Optimizing Volunteer Shifts for Dance Weekends

John Jeng, Samvel Stepanyan

January 22nd, 2015

Abstract

Yay

Contents

1 Introduction **3**

2 Computer Science Reference **3**

2.1 Introductory Notions 3
2.2 Basic Definitions 3
2.3 Priority Queues 4
2.4 Trees 4
2.5 Dijkstra 4

1 Introduction

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).

The format of dance weekends includes live bands, classe, performances, and competitions. All of these activities requires volunteers to make the weekend go smoothly. Each volunteer has a set of capabilities (sound, driving, door management, etc) and preferences for which hours (and possibly how many) 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.

Given hourly requirements for various jobs and a set of volunteers, SSC would like to minimize the total compensation payout while adhering as much as possible to the volunteers preferences. As a side note, SSC would like to pay out in passes as much as possible over cash since the realized expense is much lower than raw cash expenses.

2 Computer Science Reference

2.1 Introductory Notions

2.2 Basic Definitions

Definition 1. Big O notation:

Let f and g be functions defined on some subset of the real numbers. Then we say that $f(x) = O(g(x))$ iff there exists a positive real number C and a real number x_0 such that

$$|f(x)| \leq C|g(x)| \text{ for all } x \geq x_0$$

Definition 2. A *loop invariant* is a condition that is necessarily true at the very beginning and very end of each iteration in some loop.

2.3 Priority Queues

A priority queue is a part of a class of objects in computer science called abstract data types (ADT). Real implementations vary but ADTs are used to analyze space requirements and running time.

In general, a priority queue can be thought of as an emergency room. Higher priority data (objects, patients) gets dequeued (processed, treated) before those of a lower priority. Typically this means there is a least value that is set as the highest priority.

The most basic priority queue just has the following methods (equivically understood as functions, jobs, utilities, capabilities, or features):

1. Add(Item i): This tells the priority queue to "enqueue" Item i which is to instruct it to appropriately place Item i into the wait list based on its priority.

2. Extract_min(): This tells the priority queue to return the

This allows the program to store items based on their priorty and retrieve the highest priority one.

Running time for each operation varies on the implementation, but we will assume a fibonacci heap in our case. Thus Add and Update_priority will run in $O(1)$ time and Extract_min will run in $O(\log n)$ time where n is the number of items in the priority queue.

2.4 Trees

For these next two subsections, our starting novice may do well to read the introductory section on Graph Theory.

2.5 Dijkstra

In this section we introduce Dijkstra's a algorithm for shortest path on general graphs with non-negative edge weights as we will emulate it in our main result.

The idea behind Disjkstra's algorithm is to mark all nodes that aren't the source node to initiailly to have a distance (denoted $d[v]$) equal to infinity. Then, as we traverse the graph from the source node, we update the shortest path to each v until we have traversed the entire graph.

When implemented with a fibbonacci heap, Disjkstra's algorithm runs in $O(m + n \log n)$ time where m is the number of edges and n is the number of vertices. We will not prove its correctness or running time here as it can be found in any introductory text on algorithms.

Algorithm 1: Dijkstra's algorithm

```
1  dist[source] := 0
2  foreach vertex v in Graph do
3      if v ≠ source then
4          dist[v] := infinity
5          previous[v] := undefined
6      PQ.add(v, dist[v])
7  while PQ is not empty do
8      u := PQ.extract_min()
9      foreach neighbor v of u do
10         alt = dist[u] + length(u,v)
11         if alt < dist[v] then
12             dist[v] := alt
13             previous[v] := u
14             PQ.decrease_priority(v,alt)
```

References

[1] Greg N Federickson. Fast algorithms for shortest paths in planar graphs, with applications. SIAM Journal on Computing, 16(6):1004-1022, 1987

[2] Monika R. Henzinger, Philip Klein, Satish Rao, Sairam Subramanian. Faster Shortest-Path Algorithms for Planar Graphs Journal of Computer and System Sciences Article NO. SS971493 1997