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 usually done by hand, but becomes harder with larger events and does not take into account many variables that could be considered such as volunteer time preferences and minimize the number of volunteers used. 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 volunteers. 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). Our focus is on these "full-featured" dance events where there are classes, evening dances, performances, and competitions. At each of these events, volunteers are required to make the weekend go smoothly.

Each volunteer has different jobs they are able (or allowed) 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 might mean full pass (access to all classes and evening dances) while fewer hours of volunteering might just mean a dance pass (only access to evening dances) or a partial full pass (eg. classes and evening dances on Saturday only). 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' time preferences and availability. 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. While at this point most volunteers relent to this fact, it is an improvement that we seek to provide.

We will create a schedule that is not too hectic and is easy for a supervisor to understand. For example, though it might minimize the number of volunteers if Volunteer A does the cashier job in Ballroom B, then runs across the street to to sales, back to Ballroom B for another shift, and finally ends the day driving instructors home, SSC would like to have the simplified to having Volunteer A just do 3 shifts as a cashier in Ballroom B unless the trade off is warranted.

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.

1.1 Specific Challenges

Initially we considered simplifying the problem with 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 we 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 make some observations about the structure of volunteer schedules by inspecting events in North America that ranged from ones with fewer than 100 people to the largest event in North America, Lindy Focus, that spanned 5 days and had over 1000 people to its name.

- All shifts were generally about an hour long.
- 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 in a day.

Through some surveys, we were able to learn that most 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.

We would like to note that each of these assumptions could also be valid conditions in a "real" problem. Most of the majority of volunteering jobs do not require any special skill to accomplish. Checking people off at registration and making sure non-paying people don't get into classes is not particularly difficult. On the flip side, at large events such as Lindy Focus, many volunteers are screened before hand so they already have a predetermined role.

Having each Job consist of 1-12 shifts is reasonable since that is more than the number of shifts any job has at Lindy Focus and that is the largest event we want to consider.

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.

The last assumption is actually available as input in the final program but for our test purposes, we limited daily work time to 4 hours because that also happened to be the most any one person worked in a day at Lindy Focus.

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 4 hours

3 Formulation

Our problem can be approached multiple ways. Considering 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 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 $\left(\sum_{i=1}^{n} a_i\right)/c$.

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 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.

Unfortunately, the problem is only similar to both of these. We would like to partition intervals into pre-existing containers that have limitations on what can be stored in them. We would like to store all the work shifts in volunteers but certain shifts will not "fit" for reasons other than being too large.

3.1 Adjustments

3.2 Correctness

4 Results

Based on our randomly generated data, our solution to the problem is typically off the optimal number of volunteers by no more than 2 volunteers. For example, if there are 40 man hours in a day to work and each volunteer can work no more than 5 hours, then we need at least 8 volunteers just to satisfy the man hours, not considering any potential conflicts.

INSERT GRAPHICAL REPRESENTATION

5 Future Work

We have so far been unable to finish the program part of our problem though we have (in our opinion) 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 on them in the fall and attempt to actually implement our system for choosing volunteers.

6 Appendix

```python
#!/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)] # this is a bad thing to do :-(

    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


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

    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


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) # This makes it a BFD instead of an
            FFD algo
        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)


NUM_OF_JOBS = 15
NUM_OF_VOLUNTEERS = 60
volunteers = make_random_volunteers(NUM_OF_VOLUNTEERS, (2,6))
jobs = make_random_jobs(NUM_OF_JOBS)

# estimating lower bound by dividing sum of shifts by average volunteer capacity
total_job_hours = sum([j.length for j in jobs])
avg_vol_capacity = sum([v.capacity for v in volunteers])/float(len(volunteers))
estimated_lower_bound = int((total_job_hours/avg_vol_capacity)+1)

current_best_schedule = []
min_volunteers_needed = 60

print "Estimated Lower Bound: %s..." % estimated_lower_bound

# Here we start randomizing the volunteer set to get different solutions
# we keep track of the smallest solution and store it
# we can choose to terminate before the optimal sol is found
# we will either get the current most optimal solution
# or an unfeasible, partial solution, with a list of anassignable jobs.
while(min_volunteers_needed > estimated_lower_bound):
    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])
           if current_min < min_volunteers_needed:
               min_volunteers_needed = current_min
               current_best_schedule = copy.deepcopy(volunteers)
               print "Current min volunteers needed: %s" % min_volunteers_needed

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

    except KeyboardInterrupt:
       print "Terminating...\n\n"
       if len(unassigned_jobs) == 0:
           print "Min # of volunteers found so far: %s" % min_volunteers_needed
           print "Schedule: "
           show_schedule(current_best_schedule)
           print "\n"
       else:
           print "Not able to find feasible solution"
           print "Unassigned jobs: %s" % unassigned_jobs
           print "Current best schedule: "
           show_schedule(volunteers)
       sys.exit()

print "Finished with optimal (%s) number of volunteers." % min_volunteers_needed
show_schedule(current_best_schedule)
```

References

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