A PROGRAMMING COMPETITION MODEL FOR SMALL LIVE CONTESTS [*]

Dr. William Confer
SUNY Polytechnic Institute
william.confer@sunyit.edu

ABSTRACT

The ACM International Collegiate Programming competition (ICPC) is one of the most recognized and attended live programming competitions. Its paradigm has been replicated in regional contests and numerous conferences, but the ICPC model offers unique challenges when implemented at small scales. This paper introduces a novel competition model developed to address the challenge of offering an exciting ICPC-like live competition experience to departmental students on a single small campus. Further, this new model may offer a compelling alternative for regional (or larger) competitions.

INTRODUCTION

Every year, the ACM organizes an international programming competition in two stages: a series of one or more regional qualifiers and the World Final. The best teams from each region go on to the World Final, but most teams only get the opportunity to compete at a regional event. Even so, an ICPC regional qualifier is an exciting event for students and team coaches.

For three years beginning in 2012, the Computer and Information Sciences (CS) department at the State University of New York Polytechnic Institute (SUNY Poly) has held a Spring programming competition for its students based on the ICPC. While students typically looked forward to these departmental competitions, the contests never captured the motivating energy of the ICPC, and for the last hour of each event, teams seemed to decrease their effort significantly, as if waiting for the clock to just run out. In 2015, the department restructured its competition model to improve the overall experience and replicate some of the energy of the ICPC at a small scale.

[*] Copyright © 2016 by the Consortium for Computing Sciences in Colleges. Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the CCSC copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Consortium for Computing Sciences in Colleges. To copy otherwise, or to republish, requires a fee and/or specific permission.

The remainder of this paper revisits the common aspects and rules of most ICPC regional events and examines how these translated to early departmental competitions at SUNY Poly. It will identify the challenges discovered in hosting these events and outline a new competition model designed to overcome those issues. Finally, the paper will provide an example problem from the 2015 departmental competition, report on the results of and lessons learned from that event, and outline plans for future competitions.

THE ICPC REGIONALS

Before a team can participate in the ICPC World Final, they must be the winning team in one of several regional contests that year. These regional competitions are broken down into sub-regional competitions, as well, and sometimes further. It's these low-level regional competitions most teams participate in. The operation, rules, and development environments vary a bit between contest locations, but they are all based on the published ICPC Regional Rules [1]. This section walks through those ICPC rules pertinent to the work described in this paper.

Teams may have up to three student members from the same college who must each be certified as eligible by the faculty advisor (often the coach and advisor are the same person). Essentially, competitors should be no older than 23 and started their college careers no more than four years ago. Teams cannot speak with anyone other than their teammates once the problem set is distributed. Development languages are typically restricted to C/C++ and Java, as in the World Final, and since 2010, nearly all I/O is done through stdin and stdout.

There are six problems or more distributed at each contest (sometimes as many as 12). The problems each have a short description, example input, and precisely formatted expected output. The problem sets are still heavily influenced by Verhoeff's guidelines [5] and typically include a spectrum of domains, most commonly: search, graph theoretic, geometric, dynamic programing, trivial, and non-standard [4]. Additionally, there are usually one or two noticeably easy and one or two noticeably difficult problems. Teams may request the judges clarify a problem or review its write-up for an error.

Teams submit their programs for judging to have them tested against the judges' secret test data once it passes the team-made tests. A correct program's output must be essentially identical to the judges' expected output on their secret test data. Incorrect submissions add a 20-minute penalty to the team's time; however, many regional events waive the penalty. Incorrect submissions are returned with a note from the judges: filename error, compilation error, run-time error, wrong answer, time limit exceeded, or presentation error. Teams may request the judges review the secret test files for error.

Final team ranking is determined, first, by the number of problems each team solved, and then by total time expended by the team, including penalties. The ICPC indicates this ranking method is designed to incentivize teams to solve problems completely, quickly, and one after another [1]. Criticisms of this method focus on its disregard for disciplined, best practice software engineering [2, 3].

SMALL SCALE ICPC

The ICPC model may work well for large regional contests with tens of teams from different schools, but repeatable issues emerge duplicating it at the small scale of single department competitions.

The SUNY Poly CS department serves both undergraduate and graduate constituencies, but ICPCs eligibility criteria prohibits most graduate students. Additionally, students in their first semesters of the program are unlikely to compete, realistically, because completely correct programs are required. The side effect of the eligibility requirements and judging standards effectively reduce the competitor pool to the upperclassmen, only a subset of whom chose to compete anyway. Another way to view this is well over half the department's students are unserved by these competitions. The few entry-level students that did participate generally seemed demoralized by upperclassmen completing the “easy” problems they are struggling to begin.

Producing a balanced problem set of 6-12 problems also poses risks. Only one or two faculty drive the departmental competition each year while the ICPC has the benefit of many times more minds at work on new problems. It is difficult for so few faculty to develop novel problems, useful test data, judges' solutions, and expected output for very many problems (error-free, no less) without reusing existing ICPC problems or dumbing down the problems in general. The large problem set is necessary in the ICPC, though, to ensure even the top teams are working the entire time.

