Optimizing Volunteer Shifts for Dance Weekends

John Jeng, Samvel Stepanyan

January 22nd, 2015

Abstract

The Savoy Swing Club (SSC) in Seattle, WA is a non-profit that organizes dance weekends for the general public and has outreach programs to middle schools and high schools in the greater Seattle area. All events require volunteers to run the event. Volunteer scheduling is currently done by hand. Small events are easy, but larger events become more of a headache for event planners. Our objective was to provide a program which would take a pool of volunteers and set of shifts and jobs, and return a schedule of volunteers that optimized for work balance, number of volunteers, "choas", and shift preferences. We were able to create schedules that came within the absolute minimum number of volunteers by 1 or 2. We consider these schedules better than the most optimal ones because cutting it as close as exact leads to disastrous problems.

1 Problem Description

Savoy Swing Club (SSC) is local non-profit organization that currently seeks to teach swing dancing to middle and high school kids in the greater Seattle area. They are also heavily involved with 2 "dance weekends": Seattle Lindy Exchange (SLX) and Killerdiller Weekend (KDW). For this paper, we will focus on full-featured dance events where there are classes, evening dances, performances, and competitions. These are the events with the largest number of participants and can span any number of days and volunteers are required to make the weekend go smoothly. Participants in these events are those people who take part in classes, dances, or competitions. Volunteers may be participants and vice versa. If the event isn't adequately staffed, then the event participants suffer poor service. If the volunteers time constraints and preferences aren't taken into account in some degree, then performance is worse which leads to a worse participant experience.

Each volunteer has different jobs they are able to perform and preferences for which hours they are volunteering. Volunteers are compensated based on the number of hours they work and the type of work. For example, 8 hours of volunteering which required basic skills might grant access to all classes and evening dances while fewer hours of volunteering might just mean access to evening dances. A technician however, might receive a full weekend pass and additional cash compensation for just 6 hours of work. While we could optimize on some sort of cost function under conditions similar to the examples above, SSC has determined that since developing the cost function itself would require analysis of its own right, it is sufficient to just reduce the number of volunteers necessary for any given event.

Thus our objective is to develop a program that will return a schedule that is in some sense "optimal" for both the volunteers and SSC. We achieve this objective through the following means.

We will take into consideration the volunteers' general time preferences and availability. Specifically, we will allow them to choose a time frame that they would like to be scheduled to volunteer and to list times that they absolutely cannot volunteer. For example a volunteer might decide that they don't want to take classes but they want to be able to participate fully in the evening dances. However, they have a late lunch date so they cannot volunteer from 1:30 - 3:00 pm. This volunteer could mark that they would like to be scheduled between 9 am and 5 pm but be completely unavailable from 1:00 - 3:00 pm. This is an improvement over the current system because typically, volunteers do not know exactly what hours they will be working until at most a few days before the event. This means that volunteers must commit their entire weekends even though they may only be working 6 to 8 hours over 2 days.

We will create a schedule that is not too hectic and is easy for a supervisor to understand. For example, the need for fewer volunteers may give way to not having a schedule that had many volunteers switching jobs and shifts throughout the day. Chaotic schedules have a very real overhead cost since more effort must be put into logistics instead of just doing the jobs well.

Finally, we will create a program that is easy for an event planner to input work shift and volunteer data which will return our schedule. This last goal is secondary to the first two because SSC does not have any events to run until this fall. This paper shall work through a rudimentary program but with the full algorithm and solution.

1.1 Specific Challenges

Initially we considered simplifying the problem with so many assumptions and constraints that resulted in schedules that were very similar to current ones done by hand and left little to be optimized or improved.

Even though we have volunteer schedules from real events from around the United States, all schedules are final products after last minute adjustments and sometimes shift swaps engineered by the volunteers themselves. This poses a problem since we don't have each event's potential volunteer pool and for many cases, we don't know what the initial schedule was. We are also missing data concerning the times that its volunteers would prefer to have worked or not worked.

2 Data Collection

We were able to examine volunteer schedules from events in North America that ranged from ones with fewer than 100 participants to the largest event in North America, Lindy Focus, that spans 5 days and has over 1000 participants. We were able to make the following observations.

- All shifts were generally about an hour long.
- Odd lengthed shifts were usually set up and tear down.
- For any volunteer, it was uncommon for them to work more than 4 hours in a day and extremely rare for them to work more than 6 hours in a day.
- The majority of volunteering jobs do not require any special skill to accomplish.
- The longest job we found was 12 hours which consisted of 12 one hour shifts covered evenly by 3 people working 4 hours each.

