Backtracking Algorithms

Finding all the solutions

Timothy Rolfe

Backtracking algorithms let you systematically search for solutions to problems among all available options. Backtracking is illustrated in the Eight-Queens problem—finding the chessboard configurations for eight queens in which no queen is attacking another queen. You start in the first row of the chessboard and generate candidate solutions by positioning your first queen in each of the eight positions. After each positioning, you move to the second row and position your second queen on each of that row’s squares. Each time that the queen on the current row is under attack from above, you can simply move on to the next square, and you need not examine any boards generated from the partial board.

There is, however, another interesting problem that can also be approached with backtracking algorithms.

In his August 1986 column for Isaac Asimov’s Science Fiction Magazine (http://www.asimovs.com/), Martin Gardner presented this problem:

Timothy is a professor of computer science at Eastern Washington University. He can be contacted at trolfe@ewu.edu.

Now for a curious little combinatorial puzzle involving the twelve numbers on the face of a clock. Can you rearrange the numbers (keeping them in a circle) so no triplet of adjacent numbers has a sum higher than 21? This is the smallest value that the highest sum of a triplet can have.

I know of no procedure for finding such a permutation, but there must be a way to write a computer program that will print all such permutations in a reasonable time.*


This puzzle amounts to constraining a circular permutation of the numbers from 1 to 12, but you can start with a simpler problem. Just as the Eight-Queens problem can be generalized to an N-Queens problem, you can generalize the clock-face puzzle to faces with fewer numbers than 12. After all, there are 12! permutations of the numbers from 1 to 12. You can also ignore configurations that only amount to rotations of other configurations. You can do that by nailing down one number—comparable with nailing down the “12” in the 12:00 position on the standard clock face. That at least brings you down from N! to (N−1)!, but you’re still dealing with a rather ugly number of potential configurations. You can reduce the number just a bit more: Ignore each configuration that is just the mirror image of another configuration, something you can do by forcing an order on the two numbers adjacent to the top number. In terms of the standard clock face, those would be the numbers in the 11:00 position and the 1:00 position. Figure 1 shows the single circular permutation of the numbers from 1 to 6, where we enforce 11 as the largest sum.

Now you’ve taken the N-entry clock face from N! potential configurations down to (N−1)!/2 configurations, but that is still a huge number. For instance, if you don’t find an intelligent approach, the standard clock face would require checking 11!/2 configurations—to be precise, 19,958,400 configurations. (At least that’s better than 12!—479,001,600—without the symmetry corrections.)

This problem was used in the Pacific Northwest Regional part of the ACM International Collegiate Programming Contest (http://icpc.baylor.edu/icpc/). It is possible that some of the contestants ran afoul of this nasty exponential explosion. The C++ STL’s next_permutation function provides a snare and a trap. While 6! isn’t too bad, 12! (and especially 13!) gets truly nasty—and the contest rules require that programs finish within 120 seconds. When I encoded the direct calculation using
ATTEND THE PREMIER CONFERENCE DEVOTED TO NETWORK SECURITY TECHNOLOGY

14TH ANNUAL

NetSec2004

JUNE 14-16, 2004 • SAN FRANCISCO HYATT REGENCY EMBARCADERO

FEATUREING:
- 2 DEDICATED TECH TRACKS
- SPAM
- PATCH MANAGEMENT
- RADIO FREQUENCY ID
- INTRUSION PREVENTION
- FORENSICS
- WIRELESS
- NEW PRIVACY LAWS
- VPNs
- INTERNET/INTRANET SECURITY
- REMOTE ACCESS
- AWARENESS
- CRYPTOGRAPHY
- COMPUTER CRIME
- BUSINESS SURVIVAL
- HOT BUTTONS
- SSCP PREP

Building The Secure Enterprise

120 SESSIONS
85 EXHIBITORS
next_permutation(), the 12-entry clock face required 36 seconds to test for the triple sum of 21, but checking the 13-entry clock face for the triple sum of 22 required 416 seconds.

With backtracking, however, you can start building your permutations and, at the point where a candidate partial permutation has a triple that sums to more than the allowed value, you can discard all the permutations generated beyond that.

