Creating Informatics Olympiad Tasks: Exploring the Black Art

Benjamin A. BURTON
Department of Mathematics, SMGS, RMIT University
GPO Box 2476V, Melbourne, VIC 3001, Australia
email: bab@debian.org

Mathias HIRON
France-IOI
5, Villa Deloder, 75013 Paris, France
email: mathias.hiron@gmail.com

Author's self-archived version
Available from `http://www.maths.uq.edu.au/~bab/papers/`

Abstract. Each year a wealth of informatics olympiads are held worldwide at national, regional and international levels, all of which require engaging and challenging tasks that have not been seen before. Nevertheless, creating high quality tasks can be a difficult and time-consuming process. In this paper we explore some of the different techniques that problem setters can use to find new ideas for tasks and refine these ideas into problems suitable for an informatics olympiad. These techniques are illustrated through concrete examples from a variety of contests.

Keywords. programming contests, task creation

1. Introduction

Like writing music or proving theorems, making new informatics olympiad tasks is a highly creative process. Furthermore, like many creative processes, it is difficult to create tasks on a tight schedule. With a rich yearly programme of national and international contests however, this is a problem that many contest organisers face. How can organisers come up with fresh tasks, year after year, that are not simple variations of problems that students have already seen?

We do not hope to solve this problem here – this paper does not give an "algorithm" for creating good tasks. What we do offer is a range of techniques for creating new tasks, drawn from the collective experience of the scientific committees of two countries. Of course each contest organiser has their own methods for setting tasks; the hope is that readers can blend some of these techniques with their own, and that this paper can encourage a dialogue between contest organisers to discuss the many different methods that they use.

We begin in Section 2 by describing the qualities of a "good" informatics olympiad task, and in Section 3 we run through different characteristics that make one problem different from another. Section 4 covers the difficult first step of finding an original idea; the methods offered span a range of creative, ad-hoc and methodical techniques. In Section 5 we follow up with ways in which an original idea can be massaged and

modified into an idea suitable for a real contest. Finally Section 6 reformulates the ideas of the earlier sections in a more abstract, "algorithmic" setting.

Creating new tasks is only part of the work required to organise a contest. Diks et al. (2007) discuss the work that follows on from this, such as task write-ups, analysis, documentation, solutions, test data, and final verification.

2. What Makes a Good Task?

Before we set out to create new tasks, it is important to understand what we are aiming for. In an informatics olympiad, the target audience (talented high school students), the tight time frame (typically at most five hours for a contest) and the need to fairly rank competitors all place restrictions on what tasks can be used.

The following list identifies some features of a good informatics olympiad task. This list is aimed at contests modelled on the International Olympiad in Informatics (IOI) – other programming contests have different goals, and so their tasks might have different needs. Additional notes on the suitability of tasks can be found in Diks et al. (2007) and Verhoeff et al. (2006).

- It should be possible to express the task using a problem statement that is (relatively) short and easy to understand. The focus of the task should be on problem solving, not comprehending the problem statement. It is also nice to include some tasks that relate to real-world problems, though this is by no means necessary.
- Ideally the algorithm that solves the task should not directly resemble a classic algorithm or a known problem. It might be a completely original algorithm, it might be a modified version of a classic algorithm, or it might resemble a classic algorithm after some transformation of the input data.
  An example of transforming a classic algorithm is *Walls* from IOI 2000. Competitors essentially read a planar graph from the input file. If they convert this graph into its dual, the problem becomes the well-known breadth-first search, with some minor complications involving multiple starting points.
- The task should support several solutions of varying difficulty and efficiency. This allows weaker students to gain partial marks by submitting simple but inefficient solutions, while stronger students can aim for a more difficult algorithm that scores 100%.
- The official solution should allow a reasonably concise implementation (at most a few hundred lines of code). It should also be possible to estimate during the pen-and-paper design stage whether this solution can score 100% for the given time and memory constraints.
- Ideally the official solution should also be the best known solution for the task, though whether this is desirable may depend on the intended difficulty of the task.
- For contests aimed at experienced students, it is nice to have tasks where it is not obvious in advance what category (such as dynamic programming or graph theory) the desired algorithm belongs to. It is also nice to have solutions in which two or more different algorithms are welded together.

3. Characteristics of Problems

In this section we run through the different types of characteristics that a task can have. This is useful for categorising problems, and it also highlights the different ways in which a task can be changed.

Although this paper is not concerned with categorisation per se, understanding the different characteristics of old tasks can help problem setters create new tasks that are different and fresh. This categorisation is particularly important for the techniques of Section 4.4, and it plays a key role in the algorithmic view of task creation described in Section 6.

Examining characteristics of tasks also helps us to experiment with the many different ways in which a task can be changed. Changing tasks is a critical part of the creation process; for example, tasks can be changed to make them more interesting or original, or to move them into a desired difficulty range. The process of changing tasks is discussed in detail in Section 5.

Each task has many different characteristics; these include characteristics of the abstract core of the task, of the story that surrounds the task, and of the expected solution. These different types of characteristics are discussed in Sections 3.1, 3.2 and 3.3 respectively. This list does not claim to be exhaustive; readers are referred in particular to Verhoeff et al. (2006) for additional ideas.

3.1. Characteristics of the Abstract Core

Here we reduce a task to its abstract, synthesised form, ignoring the elements of the story that surrounds it. Different characteristics of this abstract form include the following:

- *What kinds of objects are being manipulated?*
  Examples of objects include items, numbers, intervals, geometric objects, and letters. Some tasks involve sets or sequences of these basic objects; these sets or sequences can be arbitrary (such as a sequence of numbers read from the input file) or described through rules (such as the set of all lattice points whose distance from the origin is at most d). A task may involve more than one kind of object.
- *What kinds of attributes do these objects have?*
  Attributes can be numerical, such as the weight of an edge in a graph, or the score for a move in a game. They can also be geometric, such as the coordinates of a point or the diameter of a circle; they can even be other objects. Attributes can have static or dynamic values, and can be given as part of the problem or requested as part of the solution.
- *What are the relationships between these objects?*
  Relationships can be arbitrary (such as the edges of a given graph forming a relationship on the vertices), or they can be described by rules (such as intersections, inclusions, adjacencies, or order relations). They can relate objects of the same kind or of different kinds, and they can even be relations between relations (such as relationships between edges of a graph). Relationships can be constraints that one object places upon another (such as a maze, which constrains

the coordinates of objects inside it); or they can constrain the successive values of an attribute (such as a graph in which some object can only move to adjacent vertices).

