Programming Contest Strategy

Andrew Trotman and Chris Handley
andrew@cs.otago.ac.nz / chandley@cs.otago.ac.nz
Department of Computer Science, University of Otago, Dunedin, New Zealand

Abstract

Each year the ACM hosts a truly international programming contest – the International Collegiate Programming Contest (ICPC). Dating back to a contest held by Texas A&M University in 1970, this annual event, along with the associated regional contests, has grown to 5,606 teams from 1,733 universities in 84 countries (in the year 2006).

Despite the maturity of the event, and the number of competitors, there has never been a systematic examination of contest strategy. Herein several strategies are proposed and examined to determine whether a team can gain an advantage by choosing a good strategy; and, if so, then what that strategy should be.

We show that a team can gain an advantage by choosing a good strategy, but that there is no one best strategy. A team must choose between winning by number of solved problems and winning by points. Finding the optimal strategy to win by problems is shown to be NP-complete, while to win by points a team must solve problems in order from easiest to hardest.

1. Introduction

In 1970, the Alpha chapter of the UPE Computer Science Honor Society ran a programming contest at Texas A&M University. By 1977, this contest had grown to become annual and multi-tiered, with US regional contests followed by finals conducted at the ACM Computer Science Conference. Today it is a truly international event held each year [13].

The 2006 World Finals was attended by 83 teams, representing participation of over 5,606 teams from 1,733 universities in 84 countries competing in regional contests at 183 sites. Official contest estimates suggest that selection is from over 300,000 students each year [13].

A contest team consists of three students sharing one computer (with one keyboard and one mouse) working together to solve as many problems as possible from a "set" of about 9 previously unseen problems. Within a region all teams are tackling the same set of problems at the same time. Different regions usually use different problem sets, and are run at different times. Some regional competitions are distributed across multiple sites and run concurrently [2] using software such as PC^2 or Mooshak [14].

At the start of the contest teams are presented with the problems and the computer. They are given five hours to solve as many problems as they can. During this time each team must read, understand, and solve the problems. Working in any of C, C++, C#, or JAVA, teams electronically submit solutions (called runs) to judges for scrutiny.

Once the judges receive the run they compile and test it on data unseen by the contestants. The submission is then assigned a judgment of one of:

- **Compile-time error**: The run did not compile.
- **Contest rule violation**: The run required libraries forbidden by the rules (such as sockets).
- **Run-time error**: The run crashed.
- **Time-limit exceeded**: The run exceeded the time limit for the problem (probably an infinite loop, hang on reading input, or a naïve algorithm that did not complete in time).
- **Wrong answer**: The run produced the wrong answer.
- **Presentation error**: The run produced the right answer, but in the wrong format.
- **Accepted**: The run is a correct problem solution.

No further information is returned to the team. They are, in essence, working blind.

Contact between contestants and judges is limited to problem clarifications. Unless there truly is an error in the problem, these typically remain unanswered.

The team that solves the most problems in the time period wins. Ties often occur and are broken on sum of completion time for each problem, calculated as: the elapsed time (in minutes) from the beginning of the contest up to the point that the problem is successfully solved. Thus, if a team solves two problems, the first after 8 minutes, and the second 7 minutes later (15 minutes into the contest), the team's contest time (tie-break score) is 23 minutes. Additionally, each incorrect attempt at a later solved problem attracts a 20 minute tie-break penalty. Teams are therefore encouraged to submit as many problem solutions as possible, in as short a time as possible, without making any mistakes.

The programming contest offers a unique environment for research in several areas of computer science – in particular computer science education. We briefly discuss this before focusing specifically on team strategy. We are seeking a strategy that will increase a team's chance of winning should the team be matched against other teams of equal strength.

1.1. Computer Science Education Research

It is not clear why students choose to participate in contests. Should we be able to identify this, we would be able to attract additional contestants. One motivating factor could be the substantial prize money and (typically subsidized) overseas trip to the World Finals. Or it may be that competitions are fun [17]. Manne surveyed 23 participants of the University of Bergen qualifying rounds: 19 responded, of whom 13 claimed to enter for fun, and 17 responded they would enter the following year. Fitzgerald and Hines [9] note that some students become dissatisfied with programming contests, they note the low participation from female students (as did Manne [15]) and what they call minority students. Even so, the 2004 World Finals had an all female team.

There is a diverse set of programming and problem solving skills needed for the contest. Skiena and Revilla [19] present an excellent curriculum. They analyze previous contest problems and divide them into 13 categories including, amongst others, sorting, arithmetic and algebra, graph traversal, and dynamic programming. Each chapter of their book addresses one of these topics. Divided into two parts, a chapter presents a revision course in theoretical computer science of the given topic followed by a set of problems in the area.

We believe that Skiena and Revilla's work will have several effects: most noticeably the knowledge and understanding of participants in the chosen problem domains will increase considerably – the book is already being used as a training manual at several institutes. Secondly, those responsible for authoring the problems will deliberately devise problems outside the discussed domains. It is our opinion that this is already evident. At the 2004 ICPC World Finals, several problems requiring a simple brute-force solution were posed, an area not previously well represented. We believe that the problem authors and analyzers will enter a chase not dissimilar to that of virus writers and virus-protector software authors – each time a new strain is discovered and described a newer strain is developed. In this case, the strain is a programming contest problem domain.

The problem writers are at a clear disadvantage because they are constrained in the problems they can set. Problems must be tractable, by the contestants, during the contest, and potential solutions should be short (about 150 lines of code) [6]. Problems should neither be overly dependant on floating point computation, nor require exhaustive search. They must be Computer Science problems and not require knowledge from other disciplines. Of course, problems must read input and produce output.

One area of significant collaboration is web-hosted online judges. These web sites, typified by the Valladolid Online Judge [23], house many hundreds of problems from regional and World Finals dating back many years. Students are encouraged to download problem descriptions, solve the problem, and submit their source code (online) for electronic judging. Occasionally live online contests are held in order to simulate the environment of a real contest. As many as 80% of the World Finalists train using the Valladolid site [19].

This kind of resource is in the interests of not only the contestants, but also the judges and the problem authors. For the contestants there is a huge resource of problems to tackle in a submit-it-and-see environment. For the judges it removes any unfair advantage one team might have over others as a consequence of their institute housing a larger archive than any other. For the problem authors these archives serve as a resource ensuring the true originality of new problems. We foresee these sites expanding and becoming more sophisticated.

The competition is an environment in which aspects of Computer Science such as teamwork can be taught [15]. A team must extract the essence of the description, formulate it as a mathematical problem, and then apply robust Computer Science theory to solve it. An example of how this might be done is given by Shilov and Yi [18]. Old programming contest problems could be used to teach data structures and algorithms as is suggested by Szuecs [20]. Equally, the competition is an environment in which aspects of Computer Science such as object oriented design could be introduced [1].