Through some informal conversations, we were able to learn that many people who volunteer at events really would like to know their schedules sooner and be able to guarantee that they won't be working certain times or certain shifts. Though we were already planning to implement this feature, it was good to know that it was actually desired in the community.

Checking people off at registration and making sure non-paying people don't get into classes is not particularly difficult. At large events such as Lindy Focus, many volunteers are screened before hand so they already have a predetermined role.

Most of these events are hosted in a single hotel so traveling from one room to the next takes less than 5 minutes. We decided that factoring in this time would not be valuable and instead decided to make the program more likely to match a volunteer who is already matched to the first shift in a job to the second shift in that same job (assuming all requirements of that job are met). The same is true for breaks. We leave it to the organizers to determine breaks and whether or not individual shifts have, say, a 10 minute break at each end.

We assume disjoint days because no one we talked to were aware of any Lindy Hop event where there are activities between 5 am and 8 am.

2.1 Simplifying Assumptions

From previous volunteer schedules and discussions with event planners, were able to make the following initial simplifying assumptions:

- All volunteers can do all jobs. This assumption is also equivalent to assuming that all volunteers are preassigned a job type.
- Each job consists of 1 - 12 shifts.
- Each volunteer can get from shift to shift in very little time.
- There are no breaks.
- All shifts for the day can be contained in a 24 hour period.
- Each volunteer has a maximum daily work time of X hours where X is determined by the event planners.

3 Formulation

Our problem can be approached multiple ways. Chromatic numbers on graphs of jobs, integer programming on time slots, and the bin packing problem are all viable approaches. Our problem is specifically a variation on the *interval partitioning* problem which is a specific case of the bin packing problem.

The interval partitioning problem asks "Given a set of intervals $\{(s_1, f_1), (s_2, f_2), \ldots, (s_n, f_n)\}$, what is the minimum number of containers needed to store all the intervals such that in each container, no interval overlaps with another? There is a common greedy algorithm for this problem that runs in $O(n \log n)$ [CITATION NEEDED]. For this problem, the lower bound on number of containers needed is the largest number of overlapping intervals, the aforementioned greedy algorithm can be proven to find this optimal lower-bound solution. We would like to store all the work shifts in volunteers but certain shifts will not "fit" for reasons other than being too large. For this reason, we approached our problem from the bin packing problem.

The general bin packing problem asks "Given some numbers $a_1, \ldots, a_n$, what is the least number of bins with capacity c needed to store all numbers a_i?" An immediate lower bound to this problem is $\left(\sum_{i=1}^{n} a_i\right)/c$.

Translating to our problem, volunteers are the "bins" and the "items" we fill them with are the shifts that we assign.

The bin packing problem is NP hard[citation] but by Korf, we can find an exact solution on the minimum number of bins required fairly quickly for small input sizes. The Korf algorithm proved to be too memory intensive when applied to our problem. Korf's algorithm uses a branch and bound approach, pruning less-than-optimal partial solutions along the way. But in order to achieve this, all un-dominated completions of a certain bin needed to be computed. Because all of our shifts are of length 1, the number of completions we had to keep track of grew very quickly, and these shifts being Python objects took up quite a bit more space than your typical 4 byte integer. Thus an exact algorithm is not practical in our case, and we resort to a near optimal algorithm, the Best First Decreasing (BFD) algorithm. The BFD algorithm is an approximation algorithm that runs in polynomial time. It does not guarantee an optimal solution but does guarantee a solution that is no more than 11/9 * OPT + 4 bins more than OPT (the actual optimal number of bins). Also, according to Korf, this algorithm produced the optimal solution 94.8% of the time on average. For our problem, this approximation is plenty accurate as well as when we consider the fact that the truly "optimal" algorithm could be suboptimal anyways when taking into account our other constraints.

4 Solution

Our actual approach at producing an optimal schedule uses a mix of the BFD and randomization to look for the best solution. It works as follows: We first find a rough lower bound by taking the MAX(overlapping lower bound, capacity lower bound). The overlapping lower bound tells us that if there are n shifts scheduled all at the same time, then we must have at least n volunteers, since no volunteer can be at more than one shift at one time. The largest of these numbers gives us the first lower bound. However, this is not enough, as we could have non-overlapping shifts that cause the total number of shift hours to be greater than the total number of hours that are available from all volunteers. Thus, for our second lower bound, we divide the total number of shift-hours by the average number of available hours a volunteer has. Note that neither of these lower bounds take into account the number of particular volunteers can actually work any particular shift. These minimums give us a practical lower bound which we use as a benchmark for determining how good our solution is.

