Should We Teach Algorithms?

M. Ashraf Iqbal and Sara Tahir

***Abstract*—A teacher of Computer Science and Mathematics has two options: use precious classroom time in routine operations and boring formulas, thus killing the interest of students and hampering their intellectual development, or challenge their curiosity by formulating interesting and stimulating questions giving them a taste for independent thinking. The teacher need only provide the building blocks and let students themselves form more complex structures, providing them timely hints when needed. In this paper we demonstrate how a very simple procedure can be used, with minor modifications, as a building block to solve a variety of seemingly unrelated problems in the field of graph algorithms.**

***Index Terms*—Discover, design, problem solving, graph algorithms**

I. Introduction

TEACHING the standard course "*Analysis & Design of Algorithms*" at an undergraduate level in a typical Computer Science program has essentially two objectives. The first objective, dealing with *analysis*, is to familiarize students with existing algorithms. The second one, which is perhaps far more important, is to equip the students with the necessary tools and techniques, and above all the confidence*,* required in solving a non-textbook problem. This second objective, concerned with the *design* of algorithms, is essentially a creative effort containing all the ingredients of a thriller: adventure, excitement, challenge, and suspense.

There is no guarantee that one who critiques literature can learn to write beautiful poetry. Similarly the ability to understand and analyze algorithms does not guarantee that one could become an efficient algorithm designer. The study of the methods and rules of discovery and invention is a field in its own right. Though there are rules of thumb that can be followed to help an individual design an algorithm, there is no precise algorithm available that can be used to design new algorithms. Despite the fact that one cannot guarantee that a student could become an efficient algorithm designer, we believe that the instructor, following our approach and providing proper guidance, can sow the seeds that could blossom into the genius that produces efficient yet astonishingly simple algorithms.

Our experience of teaching algorithms indicates that creativity in algorithm *design* depends, to a large extent, on how we deal with the *analysis* phase. We stress that while we are familiarizing students with existing algorithms, we should not formally *teach* anything. Instead we should encourage, rather incite, students to create algorithms themselves using some very fundamental concepts. The objective is that students should experience the tension and excitement of discovery even during the initial phases of understanding existing algorithms. Polya [1] remembers the time when he was a student himself: he was always perturbed by the question: *Yes the solution seems to work, it appears to be correct; but how is it possible to invent such a solution? How could I invent or discover such things by myself?* We feel that with the availability of some pre-requisite knowledge, timely hints, and stimulating questions posed by the instructor, one can always encourage students to redesign an algorithm right from scratch. It is important now to find a good working definition of *design* (of algorithms).

A. What is Design?

According to the Webster's dictionary [2], *design* is to *conceive and plan out in the mind.* In the words of Miller [3], *Design is the thought process comprising the creation of an entity.* Rine [4] defines *design* as a *systematic, directed set of decisions that are introduced, made and deployed, leading to an effective or efficient outcome, solution, or technology.* The last definition suits our discovery based learning approach in which a teacher formulates a directed set of questions and hints in order to help his/her students *design* algorithms. It is interesting to note that our approach is similar in some respects to the so-called *Moore Method* of teaching and learning.

B. The Moore Method

R. L. Moore was a professor of mathematics at the University of Texas. In the words of Hale [5]: *What was so special about his mode of teaching was that he did not lecture, he did not profess. He sat in the back of the room, mostly quiet, occasionally asking a question, allowing his students to find the answers in their own ways.* Many professors still use his teaching style not only in his subject of specialization (topology), but in analysis, algebra, game theory, and other courses, and have advanced or modified the Moore Method in a number of ways [6], [7]. Taylor [8], while characterizing (his version of) the Moore method of teaching, does not allow collective effort on the part of the students inside or outside of class. He also does not allow the use of any source material. We, on the other hand, encourage lively discussions inside as well as outside the classroom. The teacher, in our model, starts with something (very simple), and then actively guides the students in their path of discovery.

In this paper, we provide a detailed study of a number of graph algorithms that have applications in diverse

Manuscript received October 24, 2002; revised July 18, 2003.