1.2. Other Notable Contests

The annual ICPF functional programming contest gives participating teams of any size 72 hours to solve the one given problem. Teams may use any language of their choice (functional or not) and languages like OCaml and Haskell are frequently used. The 2005 contest specifically tested a team's ability to write adaptable code. Two weeks after the initial submission date the problem definition changed and teams got a further 24 hours to adapt their solutions to the new specification [12]. The 2005 contest was the 8th annual contest.

TopCoder [22] run a 90-minute contest every week. It attracts both developers and employers, using contests as a selection method for matching the two. TopCoder also offers a component development program in which developers compete to build components that are then made available.

At the Cyber Defense Exercise students from the US Service Academies develop a robust information system which is then attacked by other teams (for a period of 4 days) [7]. Competitions of this nature focus on a particular problem in the aim of both teaching domain specific principles and encouraging research in the domain.

It is not uncommon for a funding agency to offer prize money as an incentive to stimulate research into a difficult problem. The DARPA Grand Challenge [10] prize money was won after only 2 years and resulted in innovation in autonomous robotic vehicles and computer navigation systems. The ten million dollar Ansari X-Prize was won in October 2004 by SpaceShipOne for flying twice into space in a two week period [24]. Similarly, in 2002 Google ran a contest in which task was to write a program that "does something interesting" with 900,000 web pages [11], no doubt stimulating research into web information retrieval.

Time
- *ongoing*
- *event*
- *ongoing-event*

Audience
- *pre-tertiary*
- *tertiary*
- *post-tertiary*
- *open*

Location
- *local*
- *regional*
- *international*
- *locationless*

Purpose
- *special purpose*
- *general purpose*

Figure 1: Programming Contest Taxonomy

1.3. Competition Taxonomy

There are many different competitions with different purposes, targeted at different audiences, and with different prizes. Figure 1 shows a taxonomy which we discuss further in the remainder of this section.

1.3.1. Time

Some organizations are frequently and regularly running competitions. TopCoder, for example, runs a 90 minute competition every week. Frequent competitions are also held on the Valladolid web site. Some of these *ongoing* contests score participants based not only on single contests but also on an ongoing basis.

The ACM contest is held once a year with regional contests also held once a year. At such *event* contests teams come together in one place at one time and often participate in activities other than just the contest. The Java Challenge, for example, has been held several times at the ACM World Finals.

Still other contests are both ongoing and events, they are *ongoing-events*. The DARPA Grand Challenge, although an event, requires considerable ongoing activities by participating groups – in this case it took the teams two years and two annual events before a winner was declared. The ICPF functional programming contest is ongoing in so far as it takes longer than a day, but also an event in so far as it is run annually.

1.3.2. Audience

The entry criterion for many contests is often strictly enforced. The New Zealand Programming Contest has several categories including a *pre-tertiary* category designed to encourage high-school participation. The South East Asia Regional Computer Conference (SEARCC) also has a pre-tertiary contest.

The participation rules for the ACM contest are complex, but designed to ensure *tertiary* only participation. Such rules are necessary as the duration of an undergraduate degree vary from country to country.

Although we are not aware of any *post-tertiary* contests there are several *open* contests in which there is no audience restriction. The New Zealand Programming Contest has an open category, as does TopCoder.

1.3.3. Location

Local contests are often held by universities looking for the best teams to send to other contests. We define a local contest as a contest held within a city or smaller geographical area (such as a single institution).

Participants in *regional* contests are often required to travel to a given center to participate, or alternatively the contest is held at many different sites concurrently. The South Pacific Regional Finals of the ACM contest are, for example, held concurrently at multiple sites in both Australia and New Zealand.

The ACM World Finals are an *international* contest in which participants from all over the world are required to travel to a single destination to compete. The country in which this contest is held varies from year to year.

With the advent of online judging came the *locationless* contest. The only location requirement to enter such contests is an internet connection and being acclimated to the time zone in which the contest is run.

1.3.4. Purpose

Contests such as the Ansari X-Prize and the DARPA Grand Challenge have a *special purpose*. Both contests were designed to stimulate research. In our observation these special purpose contests tend to be ongoing or ongoing-events.

Other more *general purpose* contests are held for the pleasure of competing or to pit teams against each other to see which will outperform the others. The ACM International Collegiate Programming Contest (ICPC) is an example, and we focus the remainder of our investigation on this contest due to its prestige and its connection with tertiary computer science education.

2. Strategy of the ICPC

Skiena and Revilla [19] suggest that a team be made up of members with quite different duties. The *coder* has strong programming language skills and can type quickly. The *algorist* is a good problem solver and communicator. The *debugger* is good at debugging incorrect solutions on paper. As they point out, the duties might change during the course of the contest, especially at the request of the designated *leader*. They work together, solving either one or two problems at a time, coordinated by the *leader*.

Van Brackle [4] and Manzoor [16] suggest that team members should solve problems independently, only working together if necessary. In this way problems can be solved three at a time, increasing the throughput of the team.

Skiena and Revilla suggest that problems should be tackled in order of easiest to hardest (as do Van Brackle [4], and Manzoor [16]) with the caveat that should other teams be successful in solving a problem previously considered difficult, then perhaps the classification is incorrect. Manzoor suggests that if a team member is struggling with an easy problem then a different team member should redo the solution.

Ernst *et al.* [8] suggest that hard problems should be tackled early in the contest so that there is time to solve them, and easy problems must be solved quickly. This is to burn the candle at both ends; to solve easy and hard problems in parallel.

Each of the above suggestions was presented in the context of what worked for the respective authors. We choose a different approach: to propose a team strategy through analysis.

Imagine a competition in which all teams are equally strong and all teams solve the same problems in the same order and at the same time. All teams draw equal and first. What if one team chooses to solve three problems in parallel while the others work sequentially? The same outcome is not expected and does not occur. Increasing parallelism increases throughput.

What if the teams choose equal parallelism, but solve the problems in a different order? One would hope for a draw, but, as is shown below, this does not occur; some orders are better than others. Specifically, we identify the increasing complexity / diminished time problem that we describe next.

Throughout this investigation we make several assumptions. First, all team members are equal. Second, all teams are equal. These are reasonable assumptions for the top few teams at any contest. Third, once a team member starts on a problem they work continuously on that problem and only that problem until it is solved; problems cannot be transferred between team members. Fourth, there is no waiting time at the computer when a problem is ready for coding. Our experience with actual contests suggests these two assumptions are not unreasonable. Finally, a problem takes the same time to solve regardless of the number of people working on it. We believe the final assumption is acceptable, but questionable, an alternative could be to reduce the problem solution time be some factor for each additional team member working on it.

2.1. The Increasing Complexity / Diminished Time Problem

When solving in order of easiest to hardest, as the contest progresses the team is attempting to solve harder and harder problems (i.e. ones that take longer and longer to solve) whereas the time available to solve them is strictly decreasing. Eventually there comes a point at which there is not enough time to solve the next problem. This could occur one or even two hours before the end of the contest.

