Problem-Solving Using the Extremality Principle

Jagadish M
Dept. of Computer Science and Engg.
Indian Institute of Technology Bombay
Mumbai 400076, India
jagadish@cse.iitb.ac.in

Sridhar Iyer
Dept. of Computer Science and Engg.
Indian Institute of Technology Bombay
Mumbai 400076, India
sri@iitb.ac.in

ABSTRACT

The extremality principle is one of the commonly used problem solving strategies. It involves looking at the extremal cases of a problem in order to obtain insight about the general structure. Though the principle is widely known, its use in designing algorithms is rarely discussed in CS literature. We present a methodology based on the extremality principle that is useful in solving a wide variety of algorithmic problems. We illustrate the effectiveness of the methodology by deriving solutions to three difficult problems. We believe that the key steps involved in our methodology can be taught to students as individual drills. We have anecdotal evidence for the teachability of the method.

Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computer and Information Science Education

General Terms

Algorithms

Keywords

Problem-solving, Extremality

1. INTRODUCTION

The extremality principle is a problem-solving strategy that involves studying objects with extreme properties in order to reason about more general objects. Although the principle is intuitive and well-known, its application to a specific problem can be difficult. There are many books on problem-solving that illustrate the use of extremality principle in solving mathematical problems ([2], [6]). These books typically contain several topic-specific problems that train students in identifying situations where extremality can be useful. However, the current texts on extremality principle focus mostly on math topics like geometry, number theory and combinatorics.

ITiCSE'14, June 21–25, 2014, Uppsala, Sweden.
Copyright 2014 ACM 978-1-4503-2833-3/14/06 ...$15.00.
http://dx.doi.org/10.1145/2591708.2591718.

In computer science, extremality is discussed only in the context of greedy algorithms. A greedy algorithm makes a local choice at each step that is extremal in some sense. We show that the principle can be used in more general ways. We have devised a problem-solving methodology based on the extremality principle, which we call WISE, that can be used to solve a wide range of problems. WISE stands for "Weaken-Identify-Solve-Extend". For the purpose of illustration, we derive solutions to three hard problems using this methodology. The ideas in our technique are well-known to experts who probably apply them implicitly. Our contribution is to operationalize the extremal principle into a methodology that can be directly taught to students.

2. SOURCE OF DIFFICULTY

In this section, we explain why the extremality principle is too general to be useful directly and how WISE addressees some of the key difficulties. The main difficulty arises due to the multiple ways in which instances can be represented.

Choice of representation of instances. In many cases a single instance can be represented in multiple ways. The extreme values of the instance in one representation may be unrelated to extreme values in another representation. For example, let us take the case of numbers. A number is often represented in decimal notation. However, we could also represent it using a different base system or express each number by their prime factorization. For example, the number 15 can be represented as 1111 in base 2 system or as 3x5 in factorized notation. So the numbers 15, 31, 63 do not seem to be extremal when considered in decimal or factorized representation ($3 \cdot 5, 31, 3 \cdot 3 \cdot 7$), but are extremal when represented in binary (1111,11111,111111). We illustrate the use of representation by an example.

Problem. Consider the program given below. For a given value of `n`, what is the value of `p` upon termination of the program?

```
int main(){
        int n; int p=0;
        cin >> n;
        while(n > 0){
           if( n%2 == 1 ) p++;
           n = n/2;
        }
        cout << p << endl;
        return 0;
}
```

Sol. Let us inspect the behavior of the program for small values of `n`.

