 0, 0]), 3, 5)
(Tara, 0, 7, 7, 16, array([0, 0, 7, 0, 0, 0, 0]), 1, 16)
(Vinny, 3, 8, 5, 25, array([0, 4, 0, 0, 0, 4, 0]), 4, 4)
(Yolanda, 1, 7, 6, 15, array([0, 2, 0, 2, 3, 0, 0]), 2, 6)
```

```
(Zack, 0, 10, 10, 91, array([2, 0, 4, 1, 2, 0, 1]), 1, 18)

22 workers scheduled out of 24 workers available

Workers that were not scheduled are:
Carol 8 1
[[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]]

Waldo 4.0 1
[[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]]
```

B Source Code Listings

B.1 sledgehammer.py

```
#!/usr/bin/env python
from Numeric import *
from copy import copy
from fakekjset import kjSet
from dataInput import *
from deskWorker import deskWorker
from solver import *

blocks = makeBlocks(open('../inputs/raw-data.txt', 'r'))
prefs = map(workerPrefs, blocks)

rankData = loadRankData('../inputs/raw-rank-data.txt')
nightwatchData = loadGenericData('../inputs/raw-nightwatch-exceptions.txt')
unreliableData = loadGenericData('../inputs/raw-unreliable-data.txt')

# now we bring it all together...
for pref in prefs:
    name = pref.name
    pref.averageHours = averageHours(rankData[name])
    pref.nightwatchExceptions = nightwatchData.get(name, ())
    pref.unreliableDays = map(int, unreliableData.get(name, ()))

# data loading is done, so now we move up to a higher abstraction
# data loading and data mangling/processing should be separate and
# isolated componants: thats why we have separate workerPref and deskWorker classes

workers = map(deskWorker, prefs)
workerDict = {}
conditionalWorkers = []
unconditionalWorkerMask = zeros((7, 18))

for worker in workers:
    workerDict[worker.name] = worker
    if not(worker.conditionalWorker):
        # this is a normal worker
        unconditionalWorkerMask = logical_or(
            unconditionalWorkerMask,
            worker.workMask)
    else:
        # this is a conditional worker
```

```
        conditionalWorkers.append(worker)

# we need to reduce the workMask for all conditional workers
# to only include times they can staff that no normal worker
# can staff.

for u in conditionalWorkers:
    u.workMask = logical_and(u.workMask,
                             logical_not(unconditionalWorkerMask))

# we really should update the maxHoursPerWeek in the deskWorker
# constructor for all unconditional workers, but its really not
# that important

workerIndexMap = {}
workerIndexReverseMap = {}
workers.sort()
for i in range(len(workers)):
    w = workers[i]
    workerIndexMap[w] = i
    workerIndexReverseMap[i] = w


def outputAnalysis(solution):
    data = []
    above = 0
    below = 0
    histogram = {}
    for w in solution.currentWorkers.items():
        shifts = sum(equal(solution.state, w), 1)
        #print w, equal(solution.state, w), sum(equal(solution.state, w), 1)
        total = sum(shifts)
        assert total == solution.weeklyHours[w]
        diff = total - w.averageHours
        data.append((w, diff, total, w.averageHours,
                     w.maxHoursPerWeek, shifts, w.minHoursPerShift, w.maxHoursPerShift))
        if diff > 0:
            above += diff
        if diff < 0:
            below -= diff

        if not(histogram.has_key(int(diff))):
            histogram[int(diff)] = 0
        histogram[int(diff)] += 1
```

```
        for s in shifts:
            if s and not(w.minHoursPerShift <= s <= w.maxHoursPerShift):
                print 'Aghhhhhhhhh! min/max violation for worker', w
    print 'above =', above, '   ', 'below =', below
    h = histogram.items()
    h.sort()
    print h

    data.sort()
    return data


# build initial state and domains
initialState = zeros((7, 18), PyObject)
initialDomains = zeros((7, 18), PyObject)
for day in range(7):
    for hour in range(18):
        x = kjSet()
        for w in workers:
            if w.workMask[day][hour]:
                x.add(w)
        initialDomains[day, hour] = x

initialCandidate = candidateSolution(initialState, initialDomains)
initialCandidate.constrain()