For example; should there be three problems in a 5 hour competition, the first of which will take 2 hours to solve, the second 3 hours to solve and the last 4 hours to solve; and should the team be concurrently solving two problems at a time; then a team solving the problems easy to hard, two at a time, will only solve two problems (the two easy ones). Meanwhile a team starting on the hardest and easiest problem concurrently will solve first the easiest, then the hardest, then the remaining problem. This is illustrated in Figure 2.

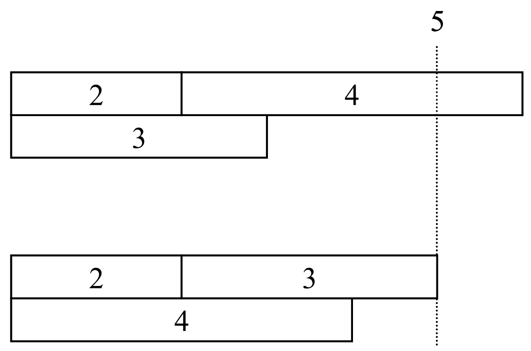

Figure 2: Problem solutions may not be maximized if tackled easiest to hardest (top).

2.1.1. Proof

This problem is the well understood scheduling concept of *makespan* minimization [5]. We include a proof for completeness.

Definition 1
Let Θ be a contest. Let $|\Theta|$ be the duration of the contest (in minutes).

Definition 2
Let X be a student participating in the contest. Let X' and X'' be team members with identical skills to those of X. Let χ be the set of X, X' and X''.

Definition 3
Let p_i be a soluble contest problem and $|p_i|$ be an estimate of the time (in minutes) it will take X to solve problem p_i in a competition environment. χ never submits incorrect solutions. Let $|p_i| \leq |\Theta|$. Let Π be a set of problems. Let $|\Pi|$ be the number of problems in the set Π.

Definition 4
The complexity of a problem p_i is $|p_i|$. If $|p_i| < |p_j|$ then p_i is said to be easier than p_j. If $|p_i| > |p_j|$ then p_i is said to be harder than p_j.

Our definition of complexity is not a traditional measure of difficulty, but of duration. For the purpose of programming contests the two are essentially equivalent. We also note that a problem which, from visual inspection, appears difficult may in fact have a low duration to solve. Manzoor [16] notes that problems with long descriptions are often easy and *vice versa*.

Theorem 1
In Θ the optimal number of problem solutions is not always achieved if the problems are tackled in order of easiest to hardest (in order of increasing complexity).

Proof
We examine three cases: solving one problem at a time, two at a time, and three at a time (the only possible cases with a team of three):

Sequential Problem Solving
The sequential case can be reformulated thus: What is the maximum number of problems, m, that can be selected from Π such that $\Sigma_{i \in m}(p_i) \leq |\Theta|$. This is a one dimensional box packing problem, which can be solved by queuing.

Let there be a problem, p_q, such that $|p_q| = |\Theta|$. By solving p_q, only one problem will be solved. However, by solving a different problem, p_r ($|p_r| < |p_q|$), it is possible to solve an additional problem, p_s, ($|p_s| + |p_r| \leq |\Theta|$). Let $|p_r|$ be the largest $|p_i|$ such that there exists a p_s. This has partitioned the contest,

Θ, into three parts, Θp_r, Θp_s, and e ($|\Theta p_r| + |\Theta p_s| + |e| = |\Theta|$, $e \geq 0$), where e is the lost time at the end of Θ. Now, add $|e|$ to $|\Theta p_r|$ and apply the argument recursively.

So, in the case of solving sequentially, the optimal number of problems is solved by addressing the problems in order of increasing complexity.

Solving Two Concurrently
A proof has already been given in Figure 2.

Solving Three Concurrently
Extending the proof for the case of two concurrent problems, we introduce a new problem, p_4, $|p_4|=240$. Now, solving three at a time in order of increasing $|p_i|$, first p_1, then p_2 then p_3 will be solved. No time remains for p_4. Solving in order of decreasing $|p_i|$, in other words p_4, p_3, p_2, p_1, all problems will be solved. The theorem holds in this case.

2.1.2. Consequences of the Proof

The proof demonstrates that the optimal number of problems may not be solved if tackled in order of easiest to hardest. Consequently, not all strategies are equal. Of two equal teams, the team that chooses the better strategy will win. To this end, we now investigate the optimal strategy.

3. Team Strategy

3.1. Proposed Teamwork Methods

Programming contest problem sets are deliberately designed so that an individual cannot, on their own, solve all the problems. In other words, it takes a team to win.

Before the 1991/1992 contest a team consisted of four members [3]. Today a team consists of three. This change was necessitated by the observation that some teams were not working as a team, but rather as a pair of couples. Before the contest began the decision would be made as to who was working in which couple. At the beginning of the contest the team would divide the problems into two subsets, one subset for each couple. The first couple to solve a problem on paper gained access to the keyboard while the remaining couple continued to solve problems on paper.. This is not in the spirit of the contest and consequently was outlawed in the only practical way: team size was reduced to three.

Today there are three ways a team can work: as a single entity, as a group of three disconnected individuals, or in a pair swapping manner.

3.1.1. Pure Teamwork

At any one time only one problem is being tackled by the team. The team is working purely as a team, three minds working as one. Tackling the problems can be done in a number of ways. Skiena and Revilla suggest a *coder*, *algorist*, and *debugger* split – however this is only one of many possible splits.

Only one person can be interacting with the computer at any given time – this person, we concede, should be called the *coder*. However, we have observed teams in which the *coder* is necessarily the *algorist*. Teams following this approach would argue, perhaps correctly, that the best person to code the solution is the person who solved it. The remaining two team members take on other duties. As each problem is solved, the duties might change; the *coder* might become the *debugger*, and so on.

A team might argue that most mistakes are made not in problem comprehension but in translation into programs. These teams are characterized by having two people in front of the computer at any given time – the *coder*, and the *observer*. The third team member is meanwhile designing test data in order to foil the implementation.

Perhaps the most visually obvious characteristic of teams employing pure teamwork is the three way discussion that occurs before a problem solution is attempted. Each team member reads the problem (at World Finals three copies are provided, but often not at regional contests). When, through discussion, the problem is solved, coding begins.

3.1.2. No-teamwork

In contrast to the sequential problem solving of pure team work, the no-teamwork approach advocates each individual solving problems entirely on their own. Parallel problem solving is far more efficient than sequential solution, however not necessarily three times so in this context.

Van Brackle [4] and Manzoor [16] advocate this strategy. It maximizes throughput, reduces communication and avoids programming style debates. Teams only work as a team when there are fewer problems left than team members, or when a problem requires the specific skills of more than one team member. Ernst *et al.* [8] give anecdotal evidence that this approach worked for their team.

Teams solving more than one problem concurrently must avoid a computer bottleneck. For example, imagine that there are three problems in the problem set. Each individual in the team takes a different problem. Each individual, on their own, can correctly solve their problem in 20 minutes, and correctly code their solution in 10 minutes. After 20 minutes there are three solved problems and a queue for the keyboard. At 30 minutes there is one correct solution, at 40 a second, and at 50 finally a third.