- *What kind of question is asked about these objects?*
  The question might involve finding a specific object (or set of objects) that has some property, or that maximises or minimises some property. It might involve aggregating some attribute over a set of objects, or setting the attributes of each object to reach a particular condition.
- *Does the task ask just one question, or does it ask many questions over time?*
  The same question can often be asked in two ways: (i) asking a question just once, or (ii) asking it many times, where the objects or their attributes change between successive questions. An example of the latter type is *Trail Maintenance* from IOI 2003, which repeatedly asks for a minimal spanning tree in a graph whilst adding new edges between queries. These changes might be given as part of the input or they might depend on the previous answers, and questions of this type can lead to both interactive and non-interactive tasks.
- *What general constraints are given on the structures in the task?*
  Examples for graph problems might include general properties of the graph, such as whether it is directed, sparse, acyclic or bipartite. Constraints for geometric problems might include the fact that some polygon is convex, or that no two lines in some set are parallel.

Some of these characteristics – in particular, the objects, attributes and relationships – are reminiscent of other fields of computer science, such as object-oriented programming or database design. However, in the more flexible domain of informatics olympiad tasks, these characteristics can cover a much broader scope.

3.2. Characteristics of the Story

The same abstract task can be presented in many different ways, by wrapping it inside different stories. Selecting a good story can have a great influence on the quality and accessibility of a task. As discussed in Section 5, it can also affect the difficulty of the task through the way it hides elements of the core problem. Characteristics of the story include the following:

- *What real world item do we use to represent each object?*
  Typical items include roads, buildings, cows, farmers, strings; the list goes on. Rectangles might become cardboard boxes; a grid of numbers might become heights in a landscape. The possibilities are immense.
- *What kind of measures do we use for each attribute?*
  Numerical attributes might represent time, weight, age, quantity or price. Symbolic attributes could represent colours or countries; geometric attributes such as coordinates or diameter could describe a location on a map or the range of a radio tower.
- *How do we justify the relationships between objects?*
  This typically depends upon how the objects themselves are represented. For instance, if the objects are people then relationships could be described as friendships or parent/child relationships. Often the descriptions of relationships follow naturally from the descriptions of the objects themselves.

- *How do we explain why the question needs to be solved?*
  This explains the motivation for the task, and often provides an overall storyline for the problem.

3.3. Characteristics of the Solution

The solution of a problem is its most important aspect – a large part of what makes a task interesting is how interesting and original the solution is. Unfortunately it can be difficult to change the solution directly without having to make significant changes to the entire task. Characteristics of the solution include:

- *What domains of algorithmics are involved?*
  Examples include graph theory, dynamic programming and computational geometry. Verhoeff et al. (2006) provide an excellent reference of domains that are relevant for the International Olympiad in Informatics (IOI).
- *What mathematical observations need to be made?*
  Often a solution relies on some property of the task that is not immediately obvious. For instance, students might need to realise that a graph contains no cycles, that some attribute can never be negative, or that a simple transformation can always convert a "sub-optimal" answer into a "better" answer.
- *What is the main idea of the algorithm?*
  This is the core of the solution, and describes the key steps that are eventually fleshed out into a detailed algorithm and implementation. Is it a greedy algorithm? Does it build successive approximations to a solution? Does it solve a problem by iteratively building on solutions to smaller problems? What are the important loop invariants?
- *What data structures can be used?*
  In some cases the data structures form an integral part of the algorithm. In other cases, clever data structures can be used to improve an algorithm, such as implementing Dijkstra's algorithm using a priority queue. For some algorithms there are several different choices of data structure that can lead to an efficient solution.

4. Finding a Starting Point

One of the most difficult steps in creating a new task is finding a starting point – that elusive new idea for a task that is like nothing students have seen before. In this section we explore some of the different techniques that can be used to find an original new idea.

The techniques presented here are based on personal experience. This list was compiled by approaching several problem setters from the authors' respective countries, and asking them to reflect on past contests and analyse the different processes that they used to generate ideas.

Many of the examples in this section are taken from the French-Australian Regional Informatics Olympiad (FARIO), the France-IOI training site, and the Australian Informatics Olympiad (AIO). Full texts for all of these problems are available on the Internet: the FARIO problems from `www.fario.org`, the France-IOI problems from `www.france-ioi.org` (in French), and the AIO problems from the Australian

training site at `orac.amt.edu.au`. The last site requires a login, which can be obtained through a simple (and free) registration.

Finding a new idea is of course only the beginning of the task creation process, and is usually followed by a round of modifications to improve this initial idea. These modifications tend to follow a similar pattern regardless of how the initial idea was formed, and so they are discussed separately in Section 5.

4.1. Looking Around

One of the most common techniques, though also one of the least predictable, is to "look around" and take inspiration from things that you see in real life. An alternative is to find somebody unfamiliar with computer science or programming contests, and to ask them for ideas.

An example is *Guards* (FARIO 2006), which describes a circular area with roads entering at different points on the edge of the circle (Fig. 1). The task is then to place guards at some of these roads so that every road is "close enough" to a guard, and to do this using as few guards as possible in linear time. This turned out to be a relatively tricky sliding window problem, and has proven useful for training in the years since. The inspiration for this problem was the 2006 Commonwealth Games in Melbourne, where one author had a large part of his suburb barricaded off for the netball and hockey matches.