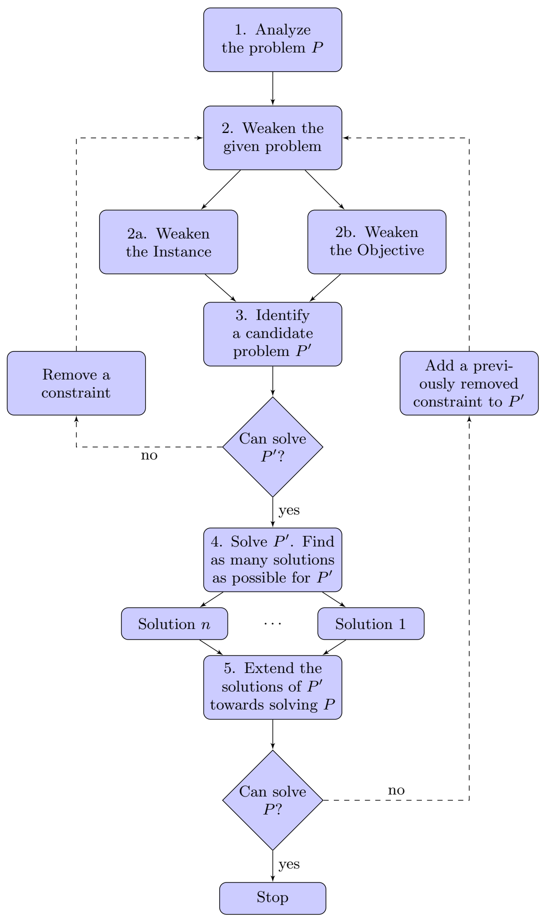

1. Analyze the problem P
2. Weaken the given problem
2a. Weaken the Instance
2b. Weaken the Objective
3. Identify a candidate problem P'
Remove a constraint
Add a previously removed constraint to P'
Can solve P'?
no
yes
4. Solve P'. Find as many solutions as possible for P'
Solution n
...
Solution 1
5. Extend the solutions of P' towards solving P
Can solve P?
no
yes
Stop

Figure 1. The WISE methodology based on extremality principle. (WISE is short for "Weaken-Identify-Solve-Extend")

n	1	2	3	4	5	7	10	15
p	1	1	2	1	2	3	2	4

It is difficult to find the relationship between `n` and `p` from the above table. However, if we switch the represention of `n` from decimal to binary the relationship becomes obvious.

n	01	10	11	100	101	111	1001	1111
p	1	1	2	1	2	3	2	4

It is now easy to see what the program does: At each iteration of the loop, `p` is incremented if the last bit of `n` is 1. Variable `n` is also right shifted by one bit at each iteration. Hence, when the program terminates, the value of `p` contains the number of 1 bits in the binary representation of original value of `n`.

The key to solving the above problem was to find the right representation for numbers. The importance of representation is mentioned in [5] in the context of 'transform and conquer' technique. It plays an important role in our methodology in finding extremal instances.

3. METHODOLOGY

We describe the steps involved in the WISE methodology. We elaborate the steps that are shown in Fig. 1.

Step 1 *Analyze the given problem P*

The first step of the method is to identify the instances, constraints and the objective of the problem.

Instances and constraints in the problem are easy to identify by looking at the *nouns phrases* and *verb phrases* in the problem description, respectively.

For each instance, we select a *representation* and list their *properties*. For example, a graph can be represented either as an adjacency matrix or adjacency list. Graphs have properties like maximum degree of a vertex, diameter, connectivity, etc. The properties may depend on the choice of representation. A property can be intrinsic to an instance or depend on the choice of representation. For example, diameter is a property that is intrinsic to a tree, but height is a property that is applicable only if the tree is represented as a rooted tree. Table 1 shows representations and properties of common instances we encounter in algorithmic problems.

Instance	**Properties (Representation)**
Number	Value, Parity, Sign, Number of prime factors, Number of digits (Decimal), Numbers of bits (Binary).
Lines	Length and Slope.
Array	Length, Values, Number of inversions, etc.
Tree	Maximum degree of vertex, Number of leaves, Diameter, Height (Rooted tree).
Graph	Number of edges, Diameter, Connectivity, Regularity, Planarity, Bipartiteness, Number of cycles, Chromatic number, Girth, Number of overlapping back-edges (DFS Tree) Height (BFS Tree).

Table 1. Common instances and their properties.

Step 2 *Weaken the given problem*

If the problem is hard to solve, we look for a problem that is simpler to solve first. This is one of the most common problem-solving techniques [6]. We describe two common ways of weakening.