The bottleneck can disable teams. We agree with the comments of Skiena and Revilla [19], "teams which fight for the terminal go nowhere"; those of Cormack [19], "always use the keyboard, even if you are just typing in the shell of a program for reading input"; and those of Manzoor [16], "real-time debugging is the ultimate sin". Throughout this investigation the bottleneck is ignored, primarily because we believe that solving the problems takes longer than coding the solution.

3.1.3. Paired Methods

The obvious middle ground between pure teamwork and no-teamwork is to work in pairs. As three cannot work in pairs, this must be interpreted with respect to other methods. The paired method is, therefore, to solve two problems at a time.

Since only one team member can interact with the computer at any one time, the other two are forced to work together. One of this pair takes ownership of the problem; that team member is the *algorist* and *coder* for the problem. Before being allowed access to the keyboard the *algorist* must, to the satisfaction of the other member of the pair, explain the solution, then code it on paper. The remaining member of the pair, meanwhile, devises test cases for the solution. When the keyboard becomes available, the pairs necessarily shift. At the beginning of the contest the group can either work with pure teamwork or no-teamwork.

Ernst *et al.* [8] use a paired strategy called *think-tank*. The two best problem solvers read the problems while the third team member (the *programmer*) codes standard subroutines. The *think-tank* gives the easiest problem to the *programmer*, along with a solution. The *think-tank* decides how many problems are to be solved, allocates the easiest problems to the *programmer*, and then divides the remaining problems between themselves.

When tackling high complexity problems, shifting pair strategies can benefit from team member "cycling". That is, as the contest progresses, the pair working on the given problem changes. At one point *X* and *X'* might be working on the problem, at a later moment, *X'* and *X''*, then later still *X''* and *X*. Each time a substitution is made, the advances made on the problem must be communicated to the team member, thus increasing the overall understanding of the whole team.

3.2. Proposed Problem Order

Accepting the variety of working models, and team choice of how to organize itself, there remains the equally important question of what order the problems should be tackled. Skiena and Revilla [19], Van Brackle [4], and Manzoor [16] suggest working from easiest to hardest; above we prove this is only optimal if the team is solving problems strictly sequentially.

For the purpose of this investigation we assume there are 9 problems in the set ($|\Pi| = 9$), and a competition runs for 5 hours ($|\Theta|=300$), for no reason other than this being the case at the 2005 South Pacific Regional Finals in which the authors were involved organizationally. For clarity we choose to describe possible orders as if the team is employing pure teamwork; we believe adjusting to other teamwork methods is intuitive. For the purpose of this discussion we distinguish problems from each other using a unique identifier in the range 1-9.

3.2.1. Numeric Order

Upon seeing the problems the team attempts to solve the problems in numeric order. Some contests (e.g. South Pacific Regional Finals) openly rank the problems in order of complexity. Under this circumstance numeric order is the same as order of increasing complexity. At World Finals the problems are assigned in no particular order. Under this circumstance, numeric order is equivalent to random order.

Although solving in numeric order is viable if all problems are soluble, should the first problem be insoluble the team will immediately stall. This was evidenced at the 2004 World Finals; from the final scoreboard given after 240 minutes, none of the top 30 teams tackled the numerically first problem whereas one team that failed to solve any problems had tackled only the numerically first problem.

3.2.2. Order of Ease

At the beginning of the contest the team ranks the problems from easiest to hardest and then solves the easiest problem, progressing onto the next easiest only upon a correct solution of the first. This is the order advocated by Van Brackle [4], and Manzoor [16]. It is noted as "correct" by Salenieks and Naylor [17].

3.2.3. Reverse Order of Ease

The team tackles the problems in reverse order of complexity. When the contest starts the team ranks the problems hardest to easiest and tackles the hardest. Upon solving this, they move on to the remaining hardest unsolved problem.

Advantageously, as the contest progresses, the problems become easier while the remaining time decreases; there is no diminishing time / increasing complexity issue. Disadvantageously, the hardest problem may be insoluble in the time given. This is evidenced by the 2004 South Pacific Regional Finals where no team solved problem 9 even though the winning team, having solved 8 of the 9 problems in 156 minutes, worked continuously on the remaining problem for at least 144 minutes without solving it (they were granted a solution *post facto*).

A team working from the hard end must, before starting, decide how many problems it will solve during the competition, and work backwards from there – it is quite usual for the sum of completion times to be very much larger then the duration of the contest.

Assuming all teams can solve all problems; this strategy is prone to high tie-break times. The time to first submission is high. The time to second submission is higher, and so on. Since the scoring is based on total elapsed time to submission, the long time to the first submission is incrementally included in every submission.

3.2.4. Burn the Candle at Both Ends

In an attempt to decrease the high tie-break times, while at the same time not suffer from the diminishing time / increasing complexity problem, a team might choose to rank the problems in order of complexity then solve first the easiest, then the hardest, then remaining easiest, remaining hardest and so on.

This approach benefits from the no-teamwork and paired teamwork methods. One person (or pair) starts on the hardest problem; meanwhile the easier problems are being solved easiest to hardest. High tie-break scores are alleviated while time is available for solving complex problems. Ernst *et al.* [8] use this approach with two team members working from the hard end and one from the easy end.

3.2.5. Middle Out

Ranked from easiest to hardest, the problems are tackled middle out. In other words, if the problems are ranked 1 to 9, then problem 5 is tackled first. This approach manifests itself in two forms. We call these middle(5,4) and middle(5,6) where the two numbers represent the identifier of the first and second problem tackled.

3.2.6. Others

Solving in order of ease is equivalent to forming a queue of problems in order of increasing complexity, then solving in order of queue. Reverse order of ease is equivalent to queuing in order of decreasing complexity and solving in order of the queue. In fact, any problem order can be described in this way. The number of possible problem solution orders is, therefore, the number of unique queues that can be formed from the problems, i.e., the number of permutations of problems ($|\Pi|!$). For a contest of 9 problems there are 362,880 orders. When solving problems in parallel, some of these are equivalent ({1, 2, 3, 4} and {3, 2, 1, 4} are the same when three problems are solved in parallel). $|\Pi|!$ is, therefore, an upper bound on the number of distinct orders.

4. Analysis

A team must choose both a teamwork method and an order to tackle the problems (together a strategy). It must choose these so as to increase the number of problems solved, while decreasing the tie-break score.

Teamwork method and problem order are orthogonal. Any combination of one of each is a valid strategy. Above, 3 teamwork methods are given along with 5 problem order approaches. This provides 15 possible strategies. These are analyzed in light of different models for the complexity of the problems.

4.1. Problem Complexity Patterns

Judges deliberately look for problems of differing complexity [6], in part so that every team will solve at least one problem. Accepting this simplification, there are several possible complexity models for problem sets and we examine 4 of these.

Equal Complexity – All problems are of equal complexity. No one problem is either harder or easier than any other problem.

Linear Increasing Complexity – The first problem takes time $|p_i|$, the second twice that ($2 \times |p_i|$), the third three times ($3 \times |p_i|$) and so on.

