PROGRAMMER'S CHALLENGE

by Bob Boonstra, Westford, MA

DNA MATCH

This month's Challenge is based on a suggestion submitted by Vicente Giles of the Universidad de Málaga. Vincente faces a real-world problem to look for all the genomic sequences that match certain criteria, given a DNA database sequence and a problem sequence. A DNA sequence is a string of the four different nucleotides involved in the genetic code, denoted 'A', 'C', 'G', and 'U', which stand for adenine, cytosine, guanine, and uracil. The problem is to find all possible matches of the problem sequence in the database sequence, allowing a specified number of differences.

The prototype for the code you should write is:

```
long FindMatch(
  char *alphabet,        /* legal characters for database and fragment strings */
  char *database,        /* reference string to compare fragments against */
  void *storage,         /* storage preallocated for your use */
  char *fragment,        /* string to match against database */
  long diffsAllowed,     /* differences allowed between fragment and database */
  long matchPosition[]   /* return match positions in this array*/
);

void InitMatch(
  char *alphabet,        /* legal characters for database and fragment strings */
  char *database,        /* reference string to compare fragments against */
  void *storage         /* storage preallocated for your use */
);
```

Because we would like our DNA-matching algorithm to be useful even if scientists discover an extraterrestrial genetic code based on other nucleotides, the algorithm accepts the genetic `alphabet` as a parameter. In the problem posed by Vincente, this would be the string "ACGU", but in our Challenge it might include any of the characters `'a'..'z'` or `'A'..'Z'` (Extraterrestrial DNA is case sensitive). The null-terminated reference string contained in the `database` parameter can be up to 1000000000 (10^9) characters long. The `fragment` that you are to match is also null-terminated, but will be significantly shorter on average (up to 10000 characters) than the `database` string. You should compare the input `fragment` against `database`, finding all occurrences of `fragment` that differ in no more than `diffsAllowed` positions from a substring of `database`. Your code should populate one entry in the preallocated `matchPosition` array for each match found, storing the offset of the character in `database` that corresponds to the first character of `fragment`. The `FindMatch` function should return the number of matches found.

As an example, given the following input …

```
    alphabet:          ACGU
    database:          ACGTACGTACGTAAAAAATACGTACGTATA
    fragment:          ACGTACGTAC
    diffsAllowed:      5
```

… your code should find 7 matches and store the following values in `matchPosition`:

```
    -4  0  4  8  15  19  23
```

Notice that partial matches can occur at the beginning or the end of `database`, and as a result, the offsets returned in `matchPosition` can be negative or greater than `strlen(database) – strlen(fragment)`.

To allow you to do some preprocessing, your `InitMatch` routine will be called once before a sequence of calls to `FindMatch`. `InitMatch` will be called with the same `alphabet` and `database` parameters provided to subsequent `FindMatch` calls. Both routines will also be given the same `storage` parameter that points to at least 1MB of memory allocated and initialized to zero by the calling routine. `FindMatch` will be called between 100 and 1000 times, on average, for each call to `InitMatch`. The winning solution will be the one with the fastest execution time, including the execution time for both `InitMatch` and `FindMatch`.

Other fine print: The `alphabet` characters will be provided in increasing ASCII order. The offsets you store in `matchPosition` need not be in any particular order. The value for `diffsAllowed` will typically be smaller than 50% of `strlen(fragment)`. Finally, you should not allocate any dynamic storage in your solution beyond that provided in the `storage` parameter.

This will be a native PowerPC Challenge using the latest Symantec environment. Solutions may be coded in C or C++.

Two Months Ago Winner

Congratulations to **Randy Boring** for submitting the fastest entry to the A-Maze-ing Programmer's Challenge. The Challenge this month was to write code that would find a path leading out of a three-dimensional maze. The solutions were provided with the maze size, an initial position, some storage for use in mapping the maze, and a callback routine. The callback provided the result of attempting to move in a given direction, indicating whether the attempt to move succeeded, failed because there was no opening in the specified direction, resulted in a fall down a shaft in the mine, or found an exit to the mine. Of the four entries submitted, only two successfully solved all of my test mazes; one of the entries crashed, and one went into an infinite loop.

The table below summarizes the results for each correct entry, including the language in which the solution was written, the size of the solution code, the amount of static data used by the solution, the total execution time for all test cases, and the number of moves needed to solve the mazes.