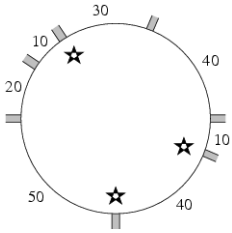

Fig. 1. The circle of roads for the task *Guards* (FARIO 2006)

Another example is *Banderole* (France-IOI), which gives a line of vertical sticks and asks for the number of integer positions in which a rectangle can be balanced (Fig. 2). The idea for this problem came from a France-IOI team dinner, where people were balancing the objects in front of them (plates, cider bottles and so on).

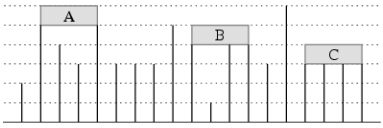

Fig. 2. Examples of balanced rectangles for the task *Banderole* (France-IOI)

Looking around for inspiration has some disadvantages. It requires a great deal of effort trying to solve the problems that you create, since you do not have a solution in mind in advance. Furthermore (as is frequently found in the real world) these ideas can

often lead to problems that are NP-hard, though the techniques of Section 5 can be used to convert them into more tractable tasks.

However, the same cause of these difficulties – that you do not have a solution ready in advance – means that this method can sometimes lead to highly original problems that do not rely on simple modifications of standard algorithms. One of the first author's favourite problems is *Citizenship*, from the Australian team selection exam in 2006; this gives rules for gaining and losing citizenship of different countries, and asks for the largest number of countries for which you can hold citizenship simultaneously. The solution involves an interesting representation of the input data and unusual ad-hoc techniques on directed graphs. The motivation for this problem was a dinner conversation at the 2006 Australian Linux conference (which was held in Dunedin, New Zealand).

There are many places to "look around" for inspiration. One is your immediate environment – focus on a nearby object, or an action that somebody performed. Another is to recall events in your life or a friend's, or even in the news. Plots from books or movies can also be used as a starting point.

Once you have picked a specific object or event, you can focus on that object or event and work to create a task around it; this is usually more efficient than looking in lots of different places until inspiration comes. Restricting your attention to a given branch of algorithmics can also help; for instance, you might focus on creating a graph problem involving light bulbs. Working with unusual objects, actions or events can help to create original ideas.

4.2. Drawing on the Day Job

Another technique employed by many problem setters is to use tasks that arise in their daily work. For instance, you might be working on a large and complex research problem but find that a small piece of this problem is just the right difficulty for an informatics olympiad. Alternatively, you might be reading through a research paper and find an interesting algorithm that can be adapted for a programming contest.

One problem of this type is *Giza* (FARIO 2007), which was inspired by real-world work in tomography. This task is a discrete 2-dimensional version of the general tomography problem. Specifically, it asks students to recover a secret image formed from grid cells, given only the cell counts in each row, column and diagonal (Fig. 3).

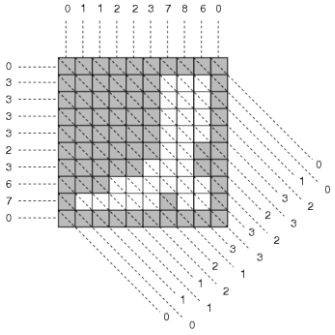

Fig. 3. A secret image and cross-section totals for the task *Giza* (FARIO 2007)

Another example is *Frog Stomp*, from the 2007 Australian team selection exam. This is an output-only task that asks students to find "sufficiently short" integer sequences with particular properties. This problem arose as part of a larger research project in eliminating statistical biases in financial data, and was easily extracted from its financial setting into a standalone mathematical task.

In contrast to "looking around", with this technique you already begin with a solution – either you have already solved it yourself, or you have read the algorithm in a research paper. While this is convenient, it also risks underestimating the difficulty of such tasks. Personal research projects typically involve ideas that you have worked on for months, and research papers rarely give an indication of *how* they arrived at an algorithm or how difficult this was. It is usually best to give the problem to other people to solve, in order to gain a better estimate of its true difficulty.

4.3. Modifying a Known Algorithm

One of the simpler techniques for producing new tasks is to begin with a standard algorithm (such as quicksort or a breadth-first search) and set a task that modifies it in some way.

An example is *Chariot Race* (FARIO 2004), which requires a modification to Dijkstra's algorithm for the shortest path through a graph. In this task the problem setters began with Dijkstra's algorithm and set out to remove the "no negative edge weights" constraint.

If we simply allow negative edge weights, this creates difficulties with infinite negative cycles. To avoid this, the task was changed so that, instead of edges with fixed negative weights, we allow *wormholes* – edges that "travel back in time" by dividing the current total distance by two (Fig. 4). By making this division round to an integer, the problem setters were able to avoid infinite negative cycles entirely.

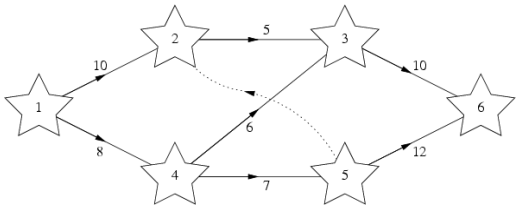

Fig. 4. A graph for *Chariot Race* (FARIO 2004), with the wormhole as a dotted arrow

The solution to Chariot Race is essentially Dijkstra's algorithm with adaptations to deal with (i) the fact that the total distance can decrease after following an edge, and (ii) the fact that the precise *change* in distance when following a wormhole is not a constant, but instead depends upon the total distance so far. Good students should be able to deal with both adaptations easily enough, and indeed this turned out to be the easiest problem on the FARIO contest paper.