Weaken the instances. The most common way to simplify the problem is to retain the objective but restrict the instances to a particular type. In this section, we discuss ways to weaken the instances.

Extremal instances are those which optimize a function on properties subject to some constraints. Due to the number of ways in which we can combine properties, there are several extremal instances one can derive. We list some simple extremal examples in Table 2 that are useful in many problems. Solving the problem on a simple extremal instance usually gives a clue to what other extremal instances might be interesting to consider.

Emergent Properties. Interaction between instances can give rise to new properties which we call as *emergent* properties. For example, a pair of lines can have an *intersection point* which is not a property of either of the lines. Interaction could occur between multiple instances of the same kind or different ones. Since instances could interact in many ways, there could be a large number of emergent properties.

Weaken the objective. In this step, we identify the objective and relax the constraints of the problem. The common way to do this is to relax the individual properties of

Instance	Extremal function	Constraints	Extremal Instance
Number	Min. the number of prime factors	-	Numbers of the form p^n where p is prime
	Min. the number of 1s in bits	-	`10000,01000,00001, etc.`
Lines	Maximize or Minimize slope	-	Horizontal and Vertical lines
Array	Number of values	-	Array consisting of only 1s and 0s
Tree	Minimize the number of leaves	-	Path
	Minimize height	-	Star graph
	Minimize height	Max. degree is three	Complete binary tree
Graph	Maximize number of edges	-	Complete graph
	Minimize number of edges	Preserving connectivity	Tree

Table 2. Examples of extremal objects.

the constraints or operations. It is well known that simplifying constraints by itself can lead to insight [3]. Combining this with extremality makes it more powerful.

Step 3 Identify a candidate problem P'

Once we have a list of weaker problems, we identify all the trivial problems that can be easily solved. Usually, many of these problems do not give much insight. So we need to pick a candidate problem that gives some insight into original problem. The candidate problem is the *simplest non-trivial* problem to which we do not have a solution. If the candidate problem itself is too hard then we go back to Step 2 by and weaken it further. By repeated application of weakening the problem or objective, we obtain a candidate problem P' that retains some aspects of the original problem but is easy enough to be solved. Since the problem can be weakened in several ways, we usually end up with multiple candidate problems. We choose to pursue one candidate problem at a time.

Step 4 Solving the candidate problem P'

In this step, we can apply any of the commonly available techniques like greedy, divide-and-conquer, dynamic programming, etc. to solve P'. It usually helps to solve the candidate problem in multiple ways. Solutions differ in their strengths and weaknesses and may give different insights into the problem.

Step 5 Extend the solutions

We use each solution of the candidate problem to get insight into the original problem. One of the common ways of doing this is to first see if the solution applies to near-extremal instances *i.e.* extremal instances which are slightly perturbed. For example, we can try to extend a solution on prime numbers to numbers with two prime factors. Most often this extension gives an idea that works for arbitrary numbers. Similarly, we may try to extend a solution that works for a path to trees with only two paths. The solution to two paths may generalize to trees with fixed number of leaves and then to general trees.

However, if the solution does not extend to a general case then we add the difficult case to the candidate problem, go back to Step 1 and tackle it as a new problem.

4. ILLUSTRATIVE EXAMPLE I: COINS IN A ROW

We apply the WISE methodology to a problem that appears in the book "Mathematical Puzzles: A Connoisseurs Collection" by Peter Winkler [7]. The book is a collection of hard and interesting puzzles. This is the first of the three problems with which we illustrate the effectiveness of WISE.

Problem. COINS IN A ROW. On a table is a row of fifty coins of various denominations. Alice picks a coin from one of the ends and puts it in her pocket; then Bob chooses a coin from one of the (remaining) ends, and the alternation continues until Bob pockets the last coin. Prove that Alice can play so as to guarantee collecting as much money as Bob.

Step 1 Analyze the problem

We identify the instances, constraints and the objective of the problem. Noun phrases in the problem description usually correspond to instances and verb phrases to constraints and objectives. The cue phrases in the problem are shown in italics below.