Exponential Increasing Complexity – Each problem takes twice as long as the previous problem. The hardest problem takes half the contest, the second hardest taking a quarter, and so on.

Stratified Complexity – At some contests, including the New Zealand Programming Contest (which is not affiliated with ICPC), the problems are divided by complexity level. We assume three complexity levels: easy, intermediate, and hard; with three problems at each level. The time to solve each problem at a given level is constant whereas the time to solve a problem of the next highest complexity is twice that to solve a problem at the previous level. In competitions that do not strictly divide problems into levels, Manzoor [16] suggests that the team do this themselves anyway.

4.2. Simulation

All 15 strategies are tested against the four complexity patterns. For the purpose of the simulation several assumptions are made. Consistent throughout this investigation $|\Pi| = 9$ and $|\Theta| = 300$; there are 9 problems, the contest runs 300 minutes. A team, χ, consists of three identical members all with identical skills. No problem is ever submitted incorrectly so there are no penalties. Teams always make an accurate estimate of the time, $|p_i|$ to solve problem p_i. There is no keyboard queue. Problems must be solved to completion by one team member.

Problem	1	2	3	4	5	6	7	8	9
Equal	0:33:20	0:33:20	0:33:20	0:33:20	0:33:20	0:33:20	0:33:20	0:33:20	0:33:20
Linear	0:06:40	0:13:20	0:20:00	0:26:40	0:33:20	0:40:00	0:46:40	0:53:20	1:00:00
Exponential	0:00:35	0:01:10	0:02:20	0:04:41	0:09:22	0:18:45	0:37:30	1:15:00	2:30:00
Stratified	0:14:17	0:14:17	0:14:17	0:28:34	0:28:34	0:28:34	0:57:08	0:57:08	0:57:08

Table 1: Time (in hours, minutes and seconds) that the hypothetical team estimates each problem will take to solve. Problems are presented to the team in order from easiest (1) to hardest (9).

In Table 1, the time required (in hours, minutes and seconds) to solve each problem is given. The problems have been sorted in order of complexity with the easiest problem listed first. Each of the complexity patterns discussed above is represented as a single row of this table. Due to rounding errors, not all simulated contests are of equal length; this does not matter as there are no comparisons that assume they are.

	Ease		Reverse Ease		Candle		Middle(5,4)		Middle(5,6)	
	Time	Score	Time	Score	Time	Score	Time	Score	Time	Score
Equal	5:00:00	1500	5:00:00	1500	5:00:00	1500	5:00:00	1500	5:00:00	1500
Linear	5:00:00	1100	5:00:00	1900	5:00:00	1567	5:00:00	1433	5:00:00	1567
Exponential	4:59:23	593	4:59:23	2400	4:59:23	2158	4:59:23	835	4:59:23	1108
Stratified	4:59:57	1114	4:59:57	1885	4:59:57	1628	4:59:57	1371	4:59:57	1500

Table 2: Time to complete the problem set and tie-breaking score when problems are solved sequentially.

Presented in Table 2 are the times required to complete all problems (and the tie-break score) for each complexity model when problems are all tackled using the pure team work model. When the problems are of equal complexity, all strategies are equal, and there is no difference in completion time, or tie-breaker score. Although each strategy results in the competition completing at the same time, the tie-breaking score is always lowest when order of ease is used, and highest when reverse order of ease us used.

	Ease		Reverse Ease		Candle		Middle(5,4)		Middle(5,6)	
	Time	Score	Time	Score	Time	Score	Time	Score	Time	Score
Equal	2:46:40	833	2:46:40	833	2:46:40	833	2:46:40	833	2:46:40	833
Linear	2:46:40	633	2:33:20	1020	2:40:00	707	2:46:40	767	2:40:00	833
Exponential	3:19:47	397	2:30:00	1200	2:30:00	444	3:19:47	472	3:22:43	563
Stratified	2:51:24	643	2:37:07	1000	2:51:24	728	2:51:24	743	2:37:07	800

Table 3: Time to complete the problem set and tie-breaking score when two problems are solved concurrently.

Examining the case of two problems solved concurrently (Table 3) again all methods are identical if the problems are of equal complexity. To complete the most problems in the shortest period of time, reverse order of ease is best for all complexity models. However this is not a winning strategy; the winner must optimize both the number of problems solved and the tie-break score. The lowest tie-break score is achieved when problems are solved in order of ease. The team must choose a strategy, one of either order of ease or reverse order of ease, based on whether they wish to finish first, or reduce the tie-break score.

	Ease		Reverse Ease		Candle		Middle(5,4)		Middle(5,6)	
	Time	**Score**	**Time**	**Score**	**Time**	**Score**	**Time**	**Score**	**Time**	**Score**
Equal	1:40:00	600	1:40:00	600	1:40:00	600	1:40:00	600	1:40:00	600
Linear	2:00:00	480	1:46:40	720	1:53:20	507	2:00:00	560	2:06:40	593
Exponential	2:51:05	340	2:30:00	675	2:30:00	347	2:49:20	369	2:49:55	406
Stratified	1:39:59	471	1:39:59	728	1:54:16	514	1:54:16	528	1:54:16	571

Table 4: Time to complete the problem set and tie-breaking score when three problems are solved concurrently.

Tackling three problems at a time (Table 4) shows a pattern similar to two at a time. A team solves the most problems in the shortest time when working from hard to easy, but achieves the lowest tie-break score when working from easy to hard.

Comparing Tables 2, 3, and 4, in all cases, as the concurrency increases, both the time to solve all problems and the tie-break score decrease (as expected).

4.3. Discussion

The analysis shows that problems should be solved in parallel if possible. Solving problems in order of ease results in the lowest tie-break score, but takes longer to solve the problems. In a programming competition a winning team must optimize both the number of solved problems and the tie-break score to win.

Should the team be confident in completing all problems; having longer to do so, while at the same time reducing the tie-break, is a good strategy. Subconscious analysis of a problem will begin as soon as the problem is read. By reading the problem at the beginning of the contest, more time can be spent in subconscious analysis; while at the same time the conscious analysis time remains constant. At the South Pacific Regional Finals, and at World Finals, it is highly unusual for any team to solve all the problems in the allotted time.

To find a winning strategy it is necessary to examine two independent events: first is how tie-break score increases with solved problems, and second, the number of problems that can be solved in a given time period. These are presented for the case of three problems solved concurrently. The case of problems with equal complexity is not shown as all strategies are equal in this case.

As the number of problems solved increases, so too does the score. From Figure 3, at all times and irrespective of complexity model, solving problems easiest to hardest is (at worst) the best strategy. The tie-break score is always minimized. Should a tie occur; the team can maximize the chance of winning by adopting this strategy.