The advantages of beginning with a standard algorithm are that it is easy to create new tasks (there are plenty of standard algorithms, and plenty of modifications to make), and that you typically have a good idea of what the solution will look like in advance. The main disadvantages are that the resulting tasks will often be similar to

well-known problems, and that good students should see through this easily. It is therefore difficult to create highly original problems using this technique.

It should be noted that the algorithm you begin with does not need to be one of the fundamental algorithms that appear regularly in olympiad problems. There are plenty of standard algorithms that are less commonly taught; examples include graph vertex connectivity, permutation generation, and union-find data structures. As a possible starting point, Skiena (1998) offers an excellent catalogue of standard algorithms and their variants, and Verhoeff et al. (2006) propose a detailed syllabus of topics that relate to the IOI.

4.4. Filling the Holes in a Domain

Here we describe a technique that focuses on a particular domain, such as shortest path problems or binary trees, where you already have a pool of problems in stock that you cannot reuse. This technique is useful when preparing training sessions for experienced students who have seen many problems before; it can also help create new tasks for programming contests.

The technique essentially begins with a large pool of tasks, examines the characteristics of these tasks, and then forms new combinations of these characteristics. In detail:

- The first step is to compile a thorough list of "old" tasks in the domain of interest. This can be time consuming, but it is important to include as many tasks as possible – not only does this list remind you of tasks that you cannot reuse, but it also serves as a basis for creating new tasks in the domain.
  To work with this list efficiently, it helps to have each task clearly in mind. A suggestion is to synthesise each task and its solution in a few sentences; in many cases a quick drawing can also help. As a side-effect, compiling this list helps give a clear view of the types of problems that can be created in this domain, and new ideas might come naturally as a result.
- The next step is to examine the different characteristics of the tasks in this list; examples of such characteristics can be found in Section 3. The aim is to find characteristics that tasks have in common, and also characteristics in which tasks differ. In this way, each task can be described as a combination of different characteristics.
- Creating new tasks is then a matter of finding combinations of characteristics that have not yet been used. Blending the characteristics of two or more different problems in the domain can often lead to a meaningful (and interesting) new task.

To illustrate, we can apply this technique to the domain of sweep-line algorithms and sliding windows. A list of 30 old problems was compiled by the second author; although the individual problems and solutions are too numerous to list here, the main characteristics are as follows:

(i) The objective may be to find a point, a set of points, an interval, or a set of intervals satisfying some given properties. Alternatively, it may be to aggregate some value over all points or intervals satisfying some given properties.

(ii) The points or intervals may need to be chosen from a given list, or they may be arbitrary points or intervals on either a discrete or continuous axis.

(iii) The properties used to select these points or intervals may involve relationships amongst these objects, or they may involve relationships between these objects and some entirely different set of objects. In the latter case, this second set of objects may itself involve either points or intervals.
(iv) The properties used to select these points or intervals may be purely related to their location on the axis, or they may involve some additional attributes.
(v) When intervals are manipulated, intervals that overlap or contain one another may or may not be allowed.
(vi) In the story, the objects may appear directly as points or intervals on a given axis, or they may be projections of some higher-dimensional objects onto this axis. The axis may represent location, time, or some other measure.
(vii) Solutions may involve sweep-line algorithms, sliding windows, or possibly both.

Having listed the most important characteristics within this domain, we can now choose a new combination of values for these characteristics. We will ask for (i) an interval, chosen on (ii) a continuous axis. We will require this interval to satisfy some relationship involving (iii) a different set of intervals, given in the input file. Intervals will (iv) be assigned a colour from a given list of possible colours, and (v) no rules will prevent intervals from intersecting or containing one another. The story will use (vi) circles as its primary objects, where the intervals describe the angles that these circles span when viewed from the origin.

Now that a choice has been made for each characteristic, most of the elements of the problem are set. Fleshing this out into an abstract task, we might obtain the following:

> *A set of circles is given on a two dimensional grid. Each circle has a colour identified by an integer. The task is to choose two half-lines starting from the origin, so that at least one circle of each colour lies entirely in the region between these lines, and so that the angle between these lines is as small as possible.*

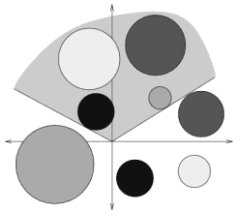

Fig. 5. Creating a new task by combining the different characteristics of old tasks

This task is illustrated in Fig. 5. A story still needs to be built around this idea, but we have a good start for a new task that is different from the 30 problems originally listed.

This technique gives an efficient way of creating many new problems in a short time within a specific domain. However, it tends to produce tasks that are not truly original, since they always share the characteristics of existing problems.

One way to obtain more original problems with this technique is to choose values for some characteristics that do not appear in *any* of the listed problems. For instance, in the example above, the objects in the original list of 30 problems include points,

intervals, segments, squares, rectangles, right-angled triangles, and circles. We might therefore try parallelograms or ellipses for something new.

4.5. Building from Abstract Ideas

If you don't have a day job or other external sources of inspiration, another way to generate new ideas is to draw random things on a blank piece of paper. Here we describe a technique that begins with vague sketchings and gradually refines them into a usable problem.

In Section 3.1 we describe several kinds of objects that can appear in a task. To search for ideas, you can pick one kind of object at random and draw some instances of it on a piece of paper. You might also choose a second kind of object and do the same. You can then search for a task that involves these objects; this might require you to find some attributes, relationships, and a question about these objects that you can ask. Drawing diagrams to represent the different elements of the task can help ideas to come.

During the initial stages of this process, what you manipulate is not always a precise problem with a solution. As you add objects, attributes and relationships one after another, you might not have specific algorithms in mind but rather vague ideas of where you are heading. Each time you define another characteristic of the task, you obtain a more precise idea of the problem, and a better idea of what kind of solution you might end up with. If at some stage you are unhappy with the direction the problem is taking, you can go back and alter one of the characteristics that you defined earlier. Little by little, you get closer to a usable problem.