This work was supported by a research grant from the Lahore University of Management Sciences, Lahore, Pakistan. A brief version of this paper was presented at INMIC 2001 (IEEE National Multi-topic Conference, 2001).

M. Ashraf Iqbal is with the Department of Computer Science, Lahore University of Management Sciences (LUMS), Lahore-54792, Pakistan (e-mail: aiqbal@lums.edu.pk).

Sara Tahir was with the Department of Computer Science, Lahore University of Management Sciences (LUMS), Lahore-54792, Pakistan. Currently she is pursuing an M.S. degree in Management Science and Engineering at the Stanford University, US (e-mail: stahir@stanford.edu).

Publisher Item Identifier S 1682-0053(03)0186

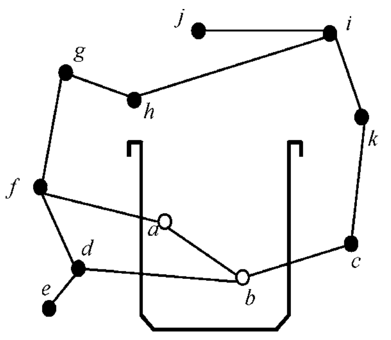

Fig. 1: The Bucket B after first iteration through Steps 2-4.

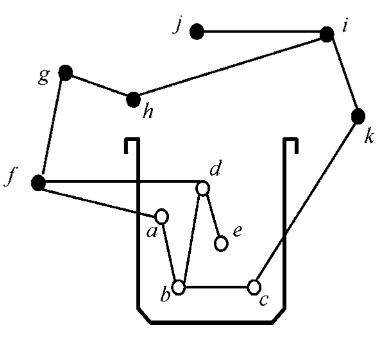

Fig. 2: The Bucket B after 4 iterations through Steps 2-4.

fields like chemistry, biology, mathematics, engineering, social sciences, and also computer science. We start with a simple algorithm known as the *Bucket Algorithm* (the bucket symbolizes a friendly container where a child puts every new toy or every new discovery) consisting of just four lines of pseudo code [9], [10]:

Bucket Algorithm (Input: G, Output: a Bucket B)

```
1. Put any vertex $x$ of Graph G in the Bucket B.
2. While there are edges coming out of the Bucket B.
3.    Select an edge connecting $u$ **in** B to $v$ **not in** B.
4.    Put $v$ in B.
```

We shall show how this primitive procedure can be used to reinvent a number of existing powerful algorithms in graph theory [11]-[17]. With some encouragement from the instructor, the students should develop a keen desire and ability to understand the motives behind, and the procedures followed in order to arrive at innovative solutions. They would learn the ways and means of devising their own algorithms. Specifically the Bucket Algorithm would be used to solve the following problems:

1. Find if a given graph is connected
2. Find the number of connected components of a graph
3. Find if a graph is a tree
4. Find a bridge in a graph
5. Find a path between two vertices in a graph provided a path exists
6. Find a spanning tree of a graph
7. Find a minimum-spanning tree of a graph:
8. Rediscovering Prim's Algorithm
9. Rediscovering Kruskal's Algorithm
10. Solve the single-source shortest-paths problem: Rediscovering Dijkstra's Algorithm
11. Conduct a breadth first search in a graph
12. Conduct a depth first search in a graph

We believe that it is possible to apply this approach in other fields as well. This technique has already been practiced with varying degrees of success in teaching subjects like Electric Circuits, Digital Circuit Design, Semi-conductor Theory, Discrete Mathematics, Computer Organization, and Data Structures, to name a few. The motivation is that once the tutor provides students with the building blocks and the confidence needed for the creative process, it becomes almost certain that the students would be able to arrive at the right conclusion with minimal direction provided by the instructor. The role of the teacher is, however, redefined: (s)he is certainly not required to reproduce what is given in the textbook: rather provide missing links in the jigsaw puzzle such that the students may recreate the bigger picture themselves.

II. Understanding the Bucket Algorithm

The Bucket Algorithm is simple and straightforward. It is just a 4-line algorithm with a simple *while* loop with no recursion. We start with something simple but potentially very powerful. Simple, because it is easy to understand and at the same time flexible enough to handle a variety of problems belonging to different categories.