Name	Language	Code Size	Data Size	Time	Moves
Randy Boring	C++	2792	484	343153	33519
Jay Negro	C++	1788	51	40945114	7120802

The test mazes used in the evaluation ranged in size from 10x20x30 to 100x100x200, and ranged in density (the percentage of open cells) from 10% to 20%. As indicated in the problem statement, a path to an exit was guaranteed to exist from any cell reachable from the starting position.

Randy's winning entry spent more time processing each move than the second place entry from first-time Challenge contestant **Jay Negro**, but Randy's code solved the maze using significantly fewer moves and executed two orders of magnitude more quickly. His code maintains a queue of what are believed to be the best moves to try. As long as there are moves in the best move list, it invokes the callback with the best move, checks for success, and then updates the map with what it has learned about the maze position it just tried. The `CalcBestMove` routine determines the best possible move (surprise!) by first moving toward an adjacent maze boundary if one exists, or moving toward the nearest maze boundary if nothing is known about the current position, or trying adjacent positions about which nothing is known, or finally by moving toward a position about which nothing is known. The `CalcBestMove` heuristics, along with judicious use of inline functions and some optimization of maze offset calculations, combined to make this an efficient solution.

Careful readers of the code will note one potential problem with the `Initialize` routine, in that it simply gives up and returns if it is unable to allocate enough memory. This could have caused the winning entry to fail for larger mazes when given only the amount of memory guaranteed by the problem. However, the size of the mazes that I could practically evaluate was limited by the speed of the other entries, and the memory problem did not show up with those cases, so I elected to ignore it. Under other circumstances, proper handling of low memory conditions would have been required to win.

TOP 20 CONTESTANTS

Here are the Top 20 Contestants for the Programmer's Challenge. The numbers below include points awarded over the 24 most recent contests, including points earned by this month's entrants.

Rank	Name	Points	Rank	Name	Points
1.	Munter, Ernst	193	11.	Cutts, Kevin	21
2.	Gregg, Xan	92	12.	Picao, Miguel Cruz	21
3.	Larsson, Gustav	87	13.	Brown, Jorg	20
4.	[Name deleted]	60	14.	Gundrum, Eric	20
5.	Lengyel, Eric	40	15.	Karsh, Bill	19
6.	Lewis, Peter	30	16.	Stenger, Allen	19
7.	Boring, Randy	27	17.	Cooper, Greg	17
8.	Beith, Gary	24	18.	Mallett, Jeff	17
9.	Kasparian, Raffi	22	19.	Nevard, John	17
10.	Vineyard, Jeremy	22	20.	Nicolle, Ludovic	14

There are three ways to earn points: (1) scoring in the top 5 of any Challenge, (2) being the first person to find a bug in a published winning solution or, (3) being the first person to suggest a Challenge that I use. The points you can win are:

1st place	20 points	5th place	2 points
2nd place	10 points	finding bug	2 points
3rd place	7 points	suggesting Challenge	2 points
4th place	4 points		

Here is Randy's winning solution:

AMAZING.CP

Copyright © 1996 Randy Boring

```
typedef Boolean (*MoveProc) (
  long xMove,long yMove,long zMove,
  long *newXPos,long *newYPos,long *newZPos
  );

// the MoveProc, MakeAMove, is a callback.  It returns true if you
// have found your way out of the maze.
// You give it (x,y,z) as a delta from your current position,
// each from [-1, 0, 1].  Straight up and straight down (and all zeroes)
// always result in no movement.

Boolean Maze (long xMove, // these are your initial position
  long yMove,
  long zMove,
  long xSize,                // these are the dimensions of the maze
  long ySize,
  long zSize,
  MoveProc MakeAMove,        // this is your callback routine
  char *mapStorage           // this is your preallocated storage
  );                         // (one char per position in maze)
```