# here we go...
bestSolution = solve(initialCandidate)

b = bestSolution
print transpose(b.state)
print
y = outputAnalysis(b)
for x in y:
    print x
print
print len(y), 'workers scheduled out of', len(workerDict.keys()), 'workers available'
print
print "Workers that were not scheduled are:"
for w in workers:
    if not(b.currentWorkers.member(w)):
        print w, w.averageHours, w.minHoursPerShift
        print w.workMask
```

```
        print


# write the graphviz log file out...
f = open('log.dot', 'w')
f.write('digraph G {\nsize="10,7.5"; ratio = fill;\nrotate=90\n')
f.write('\n'.join(theLog.log))
f.write('}')
f.close()
```

B.2 solver.py

```
from Numeric import *
from copy import copy
from fakekjset import kjSet

from logger import logger
theLog = logger()

class candidateSolution:
    def __init__(self, state, domains):
        self.state = state
        self.domains = domains
        self.weeklyHours = {}
        self.currentWorkers = kjSet()

    def add(self, worker, day, hour):
        if self.domains[day, hour].member(worker):
            if not(self.weeklyHours.has_key(worker)):
                # this is the first time this worker
                # has ever been assigned...
                self.weeklyHours[worker] = 0
            self.weeklyHours[worker] += 1
            self.state[day, hour] = worker
            self.currentWorkers.add(worker)
        else:
            raise StandardError(
                "Can't add worker %s to day %s, hour %s" % (worker, day, hour))

    def delete(self, worker, day, hour):
        self.state[day, hour] = 0
        if self.weeklyHours[worker] == 1:
            del self.currentWorkers[worker]
```

```
    self.weeklyHours[worker] -= 1

def clone(self):
    newState = copy(self.state)
    newDomains = zeros(self.domains.shape, PyObject)
    for day in range(7):
        for hour in range(18):
            newDomains[day, hour] = kjSet(self.domains[day, hour])
    myClone = candidateSolution(newState, newDomains)
    # we really should clean this up so that you can set these
    # from the constructor...
    myClone.weeklyHours = copy(self.weeklyHours)
    myClone.currentWorkers = kjSet(self.currentWorkers)
    return myClone

def variableScore(self, day, hour):
    score = 0
    for worker in self.domains[day, hour].items():
        score += self.vitality(worker, day, hour)
    return score

def pickMostConstrainedVariable(self):
    # list of (domainLen, (day, hour)) tuples for all unfilled slots
    slotRanks = [(len(self.domains[day, hour]), (day, hour))
                 for day in range(7)
                 for hour in range(18)
                 if not(self.state[day, hour])]
    slotRanks.sort()
    (minLen,(day, hour)) = slotRanks[0]
    return day, hour, minLen
    # if we find an empty domain, we'll just return that and let
    # our caller deal with it...

def constrain(self):
    # weekly hours check
    if not(self.checkWeeklyHoursConstraint(self.currentWorkers.items())):
        return 0

    # check min/max hours per day and one shift per day constraints
    for day in range(7):
        if not(self.checkPerDayConstraints(day)):
            return 0

    # 10 hour separation rule!
```

```
    for day in range(7):
        if not(self.checkTenHourConstraint(day)):
            return 0

    return 1 # if we managed to get here, then all domains have been
             # reduced and all constrains are still valid...

def fastConstrain(self, change, day):
    # this version of constrain is garunteed to give the same results
    # as the full constrain above as long as we've only added a single worker
    if not(self.checkWeeklyHoursConstraint(change.workers())):
        return 0

    if not(self.checkPerDayConstraints(day)):
        return 0

    for this_day in (day, (day - 1) % 7):
        # check for 10 hour violations both on the night of day we added
        # a worker to and on the previous day...
        if not(self.checkTenHourConstraint(this_day)):
            return 0
    return 1

def checkWeeklyHoursConstraint(self, listOfWorkers):
    # we also reduce domains here...
    # weekly hours check; pass in a sequence of worker NAMES as argument
    for w in listOfWorkers:
        x = self.weeklyHours[w]
        y = w.maxHoursPerWeek
        if x > y:
            # constraint violation!
            return 0
        elif x == y:
            # we could probably make this faster by pulling this test
            # outside the loop so it only gets done at most once...
            # we've maxed out worker w so we should remove him from
            # all other domains
            for day in range(7):
                for hour in range(18):
                    if self.state[day, hour] == 0:
                        set = self.domains[day, hour]
                        if set.member(w):
                            del set[w]
        # check to see if any domains have become empty...
```