A. How Does it Work?

We identify a Graph G and a Bucket B (See Fig. 1). Step 1 instructs us to put any node, say node a, of the Graph G into the bucket. Next we choose any edge joining a to any other node, say node b, in the graph (since all other nodes are currently outside of the bucket) and put b in the bucket. See Figure 1 for a picture of what the Bucket B will look like at this stage. Now we have a set of nodes $\{a, b\}$ in the bucket giving rise to a set of edges $\{(a, f), (b, c), (b, d)\}$ to choose from in Step 3 as we iterate through the w*hile* loop.

Fig. 2 shows the bucket B after 4 iterations through the *Bucket Algorithm*. Notice that there are two types of vertices: those inside the bucket represented by the set $\{a, b, c, d, e\}$, and those outside the bucket, the set $\{f, g, h, i, j, k\}$. These two different kinds of vertices give rise to three different kinds of edges. The first is the set of edges connecting vertices inside the bucket with each other: $\{(a, b), (b, c), (b, d), (d, e)\}$. The second is the set of edges connecting vertices outside the bucket with each other: $\{(f, g), (g, h), (h, i), (i, j), (i, k)\}$. The third is the set of edges (the "branches coming out of the Bucket B" in Step 2) connecting vertices inside the bucket to vertices outside the bucket, the set $\{(a, f), (c, k), (d, f)\}$. An edge belonging to this last set of edges is called a *cross edge* and is of most interest to us. Depending on the constraint we place on the selection of cross edges in Step 3, we can implement numerous algorithms.

B. Playing with the Algorithm

The instructor should encourage students to play around with the *Bucket Algorithm* to get comfortable with it. During this activity the instructor should ask thought provoking questions such that the students focus on multiple facets of the algorithm that would later help in designing new algorithms. Such questions could be:

1. *Under what conditions there would be no edges coming out of the Bucket?* Note that this condition should be met otherwise the algorithm would never terminate.

2. *Would all the vertices of the graph move into the bucket after the completion of the algorithm? When would this scenario be true, and when would it be false?*
3. *Does it make any difference if we have a different starting vertex?* Note that there are situations when it really makes a difference.

Students must come to realize the importance of *cross edges*: it is because of this cross edge *(u, v)* we select in Step 3 that we discover the new vertex *v*.

C. Solving Other Problems

The above questions would induce a deeper understanding amongst students about how the *Bucket Algorithm* works under different conditions and give some hints while solving more complex problems. After the students are confident that they understand the idea behind the Bucket Algorithm, the instructor can start asking them to modify it to solve more complex problems. What is the worst-case complexity of this algorithm? It is recommended that the instructor not involve the underlying data structure at this stage in order to tackle the issue of complexity. It becomes essential to include it at a later stage.

D. The Right Provocation

It is well known that a real understanding of the problem is a necessary condition to solve any problem. According to Rine [18], *" Half way home to solving a problem is a clear understanding of the problem*". Out of a sequence of six questions posed by Sakiena [11] in order to guide one to discover the right algorithm, the first question is *Do I really understand the problem*? Then comes the role of the teacher in terms of how he/she states a problem and *provokes* (or guides) his/her students to solve it in a specified manner. For example if a teacher is talking about Quick Sort, he/she cannot expect his/her students to discover the said algorithm just after understanding the sorting problem. The teacher should first make the students appreciate the need of *partitioning* the array into halves such that all numbers in the first half are smaller than each number in the second half. *Why we should do this* and *how should we do this* are both equally important to design, discover (and even understand) the said sorting algorithm. The understanding of the previous state of an abstract system and the (usefulness of the) final system state after the application of a so called *fundamental operation* [10] (for example the partitioning procedure in Quick Sort) is crucial in problem solving in computer science as in other disciplines.

III. Find if a Graph is Connected

Assuming that the students know what a connected graph is, the instructor should ask the students: *"Can you modify the Bucket Algorithm such that you may be able to determine whether a given graph G is connected?"* The emphasis should be using the existing techniques with minimum modification. The answer is simple: if, after the *Bucket Algorithm* has been applied to a graph G, there are still any nodes left outside the bucket the graph is not connected. If, however, all nodes are inside the bucket, then graph *G* is connected.