Typedefs and Constants

```
typedef long dirT;  // direction enumerator (0-24)

static
const long dir2dx[25]={9,
           -1, 0, 1,  -1, 1,  -1, 0, 1,
           -1, 0, 1,  -1, 1,  -1, 0, 1,
           -1, 0, 1,  -1, 1,  -1, 0, 1,};
static
const long dir2dy[25]={9,
             1, 1, 1,   0, 0,  -1,-1,-1,
            1, 1, 1,   0, 0,  -1,-1,-1,
            1, 1, 1,   0, 0,  -1,-1,-1,};
static
const long dir2dz[25]={9,
            1, 1, 1,   1, 1,   1, 1, 1,
            0, 0, 0,   0, 0,   0, 0, 0,
           -1,-1,-1,  -1,-1,  -1,-1,-1,};
static
const dirT di[3][3][3] = {
            {{1,2,3},{4,0,5},{6,7,8}},
            {{9,10,11},{12,0,13},{14,15,16}},
            {{17,18,19},{20,0,21},{22,23,24}}};
static const dirT kNoDir = 0;
static const dirT kFirstDir = 1;
static const dirT kLastDir = 24;
static const dirT kNumDirs = 25;      // including kNoDir at zero
static const char kUnknown = 0;       // unknown square
  // every type below is known
static const char kWall = 1;   // wall
static const char kSpace = 2;         // space-above-wall
static const char kFall = 4;   // space-above-space
static const char kTriedSpace = 10; // searched space
static const char kTriedFall = 12;  // searched fall

static const short kTakeProb = 4;   // 1/4th
static const long kSTNBlockSize = 48;

typedef long squareIndexT;
typedef struct STN {
  struct STN *parent;
  squareIndexT square; // square index of this node
  dirT  direction;  // how I got here from my parent
  } SearchTreeNode, *STNPtr, **STNHandle;
```

Globals

```
static STNPtr gTreeRoot;
static STNPtr gTreeTop;
static STNPtr gQHead;
static STNPtr gQTail;
static STNPtr gBestMoveList;
static STNHandle gTreeRootH;
static STNHandle gBestMoveListH;
static STNPtr gBestMoveListNextPos;
```

Defines

```
#define myIsUnknown(sq) (kUnknown == (sq))
#define myIsKnown(sq)  (kUnknown != (sq))
  // the below should only be used when the square is known
#define myIsWall(sq) (kWall == (sq))
#define myIsUntriedWalkable(sq)     (kSpace == (sq))
#define myIsOpen(sq) (0x00 == (0x01 & (sq)))
#define myIsWalkable(sq) (0x02 == (0x02 & (sq)))
#define myIsUntriedFall(sq)  (kFall == (sq))
#define myIsFall(sq) (0x04 == (0x04 & (sq)))
#define myIsTried(sq)  (0x08 == (0x08 & (sq)))
#define d2x(d) (dir2dx[d])
#define d2y(d) (dir2dy[d])
#define d2z(d) (dir2dz[d])
#define xvec(d) (d2x(d))
#define yvec(d,xN) (d2y(d) * (xN))
#define zvec(d,xyN) (d2z(d) * (xyN))
#define offsetD(d,xN,xyN) (xvec(d) + yvec(d,xN) + zvec(d,xyN))
#define offsetXYZ(x,y,z,xN,xyN) ((x) + (y) * xN + (z) * (xyN))
#define map(m,x,y,z,xN,xyN) (*(m + (x) + (y) * xN + (z) * (xyN)))

#ifdef powerc
#define BreakToSourceDebugger_()          Debugger()
#else    // 68K
#define BreakToSourceDebugger_()          SysBreak()
#endif  // powerc
```

isEmptySearchQ

```
static inline Boolean
isEmptySearchQ() {return (gQHead == gQTail);}
```

isFullSearchQ

```
static inline Boolean
isFullSearchQ() {return (gTreeTop <= gQTail);}
```

DeQ

```
static inline STNPtr
DeQ(void) {return gQHead++;}
```

EnQ

```
static inline STNPtr
EnQ(void) {
  STNPtr p = gQTail++;
  //if (isFullSearchQ())
  // BreakToSourceDebugger_();
  return (p);
}
```

EnQNewCandidate

// Add this move (direction to a square index) to the tree at the current node, assumes
// the square index has not already been visited by this search.

```
static void
EnQNewCandidate(STNPtr parent, const long sqi,
   const dirT d)
{
STNPtr newNode = EnQ();
newNode->parent = parent;
newNode->direction = d;
newNode->square = sqi;
}
```

isEmptySearchTree

```
static inline Boolean
isEmptySearchTree() {return (gTreeRoot == gQTail);}
```

PopSearchTree

```
static inline long
PopSearchTree(void) {return ((--gQTail)->square);}
```

NewSearchTree

```
static void
NewSearchTree(void) {
  gQTail = gQHead = gTreeRoot;
  gQTail++;                          // the only time we're called,
}                                   //  gTreeRoot is immediately the EnQed elem.
```