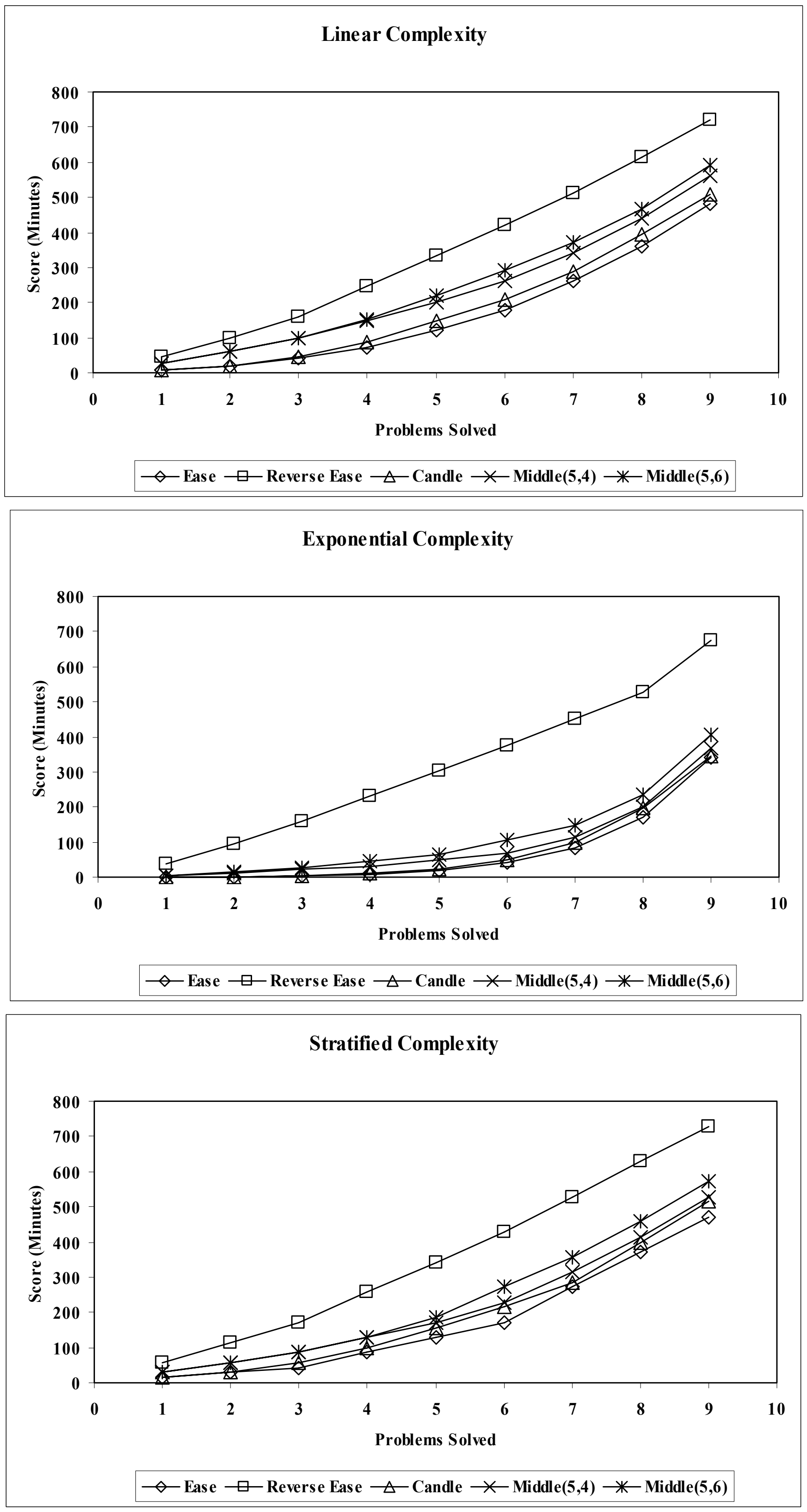

Figure 3: The score increases with solved problems for each strategy when three problems are solved in parallel.

Accumulation of tie-break score is a repeated sum. In the sequential case, time to solve the first problem is included in this sum for each solved problem. Time to solve the second problem is included one fewer times and so on until the time to solve the final problem is included only once. When multiple problems are solved concurrently, the score can be considered a sum of these scores. More formally, in the single case, tie-break score, *s*, is given by

$$s = (N)\left|p_{t,1}\right| + (N-1)\left|p_{t,2}\right| + \ldots + (1)\left|p_{t,N}\right| \tag{1}$$

in other words,

$$s = \sum_{n=1}^{N} (N-n+1)\left|p_{t,n}\right| \tag{2}$$

where N is the number of problems solved by team t, and $|p_{t,n}|$, is the time to solve the n-th problem solved by team t. For any score, that score can be reduced if the first clause $(N)|p_{t,1}|$ can be reduced, so to the second clause, and so on to the last clause. More formally,

Theorem 2
Tie-break score is minimal when problems are solved in order of increasing complexity.

Proof
This is the scheduling concept *total flow time* which is know to be optimal when the shortest jobs are scheduled first [5]. We include a proof for completeness.

Sequential Problem Solving
In the case of only one problem, p_1, there is only one order.

Examine the case of two problems, p_1 and p_2; $|p_1| < |p_2|$. Solving p_1, p_2 the score is

$$2 \times |p_1| + |p_2| \tag{3}$$

solving p_2, p_1, the score is

$$2 \times |p_2| + |p_1| \tag{4}$$

the difference, equation (4) minus equation (3), is

$$|p_2| - |p_1| \tag{5}$$

as $|p_1| < |p_2|$, this is necessarily positive. Therefore equation (3) is necessarily smaller than (4).

In the general case, take two problems, p_s and p_{s+1} ($|p_s| < |p_{s+1}|$), and solve in order p_s, then p_{s+1}. The influence of p_s on the score is $(N - s + 1)|p_s|$. For problem p_{s+1}, it is $(N - (s + 1) + 1)\ |p_{s+1}|$. The total influence on score is

$$(N - s + 1)|p_s| + (N - s)\ |p_{s+1}| \tag{6}$$

solving in the reverse order the score is

$$(N - s + 1)|p_{s+1}| + (N - s)\ |p_s| \tag{7}$$

Subtracting equation (7) minus equation (6), to find the additional cost of solving in this order gives

$$(N - s + 1)|p_{s+1}| + (N - s)\ |p_s| - (N - s + 1)|p_s| - (N - s)\ |p_s + 1| \tag{8}$$

This reduces to $|p_{s+1}| - |p_s|$, which (as $|p_{s+1}|$ is greater than $|p_s|$) is necessarily positive. Therefore there is a positive impact on the score when any two problems are solved “out of order”. The only possible order sustaining $|p_{s+1}| > |p_s|$ for all p_s is increasing order of complexity. In other words, solving in order of increasing complexity is minimal.

Solving Two Concurrently
In this case, the tie-break score is the sum of two sums, one for each concurrent solver. In the case of each solver, the score is minimized when solved in order of increasing complexity. But the interaction of the two concurrent solvers must be considered.

Given one problem, p_1, the score is constant regardless of who solves the problem.

Given two problems, p_1 and p_2, ($|p_1| < |p_2|$) which could be distributed in two ways; either one team member solves both, or each solves one. In the first case the score is $2 \times |p_1| + |p_2|$. In the second case the score is $|p_1| + |p_2|$. The second case is necessarily smaller than the first case.