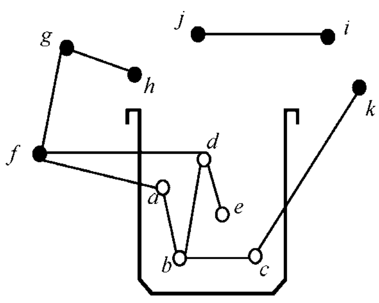

Fig. 3: A graph G that is not connected. Once the Bucket Algorithm terminates, nodes *i* and *j* will be left outside the bucket B.

Notice while students were becoming familiar with the *Bucket Algorithm*, the instructor asked when there would be nodes left outside the bucket. Brighter students would have been able to identify at that stage that some nodes will be left outside the Bucket *B* when a graph is not connected since no cross edges exist connecting them to nodes inside the bucket (Fig. 3).

Not all students may be able to identify this property of the Bucket Algorithm. The instructor in this case will have to make an extra effort to guide such students. Once all students have understood the solution (having arrived at it on their own with well-timed prodding from the instructor) the instructor should start the discussion regarding cost calculation, i.e., the complexity of the modified algorithm.

IV. The Number of Connected Components

Once the students understand how to find if a graph is connected the above problem becomes simple and very little imagination is needed to answer the above question. Applying the Bucket Algorithm once on a graph with more than one connected component would tell us that the graph is not connected as all the vertices do not end up in the bucket. The vertices that do end up in the bucket belong to a single connected component. Applying the algorithm again with a new bucket would give us a new connected component, and so on and so forth. The number of times we have to apply the Bucket Algorithm depends upon the number of connected components, and this would determine the worst-case time complexity.

V. Find a Bridge in a Graph

A *cut edge* or *bridge* is one whose removal produces a graph with more connected components than the original. There are essentially two different problems here; it is the job of the instructor to at least identify them for those students who cannot visualize the solution immediately. The first problem is to check if a *given edge* is a *bridge*. This could be solved if we remove the given edge and then check the number of connected components in the resulting graph. What would be the resulting complexity of this algorithm? The second problem is to *find* or *locate a bridge* in a given graph. Once the first problem is solved it should be a simple matter to handle it. How many times the Bucket Algorithm is applied and what is the resulting worst-case complexity of the algorithm?

VI. Find if a given Graph is a Tree

The algorithms that solve this problem depend on how we define a tree. This in not only true for this problem but is true for a majority of problems. It highlights the fact that looking at various definitions or properties is sometimes extremely useful and it provides the seed for designing a number of very powerful algorithms. Solving a problem from different angles and then making a comparison is the single most important exercise for a student studying algorithms (Rawlins [14]).

A. Every Edge in a Tree is a Bridge

We know that a tree of *n* vertices consists of bare minimum number edges, which makes it a connected graph. This implies that removing any edge would disconnect a tree. Thus every edge in a tree is a bridge. We already know how to check if a given edge is a bridge in a graph. The problem is thus reduced to repeatedly applying the algorithm in Section V. The number of times we would have to do this and finding the resulting complexity is an interesting exercise by itself.

B. The Number of Edges in a Graph

We can define a tree in a number of ways. In fact, one definition implies another. For example, a connected graph is a tree provided the number of edges in the graph is exactly equal to one minus the number of nodes in the graph, i.e., n-1. The catch is that the graph should be connected otherwise the definition would not apply. We know how to find if a given graph is connected using the Bucket Algorithm. So the problem is reduced to counting the number of edges. How complex is this problem? Is it possible to count the number of edges while we are checking if the given graph is connected? Would that perhaps reduce the complexity?

C. The Spanning Tree of a Tree

We know that a tree has the minimum number of edges required to connect a given number of vertices. A spanning tree of a given graph also satisfies this property, as it is a tree. Thus the spanning tree of a tree would be exactly the same tree. This definition or property can be used to design an algorithm to check if a given graph is a tree.

