Extending an Iterator Model for Binary Trees to Backtracking Problems

Uta Ziegler
Department of Computer Science
Western Kentucky University
Bowling Green, KY 42101
ziegler@pulsar.cs.wku.edu

Abstract

Rasala [1] developed a design for an object-oriented tree-iterator. This paper extends that design to provide an iterator for backtracking problems. The developed iterator explicitly uses a stack to store partial solutions which must be evaluated and/or extended.

The discussed design can be used as a teaching tool for recursion. Students often struggle with this paradigm [2] since - for their taste - too much is happening behind the scenes. Students can explore the explicit stack of the backtracking iterator after each recursion step.

The design can be reused for different domains, since it separates domain-specific code from general iterator code. Solutions to the set and the maze problems are shown.

Introduction

The paper by Rasala [1] provides an elegant and sophisticated object-oriented approach for an iterator class for binary search trees. The iterator class keeps track of the current state by building a linked list of the parent nodes (each with its progress indicators). Each element in the list is a tree node which must be revisited. Elements can only be added/ modified/inspected and deleted at the beginning of the list, meaning the list is used as a stack. In effect, the iterator class is (nearly) an iterative implementation (using a stack explicitly) of the recursive tree-traversal algorithm (using the system call stack as an implicit stack). Indeed, the given code would only need small changes to work entirely without recursion.

Students often struggle with recursion [2] since they would like to "see" what's happening. Using an explicit stack (which can be examined after each iteration, for example) provides a "view" into how recursion works. This paper extends the basic design from Rasala [1] to implement an "iterator" for most recursive problems which are solved by backtracking or a depth-first approach. The subset problem and the maze problem are used as examples.

SIGSCE 98 Atlanta GA USA
Copyright 1998 0-89791-994-7/98/ 2..$5.00

For the subset problem, a set of numbers S is given and a value V. The task is to find (if possible) a subset M of S, such that the sum of the elements in M is equal to V. For the maze problem, a 2-dimensional maze is given (with a treasure) and a starting point. The task is to find (if possible) a path from the starting point to the treasure.

After a short overview of the classes involved, a recursive solution to the subset problem is given. Then the design of the classes is discussed, with emphasis on showing the reader how all the parts of the recursive solution are translated into parts of the interator for backtracking problems. The paper concludes with a short overview of how the maze problem can be solved using the discussed design.

A Class Overview

It is assumed that each problem domain has its own class (e.g. maze class, set class) which allows a programmer to create/destroy and otherwise manipulate a given object.

Two more classes are defined:

- A *problemtype class* which defines the data that must be stored on the stack for each recursive step as well as the data and variables needed to specify and solve the problem.
- An *iterator class* which contains the stack and which performs the iterations.

The stack involves another class, but any well-designed stack class which provides the operations is_empty, push, pop, and topptr can be used. (For example, this project uses the one designed in [3]; only the topptr member function was added.)

The following explains some of the relationships between the classes in the current implementation. It does not make sense to pose a problem (e.g. find the treasure in the maze starting at ...) if the object itself (e.g. the maze) has not been declared. Thus the constructor of the problemtype class expects a pointer to the object. Similarly, one cannot iterate through the solution process without a problem. Thus, the constructor of the iterator class expects a pointer to the problem to be solved.

The problemtype class and in many cases the domain object class (sets, maze,...) must define the iterator class as a friend class.

A Recursive Solution to the Subset Problem

Below is one of the many solutions to the subset problem.

```
bool sset (int S[], bool M[], int match, int index)
{
        if (match == 0) return true;            // success

        if ((index >= MAX)
            || (match < 0)) return false;       // failure

        // try using the current number
        M[index] = true;
        if (!(sset (S, M, match-S[index], index+1)))
        {
                // do not use the current number after all
                M[index] = false;
                return sset(S, M, match, index+1);
        }
        else return true;
}
```

In the provided code, *M* is an array of boolean values, one for each element in S. A true value indicates that the corresponding value of the set S is included in the subset. MAX is the number of elements in the set S. Index is the subscript of the element of S currently under investigation. The recursive function performs several processing steps (in this order):

- check for immediate success or failure;
- add the current number to the subset and attempt a solution;
- remove the current number from the subset if the attempt failed and
- attempt a solution without the current number.