To illustrate this process we give a concrete example; some of the intermediate sketchings are shown in Fig. 6. We might begin by drawing circles on a piece of paper, with some intersections and inclusions. Perhaps it reminds us of a graph problem we have seen before; to avoid this we decide to add another kind of object. We choose lines, and think about ideas such as finding the line that intersect the most circles.

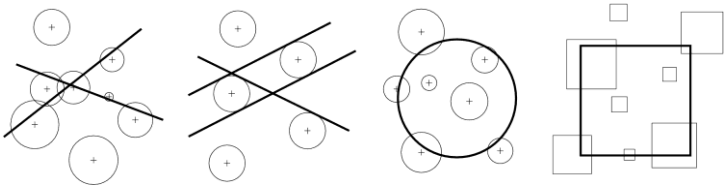

Fig. 6. Successive diagrams used whilst creating a task from abstract ideas

However, this also reminds us of problems we have seen before; we therefore try to replace lines with circles, and look for the position of a new circle that intersects the most circles from a given set. This might become a relative scoring task, which is not what we are looking for today. Replacing circles with squares yields an interesting problem for which we can find several solutions of varying complexities. We massage it a little further by changing squares into rectangles and giving them a "value" attribute, and obtain the following task:

> *You are given N rectangles ($N \leq 500$) whose sides are parallel to the x and y axes. Each rectangle is described by four integer coordinates between 0*

and 1 000 000 and a value between 1 and 1 000. Given a width W and height H (0 ≤ W,H ≤ 100 000), your task is to find a new rectangle of size W × H for which the total value of all rectangles it intersects or contains is as large as possible.

This task is not a great one but can be good enough for some contests. One solution involves a sweep-line algorithm with an interval tree.

4.6. Borrowing from Mathematics

Although ideas for tasks can come from many different disciplines, some branches of mathematics are particularly useful for creating informatics olympiad problems. Here we focus on tasks derived from the mathematical field of *combinatorics*.

To summarise, combinatorics problems typically involve counting things or arranging things according to certain rules. Examples of combinatorics problems are (i) counting different colourings of an $n \times n$ chessboard, (ii) counting the number of different triangulations of an n-sided polygon, and (iii) creating latin squares, which are $n \times n$ grids filled with the numbers 1,2,…,n so that each row and each column contains every number once.

Combinatorics problems can often lead to interesting or unusual dynamic programming tasks, because they both share a core dependency on *recurrence relations*: formulae that solve a larger problem in terms of one or more smaller problems.

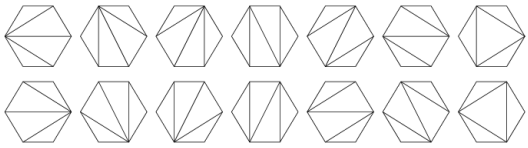

Fig. 7. The 14 different ways of triangulating a hexagon

For instance, consider example (ii) above. Let T_n be the number of ways of triangulating an n-gon; in Fig. 7 it can be seen that $T_6 = 14$. In general, T_n is the famous sequence of Catalan numbers; see van Lint and Wilson (1992) for some of the many, many other things that it counts.

The sequence $T_1, T_2, \ldots$ obeys a recurrence relation, which we can see as follows. Choose the bottom edge of the hexagon, and consider the possible triangles that this edge could belong to. There are four possibilities, illustrated in Fig. 8. In the first and last diagrams we must still triangulate the white 5-gon, whereas in the middle two diagrams we must still triangulate the white 3-gon and the white 4-gon. Thus $T_6 = T_5 + T_3T_4 + T_4T_3 + T_5$. In general, a similar argument shows that

$$T_n = T_{n-1} + T_3T_{n-2} + T_4T_{n-3} + \ldots + T_{n-3}T_4 + T_{n-2}T_3 + T_{n-1}.$$

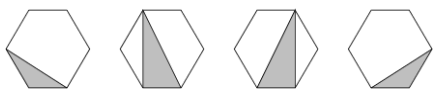

Fig. 8. The four possible triangles that might include the bottom edge

It is easy to turn this mathematical recurrence into a dynamic programming problem. One way is to ask precisely the same question – count the number of triangulations of an n-gon (possibly writing the answer mod 1000 or similar to avoid the inevitable integer overflows).

Another way is to turn it into an optimisation problem – give each triangulation a score, and ask for the largest score possible. For example, we could place a number at each vertex, and score each edge of the triangulation according to the product of the two numbers that it joins. The recurrence relation would change a little (for instance, the sum would become a maximum), but its overall structure would remain the same. This is precisely the task *Polygon Game* from the 2002 Australian team selection exam, and it was created in precisely this way. See Burton (2007) for a more detailed analysis of this problem.

Combinatorial recurrence relations are plentiful, and can be found from several sources:

- They frequently appear in the study of generating functions (Wilf, 1994). The problem *Architecture* (FARIO 2007) was based on a problem described by Wilf that counts the number of n-square polyominoes with a certain convexity property (where polyominoes are "generalised dominoes" formed by joining several squares together, similar to Tetris blocks). The final problem Architecture asks students to maximise a "prettiness" score for n-block buildings with this same convexity property.
- The online encyclopedia of integer sequences (Sloane, 2008) is full of interesting combinatorial sequences and recurrence relations; simply browsing through the encyclopedia can yield interesting results.
- Finally, it is useful to look through mathematics competitions, where combinatorics is a popular topic. There are many compilations of mathematics competition problems available; see Kedlaya et al. (2002) and Larson (1983) for examples.

Combinatorics is of course not the only branch of mathematics that can yield interesting problems. We focus on it here because many combinatorial problems are easily accessible to students with no formal mathematics background. For different examples the reader is referred to Skiena and Revilla (2003), who work through problems from several different branches of mathematics.

4.7. Games and Puzzles

Another concrete source of ideas is games and puzzles. Many obscure games can be found on the Internet, and puzzles are readily available from online sources such as the `rec.puzzles` newsgroup, and from friends' coffee tables.