Objective Find a strategy for Alice to *collect more money* than Bob.

Constraint Coins must be *picked alternatively* from both ends.

Instances *Coins* and a *sequence* of fifty coins.

Emergent Property. Since the game is deterministic, for any sequence of coins S, there is a unique value which denotes the maximum amount Alice can collect on S, assuming that both Alice and Bob choose optimally. Let *profit* denote the difference between Alice's amount and Bob's amount for a given sequence of coins. Note that profit is an emergent property of the sequence.

Step 2 Weakening

Weaken the instance. One way to weaken the input sequence is to restrict the values of coins to 1s and 0s. Zeroes and ones are extremal because they are the smallest two non-negative values. A sequence of all zeroes or all ones is also extremal, but this makes the problem trivial.
Weaken the objective. There are many options to relax the objective or constraints: Can Alice pick the largest coin always? Can Alice force Bob to pick a particular coin? Can Alice collect at least half the amount as Bob?

Step 3 Identify a candidate problem

Candidate problem. Given a sequence consisting only zeroes and ones, find a strategy for Alice to maximize her profit.

Step 4 Solving a candidate problem

This is the step in which extremality is most useful. We would like to identify problem-specific extremal instances of the candidate problem which can be used to get some insight into the problem.

Here is an extremal instance based on the emergent property profit: *Which configuration consisting of 1s and 0s gives the* maximum profit *for Alice?* The instance is not hard to construct since at each turn Alice should be able to pick 1 but not Bob:

0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1

In the above example it is easy to see that Alice can pick all the 1s and Bob gets zero. So the gain for this sequence is maximum over all sequences of the same length.

If 1s and 0s appear in alternate positions, Alice can always pick all the 1s.

o	*e*	*o*	*e*	*o*	*e*	*o*	*e*	*o*	*e*	*o*	*e*	*o*	*e*	*o*	*e*
0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

In general, Alice can pick all the numbers that are in positions of the same parity, regardless of the values of the coins (Key Idea). Notice how we made this observation simply by solving the *right* extremal problem: that of alternate ones and zeroes.

Step 5 *Extend the Solutions*

Near extremal instance: Alternate 1s and 0s except for one position. Alice can win by collecting 1s if x is less than the number of ones. Otherwise, she can win by collecting x. By generalizing this idea, we see that Alice has a winning strategy: Alice first sums up all the coins in even positions and all the coins in odd positions. Then, she will pick up all coins of the same parity that gives her a larger sum.

o	*e*	*o*	*e*	*o*	*e*	*o*	*e*	*o*	*e*	*o*	*e*	*o*	*e*	*o*	*e*
0	1	0	1	x	1	0	1	0	1	0	1	0	1	0	1

Sol. Alice either picks all the numbers in even positions or odd positions (whichever is greater).

5. EXAMPLE II: GRID MINIMUM

This problem is taken from the textbook by Klienberg-Tardos [4]. The authors mention that problems in each chapter appear roughly in increasing order of difficulty. Grid Minimum is the last problem in the chapter on divide-and-conquer technique.

Problem. Let G be an $n \times n$ grid graph. Each node v in G is labelled by a real number x_v; you may assume that all these labels are distinct. A node v of G is called a *local minimum* if the label x_v is less than the label x_w for all nodes w that are joined to v by an edge. You are given the grid graph G, but the labelling is only specified in the following *implicit* way; for each node v, you can determine the value x_v by *probing* the node v. Show how to find a local minimum of G using only $O(n)$ probes to the nodes of G. (Note that G has n^2 nodes.)

Step 1 *Analyze the problem*

We first identify the instances of the problem.

- Instances in the problem: Grid graph with values.
- Extremal instances
  - An $1 \times n$ grid graph (extremality in structure.)
  - A grid graph with only one node that is a local minimum (extremality in values.)

Step 2a *Weakening the instance*

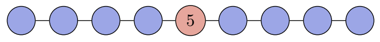
Let us consider the extremal instance when the grid graph is of size $1 \times n$. This means the graph is just a path.