Once we have a lower bound, we begin by keeping running our list of volunteers and shifts through the BFD algorithm, which will produce either a complete schedule or a partial schedule with the specific un-assignable shifts highlighted.

The BFD algorithm works by taking in our list of shifts, sorted in decreasing order of length, and our list of volunteers, ordered in decreasing order of "fullness". It then attempts to assign the first shift in the list to the first volunteer. If the volunteer can take this shift, we assign it to that volunteer, otherwise we move on to the next volunteer in our list. Once the job has been assigned, we re-sort the list of volunteers before considering the next job. This is the "best" part of the Best Decreasing Fit algorithm. We want to ensure that we always consider the volunteer with the most assigned shifts first. Now, if the a shift goes through all volunteers in our list unassigned, it goes into an "unassigned shifts" list, which we will then present afterwards with a partial schedule as incomplete.

We have also developed a notion of a job with shifts. In essence, each shift is associated with a job and we would prefer to keep the same volunteer on shifts of the same type, rather than shuffling around volunteers from shift to shift for no good reason (an optimal solution will not guarantee this consistency). We introduce a job-type weight of a volunteer which is calculated by summing up the squares of the counts of shifts of the same type. For example, if volunteer A has shifts 1.1, 2.1, 2.2, her weight will be 1^2+2^2. x in $x.y$ indicates the job and y indicates the shift number. Counting the total weight of all volunteers who have been assigned shifts can help us compare solutions and find ones that better match this preferred shift assignment scheme.

Our algorithm is an any-time algorithm. If the volunteer set and shift set will not produce a feasible solution, then it will return a partial solution immediately. However, if one solution is found, we continue by running the BFD algorithm on various random permutations of our volunteer list. This will clearly produce a different schedule, as the BFD algorithm assigns shifts in a (selective) first come first served basis. On each iteration we keep track of the number of volunteers this solution assigned as well as the total weight of the assignment. If the number of volunteers needed on any given iteration is smaller than before, we update our new solution to reflect this assignment. If that number is the same as the smallest we have found so far, but the total weight of volunteers exceeds the max weight we have found, then we also update. In other words, we want to maximize the total weight but not at the cost of adding extra volunteers.

The user can choose to terminate the running code at any time. The longer the code is left running, the better the solution might get.

5 Results

As sample data, we are currently generating a list of 60 volunteers, each of which are only able to work a random but continuous 2-6 hour chunk a day. We then create a random list of shifts. We do this by starting with a base of 15 jobs, each of which can have 1-6 hour long shifts, and each job can start at a random hour of the day. Our algorithm consistently finds solutions that are within 2-3 volunteers of the lower bound. Many times it will actually find the optimal solution. Attached in appendix B are three sample runs, each of which were terminated after 30 seconds of running.

We are fairly satisfied with the results of this algorithm. Inspecting the schedules ourselves, we have determined that there might be room for improvement in the sense of shuffling around shifts to make things more consistent. However, this should provide our friends over at Savoy Swing Club a good starting point with their scheduling pain points. They will now be able to produce a fairly optimal schedule in under a minute with which they can then further "hand-optimize" to suit their needs.

6 Future Work

We have so far reasonably solved the algorithmic problem of assigning volunteers to positions based on availability. We plan to proceed with the program and build a fully functional web app that will allow event organizers to use different approaches to solve the problem and to manually adjust any schedule that we conjure up.

SSC is unfortunately not hosting any weekend events until this fall. However, we also plan to follow up with them in the fall and attempt to actually implement our system for choosing volunteers.

7 Acknowledgments

We would like to thank our community partners over at Savoy Swing Club for working closely with us on the problem. A special thanks for Andrew Rogers who spoke to many dancers in the community on behalf of us and Ben White for providing volunteer schedules from national events.

We would also like to thank Professor Sara Billey for meeting with us to suggest ways on how we could model the problem and which general problems our particular one resembled.