Games and puzzles are useful in two ways. On the one hand, they provide ready-made tasks such as playing a game optimally or solving a puzzle using the smallest number of moves. On the other hand, they can supply interesting sets of objects and rules that can act as starting points for other tasks, using techniques such as those described in Section 4.5.

An example of a game-based problem is *Psychological Jujitsu* (AIO 2006), which involves a card game where players bid for prizes. In the real game players cannot see each others' bids. To make the problem solvable, the AIO task assumes that a player can see their opponent's bids in advance, and then asks for a sequence of bids that

allows the player to beat their opponent with the largest possible margin. The solution is an original (though relatively simple) ad-hoc algorithm.

A problem that began as a game but grew into something different is *Collier de Bonbons* (France-IOI). The original source was *Bubble Breaker*, an electronic game involving stacks of bubbles of different colours, in which clicking on a group of two or more bubbles of the same colour makes those bubbles disappear. The game was changed to use a single stack, which then became a single cycle. The task was then changed so that students must remove all of the bubbles in the cycle; bubbles can be removed individually if required, but this must be done as few times as possible. The cycle of bubbles was changed to a candy necklace, and the final solution is a nice example of dynamic programming.

One difficulty of using games and puzzles is that solving them is often NP-hard (otherwise they are not interesting for humans to play!). For this reason, tasks derived from games or puzzles often need to be simplified or constrained before they can be used in a programming contest.

4.8. Final Notes

We close this section with some final suggestions when searching for an initial idea:

- *Restrict your search to a single theme or domain.*
  To generate ideas faster, it is often useful to restrict the search space. For instance, you might search for a task relating to trains (a restriction on the story), a task involving hash tables (a restriction on the solution), or a task involving geometry (a restriction on the underlying objects). One benefit of reducing the search space is that you can quickly enumerate the classic problems in that space, which in turn makes it easier to find ideas outside these classic areas.
- *Get other people involved in the search process.*
  If you explain an idea to someone, they can often suggest improvements that you might not have considered yourself. For instance, they could suggest a different way to present the task, or add originality by modifying the task in a new direction. Moreover, explaining a task to a seasoned problem solver can occasionally give an unexpected result: by misunderstanding your explanation, they might solve the wrong task and end up with a different, more interesting version of your initial idea!
- *Be proactive!*
  Instead of only creating tasks when you need them, it helps to always be on the lookout for new ideas. It is worth keeping a file or notebook of ideas that you have had, or interesting papers you have read. Even if you just write one or two lines for each idea, it is far easier to browse through your notebook when you need new tasks than try to remember everything you thought of over the past months.

5. Improving the Task

In the experiences of the authors and most of the problem setters approached for this paper, the initial idea is typically not the final version of a task. Instead tasks are repeatedly modified, passing through several intermediate versions, until problem

setters are happy with both the originality and the difficulty of the problems and their solutions. In this section we explore this modification process in more detail.

There are several reasons a task might need modification. It might be missing one of the qualities of a good problem as outlined in Section 2. On the other hand, it might have all of these qualities but not fall within the required difficulty range. Sometimes individual tasks are fine but the contest as a whole needs rebalancing.

In Section 5.1 we examine the ways in which tasks are changed one step at a time. Section 5.2 discusses modifications with a particular focus on changing the difficulty, and Section 5.3 looks at the related issue of adding and removing subproblems. In Section 5.4 we discuss the process of modifying a task to fit an intended solution, and offer some general advice when searching for solutions to candidate tasks.

5.1. Elementary Changes

If a task is not perfect, it does not make sense to throw it away completely. Initial ideas with nice properties are hard to find, and problem setters like to preserve these properties where possible by keeping their changes small. In this way, the modification process is often a succession of small changes, where good changes are kept and bad changes are undone, until eventually all of the concerns with a task have been addressed.

It is frequently the case that a single "small change" is a result of modifying a single characteristic of the task; for instance, the principal objects of the task might change from rectangles to circles. Sometimes other characteristics must change as a result; for instance, the old attributes of position, height and width might need to become centre and radius instead.

A more interesting example is *Chariot Race* (FARIO 2004), discussed earlier in Section 4.3. The initial idea for this task was Dijkstra's algorithm; the first change was to alter the properties of the graph to allow negative edges. In order to preserve some properties of the original idea (in this case the general structure of Dijkstra's algorithm), some other changes were needed – in particular, it was necessary to replace negative edges with edges that "divide by two and round to an integer". See Section 4.3 for details.

These observations suggest a general technique for modifying a task. Examine the various characteristics of the task, as outlined in Section 3. Then try changing these characteristics one at a time, beginning with those characteristics that take the most common or uninteresting values, and keep the changes that work best.

5.2. Changing the Difficulty

It is often the case that a task is original and interesting but simply pitched at the wrong audience. Here we examine some concrete ways in which the difficulty of a task can be changed.

If a task is too difficult, simplification techniques such as those described by Ginat (2002) and Charguéraud and Hiron (2008) can be applied. Although these techniques are designed to aid problem solving, they have the side-effect of generating easier versions of a task. The general idea behind these techniques is to remove or simplify one or more dimensions of a task.

For example, consider the problem *Giza* (FARIO 2007), discussed earlier in Section 4.2. The original idea for *Giza* was the general tomography problem (recovering the shape and density of a 3-dimensional object from several 2-dimensional x-rays). The problem was simplified by removing one dimension (making the object 2-dimensional) and simplifying others (allowing only three x-rays, and restricting the densities of individual grid cells to the values 0 or 1).