The same processing steps must be performed by the iterator class. Note that this is a very common pattern for backtracking problems.

The Problemtype Class

The structure pdtype below represents the data stored on the stack for each recursive step. Several processing steps are performed on each stack element. Each processing step is associated with one of the flags in the boolean array *doneflags*. ACTS is the number of processing steps. If the processing step must still (or again) be performed, the corresponding flag is set to false; otherwise to true.

```
class problemtype {
  struct pdtype {
        int index;      // next value to examine
        int match;      // the value to sum up to
        bool * doneflags; // what's done?

        pdtype ()           // default constructor
         {      index = 0;
                match = 0;
                doneflags = new (bool[ACTS]);
                for (int i = 0; i<ACTS; i++)
                        doneflags[i] = false;

         };

        // constructor with problemtype
        pdtype (problemtype * ptr)
         {      doneflags = new (bool[ACTS]);
                *(this) = ptr->probdata;
         };
};
```

Statements printed in italics are specific for the subset problem. The remaining statements are the same for different backtracking problems.

Not shown here, but necessary for a correct working of the pdtype structure are a copy constructor, an operator= member function and a destructor.

The other data to be stored in the problemtype class depend on the problem at hand, as do the desirable constructors (besides the default constructor).

```
public:
        sets * setptr;      // a pointer to the set
        bool M[MAX];    // member array for subset
        pdtype probdata; // specify problem

        // constructor for problem type
        problemtype(sets * S)
        {  setptr = S;
           for (int i = 0; i<MAX; i++)
                M[i] = false;
          cout << endl << "Enter match \n" ;
          cin >> probdata.match;
        };

        // display the result
        void display ()
        {  for (int i = 0; i<MAX; i++)
                if (M[i])
                  cout << setptr->getelem(i) << endl;
        };
        friend class Iterator;
};
```

The member variables of the problemtype class together represent the posed problem. In most cases, this includes a pointer to the domain object (e.g. the set), the particular goal of the problem in a pdtype structure (e.g. which value to sum up to) and (if necessary) other variables to hold the result (e.g. M). The problemtype constructor provided creates a representation of the problem (assuming

the set already exists).

In this particular example, no destructor is needed, but a display function is provided to show the computed solution of the problem.

The Iterator Class

There are only three public member functions:

1) the constructor which sets up the problem solution (in other words, it pushes a description of the problem on the stack);
2) the destructor empties the stack if necessary (not shown here);
3) the *Iterate* function which processes the top node on the stack (or pops it if nothing else needs to be done). This function is in effect the Iterate function from [1], rewritten to be iterative instead of recursive and using a stack explicitly.

Note: The code for these three public functions is *independent of the particular problem domain.*

```
class Iterator {
    typedef problemtype::pdtype IteratorNode;
    Stack<IteratorNode> S;
    bool success, stop;
    int method;
    problemtype * probptr;

  public:
  // constructor
  Iterator (problemtype * problem, int Method)
   {    probptr = problem;
        // create the stacknode for the problem
        IteratorNode first (problem);
        // push the node on the stack
        S.push(first);
        // set other variables
        method = Method;
        success = false;
        stop = false;
  };

// iterate action
bool Iterate (bool & empty)
  {    stop = false;
       while (!(empty = S.is_empty()) || !stop)
            if (!Tryoptions())
                 S.pop();
       return success;
  };
```

The member variable *success* is not needed for such things as the tree-traversal, since the three actions of a tree traversal are always carried out. However, in many backtracking problems, some actions are only necessary if earlier actions failed to come up with a solution. The variable *success* records whether earlier actions did or did not find a solution to the problem. If the stack is empty and *success* is false, then no solution exists for the problem.

