CS 3505 Assignment 01

Landon Gilbert-Bland
Alex Clemmer

1 Design Purpose

Students were supposed to implement a solution to the notorious NPC (or NPH, depending on how you ask the question) scheduling problem in C++ using `g++ v4.*`. No data structures from `stl` were to be used, but the instructor explicitly allowed certain method calls (*e.g.* `tolower()` and various searches).

There was no constraint on the algorithm's time to complete the solution, which lead us to believe that the point of the assignment was not to create an amazing algorithm. As such, our primary goal was to just get the program to work.

The secondary goal was mainly to learn C++, including design philosophy (*e.g.*, templates vs Java's Generics), conventions (which method declarations go in header files as opposed to `.C` files), and traditional C++ sticking points (pointers, arrays, memory management, et al).

The third goal was to learn emacs and use subversion, which neither of us were keen on.

Our fourth major goal was to make the program at least reasonably efficient.

2 Implementation

Basic timeline:

We spent the first 2 or 3 hours planning and thinking about the algorithm. We spent about 3 hours going through the code provided by the instructor. We actually decided to adopt most of it, bar a few minor modifications, but we spent a lot of time just looking up things we didn't understand.

We spent around 3 combined hours over the course of that time debating which algorithm to use. The next 15-18 hours (per person) were spent battling C++ to get our program to work. We knew it would take a while, but we imagined that it would be more algorithm tweaking and less dealing with segfaults.

Where we succeeded:

Goal 3 was clearly accomplished. We learned emacs and used version control extensively. One of us tried to use a vim emulator, but couldn't get the hang of it and ended up just using emacs proper, and the other eventually gave up when it came down to the wire, but we both know it reasonably well now. We started off with subversion, but eventually switched to git-svn when we spent a lot of our time working on a computer with no internet access but still needed to commit regularly. Our reflections on each and the specifics are to follow in the next section.

Goal 2 was a half-success. We feel like we have learned a lot about C++, but we are not at all confident in the language. This is a work in progress, and we expect it will take at least the whole semester to complete. Both of us have C++ books that we like. On the positive side, we do feel like we have most of the conventions down, and the code is commented well enough.

Goal 1 was also accomplished, but not so clearly. The program works for big and small input and there are test cases enclosed. Unfortunately, the nature of the problem makes it difficult to tell just how well this was accomplished. We can't always get a 100% matchup, and it's hard to tell the difference. Fairly concentrated situations like the test data we were given tend to be mostly solvable, but as you get sparser and bigger this question gets a lot harder to answer.

We would like to give a detailed metric about how well we did, but we didn't have time to make this analysis. Instead we spent most of our time wrestling with C++. That said, we expect, in the end, that we did about as well as other groups, or maybe slightly worse than average in terms of both efficiency and correctness.

There is a double-edged sword here, though: our solution is not efficient (see "Where we failed"). The way we know our solution is correct is because we've tested a lot of sticky corner cases with smaller (more tractable documents). We've also run the test on a few large documents, some of which returned, and some of which we aborted.

Where we failed:

The only clear failure of the project was goal 4. We did not make this solution more efficient. It is not our opinion that we *can't* do this, but the fact of the matter is that we spent a lot more time figuring out C++ than we thought we would.

The other goal we were not as successful at was goal 2, but we don't really think it's realistic to pick up all of C++ in a week, so we're ok with kind of failing this one.

3 Project length

This project took each person 2-3 hours of planning, ∼3 hours of preparatory coding, and *at least* 15 hours of actual coding, most of which consisted of wrestling with C++ (as opposed to constructive work). This totals up to about ∼41 hours of total work, or about 20.5 hours per person.

4 Testing:

Knowing how correct our program is ends up being very difficult. The way we did this was to test various sticky cases using small, tractable documents. They are hand coded. The idea here was to test the mechanisms of the algorithm. We could have tested the program for huge input, and we did to some extent, but this only goes to show that our program works on a larger scale. We wanted to show that the specificities of the program are on track, which they were.

Actually, the best way to test software like this (*i.e.*, how John Regehr finds compiler bugs for his research) is to have lots and lots of random input for the program. We did not do this, but we would have *loved* to, if we weren't so busy figuring out C++. We will definitely try this at a later date.

5 Reflections:

There is a process to attacking large and intractable problems. We discovered in this assignment some aspects of that system.

In the first place, we became much more productive when we broke the problem down into several smaller problems. In this case, the problem of finding an efficient way to match several partners can be broken down into several questions: How can we compare to n unit vectors (which in this case represent the times each person is available)? Who should we compare first? What mechanism for comparison should be used?

Not all of these are independent, and so sometimes it's not enough to break it into several questions. Another thing we noticed is that we can get closer to an efficient solution by looking at the things that all solutions have in common.

This is called constraint propagation, and the in this project it's fairly subtle also (being a very difficult problem): we notice, for example, that if we're matching people into groups of 4, then for every person there should be 4 people who have 5 overlapping times. This sounds obvious, but if we have a list of all people who can meet for any given hour, then all we have to do is tally up who are "most available" during the hours of one person to get a rough estimate of who we can group together. There are lots more, but for brevity we'll omit them.

The other thing we noticed is that we are going to need to learn the language on our own. It is not enough to simply show up and work. Not only is it less efficient, it makes us more frustrated with each other.

We need to pipeline our version control and coordination. Currently when we operate contiguously, we overlap code and cause merging headaches. More communication and perhaps code review should fix this.

Another thing that was a sticking point was clear responsibilities. In the future, we will definitely do a better job of making sure we both know who is doing what, because we end up with a lot of overlapping code otherwise (and we were working next to each other for almost all of it).