Step 3 *Identify a candidate problem*

The candidate problem is as follows: What is the least number of queries in which we can find a local minimum on a path?

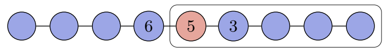
Step 4 *Solving the candidate problem*

Suppose we probe the middle node m and find its value to be 5 (say). If this node is the local minimum, then its neighbors must be larger than 5.



To check if the node m is a local minimum, we compare its value with its adjacent nodes. If m is smaller than both the neighbors, then we are done; otherwise we do the following:

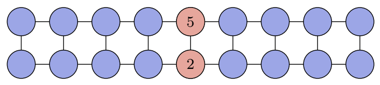
Without loss of generality, assume that the right node is smaller than the middle node. Observe that the node with the smallest value in the right half (outlined) of the path must be a local minimum, so the right half of the path contains at least one local minimum. With one probe, we can reduce the size of the path by half. We can find a local minimum in at most $O(\log n)$ probes by recursing.



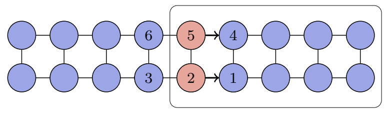
Step 5 *Extend the solutions*

Can we generalize the above idea to a $2 \times n$ size grid?

Suppose, we probe the middle nodes of the $2 \times n$ grid and find that one of them is a local minimum, then we are done.



Otherwise, it means that each middle node is adjacent to a node that has smaller value. Consider the case when both the smaller adjacent nodes are on the same side of the middle nodes (say right side as shown below). In this case, we know that there exists a local minimum in the right half of the gird (outlined portion), due to the same reason as above: the smallest valued node in the right half is definitely a local minimum. Hence, the same idea that we used for a path extends to this case. We can recurse on the right half of the grid since it is assured to contain a local minimum.



However, this idea does not extend to the case when the smaller adjacent nodes are in opposite directions (as shown in Fig. 2). This case is perplexing because we do not know if we can discard the right half or the left half of the grid (if at all we can discard). Let us add this difficulty and consider it as our next candidate problem.

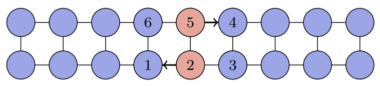


Figure 2. Should we recurse on the left half or the right?

Step 3 *Identify a candidate problem*

Candidate Problem 2: *Given a grid graph of size $2 \times n$ and the values of middle nodes and their neighbors, determine which half of the grid contains a local minimum.*

Step 4 *Solving the candidate problem*

We consider the difficult case when the smaller neighbors of middle nodes are on the opposite sides as given in Fig. 2.

Instead of approaching this problem directly, we use extremality to get some insight. Our instance is a $2 \times n$ grid. An extremal instance of a grid may be a grid with *only one* local minimum. Is it possible to extend the grid shown in Fig. 2 such that it has only one local minimum?

Note that if a node is not a local minimum, then one of its neighbors has a smaller value than itself. We can use this fact and complete the grid in Fig.2 with values such that it has only one local minimum. One such extremal case is shown in Fig. 3, where node 1 is the only local minimum.

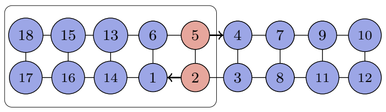


Figure 3. Node 1 is the only local minimum.

Note that the local minimum has appeared on the *left side* of the grid graph. We observe that no matter how we try to fill up the values for the remaining nodes in Fig.2, we always end up with a local minimum on the left side of grid graph. So in some sense, the middle node 2 seems to have more influence than middle node 5.

Key Fact. Node 2 is the minimum node among the middle nodes 2 and 5.

Here is the reason why the local minimum always appears on the left side of the grid graph shown in Fig.2: Suppose node 1 is a local minimum, then we are done. Otherwise, let node l be the minimum valued node in the left half of the graph (outlined portion in Fig. 3). Note that node l is less than *all* the nodes in the left half of the grid graph including the middle nodes 2 and 5. Therefore, node l is a local minimum. So in either case, we get a local minimum on the left side of the grid graph.