Given three problems, p_1, p_2, and p_3, ($|p_1| < |p_2| < |p_3|$) from the solution to two problems, there are two possible distributions; X solves p_1 and p_2, or X solves p_1 and p_3; in both cases X' solves the remaining problem. In the first case the score is $2 \times |p_1| + |p_2| + |p_3|$. In the second case the score is $2 \times |p_1| + |p_3| + |p_2|$. The score for these two are equal.

Given four problems, p_1, p_2, p_3, p_4, ($|p_1| < |p_2| < |p_3| < |p_4|$), where X is solving p_1 and p_2, and X' is solving p_3 and p_4, the individual scores $|X|$ and $|X'|$ are given by $|X| = 2 \times |p_1| + |p_2|$, and $|X'| = 2 \times |p_3| + |p_4|$. Each solver is, necessarily, solving problems in order of easiest to hardest. The effect of swapping p_2 and p_3 is $|p_3| - |p_2|$ which, as $|p_3| > |p_2|$, is necessarily positive.

Given N problems, p_1, p_2, ..., p_N ($|p_1| < |p_2| < \ldots < |p_N|$), ordered with optimal score, introduce a new problem p_0 such that $|p_0| < |p_1|$. We now ask who must solve that problem to maintain the optimal score. Introducing before p_1, solved by X, will increase the overall score of the team by $|X_{N+1}| \times |p_0|$, where $|X_N|$ is the number of problems that X has already solved. Adding before p_2, solved by X', will increase the score by $|X'_{N+1}| \times |p_0|$. The problem must be given to whoever will solve the fewest problems in the remainder of the contest. In other words, solvers must take it in turn to solve problems.

Succinctly, to minimize individual score the problems must be solved easiest to hardest, and to minimize overall score, the problems solved must be the easiest problems available.

Solving Three Concurrently
Appealing to the solution of solving two concurrently, the same holds true for three problems solved concurrently. Each solver must solve in order of increasing $|p_i|$ and problems must be distributed in increasing $|p_i|$.

Consequences of the Proof
In Figure 4, the number of problems solved at any moment in time is shown for each problem order. Examining linear complexity, first to solve four problems is order of ease, then candle, followed by middle(5,4), middle(5,6), then finally reverse. At the end of the competition, the order is different. First to finish is reverse, then candle, followed jointly by ease and middle(5,4) and finally middle(5,6). Throughout most of the competition, order of ease is a winning strategy, but at the end, reverse order is the better strategy. Of the strategies tested, there is no "always wins" strategy.

A consistent error in estimating the problem solution time is equivalent to reducing the length of the competition by the error. Imagine the linear complexity competition finishing after one hour and 50 minutes ($|\Theta| = 110$). In this case, a team adopting reverse order of ease will solve one more problem than any other team – and win.

This raises the question: What is the optimal order in which to tackle the problems so as to reduce the overall completion time?

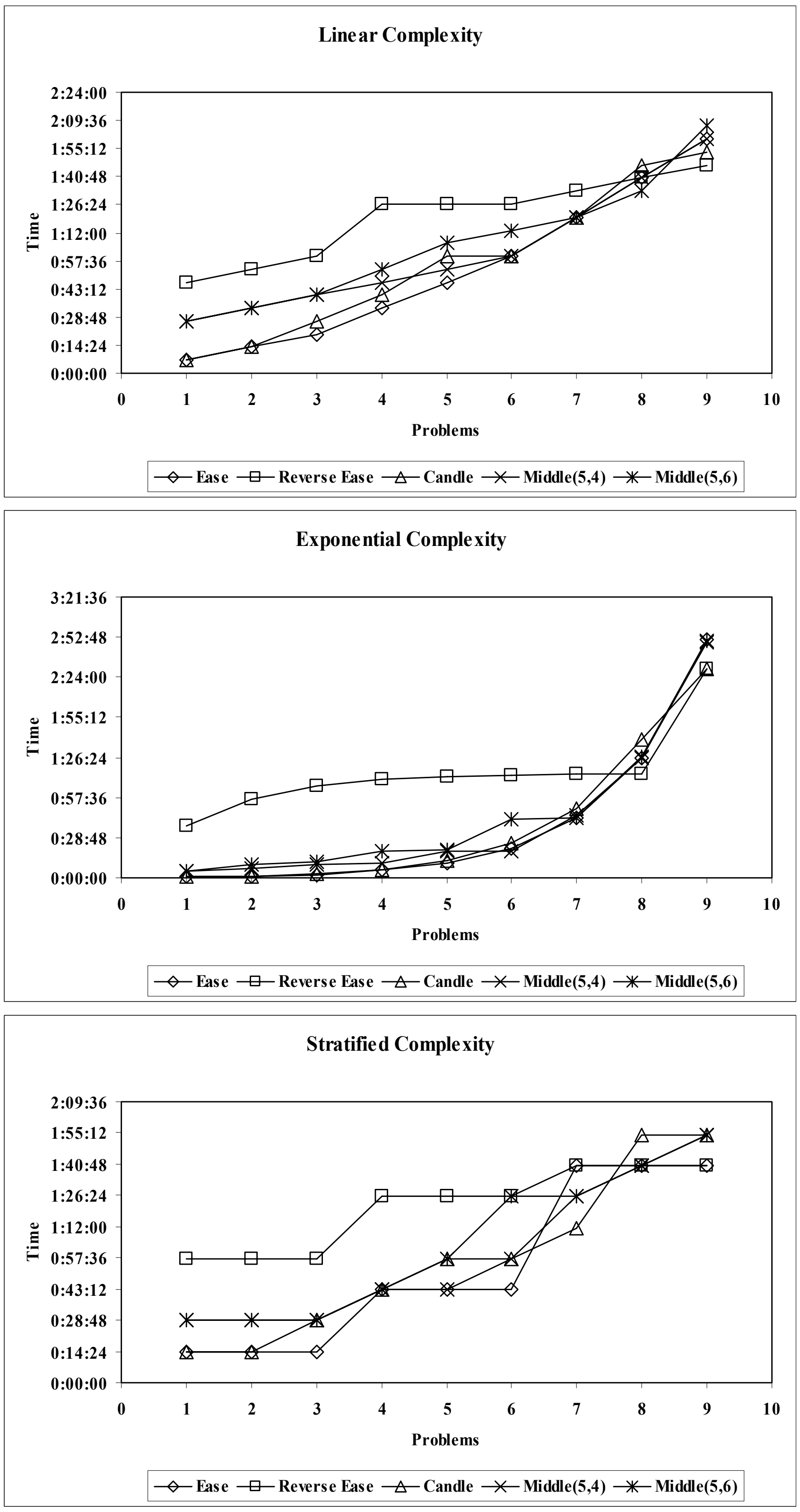

Figure 4: The number of solved problems increases with time for each strategy.

Theorem 3
There is no single a priori total time minimizing strategy.