```
            if not(alltrue(array(map(len, ravel(self.domains))))):
                # at least one domain now has length zero, so we
                # should fail now...
                return 0
    return 1

def checkPerDayConstraints(self, dayNumber):
    # check min/max hours per day and one shift per day constraints
    day = self.state[dayNumber,:]
    currentWorker = None
    workerHours = {}
    shifts = {}
    for hour in range(18):
        slot = day[hour]
        # slot is actually the name of the worker for that hour
        if slot:
            if slot != currentWorker:
                if workerHours.has_key(slot):
                    # bad news: we've got multiple shifts for this
                    # worker...we should fail now...
                    return 0
                else:
                    currentWorker = slot
                    workerHours[slot] = 0
                    shifts[slot] = []
            workerHours[currentWorker] += 1
            shifts[slot].append(hour)

    # check min/max hours per shift constraints here
    for worker, hours in workerHours.items():
        if worker: # just to eliminate the 0 entries...
            thisShift = shifts[worker]
            thisShift.sort()
            if hours < worker.minHoursPerShift:
                return 0
            if hours > worker.maxHoursPerShift:
                return 0
    return 1

def checkTenHourConstraint(self, day):
    # (22, 23, 0, 1)
    for hour in (14, 15, 16, 17):
        worker = self.state[day, hour]
        if worker and (worker in self.state[(day + 1) % 7, :(hour - 13)]):
```

```
            return 0
    return 1

def execute(self, aChange):
    for worker, day, hour in aChange.deletions:
        self.delete(worker, day, hour)
    for worker, day, hour in aChange.additions:
        self.add(worker, day, hour)

def vitality(self, worker, day, hour):
    daysAvailableToWork = logical_and(equal(self.state, 0), worker.workMask)
    # list of all domains this worker can work that are as yet unscheduled
    alternativeSlotDomains = compress(ravel(daysAvailableToWork), ravel(self.domains))
    total = 0.
    for domain in alternativeSlotDomains:
        slotTotal = 0.
        for otherWorker in domain.items():
            excess = otherWorker.averageHours - self.weeklyHours.get(otherWorker, 0)
            # only count workers who haven't yet met their weekly target
            if (excess > 0) and otherWorker != worker:
                slotTotal +=  excess
        total += slotTotal
    return total / len(alternativeSlotDomains)

def newValues(self, day, hour):
    availableWorkers = self.domains[day, hour].items()
    x = [] #map(self.workerScore, availableWorkers)
    for worker in availableWorkers:
        thisWeekHours = float(self.weeklyHours.get(worker, 0))
        avgHours = float(worker.averageHours)
        score = (thisWeekHours / avgHours) * self.vitality(worker, day, hour)
        x.append((score, worker))
    x.sort()

    changeList = []
    for score, worker in x:
        changeList.extend(self.buildChangeSet(worker, day, hour))
    return changeList

def buildChangeSet(self, worker, day, hour):
    changeList = []
    m = worker.minHoursPerShift
    if m > 1:
        for changeNum in range(m):
```

```
                startHour = hour - changeNum
                endHour = startHour + m - 1
                if (startHour >= 0) and (endHour < 18):
                    additions = []
                    deletions = []
                    for index in range(m):
                        additions.append( (worker, day, startHour + index) )
                        deletedWorker = self.state[day, startHour + index]
                        if (index != hour) and deletedWorker:
                            deletions.append( (deletedWorker, day, startHour + index) )
                    c = change(additions, deletions)
                    if c.check(self):
                        changeList.append(c)
        else:
            changeList.append(change(((worker, day, hour),) ))
        return changeList

class change:
    def __init__(self, additions, deletions = ()):
        self.additions = additions
        self.deletions = deletions

    def workers(self):
        x = kjSet()
        for w, d, h in self.additions:
            x.add(w)
        for w, d, h in self.deletions:
            x.add(w)
        return x.items()

    def __repr__(self):
        return self.additions[0][0]

    def __hash__(self):
        return hash((tuple(self.additions), tuple(self.deletions)))

    def check(self, solution):
        # verify that this change is valid: i.e., that it doesn't force
        # workers to work slots they're not open for and that it
        # doesn't delete from a slot a worker that is at his maxHoursPerWeek
        # (since we can't rollback those deletions without putting restocking
        #  the deleted worker's domains...)
        for deletedWorker, delDay, delHour in self.deletions:
            if solution.weeklyHours[deletedWorker] >= deletedWorker.maxHoursPerWeek:
```