8 Appendix A

```
#!/usr/bin/python
#
# Work objects
#


class Volunteer:
    """
    Represents a single volunteer
    String name - Name of the volunteer
    Int capacity - How many hours this volunteer has available
    Int current_capacity - How many available hours the volunteer current has
    [Job] jobs - Current list of jobs assigned to volunteer
    Boolean is_used - True if the volunteer's job list is not empty
    [Int] job_id_count - keeps count of how many shifts of each job ID a volunteer has
    """

    def __init__(self, a_name="", a_capacity=4):
        self.name = a_name
        self.capacity = a_capacity
        self.current_capacity = 0
        self.jobs = []
        self.is_used = False
        self.job_id_count = [0 for _ in xrange(15)]

    def __cmp__(self, other):
        return -cmp(self.current_capacity, other.current_capacity)

    def __repr__(self):
        return self.name

    def add_job(self, a_job):
        """Adds a job and updates state only if job is not a pseudo job"""
        self.jobs.append(a_job)
        if not a_job.name.startswith("UNAVAILABLE"):
            self.current_capacity += a_job.length
            self.job_id_count[a_job.id] += 1
            self.is_used = True

    def can_take_job(self, a_job):
        """Will return true as long as volunteer has space and no conflicting jobs"""
        for current_job in self.jobs:
            if a_job.conflicts_with(current_job):
                return False

        if (self.current_capacity + a_job.length) > self.capacity:
            return False

        return True

    def get_weight(self):
        return sum([x**2 for x in self.job_id_count])

    def print_schedule(self):
        self.jobs = sorted(self.jobs, key=lambda x: x.start)
        print "Name: %s :: Hours %s" % (self.name, self.capacity)
        for j in self.jobs:
            print "\t%s :: %s - %s" % (j.name, j.start, j.end)

    def clear_all(self):
        self.jobs = filter(lambda x: x.name.startswith('UNAVAILABLE'), self.jobs)
        self.current_capacity = 0
        self.is_used = False
        self.job_id_count = [0 for _ in xrange(15)]


class JobShift:
    """
    Represents a single shift
    String name - name of the shift
    Int start - shift's start time
    Int end - shift's end time
    [Int] interval - all hours covered by this shift
    Int length - total number of hours for this shift
    Int id - represents this shifts job ID
    """
    def __init__(self, a_name, a_time_interval, an_id):
        self.name = a_name
        self.start, self.end = a_time_interval
        self.interval = set(range(self.start, self.end+1))
        self.length = self.end - self.start
        self.id = an_id

    """Sort by start time then by job
        An earlier start time in a different job would conflict with the shift before it.
        Assume shift times include break and travel times so shifts are always directly back
            to back.
        was (self.length, other.length) (self.start, other.start)"""
    def __cmp__(self, other):
        return -cmp(self.length, other.length)
    def __repr__(self):
        return "%s :: %s - %s" % (self.name, self.start, self.end)

    def conflicts_with(self, another_job):
        return len(self.interval & another_job.interval) > 1
```

```
import sys
import random
import copy
import Work as SS


# Helper functions
def rand_time_interval(a_min, a_max, a_length):
    start = random.randint(a_min, a_max - a_length)
    return (start, start+a_length)


def show_schedule(volunteers):
    for v in volunteers:
        if v.is_used:
            v.print_schedule()


def make_random_volunteers(num_of_volunteers, capacity_range):
    V = []
    for x in xrange(1, num_of_volunteers+1):
        name = "Volunteer " + str(x)
        capacity = random.randint(capacity_range[0], capacity_range[1])
        volunteer = SS.Volunteer(name, capacity)
        available_start = random.randint(0, 24-capacity)
        volunteer.add_job(SS.JobShift("UNAVAILABLE", (0, available_start), -1))
        volunteer.add_job(SS.JobShift("UNAVAILABLE", (available_start+capacity, 24), -1))
        V.append(volunteer)
    return V


def make_random_jobs(num_of_jobs):
    J = []
    for x in xrange(1, num_of_jobs+1):
        id = x-1
        num_of_shifts = random.randint(1,6)