The judging policy and ranking system also had unexpected effect when paired with the small number of teams and large problem set. They seemed to discourage average teams from exploring the problem set in code because only completely correct programs score. Most teams in the departmental competition worked on a single problem until they finally got past the judges or gave up and began working on another problem. Also, with only a few teams and a large balanced problem set, the problems chosen by the teams (excluding the easiest problem) did not overlap much, resulting in a feeling one student described as “competing in separate competitions”. In contrast, the ICPC mood is exciting and competitive regardless of a team's problem choices because many other teams will be solving the same problems, and each problem will be repeatedly solved again and again, motivating further solutions.

Additionally, a number of students requested the departmental judges accept programs in other languages; Python, Ruby, Perl, and Go were requested specifically. This would not be allowed at the ICPC World Final, so it was not allowed at the departmental competitions.

A SMALL COMPETITION MODEL

To address the challenges discovered using the ICPC model at a departmental scale, a new competition model was developed, suitable for small environments. This model was then used in a trial competition in the Spring of 2015.

First, the ICPC eligibility rules were replaced to ensure all departmental students could participate/enjoy the competition. This adds new challenges because, in order to motivate all constituencies, the same problem set must be compelling to graduate students and approachable to those who recently started their computer science education. Only students working in the field professionally were asked, in good faith, not to compete. They were, however, encouraged to participate and share results with the competitors.

The second and most substantive change was replacing the ICPC judging method. Instead of requiring completely correct solutions to score, it was decided that teams would earn better and better scores as new submissions for the same problem improved against the judging test data. This effectively eliminates the judging notes, “wrong answer” and “presentation error”, as any solution that has the correct filename, compiles, and doesn't exceed the time limit will score, although the score could have little or no value to a team if the solution has little or no value. Allowing scores to improve with better solutions also increases the competitive energy as new solutions can affect team rankings with greater frequency than in the ICPC model, which, in previous departmental contests, afforded only a handful of scoring events. Additionally, beginner students are encouraged because they are more informed about their programs' relative merit, even if they never produce a complete correct solution.

The ICPC ranking mechanism was then modified to account for the new judging method. In homage to the ICPC standard and the incentive to promote work on as many problems as possible, rank is first based on the number of problems for which a team solution has scored. This virtually guarantees teams will at least attempt and submit a solution to each problem. In the event of a tie, rank is determined by the sum of the team's rank for each problem scored; the lowest sum is the winner. Rank for a problem is determined first by best solution score, breaking ties by submission time of the solutions. If there is a tie at the sum-of-ranks level, the winner is chosen by the head judge.

To ensure the judging could be automated and objective, the problems are designed to have explicit scoring functions, which are given to the teams with the problems. This way, even if a team believes the scoring function could be improved, everyone is aware at the beginning of the contest precisely how the score is determined and can skew their solutions based on it. Further, the judging programs (including sources) are given to the team, eliminating questions about the scoring information. Finally, the judging test data is also given to teams in its entirety. If there is a suspicion of errors in the judging program or test data, teams have each so they can adapt their solutions to deal with it.

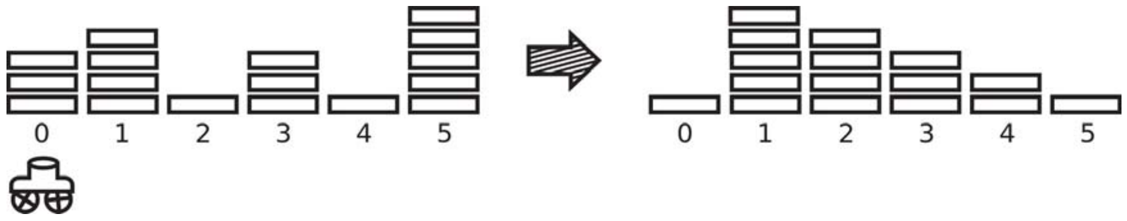
To mitigate the risk that all this openness of judging code and test data might have made the problems too easy to solve, the problems were designed to have scoring functions with competing objectives, and the test data is randomly generated with small to very large cases. The multi-objective scoring functions force the scoring landscape to be multi-modal, and the randomly generated test data guarantees no one, including the judges, know what the best possible score is for each problem. This keeps even the best teams working during the entire contest in the hope of acquiring a better score than the best known so far. Even the judges must wonder whether better solutions exist. Further, because there is no clear “end” to a problem like this, considerably fewer problems can be used to keep teams engaged over the competition period, relieving some of the challenge of building large problem sets. Fewer problems also reduce the “separate competitions” perception since all teams are effectively forced to actively work on the same problems. In the trial competition, for example, only three problems were given, and almost everyone remained engaged the entire time, in contrast to the experience using the standard ICPC model.

AN EXAMPLE PROBLEM

This section outlines one problem used in the trial departmental contest using the small competition model presented in this paper. Interesting student solutions to this problem will also be diSUNY Polyssed.