```
            return 0
        for worker, day, hour in self.additions:
            if not(solution.domains[day, hour].member(worker)):
                return 0
        return 1



def solve(candidate):
    numLeft = sum(sum(candidate.state == 0))
    print numLeft
    # this test should probably move into pickMostConstrainedVariable...
    if sum(sum(candidate.state == 0)):
        # at least one state variable hasn't been selected
        (day, hour, domainSize) = candidate.pickMostConstrainedVariable()
        assert domainSize != 0
        for newValue in candidate.newValues(day, hour):
            newCandidate = candidate.clone()
            newCandidate.execute(newValue)
            theLog.logNewSolution(day, hour, newValue)
            if newCandidate.fastConstrain(newValue, day):
                # all the constraints check out
                x = solve(newCandidate)
                if x:
                    return x
            else:
                theLog.logSolutionFailure()
            # if any constraints are violated, just move on to the
            # next possible value...
        # if we've haven't returned with any winning solutions by now,
        # we never will, so lets just fail...
        return 0
    else:
        # we've run out of variables to pick...since there were no
        # constraint violations, this solution must be a winner....yay!
        return candidate
```

B.3 dataInput.py

```
#!/usr/bin/env python

def makeBlocks(lines):
```

```
    currentBlock = []
    allBlocks = []
    for line in lines:
        if line.strip():
            currentBlock.append(line)
        else:
            allBlocks.append(currentBlock)
            currentBlock = []
    if currentBlock:
        allBlocks.append(currentBlock)
    return allBlocks

class workerPrefs:
    def __init__(self, lines):
        self.name = ''
        self.maxHoursPerWeek = 0
        self.maxHoursPerShift = 0
        self.minHoursPerShift = 0
        self.worksNightwatch = 0
        self.conditionalWorker = 0
        self.workTimes = []
        for i in range(7):
            self.workTimes.append([None] * 18)

        for line in lines:
            if line[:2].isdigit():
                time = int(line[:2]) - 8
                dayNumber = 0
                for character in line[2:9]:
                    self.workTimes[dayNumber][time] = (character == 'x')
                    dayNumber = dayNumber + 1
            elif ':' in line:
                field, value = self.parseField(line)
                field = field.lower()
                if field == 'max hours per week':
                    self.maxHoursPerWeek = self.parseNumberField(value)
                elif field == 'max hours per shift':
                    self.maxHoursPerShift = self.parseNumberField(value)
                elif field == 'min hours per shift':
                    self.minHoursPerShift = self.parseNumberField(value)
                elif field == 'works nightwatch?':
                    self.worksNightwatch = self.parseBooleanField(value)
                elif field == 'name':
                    self.name = value
```

```
            elif field == 'other notes':
                if line.lower().find('conditional worker') != -1:
                    self.conditionalWorker = 1

    def parseField(self, text):
        if ':' not in text:
            raise StandardError("Can't parse field: ", text)
        x = text.split(':')
        field = x[0].strip()
        value = x[-1].strip()
        return field, value

    def parseNumberField(self, text):
        if '-' in text:
            return 0
        elif text.isdigit():
            return int(text)
        else:
            raise StandardError("Can't parse numeric value:", text)

    def parseBooleanField(self, text):
        x = text.lower()
        if x in ('y', 't'):
            return 1
        elif (x in ('n', 'f')) or (x == ''):
            return 0
        else:
            raise StandardError("Can't parse boolean value:", text)

# toss in ranks here...
def averageHours(rank):
    if rank < .5:
        return 4.
    elif .5 <= rank <= 1:
        return 5
    elif 1.5 <= rank <= 2:
        return 5
    elif 2.5 <= rank <= 3:
        return 6
    elif 3.5 <= rank <= 4:
        return 6
    elif 4.5 <= rank <= 5:
        return 7
    elif 5.5 <= rank <= 6:
```