If you use a queue, generating the permutations is also easy. First, you enter all of the digits from 1 to N into your queue. If you have an array face[] to hold the current permutation, you can initialize face[0] with whatever value you want to nail to the top of the clock face by dequeuing a value into that position. After that, recursion is your friend. You position digits into face[1] (rotating available digits from the queue into that position) and, for each of those, you position remaining digits into face[2], on up. The following is the pseudocode for this.

To build a clock face in position k:

1. If the queue is empty, all positions are filled—you have a winner.
2. Otherwise, retain the value of the next item from the queue.
3. Loop (bottom driven) until that value comes back around.
   (a) Remove a value from the queue and put it into face[k].
   (b) If the resulting partial clock face is still valid, recursively fill position k+1.
   (c) Return the value in face[k] back into the queue.

Listing One implements this in C++; the Java version differs only in declaring the method to be public.

The validity check can also take advantage of the fact you have a valid partial solution up to the position you are filling. This is something that I failed to notice 15 years ago, when I first played with this puzzle in an article that appeared in Mathematics and Computer Education (Spring 1987; http://www.macejournal.org/). The pseudocode for the check looks like this:

To check position k of the clock face:

1. Generate the total from face[k-2] up to face[k], being careful to only go back to subscript 0, if appropriate.
2. Else, if this isn’t the last position on the face, return success.
3. Else, this is the last position on the face.
   Check the remaining two triples.

PC-lint 8.0 Bug of the Month
#802

PC-lint for C/C++ will catch this and many other bugs. It will analyze a mixed suite of C and C++ modules to uncover bugs, glitches, quirks and inconsistencies.

Not your Grandpa’s lint: PC-lint has introduced several spectacular and revolutionary innovations in the art of static program analysis. Taking clues from initializers, assignments, and conditionals, variable and member values are tracked, enabling reports on potential uses of null pointers and out-of-bounds subscripts.

New with Version 8: Interfunction value tracking - Actual argument values are used to initialize parameters; return values are computed; a multi-pass operation (you control the number of passes) allows you to plumb the depths of function behavior to arbitrary levels.

Plus Our Traditional C/C++ Warnings: Uninitialized variables, inherited non-virtual destructors, strong type mismatches, ill-formed macros, inadvertent name-hiding, suspicious expressions, etc., etc.


PC-lint for C/C++ $239
Numerous compiler libraries supported. Runs on Windows, MS-DOS, and OS/2.

FlexeLint for C/C++
The same great product for other operating systems. Runs on all UNIX systems, VMS, mainframes, etc. Distributed in shrouded C source form. Call for pricing.

30 Day Money Back Guarantee

Dr. Dobb’s Journal, May 2004 http://www.ddj.com
Listing Two implements this in C++; the Java version differs only in declaring the method to be private and calling the return value boolean.

The complete program (in C++ and Java) is available electronically; see "Resource Center," page 5. (The archive also includes BadWords.cpp, the version that uses the STL next_permutation function.)

Runs of the Java version generated the output in Figure 2. In the first run, rather than doing a check of 19,938,400 configurations requiring up to 12 checks each, we only ran 2,168,123 checks of configurations, mostly requiring a single check. The time reflects executing the program on a Compaq Presario (AMD Athlon Processor 1.2 GHz) under Java SDK 1.4.0_01 running Windows ME.

Optimization
I also used this problem as a programming assignment in my Data Structures II course, providing the students with a draft of this paper (minus the code listings, of course) to explain the problem. That motivated me to experiment with other methods to solve the problem that might execute more quickly. Some students also suggested ideas to speed execution.

Robert Lyon’s implementation moved the rejection of mirror images from where I have it (after completion of the permutation) to a check when there are only three elements in the partial permutation. This has a dramatic effect, thanks to the early backtracking. Brendan Cassida’s implementation moved the check from a function call to inline code, providing another speed increase.

I noted that I could dispense with the queue altogether, simply moving values around within the permutation array in a controlled fashion. Unlike the queue-based implementation, I also wanted to generate candidate permutation in lexicographic order. This means that, for four items, you’re looking for the following sequence:

1 2 3 4 etc.
2 1 3 4 etc.
3 1 2 4 etc.
4 1 2 3 etc.