The variable *stop* controls when the iteration stops (for example, to examine the stack). It can be set by one or more of the processing steps.

The iterate function always processes the current node on top of the stack. Tryoptions applies the processing steps to the node until one of them returns true. If none of the processing steps returns true, then the node is popped off the stack.

Hidden Member Functions

The hidden member functions of the iterator class are set up following the general principle in the article from Rasala[1]:

- The Tryoptions function executes a logical OR of several test functions (stopping as soon as one returns true). If there exists more than one method (order) in which the test functions can be executed, a switch statement will select the correct one based on the method variable set in the iterator constructor.
- Each of the test functions first checks whether it still needs to be performed. If so it changes the flag. (Note: it will directly change the flag information of the top element from the stack to avoid repeated pop/push operations.)
- Each of the test functions to be performed returns true or false, based on whether the function could perform its designated processing step or not. (Do not confuse this with success, which indicates whether a successful solution has been found.)

```
private:
bool Tryoptions()
{ switch (method){
   case LeftFirst :
     return (Checkdone() ||
          TrywithNumber() || ResultofTry() ||
          TrywoutNumber());
   case RightFirst :
     return (Checkdone() ||
          TrywoutNumber() || ResultofTry() ||
          TrywithNumber() );
        }
   return false;
};
```

```
bool Checkdone()          // are we done now?
{       IteratorNode * pptr = S.topptr();
        // if already done return false
        if (pptr->doneflags[0])
                        return false;
        pptr->doneflags[0] = true;

        // successful end of current track?
                if (pptr->match == 0)
                {       success = true;
                        S.pop();
                        return true; }

        // unsuccessful end of current track?
        if ((pptr->match <0) ||
                (pptr->index >= MAX))
                {       success = false;
                        S.pop();
                        return true;}
        // current track needs to be pursued
        return false;
};
```

We are done (and have a solution) if the numbers of the current subset M add up to the given number V. In that case the number which must *still* be matched is 0. The current track can also be abandoned (unsuccessfully), if the numbers of the current subset M add up to a value larger than V (in this case the number which must still be matched is negative), or if there are no more numbers in the set S to add to the subset M. In other words, once it is clear that a successful solution cannot be found along the track currently under investigation, a "backtrack" step must be performed (= pop the topmost state off the stack).

Notice the similarities in the code with the recursive function. Since the data is not available locally (as in the recursive function), but on top of the stack, the access to the data is more involved. However, the steps which are taken are the same.

```
bool TrywithNumber()
{       ..... check and set doneflags[1][1]
        // allow ResultofTry to be executed
        pptr->doneflags[2] = false;

        // get the value of the current set element
        int k = pptr->index;
        int number = probptr->setptr->geteleim(k);

        // add number to current solution
        probptr->M[k] = true;
        stop = true;
        // find the remainder of the solution
        int nextmatch = pptr->match - number;
        IteratorNode next(nextmatch, k+1);
        S.push(next);
        return true;
};
```

TrywithNumber adds the current number to the subset and then pushes the description of the "left-over" problem on the stack to be solved. (The "left-over" problem is: use the remaining numbers in the set, try whether a subset exists whose numbers add up to the value minus the current number.) Again, a comparison of the second processing step in the recursive function reveals that TrywithNumber performs the same task.

```
bool TrywoutNumber()
{       ..... check and set doneflags[3][1]
        IteratorNode next(pptr->match, pptr->index+1);
        S.push(next);
        return true;
};

bool ResultofTry()
{       ..... check and set doneflags[2][1]
        // if not successful, then return false
        if (!success)
          {     // don't use current number
                probptr->M[pptr->index] = false;
                return false;
          }

        // if successful, pop the node off the stack
        S.pop();
        return true;
};
```

TrywoutNumber and ResultofTry again follow the steps of the recursive function pretty closely. The return from the recursive function (in the case of a successful attempt) is translated into popping the top node off the stack.