DisposeSearchTree

```
static void
DisposeSearchTree(char *m) {
while (!isEmptySearchTree()) {
                                     // remove 'tried' mark
  *(m + PopSearchTree()) = kSpace;
  }
}
```

isEmptyMoveList

```
static inline Boolean
isEmptyMoveList()
  {return (gBestMoveListNextPos == gBestMoveList);}
```

PushMoveList

```
static inline void
PushMoveList(const dirT d) {
  gBestMoveListNextPos->direction = d;
  gBestMoveListNextPos++;}
```

PopMoveList

```
static inline dirT
PopMoveList(void) {
  --gBestMoveListNextPos;
  return (gBestMoveListNextPos->direction);}
```

NewBestMoveList

```
static void
NewBestMoveList(void) {gBestMoveListNextPos = gBestMoveList;}
```

SetBestMove

```
// Copy the sequence of best directions-to-move to the
// best move list.  (The tree is about to be freed.)
static void
SetBestMove(STNPtr node, const dirT d)
{
PushMoveList(d);
while (kNoDir != node->direction) { // the root's dir
  PushMoveList(node->direction);
  node = node->parent;
  }
}
```

Initialize

```
// Initialize everything for the routine
// map is already initialized to zeroes (kUnknown == 0)
static Boolean
Initialize(const long x, const long y, const long z,
  const long xSize, const long ySize, const long zSize,
  char *m)
{
const long xySize = xSize * ySize;
long treeBytesWanted = sizeof(SearchTreeNode)
  * ((xSize - 2) * (ySize - 2) * (zSize - 2) + 4);
long treeBytesNeeded = sizeof(SearchTreeNode)
  * ((xSize - 2) + (ySize - 2) + (zSize - 2) + 1);
Boolean succeed = true;

map(m,x,y,z,xSize,xySize) = kSpace;
map(m,x,y,z-1,xSize,xySize) = kWall;

gBestMoveListH = (STNHandle) NewHandle(sizeof(SearchTreeNode)
           * (zSize + xySize) * 2);
if (!gBestMoveListH) succeed = false;
    // unable to initialize successfully,MEM
HLock((Handle) gBestMoveListH);
gBestMoveList = *gBestMoveListH;

do {
  gTreeRootH = (STNHandle) NewHandle(treeBytesWanted);
  treeBytesWanted *= 0.9;
  }
  while (!gTreeRootH &&
    (treeBytesNeeded <= treeBytesWanted));
if (!gTreeRootH) succeed = false;
    // unable to initialize successfully,MEM
HLock((Handle) gTreeRootH);
gTreeRoot = *gTreeRootH;

NewBestMoveList();
return succeed;
}
```

DeInitialize

```
// Free everything we allocated
static void
DeInitialize()
{
HUnlock((Handle) gBestMoveListH);
DisposeHandle((Handle) gBestMoveListH);
HUnlock((Handle) gTreeRootH);
DisposeHandle((Handle) gTreeRootH);
}
```

AtEdge

```
// Are we at an edge?
// return true if we are
static Boolean
AtEdge(const long x, const long y, const long z,
   const long xMax, const long yMax, const long zMax)
{
if (0==x || 0==y || 0==z) return true;
if (xMax==x || yMax==y || zMax==z) return true;
return false;
}
```