On the other hand, suppose we wish to increase the difficulty of a task. One obvious method is to add dimensions to the problem. For instance, consider the task *Speed Limits* from the 2002 Baltic Olympiad in Informatics. Here students are asked to find the fastest route for a car travelling through a network of intersections and roads, where some roads have an additional attribute (a "speed limit") that changes the speed of the car. The solution is a classical Dijkstra's algorithm, but on an unusual graph where each (intersection, current speed) pair defines a vertex. In this way the extra dimension of "speed limit" increases the difficulty of the task, since students must find the hidden graph on which the classical algorithm must be applied.

Some other ways of increasing the difficulty include:

- Changing how some of the dimensions are described in the story. For example, a problem involving x and y coordinates could be reformulated so that x and y become time and brightness. Although the abstract task stays the same, the unusual presentation may make it harder to solve.
- Transforming the task into a dynamic task. Instead of asking just one question, the task might ask a similar question many times, changing the input data between successive queries.
- Asking for a full backtrace or list of steps instead of just a minimum or maximum value. For instance, a task that asks for the length of a shortest path might be changed to ask for the path itself. In some cases this can make a solution significantly harder to implement.
- Reducing the memory limit, which can often bring new challenges to an otherwise straightforward algorithm.

5.3. Adding or Removing Subproblems

One way to make a task more difficult and sometimes more original is to add a second problem or subproblem. This can be done in several ways:

- By transforming the input of the task, so that additional work is needed before the data can be used in the main algorithm. For instance, an input graph might be presented as a set of circles in the coordinate plane, where individual circles correspond to the vertices of the graph and intersections between these circles correspond to edges.
- By using the output of the original task as input for another task. For example, suppose the original task asks for the convex hull of a set of points. A new subproblem might then ask for the specific point on that hull from which the other points span the smallest range of angles.
- By embedding a new subtask into each iteration of the original algorithm, or by embedding the original task into each iteration of a new algorithm. For example, suppose the original task asks whether a graph with some given attribute x is connected. A new task might ask for the *smallest* value of x for which the graph

is connected. The new algorithm then becomes a binary search on x, where the original connectivity algorithm is used at each step.

Even beyond the extra work involved, adding a subproblem can make a task more difficult by hiding its real purpose, or by hiding the domain of algorithmics that it belongs to. For instance, in the first example above (graphs and circles) the extra subproblem might make a graph theory task look more like geometry.

Conversely, a task can be made easier by removing subproblems where they exist. For instance, the input data could be transformed in a way that requires less implementation work but still preserves the algorithm at the core of the task.

5.4. Aiming for a Solution

A significant part of the task creation process involves searching for solutions. This search can sometimes present interesting opportunities for changing a task.

While trying to solve a task, an interesting or unusual algorithm might come to mind that cannot be applied to the task at hand. Such ideas should not be ignored; instead it can be useful to study why an interesting algorithm does not work, and to consider changing the task so that it *does* work.

An example of this technique can be seen in the problem *Belts* (FARIO 2006), which began its life as a simple dynamic programming problem. Whilst trying to solve it the author found a different dynamic programming solution over an unusual set of subproblems, and eventually changed the task so that this unusual algorithm was more efficient than the simple approach.

As an aside, when working on a task it is important to invest serious time in searching for a solution. In particular, one should not give up too easily and simply change the problem if a solution cannot be found. If problem setters only consider tasks for which they can see a solution immediately, they risk only finding tasks that are close to classic or well-known algorithms.

Even once a solution has been found, it is worth investing extra time into searching for a more efficient solution. This proved worthwhile for the problem *Architecture* (FARIO 2007) – in its original form the task required an $O(n^5)$ solution, but after much additional thought the organisers were able to make this $O(n^4)$ using a clever implementation. This gave the task more power to distinguish between the good students and the very good students.

6. Algorithmic Task Creation

As explained earlier, the task creation techniques described in this paper are largely post-hoc – they have been developed naturally by problem setters and improved over the years through experience. As an attempt to further understand and refine these techniques, we now revisit them in a more abstract setting. In Section 6.1 we introduce the idea of an abstract "problem space", and in the subsequent Sections 6.2–6.4 we reformulate our task creation techniques as an "algorithmic" search through this problem space.

6.1. Defining the Problem Space

We began this paper by outlining our objectives, as seen in Section 2 which described the features of a good task. Essentially these features give us different criteria by which we can "measure" the quality of a task. Although these criteria are imprecise and subjective, they essentially correspond to an *evaluation function* in algorithmics; given a task, they evaluate the quality or interest of that task.

Following this, we outlined the many different characteristics that can define a problem, as seen in Section 3. Although once more imprecise, we can think of these characteristics as different dimensions of a large multi-dimensional *problem space*. Each task corresponds to a point in this space, and the characteristics of the task give the "coordinates" of this point.

With this picture in mind, creating a good task can be seen as a global optimisation problem – we are searching for a point in the problem space for which our evaluation function gives a maximal value (a problem of "maximal interest"). In the following sections we attempt to understand the particular optimisation techniques that we are using.

6.2. Finding a Starting Point

In Section 4, we devoted much attention to the many different techniques used by problem setters to find an initial idea for a new task. This essentially corresponds to choosing a starting point in our problem space from which we can begin our search.

It should be noted that different techniques leave different amounts of work up to the problem setter. Most techniques give a reasonably precise task, though sometimes this task comes without a story (for instance, when modifying a known algorithm, or borrowing from mathematics), and sometimes it comes without an interesting solution (such as when drawing on games and puzzles).

On the other hand, some techniques do not provide a specific task so much as a vague idea; this is particularly true of looking around (Section 4.1) and building from abstract ideas (Section 4.5). Although these techniques do not give a precise starting point in our problem space, they do specify a region in this problem space in which we hope to find interesting tasks.

6.3. Moving About the Problem Space

We push our analogy onwards to Section 5 of this paper, where we discussed how problem setters modify their initial ideas until they arrive at a task that they like. This modification process typically involves a succession of small changes to different characteristics of the task.

In our abstract setting, changing a single characteristic of a task corresponds to changing a single coordinate in our problem space. In this way the modification process involves a series of small steps through the problem space to nearby points, until we reach a point whose evaluation function is sufficiently high (i.e., a good task).

