Using Topcoder in Introductory Data Structures and Algorithms

Adam Disney and James S. Plank

The Journal of Computing Sciences in Colleges

Published by the Consortium for Computing Sciences in Colleges

Volume 33, Number 2, December, 2017, pages 268-274.

The online home for this paper may be found at:
http://web.eecs.utk.edu/~plank/plank/papers/CCSC-2017.html

USING TOPCODER IN INTRODUCTORY DATA STRUCTURES AND ALGORITHMS

Adam Disney, James S. Plank
Electrical Engineering and Computer Science Department
University of Tennessee
Knoxville, TN 37996
865-974-3461
adisney1@vols.utk.edu, jplank@utk.edu

ABSTRACT

Beginning computer science students often struggle with the motivation behind different data structures and algorithms. It's helpful to have examples and practice problems that are not simply descriptions of singular algorithms. Some examples include problems that require more than one data structure and/or algorithm, problems that require modifications to standard data structures and/or algorithms, and problems that require attention to efficiency both in time and space. The Topcoder competitive coding platform can be an excellent resource to provide these examples and can be used in several ways in the classroom. In this paper, we detail our experience with Topcoder in several classes and from several perspectives.

INTRODUCTION

Topcoder is a computer programming competition platform launched in 2001 [1]. Competitions are held roughly three times a month where participants are given 75 minutes to solve three algorithm puzzles, each more difficult than the last. The programming language is the choice of the competitor, with C, C++, Java, Python and C# being supported. Competitors are given points based on how quickly they submit their solution, but receive no points if their solution fails any of the hidden test cases. Additionally the program must meet memory and time constraints specified in the problem description.

After the coding phase there is a challenge phase. Competitors have a chance to look at the other competitors' code. If they think it is incorrect, they can submit a challenge test case. If the code fails the test case, the challenger will receive 50 bonus points and the other competitor will receive no points for that problem. If the code does succeed, the challenger will lose 25 points. This penalty prevents brute force challenging.

Competitors have a running profile with a rating based on the Elo rating system, similar to the one used for chess ratings and other competitive games. Based on their rating they are placed in one of two divisions. The two divisions have different problem sets during competition, and competitors only compete within their own division.

All of this competition takes place inside of a Java web applet, although there is a browser based version in development. Both provide a basic text editor, although the Java applet has a handy plug-in that reads from a file instead of the applet window. This allows a user to use any text editor, and the applet will read the

file as the user's solution. The applet also allows users to access all of the previous competition problems. Any user can open the contest as if it were the real contest, submit solutions and run the full system tests. It even scores the users just like the original competition, although it doesn't update their Elo ratings. Users can also access all of the other users' solutions to the problems. Sometimes this is helpful but often these competitors write code for speed of typing and not readability, so it can be difficult to decipher their solutions.

The Topcoder website has a searchable listing of all previous problems, so with two divisions each with three problems run roughly three times a month for over 15 years...that's a huge backlog of practice problems! The problems have categories such as Brute Force or Graph Theory to make it easier to find problems that focus on a specific topic. Additionally there are success statistics which can be an indicator of difficulty. For example, if only 10% of competitors correctly solved the problem, then it's probably much harder than one where 90% were correct.

TOPCODER IN UNIVERSITY CLASSES

In our university, we leverage Topcoder in many Computer Science classes, from the second semester Data Structures and Algorithms class, all the way to the 400 level. We use it to augment lectures, as live programming material for laboratory classes, and to help students with their presentation skills. In this paper, we describe the utility of Topcoder in all of these ways, highlighting some of the less obvious ways in which the programming competition can be used as a teaching aid.

New students often have trouble understanding the importance of data structures and algorithms given the typical dry surface explanation. Topcoder can provide interesting problems utilizing the concepts that make the topic a bit less dry and even sometimes present realistic use situations. The instructor can solve these problems live in class, involving the students as much as possible, which adds some kinesthetics to the lectures. Students witness the experienced instructor's thought process and the troubleshooting steps if the instructor makes a mistake.

A Lecture Problem With Multiple Solutions

Usually there's more than one way to solve a problem, and many Topcoder problems lend themselves to a variety of solutions. Problem solving is in some respects an art more than a science, and is one of the hardest things to teach, and for that reason, having simple, interesting problems with multiple solutions can be an excellent teaching aid. For example, the problem "ThreeTeleports" presents a problem where the player must make their way home on a two dimensional grid [2]. On this grid there are 3 teleporters that lead from some point to another. The player may move up, down, left, or right one space in one second and using a teleporter will move the player to the other end in 10 seconds. The player must return the minimum amount of time it will take to make it home.

This immediately screams shortest path algorithm but since there are only 8 positions on the grid of consequence (starting position, home, and the 6 teleporter locations) the problem is small enough to brute force enumerate. The starting position must be fixed as the first node on the path so the enumeration is only 7! or

5040 possible paths.