MapMove

```
// Map the move we just made/tried.
// I.e., store our new knowledge of the maze in the map
static void
MapMove(const long oldx, const long oldy, const long oldz,
   const long newx, const long newy, const long newz,
   char *m, const long xN, const long xyN,
   const long dx, const long dy, const long dz)
{
long x = oldx + dx; // where we tried to move
long y = oldy + dy; // (we need these in either case below)
long z = oldz + dz;
long sqi = offsetXYZ(x, y, z, xN, xyN);
if (newx==oldx && newy==oldy && newz==oldz) { // bump!
  // mark as wall the square we bumped into
  *(m + sqi) = kWall;
  // lastTried was not successful
  //gLastWasSuccess = false;
  }
else {
  // mark as clear any squares we fell through
  // actually mark them with as fall because they can't
  // really be moved 'to' only through, downwards.
  if (newz < z) {
    do {
      *(m + sqi) = kFall;
      z--;
      sqi -= xyN;
      } while (newz < z);
    }
  // mark as clear our current space
  *(m + sqi) = kSpace;
  // mark as 'wall' the square we are standing on
  sqi -= xyN;
  *(m + sqi) = kWall;
  // lastTried was successful
  //gLastWasSuccess = true;
  // direction of last success
  //gLastSuccess = lastTried;
  }
}
```

```
// return a direction in which
// there may be an adjacent exit.
// A possible exit must be both an edge and untried.
static dirT
APossibleExit(const long x, const long y, const long z,
  const long xSize, const long ySize, const long zSize,
  char *m, dirT lastTried)
{
dirT tryD = lastTried + 1; // start here
/* check for edge conditions first */
if (x > 1 && x < xSize - 2)
  if (y > 1 && y < ySize - 2)
    if (z > 1 && z < zSize - 2)
      /* no exits are possibly nearby */
      /* because no edges are nearby */
      return kNoDir; // zero

/* search the adjacent spaces for a possible exit */
/* this could improve a lot */
while (tryD <= kLastDir) {
  const long dx = d2x(tryD);
  const long dy = d2y(tryD);
  const long dz = d2z(tryD);
  if (AtEdge(x+dx, y+dy, z+dz, xSize-1, ySize-1, zSize-1))
    if (myIsUnknown(map(m, x + dx, y + dy, z + dz,
        xSize, xSize * ySize)))
      return tryD;
  tryD++;
  }
tryD = kFirstDir;
while (tryD <= lastTried) {
  const long dx = d2x(tryD);
  const long dy = d2y(tryD);
  const long dz = d2z(tryD);
  if (AtEdge(x+dx, y+dy, z+dz, xSize-1, ySize-1, zSize-1))
    if (myIsUnknown(map(m, x + dx, y + dy, z + dz,
        xSize, xSize * ySize)))
      return tryD;
  tryD++;
  }
return kNoDir; // zero, no possible exit found nearby
}
```

TotallyUnknown

```
// returns true if every move leads to an unknown square
// this is only possible at the beginning and after falling
// at least three below our last position.
static Boolean
TotallyUnknown(const long x, const long y, const long z,
  const long xSize, const long xySize, char *m)
{
dirT dir;
/* loop through the directions until we find a known spot */
for (dir = kFirstDir; dir <= kLastDir; dir++) {
  const long dx = d2x(dir);
  const long dy = d2y(dir);
  const long dz = d2z(dir);
  if (!myIsUnknown(map(m, x + dx, y + dy, z + dz,
     xSize, xySize)))
    return false;
  }
return true;
}
```

MoveTowardsNearWall

```
// Pick a move that is towards a near wall
// (NOTE: only used when all directions are unknown)
static dirT
MoveTowardsNearWall(const long x, const long y, const long z,
  const long xSize, const long ySize, const long zSize,
  char *m)
{
#pragma unused (m)
long distx = xSize - x - 1;  // distance from x==xMax edge
long disty = ySize - y - 1;  // distance from y==yMax edge
long distz = zSize - z - 1;  // distance from z==zMax edge
// [xyz] is distance from [xyz]==0 edge
long dx, dy, dz;
long xy = xSize * ySize;

if (x < distx) dx = -1;
else if (x == distx) dx = 0;
else dx = 1;
if (y < disty) dy = -1;
else if (y == disty) dy = 0;
else dy = 1;
if (z < distz) dz = -1;
else if (z == distz) dz = 0;
else dz = 1;

return di[dy+1][dz+1][dx+1];
}
```

Gravity

```
// Fall until we find a floor below us
// Only use after mapping new knowledge
// (Don't use during mapping)
static long
```

```
Gravity(long i, const char *m, const long xySize)
{
do {i -= xySize;}
  while (kWall != *(m + i));

return i + xySize;
}
```

SearchOneSquare

```
// Search from this square for an adjacent unknown square
// queueing up moveable squares (including spaces below
// falls) for further research later,
// marking enqueued squares as 'tried' (immediately meaning
// 'not-to-be-queued-for-trying', later 'actually-tried')
// NOTE: while I could mark falls as tried (upward from
// every space) they are rather unlikely to be in the
// search path, so it's not worth it.  Just re-enact
// gravity each time, and check last square for 'tried'.
static Boolean
SearchOneSquare(STNPtr startSTN,
  const long xSize, const long xySize, char *m)
{
dirT d;
const long startSquare = startSTN->square;

for (d = kFirstDir; d <= kLastDir; d++) {
  const long sqi = startSquare + offsetD(d,xSize,xySize);
  const char sq = *(m + sqi);
  if (myIsUnknown(sq)) {
    SetBestMove(startSTN, d);
    return true;
    }
  else if (myIsUntriedWalkable(sq)) {
    EnQNewCandidate(startSTN,sqi,d);
    *(m + sqi) = kTriedSpace;
    }
  else if (myIsUntriedFall(sq)) {
    const long bottomSqi = Gravity(sqi,m,xySize);
    const long bottomSq = *(m + bottomSqi);
    if (myIsUntriedWalkable(sq)) {
      EnQNewCandidate(startSTN,bottomSqi,d);
      *(m + bottomSqi) = kTriedSpace;
      }
    //else if (myIsUnknown(bottomSq)) {
    // BreakToSourceDebugger_(); // should be impossible
    // return true; //??
    // }
    else ;   //it's an already tried space, do nothing
    }
  else ;      // it's a wall or already tried space, do nothing
  }
return false;
}
```