D. A Comparison

A comparison of all these algorithms would be extremely beneficial to the students if they were encouraged to work it out independently. Once they have the answers it would again be stimulating for them to compare their findings with their colleagues within the classroom. Encouraging and initiating interesting discussions and even heated debates is one of the most important responsibilities of a teacher: (s)he must simply coordinate and make sure that the interaction is moving in the right direction. Only once the students have gained confidence that they understand the basic problem and can find an efficient solution should we move to more complex problems such as finding whether a given graph is a forest.

VII. Find the Spanning Tree of a Graph

The algorithm that we design to solve this problem depends on how we visualize the development of a spanning tree. We can start with the original graph and start with pruning or removing edges until the graph becomes a tree. Or, we can start with no edges and start growing edges until we get a tree. It is also possible to identify some of the so-called cross edges, which would constitute the spanning tree. The resulting complexity would change dramatically depending upon the approach used. Each approach has its merits and demerits and the comparison itself is very stimulating especially because each approach has more advanced applications.

A. Cutting Edges

If we remove all redundant edges from a given graph and just keep edges essential to keep it connected the remaining graph would be a spanning tree of the given graph. This idea would give birth to an algorithm: Remove all edges that do not disconnect the given graph. What would be the worst-case time complexity of this algorithm?

B. Growing Edges

We start with no edges at all but with *n* isolated vertices. We add edges out of the edge pool of the graph such that the resulting graph remains a tree. This approach is opposite to the one discussed above: instead of pruning we are growing edges. In the earlier approach we should be careful and should not disconnect the graph. In the second approach we should be careful not to create cycles in the graph. In each case the Bucket Algorithm helps us. How many times we use the Bucket Algorithm would eventually decide the overall worst-case complexity.

C. Selecting Edges

While running the Bucket Algorithm, we might have noticed that every time we discover a new vertex it is because of a cross edge (step 3), and that the number of such cross edges would be exactly equal to n-1. If we just keep a record of all such edges we might get the spanning tree of the given graph. How efficient would this be if compared with the algorithms described earlier?

VIII. Find a Minimum Spanning Tree

There could be many non-isomorphic spanning trees possible for a given graph: each approach that we have described for finding a spanning tree of a graph was flexible and there was a lot of maneuvering possible within it, thus giving rise to different spanning trees. What if we find all distinct spanning trees of a given graph using any approach and then select the one with minimum weight? Why is this approach, which looks at all possible solutions and then selects the one of our choice, not feasible?

A. Cutting or Growing Edges

Each algorithm used to find a spanning tree in the previous section could be used with proper modification to find a minimum-spanning tree of a connected and weighted graph. While cutting edges we select the edge of maximum weight (provided it does not disconnect the graph), having first sorted the edges in descending order of weight. This would give rise to an algorithm very similar to Krushkal's. Similarly, we can grow edges starting from the edge of minimum weight (making sure no cycle is created). Please note that here we are not using any fancy data structure since the objective is not to have a complicated design, unlike the approach used by some books.

B. Selecting Edges: A Greedy Algorithm

While forming a spanning tree we can select any cross edge. In order to form a minimum spanning tree, we should try to include edges of less weight thus excluding those of higher weight. It follows that among all cross edges that we may select we should pick the one of minimum weight. Using this simple technique the Bucket Algorithm can easily be modified to find a minimum spanning tree of a weighted directed or undirected graph (Fig. 4).

It is important that the minimum spanning tree problem is an optimization problem in which we intend to minimize the sum of weights of all edges in the spanning tree. In order to minimize the global sum, we are trying to minimize a local quantity. We are lucky this time: a so-called greedy approach is working optimally and is in fact optimizing the global sum also. However, although greedy approaches are relatively efficient (being based on local conditions only), they are not always optimal.

C. The Magic of Prim's Algorithm

It would be useful if the students were asked to prove that this greedy approach would actually find a minimum spanning tree. Without reading proofs given in the textbook they should come up with something of their own making. A lively discussion can be initiated to find the merits and demerits of individual work. They should also be asked to derive the time complexity of this approach. It would be useful if they compare this approach with Prim's algorithm. In fact, the two approaches look identical. However the time complexity of Prim's algorithm is better. Why? The reason is in fact more exciting because Prim's algorithm is not just greedy, there is something else, something magical which cuts down the time complexity for not so obvious reasons. What is that magic? How and why it is working? Can this magic be used elsewhere and under what conditions?