Sequential Problem Solving
The time to solve all problems is the sum of solution times, $|p_i|$, for each problem. As addition is commutative, this is constant regardless of the order in which the problems are solved. No one strategy is any better than any other.

Solving Two Concurrently
This problem can be restated: Given a set of jobs ($p_i \in \Pi$) with completion times, $|p_i|$, and two identical processors (X and X'), partition the jobs so as to minimize the completion time of the last job. This is a well known scheduling problem [21] equivalent to the 2-PARTITION problem and known to be NP-complete. The team must solve the NP-complete problem to determine the optimal strategy.

Solving Three Concurrently
Three concurrently is equivalent to the NP-complete problem 3-PARTITION.

Consequences of the Proof
There is not only no one best strategy, but, with the exception of sequential problem solving, there is no polynomial time algorithm for finding the best strategy.

4.4. The Proposed Winning Strategy

The equivalent scheduling problem is that of minimizing the *makespan* of the *total flow time* minimal schedules. This is known to be NP-complete. [5].

The most important skill of a team is solution time estimation. Given this, a team must first determine which strategies will complete the maximum number of problems in the time period. Then, from this it must determine which strategy has the lowest tie-break score. The team can use rules of thumb such as:

- Order of ease is low tie breaking, but slow,
- Reverse order of ease is fast, but high tie-breaking,
- Candle is not-so-fast and not-so-bad for tie-breaks,
- Middle strategies are bad.

To win by problems solved, reverse order of ease is effective. To win tie-breaks, order of ease is effective. There are, however many other factors affecting a real contest; we agree with Manzoor [16] that “success in programming contests in affected by factors other than skill, most importantly adrenaline, luck, and the problem set of the contest”.

5. Conclusions

Each year the ACM runs a programming contest, now known as the ICPC. In this contest teams solve previously unseen problems. The contest runs for five hours, during which time teams of three programmers are expected to solve as many of about nine problems as they can. Scoring is on a two-tier basis. The team to solve the most problems wins, unless two or more teams solve the same number of problems. In this case, a tie is broken on accumulated solution time.

Choice of strategy is shown to affect a team’s chance of winning – not all strategies are equal. Several strategies were proposed and examined. Parallelization of problem solving shows that the more problems are solved concurrently, the quicker the team will solve the problems and the lower the tie-break score. The optimal strategy for minimizing the tie-break score is shown to be solving in order of easiest to hardest. Not only is there no one single strategy that will reduce the overall time to solve all problems, but there is not even a polynomial time algorithm to find that strategy.

A team must decide, on entering the competition, how many problems it is likely to solve, and how many problems other teams will solve. Should the team suspect that more than one team will tie on solved problems, solving from easiest to hardest will increase the team’s chance of winning the tie, even though the actual time to completion is longer. Should the team suspect that it can win by solved problems alone; it should tackle the problems in order of hardest to easiest as in this investigation that strategy has always resulted in the first solution to the last problem. Using this mixed strategy a team can maximize the chance of winning.

References

[1] Andrianoff, S. K., Hunkins, D. R., & Levine, D. B. (2004). Adding objects to the traditional ACM programming contest. In *Proceedings of the 35th SIGCSE technical symposium on computer science education*, (pp. 105-109).

[2] Astrachan, O., Khera, V., & Kotz, D. (1993). The internet programming contest: A report and philosophy. In *Proceedings of the 24th SIGCSE technical symposium on computer science education*, (pp. 48-52).

[3] Bagert, D. J. (1993). Competing in the ACM scholastic programming contest. In *Proceedings of the 1993 ACM Conference on Computer Science*, (pp. 528).

[4] Chavey, D., Monrey, T. L., Brackle, D. V., & Werth, J. (1991). Preparing a team for the ACM scholastic programming contest. In *Proceedings of the 19th annual conference on computer science*, (pp. 701).

[5] Coffman, E. G. (1976). Combinatorial sequencing and allocation problems. In M. D. Atkinson & J. F. Reynolds (Eds.), *Analytical aspects of computer operating systems.*

[6] Deimel, L. E. (1988). Problems from the 12th annual ACM programming contest. *SIGCSE Bulletin,* 20(4), 19-28.

[7] Dodge, R. C., & Ragsdale, D. J. (2004). Organized cyber defense competitions. In *Proceedings of the Fourth IEEE International Conference on Advanced Learning Technologies (ICALT'04)*, (pp. 768-770).

[8] Ernst, F., Moelands, J., & Pieterse, S. (1996). Programming contest strategies - teamwork in programming contests: 3 * 1 = 4. *Crossroads,* 3(2), 17-19.

[9] Fitzgerald, S., & Hines, M. L. (1996). The computer science fair: An alternative to the computer programming contest. In *Proceedings of the 27th SIGCSE technical symposium on computer science education*, (pp. 368-372).

[10] Gibbs, W. W. (2006). Innovations from a robot rally. *Scientific American,* 294(1), 64-71.

[11] Google Inc. (2002). First annual google programming contest. Available: http://www.google.com/programming-contest/ [2006, 21 April].

[12] ICFP Contest Organizers. (2005). 2005 ICFP programming contest. Available: http://lambda-the-ultimate.org/node/744 [2006, 21 April].

[13] ICPC. (2006). The ACM international collegiate programming contest sponsored by IBM fact sheet. Available: http://icpc.baylor.edu/icpc/About/Factsheet.pdf [2006, 21 April].

[14] Leal, J. P., & Silva, F. (2003). Mooshak: A web-based multi-site programming contest system. *Software—Practice & Experience,* 33(6), 567-581.

[15] Manne, F. (2000). Competing in computing. In *Proceedings of the 2000 Norsk Informatikkonferanse*, (pp. 129-138).

[16] Manzoor, S. (2001). Common mistakes in online and real-time contests. *Crossroads,* 7(5), 4.

[17] Salenieks, P., & Naylor, J. (1988). Professional skills assessment in programming competitions. *SIGCSE Bulletin,* 20(4), 9-14.

[18] Shilov, N. V., & Yi, K. (2002). Engaging students with theory through ACM collegiate programming contest. *Communications of the ACM,* 45(9), 98-101.

[19] Skiena, S. S., & Revilla, M. (2003). *Programming challenges*. New York: Springer Verlag.

[20] Szuecs, L. (2001). My favorite programming contest problems. *Journal of Computing Sciences in Colleges,* 17(1), 225-232.

[21] Tanaev, V. S., Gordon, V. S., & Shafransky, Y. M. (1994). *Scheduling theory: Single-stage systems* (Vol. 284): Kluwer Academic Publishers.

[22] TopCoder. (2005). About topcoder. Available: http://www.topcoder.com/tc?module=Static&d1=about&d2=index [2006, 21 April].

[23] Valladolid. (2004). Valladolid online judge. Available: http://acm.uva.es/ [2004, 29 September].

[24] X Prize Foundation. (2004). The ansari legacy. Available: http://www.xprizefoundation.com/about_us/ansari_legacy.asp [2006, 25 April].