```
        return 7
    elif 6.5 <= rank <= 7:
        return 8
    elif 7.5 <= rank <= 8:
        return 9
    elif rank >= 8.5:
        return 10
    else:
        raise ValueError()

def loadRankData(fname):
    result = {}
    # skip the header line
    lines = open(fname, 'r').readlines()[1:]
    for line in lines:
        x = line.split()
        name = x[0]
        terms = x[1]
        summers = x[2]
        result[name] = float(terms) + 0.5 * float(summers)
    return result

def loadGenericData(fname):
    result = {}
    for line in open(fname, 'r').readlines():
        if line.strip():
            fields = line.split()
            result[fields[0]] = fields[1:]
    return result


if __name__ == '__main__':
    # some random test code...
    noMin = 0
    noMaxShift = 0
    noMaxWeek = 0
    for p in prefs:
        if p.maxHoursPerShift == 0:
            noMaxShift += 1
        if p.maxHoursPerWeek == 0:
            noMaxWeek += 1
        if p.minHoursPerShift == 0:
            noMin += 1
```

```
    print noMin, noMaxWeek, noMaxShift
    print len(prefs)
```

B.4 deskWorker.py

```
from Numeric import *
class deskWorker:
    def __init__(self, prefs):
        for field in ('name', 'maxHoursPerWeek', 'minHoursPerShift',
                      'maxHoursPerShift', 'worksNightwatch',
                      'averageHours', 'nightwatchExceptions',
                      'conditionalWorker', 'unreliableDays'):
            setattr(self, field, getattr(prefs, field))

        self.workMask = array(prefs.workTimes)

        if self.worksNightwatch:
            if self.nightwatchExceptions:
                for excep in self.nightwatchExceptions:
                    if excep == 'preFriday':
                        self.workMask = logical_and(self.workMask,
                                                    logical_not(self.preFridayNightwatchMask))
                    elif excep == 'postFriday':
                        self.workMask = logical_and(self.workMask,
                                                    logical_not(self.postFridayNightwatchMask))
                    elif excep == 'preSaturday':
                        self.workMask = logical_and(self.workMask,
                                                    logical_not(self.preSaturdayNightwatchMask))
                    elif excep == 'postSaturday':
                        self.workMask = logical_and(self.workMask,
                                                    logical_not(self.postSaturdayNightwatchMask))
                    else:
                        raise ValueError("Unknown nightwatch exception for worker %s: %s"
                                         % (self.name, excep))
            else:
                self.workMask = logical_and(self.workMask,
                                            logical_not(self.genericNightwatchMask))

        # this comes from assuming all slots over all days while
        # maintaining the 10 hour rest rule.
        if self.maxHoursPerWeek == 0:
            self.maxHoursPerWeek = 98
```

```
    # this is designed to try and minimize maxHoursPerWeek by taking
    # advantage of nightwatch constraints
    self.maxHoursPerWeek = min(self.maxHoursPerWeek,
                               sum(sum(self.workMask)))

    if self.maxHoursPerShift == 0:
        self.maxHoursPerShift = min(18, self.maxHoursPerWeek)

    if self.minHoursPerShift == 0:
        self.minHoursPerShift = 1

    # we still need this even though we're counting nightwatch blocked out
    # times, since this is a cheap heuristic to compensate for the fact
    # that regardless of how it affects their schedule, nightwatch people
    # can at most work 5 hours less than their stated maxHoursPerWeek
    # because they are doing 5 hours of nightwatch on average each week.
    if self.worksNightwatch:
        self.maxHoursPerWeek -= 5

    # Technically, minimizing maxHoursPerWeek won't do anything since
    # even if its much higher than the max amount of possible hours,
    # we'll still be caught by the fact that it won't show up in any
    # domains once all slots to which it belongs have been assigned.
    # But, if we ever want to use a search or selection heuristic that
    # takes into account maxHoursPerWeek, it helps to make it accurate.