FindNearestUnknown

```
// Find a move sequence that will lead to an unknown
// square (preferably an edge square?).
static dirT
FindNearestUnknown(const long x, const long y, const long z,
  const long xSize, const long ySize, const long zSize,
  char *m)
{
#pragma unused (zSize)
const long xySize = xSize * ySize;
Boolean found = false;
// make new search tree with square at x,y,z
NewSearchTree();
NewBestMoveList();
gTreeRoot->parent = nil;
gTreeRoot->square = offsetXYZ(x,y,z,xSize,xySize);
gTreeRoot->direction = kNoDir;
*(m + gTreeRoot->square) = kTriedSpace;
// fan out from this one layer at a time
while (!isEmptySearchQ() && !found) {
  STNPtr tryNode = DeQ();
  found = SearchOneSquare(tryNode, xSize, xySize, m);
  //FreeSTN(tryNode);
  }
DisposeSearchTree(m);
// found move(list) or failed
return found;
}
```

CalcBestMove

```
// Calculate the best move to try
// return true if we have found an exit or are at wit's end
static dirT
CalcBestMove(const long x, const long y, const long z,
   const long xSize, const long ySize, const long zSize,
   char *m)
{
static dirT lastTried = kNoDir;
dirT d;

if (!isEmptyMoveList()) {  // we have a pre-made list of moves
  lastTried = PopMoveList();
  }
else if (kNoDir != (d =
   APossibleExit(x,y,z,xSize,ySize,zSize,m,lastTried))
   ) {  // always check the possible exit first
  lastTried = d;
  }
// need more different cases to know whether this is useful:
else if (TotallyUnknown(x,y,z,xSize,xSize*ySize,m)) {
  lastTried = MoveTowardsNearWall(x,y,z,xSize,ySize,zSize,m);
  }
  /* insert smart cases here */
  /* use 'lastTried' here, too? */
else if (FindNearestUnknown(x,y,z,xSize,ySize,zSize,m)) {
  lastTried = PopMoveList();
  }
else
  lastTried = kNoDir;

return lastTried;  /* really next to try! */
}
```

Maze

```
// Basically, calculate the best move to make.  Try it.
// Use the results to increase our knowledge of the maze
// (by writing in the map area).  Repeat.
Boolean
Maze(long xMove, // these are your initial position
  long yMove,
  long zMove,
  long xSize,     // these are the dimensions of the maze
  long ySize,
  long zSize,
  MoveProc MakeAMove, // this is your callback routine
  char *mapStorage     // this is your preallocated storage
  )
{
long x = xMove;
long y = yMove;
long z = zMove;
Boolean done;
if (!Initialize(xMove,yMove,zMove,xSize,ySize,zSize,mapStorage))
  return false;    // out of memory, probably

do {
  long deltaX, deltaY, deltaZ;
  long oldx, oldy, oldz;
  dirT dir;
  dir = CalcBestMove(x,y,z,xSize,ySize,zSize,mapStorage);
  if (kNoDir == dir)
    break;                          // failed to find a useful move to make
  oldx = x; oldy = y; oldz = z;
  deltaX = d2x(dir); deltaY = d2y(dir); deltaZ = d2z(dir);
  done = MakeAMove(deltaX,deltaY,deltaZ,&x,&y,&z);
  if (isEmptyMoveList())         // don't remap move. we've been here
    MapMove(oldx, oldy, oldz, x, y, z, mapStorage,
      xSize, xSize*ySize, deltaX, deltaY, deltaZ);
  } while (!done);

exitHere:
DeInitialize();
return AtEdge(x,y,z,xSize-1,ySize-1,zSize-1);
}
```