        start_time = random.randint(0, 21-num_of_shifts)
        for y in xrange(1, num_of_shifts+1):
            name = "Shift " + str(x) + "." + str(y)
            time_interval = (start_time+y-1, start_time+y)
            job = SS.JobShift(name, time_interval, id)
            J.append(job)
    return J


# BFD Algo: put largest item into bin with most stuff
def assign_jobs(jobs, volunteers):
    unassigned_jobs = []
    current_jobs = sorted(jobs)
    current_volunteers = volunteers

    while len(current_jobs) > 0:
        job = current_jobs.pop(0)
        current_volunteers = sorted(current_volunteers)
        job_assigned = False

        for volunteer in current_volunteers:
            if volunteer.can_take_job(job):
                volunteer.add_job(job)
                job_assigned = True
                break

        if not job_assigned:
            unassigned_jobs.append(job)

    return (current_volunteers, unassigned_jobs)


# Initialization of random volunteers and shifts
NUM_OF_JOBS = 15
NUM_OF_VOLUNTEERS = 60
volunteers = make_random_volunteers(NUM_OF_VOLUNTEERS, (2,6))
jobs = make_random_jobs(NUM_OF_JOBS)


# Compute the capacity based lower bound for volunteers
total_job_hours = sum([j.length for j in jobs])
avg_vol_capacity = sum([v.capacity for v in volunteers])/float(NUM_OF_VOLUNTEERS)
capacity_lower_bound = int((total_job_hours/avg_vol_capacity))


# Compute the overlapping intervals based lower bound for volunteers
overlap = dict.fromkeys(range(25), 0)
for j in jobs:
    for i in range(j.start, j.end):
        overlap[i] += 1

overlapping_lower_bound = max(overlap.values())

# Taking the larger of the two lower bounds to give us a realistic one
realistic_lower_bound = max(overlapping_lower_bound, capacity_lower_bound)

# debug info:
print "\n\nTotal number of volunteers available: %s" % len(volunteers)
print "Total number of shifts that need assignment: %s" % len(jobs)
print "Total available hours of all volunteers: %s" % sum([v.capacity for v in volunteers])
print "Total hours of all jobs: %s" % sum([j.length for j in jobs])
print "Capacity lower bound %s" % capacity_lower_bound
print "Overlap lower bound %s (shifts at the same time)" % overlapping_lower_bound
print "Lower bound on the number of volunteers needed is roughly: %s" %
    realistic_lower_bound
```

```
print "\n\nPlease terminate process whenever you see fit (Ctrl-c in terminal).\n"
print "Working...\n"

# Init tracking variables
current_best_schedule = []
current_best_weight = 0
min_volunteers_needed = NUM_OF_VOLUNTEERS


# Here we start randomizing the volunteer set to get different solutions
# we keep track of the smallest solution and store it
# we also keep track of the weight of each solution
# we will either get the current most optimal solution
# or an unfeasible, partial solution, with a list of anassignable jobs.
while (1):
    try:
        volunteers, unassigned_jobs = assign_jobs(jobs, volunteers)

        if len(unassigned_jobs) == 0:
            current_min = sum([1 for x in volunteers if x.is_used])
            total_weight = sum([v.get_weight() for v in volunteers])
            if current_min < min_volunteers_needed or \
              (current_min == min_volunteers_needed and total_weight > current_best_weight):
                min_volunteers_needed = current_min
                current_best_schedule = copy.deepcopy(volunteers)
                current_best_weight = total_weight
                print "New solution found with %s volunteers and weight %s" %
                    (min_volunteers_needed, current_best_weight)
        else:
            print "Not able to find feasible solution"
            print "Unassigned jobs: %s" % unassigned_jobs
            print "Using %s volunteers" % sum([1 for x in volunteers if x.is_used])
            print "Current best schedule: "
            show_schedule(volunteers)
            break

        map(lambda x: x.clear_all(), volunteers)
        random.shuffle(volunteers)

    except KeyboardInterrupt:
        print "Terminating...\n\n"
        if len(unassigned_jobs) == 0:
            print "Volunteer count: %s" % min_volunteers_needed
            print "Job-type weight: %s" % current_best_weight
            print "Schedule produced: "
            show_schedule(current_best_schedule)
            print "\n"
        sys.exit()
```