IX. Find a Path between Two Vertices in a Graph

It is possible to find a path between two vertices provided the graph is connected. Now instead of checking whether the graph is connected or not, we had better check if the two given vertices belong to a single connected component. If we keep moving on the edges connecting one vertex to another within the graph, a time would come when we would reach our destination. What is wrong with this approach? If there are cycles in the graph it is possible that we never reach our destination. What if there are no cycles in the graph – what if we first make a spanning tree of the graph? Even now it would be difficult to find a path, since we might have to do a lot of backtracking.

A. Cutting Edges

If we remove all redundant edges from a given graph and just keep the edges essential to keep the two vertices connected, the remaining graph would be a "*straight forward*" path between the two vertices in the given graph. What would be the worst-case time complexity of this algorithm? Note that we have used a similar technique to find a spanning tree of a graph. It would be useful to pinpoint the similarities and also the differences.

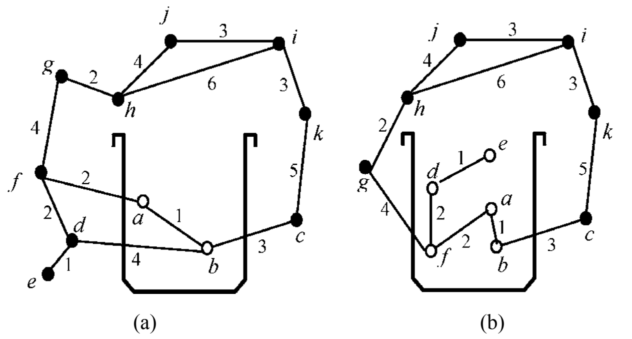

Fig. 4: Minimum spanning tree: the greedy approach (a) next edge selected: *(a, f)*, (b) next edge selected: *(b, c)*.

B. Selecting Edges

Does the problem become simpler if we first find the spanning tree of the given graph? Now if we start moving from the given vertex to the destination vertex, would it be less confusing? Perhaps, but again we may start our journey in the wrong direction and would have to backtrack. Students should experience this confusion and the resultant backtracking. Suppose we apply the Bucket Algorithm starting with the given vertex: the spanning tree thus formed would originate from the given vertex since the given vertex would be the root. We also keep a record of the parent of every vertex in the spanning tree. With this additional information would it be easier to find a path from the given vertex, now the root, to the destination vertex? The answer is still "no" because a parent may have multiple children, and thus many diversions. However if we start from the destination vertex and keep selecting the parent vertex, we would eventually reach the root without any confusion.

X. The Shortest Path Problem

If all edges in the graph were to have the same weight, would the path, found using the algorithms of section IX, be a shortest path? If not then what should be done to achieve our objective? Note that it is easier to find a shortest path in a graph with uniform edge weights, so first we should solve a simpler problem before attacking a more complex one. Now assume that the edge weights are different. Do we need a different algorithm from the one used to find a shortest path in a graph with uniform edge weights? Why?

A. First Find a Minimum Spanning Tree

If somehow we remove edges of higher weights from the graph without disconnecting the two given vertices, would the problem become simpler? What if we first find a minimum spanning tree of a graph and then move backwards from the destination to the source vertex as described in Section IX B?

B. Shortest Path from one vertex to every other vertex

Assume the given vertex goes in the bucket first. Now examine all cross edges that would be coming out of the given vertex with different weights. Identify the cross edge with minimum weight. Let us name the vertex on the other side of this edge *i*, and the weight of this edge *w*. What would be the length of the shortest path from the given vertex to vertex *i*? Under what conditions is above our solution? Note that there are situations when the above procedure would not yield the shortest path. The experience gained while answering the above questions can be utilized to modify the Bucket Algorithm to solve the problem. It is interesting to note that we are inching towards Dijkstra's Algorithm.