To illustrate, let there be a numbering of the nodes. 0 will be the starting node, 7 will be the home node, and the consecutive pairs between will represent the teleporter ends. In other words, 1 and 2 are the ends to the first teleporter, 3 and 4 are the ends to the second teleporter and 5 and 6 are the ends of the third teleporter. Now an enumerated path will simply be a permutation of these numbers.

An example path would be (0, 1, 2, 3, 4, 5, 6, 7). This means to take the shortest path from 0 to 1 then the shortest path from 1 to 2 and so on. There are some inefficiencies that the instructor can point out with this solution. For example the path (0, 7, 1, 2, 3, 4, 5, 6) ends immediately at the second node but the enumeration will go through all 6! choices that begin with 0 and 7, which ultimately evaluate to the same path. Additionally there are paths that do not make sense to take. For example, the path (0, 1, 5, 2, 4, 6, 3, 7) tags all of the teleporter ends without taking the teleport so obviously this can only make the path longer than directly moving to home from the starting point.

There's a solution to these inefficiencies. Instead of numbering the nodes, we number the teleports 0, 1, and 2. Now the enumeration will enumerate the power set of the teleports and for each set enumerate the permutations of it. For example, {0, 1} is in the power set and its permutations are (0, 1) and (1, 0). For each teleport in the permutation there is a choice to enter one of two sides. Suppose the sides of the teleports are labelled A and B then for the permutation (1, 0) the enumeration will try (A, A), (A, B), (B, A), and (B, B). All of this combined will describe the intermediate path taken between the starting point and home. This avoids the inefficient cases from the previous solution.

As a teaching aid, this problem allows the instructor to reinforce the enumeration of permutations for a quick solution; a nested power set / permutation enumeration which solves the problem more efficiently, and of course Dijkstra's shortest path algorithm, which scales much better than the enumerations. It also allows the instructor to walk through the pros and cons of each approach. The fact that the problem comes from a competition, and the fact that only 23% of the competitors solved this problem during the competition adds a separate motivational flavor to the class-room [3].

A Lecture Problem With A Non-Obvious Solution

Sometimes problems are solved in a not so obvious way. Consider the problem "Alarmed" as an example [4]. For this problem, there is a room protected by at most 50 sound alarms. Each alarm is specified by a location in the room and its sensitivity threshold. The alarm will sound if a noise is heard above its threshold. There is an attenuation function given so one can determine if a sound at a given distance from the alarm and a given noise amplitude, will trigger the alarm.

An intruder will enter the south side of the room and try to pass to the north side of the room. The program should return the maximum noise level the intruder can make without setting off any alarms while traversing the room.

A first observation to help solve the problem, is that for a given noise level, a detection radius can be determined for each alarm. Figure 1 shows the radii for the

correct answer to one of the example problems. If the intruder was any louder, then the intruder wouldn't be able to enter the room due to an alarm covering the south entrance. Furthermore it's clear to see that there is a path not covered by alarms from the south door to the north door. However, how this information maps to a solution isn't so obvious.

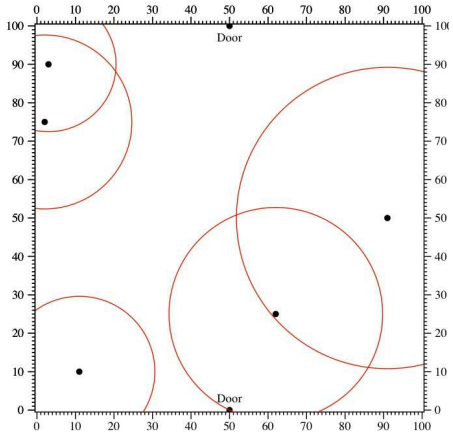

Figure 1: An example with maximum possible amplitude

Looking at another example helps to make the solution emerge. Figure 2 shows an example whose noise threshold is too high: There is no path from the south door to the north door that doesn't pass through an alarm's area. Another way to look at it is there's a path leading from the west wall to the east wall that's fully contained in the alarm area. What started as a geometrical problem has now turned into a graph problem!

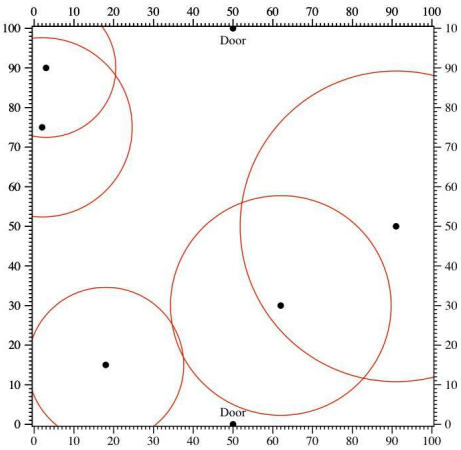

Figure 2: An example with no path from south to north

Specifically, for each potential noise threshold, we turn the problem into a graph. The graph contains a node for the west wall, the east wall, and each of the alarms. For the purposes of the solution, the west wall will include the section of the south and north walls that are west of the doors and the east wall will similarly

include the sections of the south and north walls that are east of the doors. There will be an edge between two nodes if their areas overlap. From this graph, use a depth first search (DFS) on the west wall and if the east wall is visited then there's no possible path.