The Pluck-n-Drop

For this problem, teams are given some number of linearly organized stacks, each with some blocks on it. This represents the stacks of blocks in their starting configuration. The team controls a robot to move left or right one stack and pluck (pick up at most one) and drop (put back) blocks from one stack to another. The goal is to rearrange the blocks to match a target configuration as seen in the figure below.



The scoring function is calculated as the total number of operations the robot is told to perform plus the sum of the cubed difference in block counts from the end configuration to the target configuration for each stack (see formula below). The lower the number, the better the score. In some test cases, the initial and target configuration have a different number of total blocks, so the team must do its best to get the configuration to be similar enough with the fewest operations to score well.

$$Score = OperationCount + \sum_{i}^{M} \left| ActualBlocks_i - TargetBlocks_i \right|^3, \text{ for } M \text{ stacks}$$

The problem is straightforward enough, but scoring well is not trivial because of the competition between the objectives in the scoring function: first, making the stacks look the same as (or close to the same as) the target configuration and, second, avoiding approaches with too many operations since each move, pluck, or drop adds to the cost of the solution.

The best solutions to this problem, at least those with the lowest total cost, may not even match the target configuration. Consider the stacks of the target configuration (right) in the figure above. Imagine a program has duplicated the target configuration with the exceptions that stack 0 has one too few blocks, stack 5 has one too many blocks, and the robot is located under stack 5 with no blocks plucked. Regardless of the score up to this point, correcting stacks 0 and 5 can only damage the score. Correcting the stacks will require a pluck, five left moves, and a drop, increasing the current score/cost by seven.

If the program terminates instead of correcting the stacks, the damage from the incorrect stacks is only two because both stacks are off by one block, and 2 * (1^3) = 2.

Student Solutions

During the trial competition, every team submitted multiple solutions to the Pluck-n-Drop problem. While every solution is worth mentioning, this section will focus on the unique approaches used by teams that were enabled by this small competition model.

In at least two cases, teams cleverly duplicated the scoring code from the judging program into their own. In one of those cases, the team also copied the judging program's test data input code, increasing the time they had available to explore solving the problem. In past competitions, some teams or individuals could get stalled just dealing with basic data I/O.

One team, unable to mechanize some edge cases of their “awesome” solution, detected the caveat condition and performed a few random (but valid) operations to escape it. Ultimately, that team did not discover the best solution of the day, but that little bit of randomness got them beyond their coding obstacle and into a better scoring position. This could never be done in the ICPC because solutions are either perfect or incorrect.

One team discovered a deficiency in the judging program and scoring function. The team realized that while the scoring function was evaluating the differences in the end and target configurations, whether the robot was holding a plucked block was not considered. The team leveraged this to pluck a block from a nearby stack with too many blocks just before terminating to reduce the configuration difference. Because the block differences were cubed, this single operation could significantly reduce the difference cost of that stack, incurring only the smallest operational cost.

Another team, consisting of two beginning students, opened the test data files to look at the smallest test instance, which had only five stacks. They developed an operation sequence specific to that test and wrote the equivalent of a "Hello, world" program that blindly presented that solution regardless of the test data sent to their program. While the overall score of that program was less than outstanding, it actually held the best score for that specific test for about an hour. It clearly focused the students' minds on the problem and potential solutions, and the modified judging and ranking methods kept them going even though their solutions were never perfect.

FUTURE WORK AND CONCLUSION

When SUNY Poly hosted a single departmental programing competition based on the ICPC model, a number of issues emerged related to this model when used at such a small scale. The new small competition model presented in this paper overcame these issues and was used in a successful trial competition in the Spring of 2015. This new model engaged a much larger competitor population from the department, enabled new approaches to competition problem solving, and kept students engaged in a smaller set of problems for a longer stretch of time, reducing the task of problem set generation at once.

In the Spring 2016 semester, SUNY Poly is planning to host another programming competition with this new model. This time, however, the competition will involve more than one university so more data specific to the submission stream can be collected and evaluated, and the model can be explored in slightly larger deployments.

REFERENCES

[1] ACM International Collegiate Programming Contest, ICPC regional rules for 2015, 2015, https://icpc.baylor.edu/regionals/rules, retrieved November 2015.

[2] Adrianoff, S., Hunkins, D., Levine, D., Adding objects to the traditional ACM programming contest, *Proceedings of the 35th SIGCSE technical symposium on computer science education (SIGCSE '04)*, 105-109,2004.

[3] Bowring, J., A new paradigm for programming competitions, *Proceedings of the 39th SIGCSE technical symposium on Computer science education (SIGCSE '08)*, 87-91, 2008.

[4] Manzoor, S., Common mistakes in online and real-time contests, *Crossroads*, 14, (4), 10-16, 2008.

[5] Verhoeff, T., Guidelines for producing a programming-contest problem set, 1990, http://www.win.tue.nl/~wstomv/publications/guidelines.pdf, retrieved November 2015.