The hidden member functions differ from the ones in the Rasala article in that the stack manipulations are explicit. Where in Rasala's code a *state = new IteratorNode (.....)* statement adds a new state to the linked list, the current implementation constructs an IteratorNode and then pushes it on the stack.

If desired, the switch statement in Tryoptions can be eliminated by making Tryoptions a pure virtual member function, thus changing the iterator class to an abstract class. Several derived classes can then be defined (one for each method). This approach is common [1, 4, 5].

[1] See code in Checkdone

Sketch of the Maze Solution

The treasure in the maze is indicated by the letter 'T', walls by 'W' and places already visited by 'X'. When the treasure is reached, it is changed to a 'G' (for goal).

The information to be stored in each stack element is the current place (as row and column) in the maze.

```
struct probdatatype {
        int crow, ccol;     // current row and column
        bool * doneflags;  // what's done?

        // constructors and such.....
};
```

The private member functions of the Iterator class must check whether the treasure has been reached (Checkdone) and each of the four directions: left, right, up and down. Since the four member functions are very similar the code for only one of them is provided (TryLeft). After each direction attempt, the result must be checked (with ResultofTry) to determine whether another direction should be checked next or whether a path has been found. Therefore, each test function sets the doneflag for ResultofTry before it returns. (In the code provided, objectptr is equivalent to problem->mazeptr.)

```
bool Checkdone()
{
        ... check and set doneflags[0]¹
        // get row and col
        int crow = pptr->crow;
        int ccol = pptr->ccol;
        // successful end of current track?
        if (objectptr->getmaze(crow, ccol) == 'T')
        {       objectptr->setmaze(crow, ccol, 'G');
                success = true;
                S.pop();
                return true; }
        // current track needs to be pursued further
        return false;
        };

bool TryLeft()
{ ...check and set doneflags[1]¹
   // get row and col
   int crow = pptr->crow;
   int ccol = pptr->ccol;
   char leftsquar =((ccol == 0)? 'W' :
                   objectptr->getmaze(crow, ccol-1));

   if ((ccol == 0) ||          // no place to go
     (leftsquare == 'W') || // there's a wall
     (leftsquare == 'X'))   // I've been there before
          return false;     // cannot go left
   // mark current place
   objectptr->setmaze(crow, ccol, 'X');
   stop = true;
   // find the remainder of the solution
   IteratorNode next(crow, ccol -1);
   S.push(next);
   // ResultofTry can be executed after this
   pptr->doneflags[2] = false;
   return true;
}

bool ResultofTry()
{       ... check and set doneflags[2]¹
        // get row and col
        int crow = pptr->crow;
        int ccol = pptr->ccol;
        // if try not sucessful, then return false
        if (!success)
        {       objectptr->setmaze(crow, ccol, ' ');
                return false; }

        // if try successful, nothing else needs to be done
        S.pop();
        return true;
}
```

The interested reader can appreciate the possibility of changing the above approach to a heuristic search for the treasure (assume the coordinates of the treasure are known). This would entail rewriting TryOptions to select the best direction which hasn't been tried yet based on a given heuristic.

Conclusions

The notion of recursion is fundamental to computer science. This paper discussed a reusable, object-oriented iterator which can be used to provide students with the means to explore what goes on "behind" the scene. In the beginning, the students can learn from watching how the current state of the solution changes. More experienced students can get a deeper understanding of backtracking problems by developing the necessary hidden member functions of the Iterator Class.

References

[1] Rasala, Richard, A Model C++ Tree Iterator Class For Binary Search Trees, in *The Proceedings of the 28th SIGCSE Symposium.*

[2] Roberts, Eric S, *Thinking Recursively*, John Wiley, 1986.

[3] Main, Michael and Savitch, Walter, *Data Structures and Other Objects Using C++*, Addison Wesley, 1997.

[4] Oualline, Steve, *Practical C++ Programming*, O'Reilley & Associates, 1995.

[5] Pohl, Ira, *Object-Oriented Programming Using C++*, Benjamin Cummings, 1993.