Still that's just one piece of the solution. The figures and graphs only apply to a single test of a noise level. To solve the problem, the maximum possible noise level must be found. Referring back to Figure 1, it is known that there can't be overlap on the entrance so with only a maximum of 50 alarms it's easy to check the amplitude required for each alarm to detect a noise at the entrance. The lowest amplitude required of all the alarms is an upper bound to possible answers.

The lower bound is obviously 0 so with that there's now a range of possible noise levels. The final answer will be solved with a binary search over this range until the threshold of error allowed for the problem is met. Each noise level tested will need to construct the graph and call DFS. What seemed like a problem involving circles turned out to be solved with a classic graph algorithm and binary search. This particular problem is difficult, multifaceted, and non-obvious which makes it great for illustration.

Lab Material

Topcoder in lab sections are great practice for the student. It can be fun to introduce a bit of competition here. The Topcoder applet still gives the user a score as if they were doing the original competition, so students can compare their programming speeds. Starting lab with a warm up problem is usually a good idea. Lower division first level problems often have really easy solutions that require at most 10 lines of code. We give these to students first in lab, just to "warm up their fingers," and give them a self-esteem boost.

The next problem can be something related to the current lecture topic. The instructional approach here can vary from no hints to full algorithms and code, depending on the learning focus. No hints can be extremely hard for students since they have yet to develop the problem solving skills that usually come from experience. With the no hints approach, the problems will need to tend towards the easier side. Alternately the problem can be difficult with no hints during lab, but follow up afterwards either in the lab section or lecture section with the solution in detail. This gets the students thinking about the problem before the instructor solves them. It also allows some of the stronger students a chance to be recognized as well.

A step up from no-hints is to give the students a plain language description of an algorithm, perhaps with a picture or example. This removes the problem-solving component, but instead focuses on the students' abilities to render a solution into code. Surprisingly many students tend to struggle with this, because they need the practice of actually using the programming language and its libraries. This approach also allows the instructor to introduce concepts outside the scope of the class. For example, maybe the problem requires an understanding of combinatorics, but the course where they learn it isn't a prerequisite course. The instructions can briefly describe how it works in the context of the problem and point them to extra materials if they're inclined to learn it on their own. The instructor may even

consider giving the students the code for that section of the solution.

Presentations

Topcoder problems can also be used for computer science presentation practice in possibly a later course than Introductory Data Structures and Algorithms. We have used this in a senior-level course focused on advanced algorithms and programming. Each student is assigned a Topcoder problem from the upper division (typically the easiest problem), and given 5-8 minutes to describe the problem, illuminate it with examples, detail a solution and explore its time and space complexity. So that the presentation process does not become tedious, we limit the presentations to one or two for each class period, and spread them throughout the semester.

Presentation skills are often cited by employers as lacking in computer science students in general. This is in some respects natural, as most computer science students are introverts; however, it is also because most curricula do not stress presentation skills. Our experience with Topcoder presentations has revealed that students also not experienced in developing illuminating graphics, or even designing graphs that are readable when projected. While we don't devote an enormous amount of time to this area of our students' education, the Topcoder presentations are a helpful first step. Not only are the students forced to give presentations, but they also watch other students' presentations with a critical eye. This has helped the students with subsequent job interviews.

CONCLUSIONS

Students mostly have positive reactions to Topcoder sometimes begrudgingly due to the challenge of understanding the solution or finding that tiny detail that's causing their code to fail but ultimately express that it solidified their understanding. Many realize the value after the course when they've interviewed for a position where commonly these types of algorithmic puzzles are given. Obviously a few respond negatively usually with respect to the self practice in lab sections viewing Topcoder as a distraction from their lab assignment which is worth far more points. Overall we believe instructors should consider the benefits of a problem source such as Topcoder or similar algorithm competition platforms.

REFERENCES

[1] Topcoder, About Topcoder, https://www.topcoder.com/about-topcoder/, retrieved April 17, 2017

[2] Topcoder, Problem statement for ThreeTeleports, https://community.topcoder.com/stat?c=problem_statement&pm=11554, retrieved April 17, 2017

[3] Topcoder, Problem detail, https://community.topcoder.com/tc?module=ProblemDetail&rd=14544&pm=11554, retrieved April 17, 2017

[4] Topcoder, Problem statement for Alarmed, https://community.topcoder.com/stat?c=problem_statement&pm=7479&rd=10730, retrieved April 17, 2017