9 Appendix B

Volunteers are listed with the following format:

```
Name: [Volunteer name] :: [Total hours worked]
```

```
    [Shift name] :: [Shift Hours]
    [Shift name] :: [Shift Hours]
```

The shift name "UNAVAILABLE" are times that the volunteer is unavailable. Time and hours are on a 24 hour scale. ie. 13 is 1:00 pm.

```
Run # 1

Total number of volunteers available: 60
Total number of shifts that need assignment: 46
Total available hours of all volunteers: 234
Total hours of all jobs: 46
Capacity lower bound 11
Overlap lower bound 5 (shifts at the same time)
Lower bound on the number of volunteers needed is roughly: 11


Please terminate process whenever you see fit (Ctrl-c in terminal).

Working...

Not able to find feasible solution
Unassigned jobs: [Shift 7.1 :: 0 - 1]
Using 15 volunteers
Current best schedule:
Name: Volunteer 4 :: Hours 6
   UNAVAILABLE :: 0 - 10
   Shift 6.3 :: 10 - 11
   Shift 6.4 :: 11 - 12
   Shift 4.1 :: 12 - 13
   Shift 4.2 :: 13 - 14
   Shift 4.3 :: 14 - 15
   Shift 4.4 :: 15 - 16
   UNAVAILABLE :: 16 - 24
Name: Volunteer 15 :: Hours 6
   UNAVAILABLE :: 0 - 12
   Shift 5.1 :: 12 - 13
   Shift 5.2 :: 13 - 14
   Shift 5.3 :: 14 - 15
   Shift 5.4 :: 15 - 16
   Shift 5.5 :: 16 - 17
   Shift 11.2 :: 17 - 18
   UNAVAILABLE :: 18 - 24
Name: Volunteer 7 :: Hours 6
   UNAVAILABLE :: 0 - 3
   Shift 7.4 :: 3 - 4
   Shift 1.1 :: 4 - 5
   Shift 1.2 :: 5 - 6
   Shift 10.1 :: 7 - 8
   Shift 6.1 :: 8 - 9
   UNAVAILABLE :: 9 - 24
Name: Volunteer 10 :: Hours 6
   UNAVAILABLE :: 0 - 1
   Shift 7.2 :: 1 - 2
   Shift 7.3 :: 2 - 3
   Shift 14.1 :: 3 - 4
   Shift 7.5 :: 4 - 5
   Shift 8.1 :: 5 - 6
   UNAVAILABLE :: 7 - 24
Name: Volunteer 22 :: Hours 6
   UNAVAILABLE :: 0 - 10
   Shift 12.1 :: 10 - 11
   Shift 12.2 :: 11 - 12
   Shift 10.6 :: 12 - 13
   Shift 15.1 :: 14 - 15
   Shift 15.2 :: 15 - 16
   UNAVAILABLE :: 16 - 24
Name: Volunteer 6 :: Hours 3
   UNAVAILABLE :: 0 - 13
   Shift 9.2 :: 13 - 14
   Shift 9.3 :: 14 - 15
   Shift 9.4 :: 15 - 16
   UNAVAILABLE :: 16 - 24
Name: Volunteer 5 :: Hours 3
   UNAVAILABLE :: 0 - 9
   Shift 6.2 :: 9 - 10
   Shift 10.4 :: 10 - 11
   Shift 10.5 :: 11 - 12
   UNAVAILABLE :: 12 - 24
Name: Volunteer 2 :: Hours 4
   UNAVAILABLE :: 0 - 17
   Shift 2.1 :: 17 - 18
   Shift 3.1 :: 19 - 20
   UNAVAILABLE :: 21 - 24
Name: Volunteer 9 :: Hours 4
   UNAVAILABLE :: 0 - 6
   Shift 10.2 :: 8 - 9
   Shift 10.3 :: 9 - 10
   UNAVAILABLE :: 10 - 24
Name: Volunteer 8 :: Hours 5
   UNAVAILABLE :: 0 - 15
   Shift 11.1 :: 16 - 17
   Shift 13.2 :: 17 - 18
   UNAVAILABLE :: 20 - 24
Name: Volunteer 19 :: Hours 2
   UNAVAILABLE :: 0 - 11
   Shift 9.1 :: 12 - 13
   UNAVAILABLE :: 13 - 24
Name: Volunteer 3 :: Hours 3
   UNAVAILABLE :: 0 - 14
   Shift 9.5 :: 16 - 17
   UNAVAILABLE :: 17 - 24
Name: Volunteer 25 :: Hours 4
   UNAVAILABLE :: 0 - 11
   Shift 12.3 :: 12 - 13
   UNAVAILABLE :: 15 - 24
Name: Volunteer 12 :: Hours 2
   UNAVAILABLE :: 0 - 16
   Shift 13.1 :: 16 - 17
   Shift 15.4 :: 17 - 18
   UNAVAILABLE :: 18 - 24
Name: Volunteer 13 :: Hours 2
   UNAVAILABLE :: 0 - 16
   Shift 15.3 :: 16 - 17
   UNAVAILABLE :: 18 - 24

------