Using the above observation, we can now answer candidate problem 2. Suppose we are given a $2 \times n$ grid with middle nodes a and b. Without loss of generality, assume $v_a < v_b$. Let x be the neighbor of node a with $v_x < v_a$. Then there always exists a local minimum in that half of the grid graph that contains node x.

Step 5 *Extend the solutions*

The solution to the $2 \times n$ grid gives enough insight to solve the original problem. We give a divide-and-conquer algorithm below:

Sol. Given an $n \times n$ grid, we probe all middle nodes and check if one of the middle nodes is a local minimum. If this is true, then we are done. Otherwise, we find the minimum valued node among the middle nodes (call this minimum valued node m). Since m is not a local minimum, there exits a node x, either to m's right or left such that $v_x < v_m$. Let G' be the half of the grid than contains v_x. G' is a $n \times n/2$ grid. Apply the probing procedure again to G' and partition it into two halves. Let G'' be the half that contains the local minimum. Recurse on G''. This is the divide-step of the algorithm that uses only $3n/2$ probes and reduces the problem into an instance of size $n/2 \times n/2$. The associated recurrence relation is

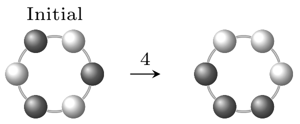
$$T(n) = T(n/2) + 3n/2$$

which implies that the algorithm runs in linear time.

6. EXAMPLE III: BEADS

This problem was posed in a regional ACM-ICPC competition that was held in the authors' institute. None of teams were able to solve it during the competition. We consider the problem to be hard since most teams that take part in ACM-ICPC are competent in common algorithm techniques.

Problem. You are given a circular necklace containing n beads. Each bead maybe black or white. You have to rearrange the beads so that beads of the same color occur consecutively. The only operation allowed is to cut the necklace at any point, remove some number of beads from both ends and put them back in any order, and join the cut ends. The cost of this operation is the number of beads removed. Any number of such operations can be used. You have to find a sequence of such operations with minimum total cost for a given initial distribution of beads. For example, if the initial string is `wbwbwb`, this can be done by a single operation of cost 4. Design a linear-time algorithm for this problem. (Problem due to Ajit Diwan.)



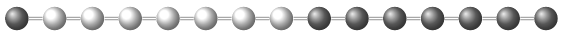
Step 1 *Analyze the Problem*

- Instances in the problem: Necklace.
- Property of the necklace: Number of misplaced beads.

Step 2a *Weakening the instance*

To simplify we use an array of beads instead of a circular necklace.

Extremal instance. Consider the problem on an array of beads with only one misplaced bead.



Step 2b *Weakening the objective*

Can we solve the problem if we allow the cut-and-join operation to be performed only *once*?

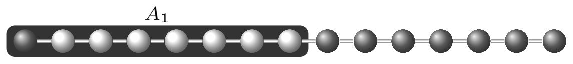
Step 3 *Identify a candidate problem*

Candidate Problem: Given an array of beads with one misplaced bead find the minimum cost by which we can sort the beads using one cut-and-join operation.

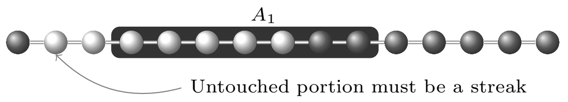
Step 4 *Solving the candidate problem*

Def. Let a maximal contiguous sequence of same colored beads be called a streak.

Observe that it is possible to solve this instance with cost of 8. But how can we prove that this is the minimum cost?



What if there exists a cut-and-join operation in the middle that achieves smaller cost? This not possible since the *untouched portion* of the array must be streak. If there are misplaced beads in the untouched portion, they will continue to remain so after the operations.



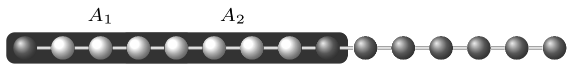
Step 5 *Extend the solutions*

In general, the untouched portions of the array must be streaks. Further, there cannot be a streak of a different color in between two streaks of the same color as this would also result in misplaced beads after the operation.