As noted earlier, sometimes a problem setter has a vague idea in mind rather than a precise task; in this case a small change may involve pinning down an additional characteristic to make the task more precise. In our abstract setting, this corresponds to

reducing the dimension of a region in the problem space, thus making the region smaller (with the aim of eventually reducing it to a single point).

Problems setters typically do not change a task just once, but instead try many different changes, keeping changes that improve the task and discarding changes that do not. Gradually these changes become smaller as the task becomes better, until the problem setter is fully satisfied.

From an algorithmic point of view, this is reminiscent of *simulated annealing*, one of the most well-known optimisation algorithms. Since simulated annealing is an efficient method of optimisation, we can hope that the techniques described in this paper are likewise an efficient way of creating new and interesting tasks.

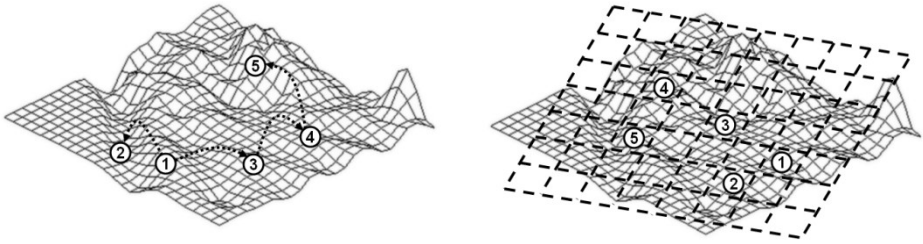

Fig. 9. Depictions of the "simulated annealing" and grid exploration methods

This overall process is illustrated in the left hand diagram of Fig. 9. Here the problem space is drawn as landscape, where the height shows the value of the evaluation function. The arrows show the progression of small changes that convert an initial idea (1) into a final task (5).

It is worth noting that simulated annealing does not guarantee a global maximum, though thankfully that is not the goal of the problem setter. The search space has many local maxima, and problem setters can look through these for suitable tasks that meet their criteria.

6.4. The Grid Exploration

The technique described earlier in Section 4.4 (filling the holes in a domain) is worth a special mention here. With this technique we make a list of existing tasks in a domain, examine their different characteristics, and then forge a new combination of characteristics that will eventually become our new task.

This technique fits naturally into our abstract setting. The domain of interest can be seen as a subspace of the larger problem space, and our list of existing tasks forms a collection of points in this subspace. By examining the different characteristics of these tasks we effectively list the "significant dimensions" of this subspace.

As a result we essentially impose a multi-dimensional grid over our subspace, where the different dimensions of this grid represent the characteristics that we are focusing on. By listing the characteristics of each task, we effectively place these tasks in individual grid cells. This is illustrated in the right hand diagram of Fig. 9.

To create a new task, we then search through empty grid cells, which correspond to new combinations of characteristics that we have not seen before. To make our tasks as

original as possible, we aim for empty cells that have few coordinates in common with our existing tasks.

6.5. Final Notes

We have seen in Sections 6.3 and 6.4 how different task creation techniques correspond to different ways of exploring and optimising the problem space. In particular, we can see how these techniques relate to well-known global optimisation methods such as simulated annealing.

As optimisation is an extensively-studied field of algorithmics, it could be interesting to take other optimisation algorithms and try to apply them to the task creation process. This in turn might help the community to produce new and original tasks into the future.

References

Burton, B. A. (2007). Informatics olympiads: Approaching mathematics through code. *Mathematics Competitions*, **20**(2), 29–51.

Charguéraud, A., M. Hiron (2008). Teaching algorithmics for informatics olympiads: The French method. To appear in *Olympiads in Informatics*.

Diks, K., M. Kubica, K. Stencel (2007). Polish olympiad in informatics – 14 years of experience. *Olympiads in Informatics*, **1**, 50–56.

Ginat, D. (2002). Gaining algorithmic insight through simplifying constraints. *J. Comput. Sci. Educ.*, April 2002, 41–47.

Kedlaya, K. S., B. Poonen, R. Vakil (2002). *The William Lowell Putnam Mathematical Competition 1985–2000: Problems, Solutions, and Commentary*. Mathematical Association of America.

Larson, L. C. (1983). *Problem-Solving Through Problems*. Springer, New York.

Skiena, S. S. (1998). *The Algorithm Design Manual*. Springer, New York.

Skiena, S. S., M. A. Revilla (2003). *Programming Challenges: The Programming Contest Training Manual*. Springer, New York.

Sloane, N. J. A. (2008). The on-line encyclopedia of integer sequences. Retrieved March 2008 from `http://www.research.att.com/~njas/sequences/`.

van Lint, J. H., R. M. Wilson (1992). *A Course in Combinatorics*. Cambridge Univ. Press, Cambridge.

Verhoeff, T., G. Horváth, K. Diks, G. Cormack (2006). A proposal for an IOI syllabus. *Teaching Mathematics and Computer Science*, **4**(1), 193–216.

Wilf, H. S. (1994). *Generatingfunctionology*. Academic Press, New York, 2nd edition.

B. A. Burton is the Australian IOI team leader, and has chaired scientific committees for national contests since the late 1990s. In 2004 he co-founded the French-Australian Regional Informatics Olympiad with M. Hiron, and more recently he chaired the host scientific committee for the inaugural Asia-Pacific Informatics Olympiad. His research interests include computational topology, combinatorics and information security.

M. Hiron is the French IOI team leader, and is the co-founder and president of France-IOI, the organisation in charge of selecting and training the French team for the IOI. He creates tasks on a regular basis for national contests and online training programs, as well as for the French-Australian Regional Informatics Olympiad, and occasionally for other international contests such as the IOI or the APIO. He is a business owner working on projects ranging from web development to image processing and artificial intelligence.