Run #2

Total number of volunteers available: 60
Total number of shifts that need assignment: 49
Total available hours of all volunteers: 219
Total hours of all jobs: 49
Capacity lower bound 13
Overlap lower bound 5 (shifts at the same time)
Lower bound on the number of volunteers needed is roughly: 13


Please terminate process whenever you see fit (Ctrl-c in terminal).

Working...

New solution found with 14 volunteers and weight 95
New solution found with 13 volunteers and weight 91
New solution found with 13 volunteers and weight 93
New solution found with 12 volunteers and weight 97
New solution found with 12 volunteers and weight 103
New solution found with 12 volunteers and weight 107
Terminating...


Volunteer count: 12
Job-type weight: 107
Schedule produced:
Name: Volunteer 55 :: Hours 6
   UNAVAILABLE :: 0 - 9
   Shift 9.3 :: 9 - 10
   Shift 10.1 :: 10 - 11
   Shift 9.5 :: 11 - 12
   Shift 12.1 :: 12 - 13
   Shift 12.2 :: 13 - 14
   Shift 12.3 :: 14 - 15
   UNAVAILABLE :: 15 - 24
Name: Volunteer 46 :: Hours 6
   UNAVAILABLE :: 0 - 3
   Shift 13.2 :: 3 - 4
   Shift 3.1 :: 4 - 5
   Shift 3.2 :: 5 - 6
   Shift 3.3 :: 6 - 7
   Shift 9.1 :: 7 - 8
   Shift 7.1 :: 8 - 9
   UNAVAILABLE :: 9 - 24
Name: Volunteer 24 :: Hours 6
   UNAVAILABLE :: 0 - 5
   Shift 13.4 :: 5 - 6
   Shift 13.5 :: 6 - 7
   Shift 11.1 :: 7 - 8
   Shift 9.2 :: 8 - 9
   Shift 7.2 :: 9 - 10
   Shift 9.4 :: 10 - 11
   UNAVAILABLE :: 11 - 24
Name: Volunteer 7 :: Hours 6
   UNAVAILABLE :: 0 - 1
   Shift 5.1 :: 1 - 2
   Shift 5.2 :: 2 - 3
   Shift 14.4 :: 3 - 4
   Shift 13.3 :: 4 - 5
   Shift 8.1 :: 5 - 6
   Shift 8.2 :: 6 - 7
   UNAVAILABLE :: 7 - 24
Name: Volunteer 9 :: Hours 5
   UNAVAILABLE :: 0 - 12
   Shift 9.6 :: 12 - 13
   Shift 6.1 :: 13 - 14
   Shift 6.2 :: 14 - 15
   Shift 6.3 :: 15 - 16
   Shift 6.4 :: 16 - 17
   UNAVAILABLE :: 17 - 24
Name: Volunteer 56 :: Hours 4
   UNAVAILABLE :: 0 - 2
   Shift 1.1 :: 2 - 3
   Shift 1.2 :: 3 - 4
   Shift 1.3 :: 4 - 5
   Shift 1.4 :: 5 - 6
   UNAVAILABLE :: 6 - 24
Name: Volunteer 43 :: Hours 5
   UNAVAILABLE :: 0 - 0
   Shift 4.1 :: 0 - 1
   Shift 4.2 :: 1 - 2
   Shift 4.3 :: 2 - 3
   Shift 14.5 :: 4 - 5
   UNAVAILABLE :: 5 - 24
Name: Volunteer 53 :: Hours 4
   UNAVAILABLE :: 0 - 15
   Shift 2.1 :: 15 - 16
   Shift 2.2 :: 16 - 17
   Shift 6.5 :: 17 - 18
   UNAVAILABLE :: 19 - 24
Name: Volunteer 42 :: Hours 3
   UNAVAILABLE :: 0 - 0
   Shift 14.1 :: 0 - 1
   Shift 14.2 :: 1 - 2
   Shift 13.1 :: 2 - 3
   UNAVAILABLE :: 3 - 24
Name: Volunteer 29 :: Hours 3
   UNAVAILABLE :: 0 - 7
   Shift 13.6 :: 7 - 8
   Shift 11.2 :: 8 - 9
   UNAVAILABLE :: 10 - 24
Name: Volunteer 1 :: Hours 5
```

```
   UNAVAILABLE :: 0 - 1
   Shift 14.3 :: 2 - 3
   Shift 14.6 :: 5 - 6
   UNAVAILABLE :: 6 - 24
Name: Volunteer 44 :: Hours 2
   UNAVAILABLE :: 0 - 11
   Shift 10.2 :: 11 - 12
   Shift 15.1 :: 12 - 13
   UNAVAILABLE :: 13 - 24

------
Run #3


Total number of volunteers available: 60
Total number of shifts that need assignment: 53
Total available hours of all volunteers: 235
Total hours of all jobs: 53
Capacity lower bound 13
Overlap lower bound 5 (shifts at the same time)