This suggests that you perform a single swap with the current position, eventually moving the last entry to the front, but having the rest in order and offset by one position. The restoration is a matter of saving the front value, then moving the rest of the array forward by one position and placing the saved value at the back—effectively rotating the array left by one position. Here’s what the pseudocode looks like:

To build a clock face in position k more quickly:

1. If k is past the end of the array, all positions are filled—you have a winner.
2. Otherwise, loop as j takes on values from k to the end of the array.
   (a) Swap entries j and k.
   (b) If the resulting partial clock face is still valid, recursively fill position k+1.
3. Restore the initial state of the array.
   (a) Save the value in position k.
   (b) As j goes from k+1 to the end, move elements from j/l to j-1.
   (c) Position the saved value at the end of the array.

Listing Three dispenses with having a clock class and passes all necessary information as function parameters. As you’d expect with optimizations, it is longer than the earlier implementation.

Acknowledgments
Thanks to Kris Rudin, director of the Pacific Northwest Region for the ACM International Collegiate Programming Contest, and Tom Capaul, faculty advisor for the ACM student chapter at Eastern Washington University, for giving me the motivation to revisit and rethink this problem in preparation for the 2002 regional competition. Thanks also to Tom Capaul for providing some useful suggestions on this article. A special thanks to the students in the Fall-2002 CScD-327 class at Eastern Washington University, particularly Robert Lyon and Brendan Cassida.

Listing One
// Set remaining values into position size and recurse (if needed) void Cloque::build ( IntQueue &work, int posn ) {  if ( work.isEmpty() ) // Face has been completed  {   if ( face[posn-1] < face[0] ) // Exit the mirror image process();  }  // Process as a valid clock face.
  // Next possibility:  intermediate case (incomplete permutation)
  else
  {   int marker = work.nextItem();
   do
   {    face[posn] = work.dequeue();
    // Note that X[1] contains the value N-1.
    // Check for mirror-image rejection
    if ( marker != work.nextItem() )
     { // and if ( work.isEmpty() ) / else
      // / and build()
   } // else if ( work.isEmpty() )
  } // else
  }

Listing Two
// Check whether the current clock face meets the restriction on // the maximum allowed triplet sum. Note that this presumes that // the face is valid UP TO making an entry into this position. bool Clique::check ( int posn ) // i.e., position being filled { int idx, total = 0;
  numCheckCalls ++;
  idx = posn - 1;
  if ( posn == 0 )
   idx = 0;
  while ( idx < posn )
   { total += face[idx++];
    if ( total > maxTot )
     return false;
   } // At the end, test the wrap-around cases;
  if ( posn == maxsize-1 )
   { total = face[posn-1] + face[posn] + face[0];
    if ( total > maxTot )
     return false;
    total = face[posn] + face[0] + face[1];
    if ( total > maxTot )
     return false;
   } // end if ( posn... return true;
  } // end check()

Listing Three
void Permute ( int X[], int Lim, int N, int MaxSum ) // int j; // Loop variable // First possibility: start-up case moving values through [N] // ( Lim == 0 )
{  Permutes ( X, 2, N, MaxSum );
  if ( j+2 < j )
  {   Swap ( j, 0, X );
   // Note that X[1] is left unchanged at N-1.
   Permutes ( X, 2, N, MaxSum );
   // We will not bother regenerating the array here.
  } // Next possibility: intermediate case (incomplete permutation)
  else if ( Lim < N-1 )
  {   int hold;
    int PairSum = X[Lim-2] + X[Lim-1];
    // Check the initial state
    if ( X[Lim]+PairSum <= MaxSum )
     Permutes ( X, Lim+1, N, MaxSum );
    continue;
    // Check for mirror-image rejection
    if ( X[Lim]+PairSum > MaxSum )
     continue;
    Permutes ( X, Lim+1, N, MaxSum );
    // Regenerate the array state.
    hold = X[Lim];
    for ( j = Lim+1; j < N; j++ )
     X[j] = hold;
    // Final possibility: complete permutation, so check for validity
    else if ( check ( X, Lim, N, MaxSum ) )
     Process ( X, N );
} // else it fails at the very end

http://www.ddj.com Dr. Dobb’s Journal, May 2004

DDJ