XI. Graph Traversal Techniques

Again, it is possible to traverse a graph in a haphazard manner. Efficiency demands that we do not visit the same vertex again and again. We must make every move in a systematic manner to ensure that we do not miss out any vertex belonging to the same connected component [6]. You might have noticed that the Bucket Algorithm is essentially a graph traversal algorithm. However, in most of the current textbooks, you would only find some very specific techniques like the *Breadth* and *Depth First Search* traversal algorithms – more complicated than our Bucket Algorithm.

A. Traditional Techniques & the Bucket Algorithm

It is interesting to note that the Breadth as well as Depth First Searches are two different implementations of the Bucket Algorithm. While introducing this algorithm we purposely did not disclose the implementation details ignoring the underlying data structure required to program the algorithm. The objective was to highlight the basic idea and initially suppress the programming details. Baase [13] uses JAVA to describe algorithms and this may be one reason why the book is relatively difficult to read even if students have prior knowledge of the language. Cormen [12] and Skiena [11] use a pseudo programming language and operate at a slightly higher level. The Bucket Algorithm is simple because it is more abstract and flexible.

B. The Underlying Data Structure

We know that we use a *cross edge* to discover a new vertex in the Bucket Algorithm (step 3). Some of these cross edges come from vertices that entered the bucket earlier, others from vertices that are new comers in the bucket. Our decision of which vertex to choose would convert the Bucket Algorithm into a Breadth First Search, Depth First Search, or a combination of the two. Using a Last in First Out (LIFO or a stack) or a First In First Out (FIFO or a queue) data structure to store the already discovered vertices would make all the difference: a stack implementation would convert the Bucket Algorithm into a Depth First Search while a queue would transform it into a Breadth First Search.

XII. Conclusions

The most important task of a teacher should be to enable the students to discover and acquire experience of independent work. According to Polya [1]: *If the student is left alone with his problem without any help or with insufficient help, he may make no progress at all. If the teacher helps too much, nothing is left to the students. The teacher should help, but not too much and not too little, so that the student shall have a reasonable share of the work.*

In this paper we have demonstrated how a teacher can help students discover a number of graph algorithms with some initial help, starting with something seemingly simplistic yet capable of being transformed into a number of powerful algorithms with minor modifications. We have shown that by asking thought provoking questions it becomes possible for the teacher to guide the students while solving difficult problems. We have also shown that making comparisons between various techniques and solutions provides a deep insight which itself is very useful in solving otherwise difficult problems [19]. It is also important to differentiate between complex problems and difficult problems: *complex problems* or NP-complete problems are those for which no polynomial-time algorithm has yet been discovered [11], [12]. On the other hand, a problem may be *difficult* (to solve) simply because it is not well understood or the students fail to find a suitable strategy with the existing tools and techniques without external help.

At times it is almost impossible to solve a given problem while it is easy to solve a related problem (the shortest path problem is solvable while the longest path problem is unsolvable in polynomial time). Similarly sometimes a problem is so complex in its original form while it is easier to solve it while placing certain restrictions (the graph isomorphism problem is solvable for trees but is difficult to solve in general). It is extremely useful to find why a certain technique works under certain conditions and why it fails in others (greedy methods provide optimal solutions in finding the shortest path but fail to find the longest path). The theory of NP-Completeness connects all problems that are NP-Complete: it is also possible to find a useful relationship among (some) solvable problems and this is what we have attempted to do in this paper. According to Hale [5], *There are different kinds of learning, but I refer here to the intellectual kind. To learn means to cause your mind to function in a different way: new memories are created and/or new connections are forged.* These relationships (or connections) provide the algorithm designer a perspective that proves invaluable when solving new problems and analyzing old ones.

Acknowledgement

We are thankful to the Department of Computer Science, Lahore University of Management Sciences for providing support for this research. We wish to specially thank R. Khan for providing motivation as well as inspiration for this research. We are thankful to A. Alvi who has suggested a number of modifications and helped us in making some improvements. We also wish to thank S. Skiena, S. Baase, S. Mahkari, H. Mian,, K. Fahd, J. Ikram, M. Maud, and T. Jadoon for their help and encouragement.

References

[1] G. Polya, *How to Solve it; A New Aspect of Mathematical Method*, Princeton University Press, 1988.