def __str__(self):
    return self.name
def __repr__(self):
    return self.name
def __hash__(self):
    return hash(self.name)
def __cmp__(self, other):
    if isinstance(other, self.__class__):
        return cmp(self.name, other.name)
    elif other in (None, 0):
        return -1
    else:
        raise StandardError("This is never supposed to happen!")

preFridayNightwatchMask = transpose(array(
    [[0, 0, 0, 0, 0, 0, 0],
     [0, 0, 0, 0, 0, 0, 0],
     [0, 0, 0, 0, 0, 0, 0],
```

```
    [0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 1, 0, 0],
    [0, 0, 0, 0, 1, 0, 0],
    [0, 0, 0, 0, 1, 0, 0],
    [0, 0, 0, 0, 1, 0, 0],
    [0, 0, 0, 0, 1, 0, 0],
    [0, 0, 0, 0, 1, 0, 0],
    [0, 0, 0, 0, 1, 0, 0],
    [0, 0, 0, 0, 1, 0, 0],
    [0, 0, 0, 0, 1, 0, 0],
    [0, 0, 0, 0, 1, 0, 0],
    [0, 0, 0, 0, 1, 0, 0]]))

postFridayNightwatchMask = transpose(array(
    [[0, 0, 0, 0, 0, 1, 0],
    [0, 0, 0, 0, 0, 1, 0],
    [0, 0, 0, 0, 0, 1, 0],
    [0, 0, 0, 0, 0, 1, 0],
    [0, 0, 0, 0, 0, 1, 0],
    [0, 0, 0, 0, 0, 1, 0],
    [0, 0, 0, 0, 0, 1, 0],
    [0, 0, 0, 0, 0, 1, 0],
    [0, 0, 0, 0, 0, 1, 0],
    [0, 0, 0, 0, 0, 1, 0],
    [0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0]]))

preSaturdayNightwatchMask = transpose(array(
    [[0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 1, 0],
```

```
        [0, 0, 0, 0, 0, 1, 0],
        [0, 0, 0, 0, 0, 1, 0],
        [0, 0, 0, 0, 0, 1, 0],
        [0, 0, 0, 0, 0, 1, 0],
        [0, 0, 0, 0, 0, 1, 0],
        [0, 0, 0, 0, 0, 1, 0],
        [0, 0, 0, 0, 0, 1, 0],
        [0, 0, 0, 0, 0, 1, 0],
        [0, 0, 0, 0, 0, 1, 0],
        [0, 0, 0, 0, 0, 1, 0],
        [0, 0, 0, 0, 0, 1, 0]]))

    postSaturdayNightwatchMask = transpose(array(
        [[0, 0, 0, 0, 0, 0, 1],
        [0, 0, 0, 0, 0, 0, 1],
        [0, 0, 0, 0, 0, 0, 1],
        [0, 0, 0, 0, 0, 0, 1],
        [0, 0, 0, 0, 0, 0, 1],
        [0, 0, 0, 0, 0, 0, 1],
        [0, 0, 0, 0, 0, 0, 1],
        [0, 0, 0, 0, 0, 0, 1],
        [0, 0, 0, 0, 0, 0, 1],
        [0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0]]))

    genericNightwatchMask = logical_or(logical_or(preFridayNightwatchMask,
                                                  postFridayNightwatchMask),
                                       logical_or(preSaturdayNightwatchMask,
                                                  postSaturdayNightwatchMask))
```

B.5 fakekjset.py

```
try:
    from kjbuckets import kjSet
except ImportError:
    print "Warning: can't find real kjbuckets.so, so using slower version"
```

```
class kjSet:
    def __init__(self, other = None):
        self.dict = {}
        if other:
            for k in other.items():
                self.add(k)
    def add(self, value):
        self.dict[value] = 1
    def items(self):
        return self.dict.keys()
    def __delitem__(self, key):
        del self.dict[key]
    def __len__(self):
        return len(self.dict.keys())
    def member(self, m):
        return self.dict.has_key(m)
```

B.6 logger.py

```
class logger:
    # this really should be a singleton class...
    def __init__(self):
        self.mostRecentID = 0
        self.stack = [('start', 99,99, 'boo')]
        self.log = []

    def currentID(self):
        return self.mostRecentID

    def nextID(self):
        self.mostRecentID += 1
        return "x%i" % self.mostRecentID

    def logNewSolution(self, day, hour, change):
        # the OOP gods hate me so much...
        name = change.additions[0][0].name
        # encapsulation?! we don't need no stinking encapsulation...
        if not(change.deletions):
            name += '*'
            # use a * after the name to signify that this change
            # requires deletions...
        pastSolution = self.currentSolution()
        ID = self.nextID()
```