Lower bound on the number of volunteers needed is roughly: 13


Please terminate process whenever you see fit (Ctrl-c in terminal).

Working...

New solution found with 17 volunteers and weight 103
New solution found with 17 volunteers and weight 109
New solution found with 16 volunteers and weight 93
New solution found with 15 volunteers and weight 105
New solution found with 15 volunteers and weight 127
New solution found with 14 volunteers and weight 115
New solution found with 13 volunteers and weight 115
New solution found with 13 volunteers and weight 131
New solution found with 13 volunteers and weight 161
Terminating...


Volunteer count: 13
Job-type weight: 161
Schedule produced:
Name: Volunteer 9 :: Hours 6
   UNAVAILABLE :: 0 - 14
   Shift 2.1 :: 14 - 15
   Shift 2.2 :: 15 - 16
   Shift 2.3 :: 16 - 17
   Shift 2.4 :: 17 - 18
   Shift 2.5 :: 18 - 19
   Shift 2.6 :: 19 - 20
   UNAVAILABLE :: 20 - 24
Name: Volunteer 46 :: Hours 6
   UNAVAILABLE :: 0 - 12
   Shift 3.1 :: 12 - 13
   Shift 3.2 :: 13 - 14
   Shift 1.1 :: 14 - 15
   Shift 1.2 :: 15 - 16
   Shift 1.3 :: 16 - 17
   Shift 1.4 :: 17 - 18
   UNAVAILABLE :: 18 - 24
Name: Volunteer 51 :: Hours 6
   UNAVAILABLE :: 0 - 4
   Shift 5.1 :: 4 - 5
   Shift 5.2 :: 5 - 6
   Shift 5.3 :: 6 - 7
   Shift 5.4 :: 7 - 8
   Shift 4.2 :: 8 - 9
   Shift 4.3 :: 9 - 10
   UNAVAILABLE :: 10 - 24
Name: Volunteer 21 :: Hours 6
   UNAVAILABLE :: 0 - 9
   Shift 7.2 :: 9 - 10
   Shift 8.2 :: 10 - 11
   Shift 8.3 :: 11 - 12
   Shift 8.4 :: 12 - 13
   Shift 8.5 :: 13 - 14
   Shift 10.1 :: 14 - 15
   UNAVAILABLE :: 15 - 24
Name: Volunteer 34 :: Hours 6
   UNAVAILABLE :: 0 - 3
   Shift 12.2 :: 3 - 4
   Shift 11.1 :: 4 - 5
   Shift 12.4 :: 5 - 6
   Shift 12.5 :: 6 - 7
   Shift 12.6 :: 7 - 8
   Shift 7.1 :: 8 - 9
   UNAVAILABLE :: 9 - 24
Name: Volunteer 18 :: Hours 5
   UNAVAILABLE :: 0 - 3
   Shift 9.1 :: 3 - 4
   Shift 9.2 :: 4 - 5
   Shift 9.3 :: 5 - 6
   Shift 9.4 :: 6 - 7
   Shift 4.1 :: 7 - 8
   UNAVAILABLE :: 8 - 24
Name: Volunteer 11 :: Hours 4
   UNAVAILABLE :: 0 - 12
   Shift 6.1 :: 12 - 13
   Shift 6.2 :: 13 - 14
   Shift 14.1 :: 14 - 15
   Shift 10.2 :: 15 - 16
   UNAVAILABLE :: 16 - 24
Name: Volunteer 28 :: Hours 3
   UNAVAILABLE :: 0 - 14
   Shift 3.3 :: 14 - 15
   Shift 3.4 :: 15 - 16
   Shift 3.5 :: 16 - 17
   UNAVAILABLE :: 17 - 24
Name: Volunteer 27 :: Hours 3
   UNAVAILABLE :: 0 - 9
   Shift 8.1 :: 9 - 10
   Shift 13.1 :: 10 - 11
   Shift 13.2 :: 11 - 12
   UNAVAILABLE :: 12 - 24
Name: Volunteer 12 :: Hours 4
   UNAVAILABLE :: 0 - 14
   Shift 14.2 :: 15 - 16
   Shift 10.3 :: 16 - 17
   Shift 10.4 :: 17 - 18
   UNAVAILABLE :: 18 - 24
Name: Volunteer 55 :: Hours 3
   UNAVAILABLE :: 0 - 2
   Shift 12.1 :: 2 - 3
   Shift 12.3 :: 4 - 5
   UNAVAILABLE :: 5 - 24
Name: Volunteer 56 :: Hours 3
   UNAVAILABLE :: 0 - 17
   Shift 3.6 :: 17 - 18
   UNAVAILABLE :: 20 - 24
Name: Volunteer 20 :: Hours 6
   UNAVAILABLE :: 0 - 11
   Shift 15.1 :: 13 - 14
   Shift 14.3 :: 16 - 17
   UNAVAILABLE :: 17 - 24
```

References

[1] Richard E. Korf A New Algorithm for Optimal Bin Packing AAAI-02 Proceedings. Copyright 2002, AAAI 2002