[2] *Webster's Third New International Dictionary*, Merriam-Webster Inc., Springfield, Massachusetts, US, 1981, P. 611.

[3] W. R. Miller, http://www.tcdc.com/dphils/dphil1.htm#top.

[4] D. Rine, *Lectures in Patterns-Directed Design*, George Mason University, 2002.

[5] M. Hale, http://www.stetson.edu/~mhale/teach/index.htm.

[6] D. R. Chalice, "How to teach a class by the modified Moore Method," *Amer. Math. Monthly*, vol. 102, no. 4, pp. 317-321, Apr. 1995.

[7] P.R. Halmos, "The teaching of problem solving," *Amer. Math. Monthly*, vol. 82, no. 5, pp. 466-470, May 1975.

[8] D. Taylor, *Creative Teaching: Heritage of R. L. Moore*, University of Houston, 1972, p. 149.

[9] M. A. Iqbal, *Class Notes for Discrete Mathematics & Analysis of Algorithms*, Lahore University of Management Sciences, Lahore, 2001.

[10] M. A. Iqbal, *Teaching Computer Science*, Technical Report, Department of Computer Science, Lahore University of Management Sciences, Lahore, 2003.

[11] S. S. Skiena, *The Algorithm Design Manual*, Springer-Verlag New York, Inc., 1997.

[12] T. Cormen, C. Leiserson, R. Rivest, and Clifford Stein, *Introduction to Algorithms*, MIT Press, Cambridge MA, 2001.

[13] S. Baase and A. V. Gelder, *Computer Algorithms: Introduction to Design and Analysis*, Addison-Wesley, 2000.

[14] G. Rawlins, *Compared to What*? Computer Science Press, New York, 1992.

[15] K. A. Ross and C. R. B. Wright, *Discrete Mathematics,* Prentice Hall International, 1992.

[16] K. H. Rosen, *Discrete Mathematics and Its Applications*, WCB/McGraw-Hill, 2003.

[17] L. R. Foulds, *Graph Theory Applications*, Narosa Publishing House, 1992.

[18] D. Rine, Lecture, National Computer Conference, 1980.

[19] M. A. Iqbal and A. Alvi, "The Magic of Dynamic Programming," accepted for presentation at the *2003 Int. Conf. on Engineering Education*, Valencia, Spain, 2003.

M. Ashraf Iqbal was born in Multan, Pakistan, in 1951. He graduated in Electrical Engineering from the University of Engineering & Technology, Lahore, Pakistan, in 1975. He received the M.Sc. degree in Electrical & Computer Engineering in 1983, and his Ph.D. degree from the same university in 1991. He joined the Department of Electrical Engineering at the Engineering University, Lahore, as a lecturer in 1975. He was promoted to Professor in 1992.

In 1985 and 1986, Dr. Iqbal worked as a graduate fellow at the Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center, Hampton, Virginia, USA. He conducted research as a Fulbright Scholar at the Department of EE-Systems, University of Southern California, Los Angeles, California, in 1992. In 1996 he worked as a DAAD Research fellow at the University of Stuttgart, Institute for Parallel & Distributed High-Performance Systems (IPVR), Stuttgart, Germany. He joined the department of Computer Science, Lahore University of Management Sciences (LUMS) in 2000, and worked as the head of the department from 2000 to 2003. Currently, he is working as a Professor in the same department.

Dr. Iqbal's research interests include distributed and parallel computing. Presently he is active in designing innovative techniques of teaching computer science and mathematics. He is a member of the Institute of Electrical and Electronics Engineers (IEEE). He is also an Urdu poet and his first collection of poems was published in 2001.

Sara Tahir was born in Karachi, Pakistan, in 1979. She graduated with distinction, on the Dean's honor roll, from the Lahore University of Management Sciences (LUMS), Pakistan, in June 2000 with a B.Sc. (Honors) Degree in Computer Science and a minor in Mathematics. Pursuing a career as a software engineer, she worked with SoftWeb, Pakistan for a year after graduation before joining Habib Bank AG Zurich, Dubai, U.A.E. In September 2002 Sara joined Stanford University, U.S. to pursue a Master of Science degree in Management Science and Engineering. She expects to graduate by June 2004.