```
        self.stack.append((ID, day, hour, name))
        self.writeLog(pastSolution, self.currentSolution())

    def currentSolution(self):
        return self.stack[-1]

    def logSolutionFailure(self):
        self.stack.pop()

    def writeLog(self, oldSolution, newSolution):
        # grungy graphviz stuff goes here
        oldID, oldDay, oldHour, oldName = oldSolution
        newID, newDay, newHour, newName = newSolution
        self.log.append('%(newID)s [label="%(newDay)i, %(newHour)i: %(newName)s" shape=plaint
        self.log.append("%(oldID)s -> %(newID)s;" % locals())
        # add node for new solution
```

B.7 experimental.py

```
class improvedCandidateSolution(candidateSolution):
    def pickMostConstrainedVariable(self):
        workerGrid = zeros((len(workers), len(workers)))
        workerSet = kjSet()
        for set in compress(ravel(equal(self.state, 0)), ravel(self.domains)):
            workerSet = workerSet + set
            for a in set.items():
                for b in set.items():
                    if a != b:
                        workerGrid[workerIndexMap[a], workerIndexMap[b]] += 1

        w = workerGrid / float(sum(sum(workerGrid)))

        workerRanks = zeros(len(workers)) * 1.0
        for worker in workerSet.items():
            workerRanks[workerIndexMap[worker]] = float(
                worker.averageHours - self.weeklyHours.get(worker, 0))

        h = clip(workerRanks, 0, 10000)
        h = transpose(h) / float(sum(h))

        for i in range(4):
            a = matrixmultiply(w, h)
            h = matrixmultiply(transpose(w), a)
```

```
    slotList = []
    for day in range(7):
        for hour in range(18):
            if not(self.state[day, hour]):
                total = 0
                for w in self.domains[day, hour].items():
                    total += h[workerIndexMap[w]]
                slotList.append((len(self.domains[day, hour]), total, (day, hour)))
    slotList.sort()
    print slotList[:4]
    (day, hour) = slotList[0][-1]
    return day, hour, len(self.domains[day, hour])


def vitality(self, worker, day, hour):
    workerGrid = zeros((len(workers), len(workers)))
    workerSet = kjSet()
    for set in compress(ravel(equal(self.state, 0)), ravel(self.domains)):
        workerSet = workerSet + set
        for a in set.items():
            for b in set.items():
                workerGrid[workerIndexMap[a], workerIndexMap[b]] += 1

    w = workerGrid / float(sum(sum(workerGrid)))

    workerRanks = zeros(len(workers))
    for worker in workerSet.items():
        workerRanks[workerIndexMap[worker]] = float(
            worker.averageHours - self.weeklyHours.get(worker, 0))

    h = clip(workerRanks, 0, 10000)
    h = transpose(h) / (float(sum(h)) or 1.0)

    for i in range(4):
        a = matrixmultiply(w, h)
        h = matrixmultiply(transpose(w), a)

    return self.h[workerIndexMap[worker]]
```

C Sample Data Files

C.1 Rank Data

```
Deskworker      Terms   Summers
Alan            3       2
Bob             3       1
Carol           5       3
Doug            0       0
Edward          2       0
Flora           0       0
Greg            1       1
Hal             1       0
Ingrid          1       2
Julie           0       0
Kim             1       0
Lucy            2       1
Mike            0       0
Nancy           4       0
Oscar           5       3
Paula           1       1
Quentin         1       1
Roger           3       2
Susan           0       1
Tara            5       2
Vinny           1       0
Waldo           0       0
Yolanda         2       1
Zack            7       3
```

C.2 Nightwatch Exceptions

```
Alan postFriday preSaturday postSaturday
Bob preSaturday postSaturday
```

C.3 Unrealiable Workers

```
Lucy 4 5 6
Zack 4 5 6
```

C.4 Worker Preferences Data

This file is identical to the worker prefernces data file distributed with the project.