Adding a difficulty

Candidate Problem 2: Given an array of beads with one misplaced bead find the minimum cost by which we can sort the beads using *two* cut-and-join operations.

Would the minimum cost of operation remain 8, if we allowed two cut-and operations? Would it be possible to achieve lower cost by using two smaller operations?



Suppose A_1 and A_2 are two operations. The cost of these operations would be $|A_1 + A_2|$ which is more than 8 if the operations overlap. Hence, it is not useful to have to consecutive overlapping operations.

This idea coupled with the idea above gives enough information to resolve the problem.

Sol. Suppose the initial configuration of beads is represented by an array $A[1..n]$. The cut-and-join operation can be seen as rearranging a sub-array $A[i..j]$ while paying a cost of $j - i + 1$, allowing the sub-array to wrap around.

Suppose $A_1, A_2, \dots, A_n$ is an optimal sequence of cut-and-joins. If A_i and A_{i+1} overlap or are next to each other they can be combined into one cut-and-join operation without increasing the cost. So there exits an optimal sequence in which A_is are separated from one another by streaks. Each A_i has at least one black bead and one white bead. There cannot be more than two A_is since this would result in two streaks of the same color after rearrangement. Since our goal is to get one streak of each color, the optimal strategy is to leave the maximum streak of white beads and maximum streak of black beads untouched and use (at most) two cut-and-join operations to rearrange the rest of the beads. This can be done in $O(n)$ time.

7. DISCUSSION ON TEACHABILITY

There are many more problems that can be solved using the WISE methodology. We have chosen to discuss three difficult problems as illustrations. The authors will be happy to share more examples upon request. Such illustrations can also be used for teaching WISE, through a cognitive apprenticeship model. The cognitive apprenticeship model is known to be a pedagogically sound way of teaching problem-solving [1].

We believe that the key steps involved in WISE can be taught to students as individual drills. The key steps involved in WISE are pervasive in all problem-solving techniques and not just limited to CS topics. Mastering these steps is likely to help the students in other areas too. These skills are not sufficiently emphasized in the current practice of teaching. For example, being able to formulate weaker problems is an important skill in any kind of problem-solving. However, there are hardly any questions in textbooks whose objective is to come up with a list of weaker problems for a given problem. Most instructors assume that this step will be implicitly done as part of solving a problem. In our preliminary experiments, we find that students tend to approach problems too directly and often get stuck in blind alleys.

WISE focuses on the method of inquiry rather than directly looking for a solution. This helps in the case of hard problems, when one cannot come up with the required insight directly. Often getting unstuck involves picking up a new line of approach. WISE does this by encouraging students to look at several simpler candidate problems and also gives explicit instructions for coming up with such problems. We have anecdotal evidence to show that the steps involved in WISE can be taught to students as individual drills. Based on our preliminary experiments, we find that students able to apply the first three steps of the WISE method but have difficulty in the 'Extend' step. In our future work, we plan to give guidelines to overcome this difficulty.

8. REFERENCES

[1] Vanessa P Dennen and Kerry J Burner. The cognitive apprenticeship model in educational practice. *Handbook of research on educational communications and technology*, pages 425–439, 2007.

[2] A. Engel. *Problem-solving strategies.* Springer, 1998.

[3] D. Ginat. Gaining algorithmic insight through simplifying constraints. *JCSE Online*, 2002.

[4] Jon Kleinberg and Éva Tardos. *Algorithm Design.* Addison Wesley, second edition, 2006.

[5] Anany Levitin and Mary-Angela Papalaskari. Using puzzles in teaching algorithms. In *Proceedings of the 33rd SIGCSE technical symposium on Computer science education*, SIGCSE '02, pages 292–296, New York, NY, USA, 2002. ACM.

[6] G. Polya. *How to Solve It - a New Aspect of Mathematical Method.* Princeton University Press, Princeton, 2 edition, 1957.

[7] Peter Winker. *Mathematical puzzles: a connoisseur's collection.* CRC Press, 2003.