CS106B
J Zelenski

Handout #19
Feb 1, 2008

Exhaustive recursion and backtracking

In some recursive functions, such as binary search or reversing a file, each recursive call makes just one recursive call. The "tree" of calls forms a linear line from the initial call down to the base case. In such cases, the performance of the overall algorithm is dependent on how deep the function stack gets, which is determined by how quickly we progress to the base case. For reverse file, the stack depth is equal to the size of the input file, since we move one closer to the empty file base case at each level. For binary search, it more quickly bottoms out by dividing the remaining input in half at each level of the recursion. Both of these can be done relatively efficiently.

Now consider a recursive function such as subsets or permutation that makes not just one recursive call, but several. The tree of function calls has multiple branches at each level, which in turn have further branches, and so on down to the base case. Because of the multiplicative factors being carried down the tree, the number of calls can grow dramatically as the recursion goes deeper. Thus, these *exhaustive recursion* algorithms have the potential to be very expensive. Often the different recursive calls made at each level represent a decision point, where we have choices such as what letter to choose next or what turn to make when reading a map. Might there be situations where we can save some time by focusing on the most promising options, without committing to exploring them all?

In some contexts, we have no choice but to exhaustively examine all possibilities, such as when trying to find some globally optimal result, But what if we are interested in finding any solution, whichever one that works out first? At each decision point, we can choose one of the available options, and sally forth, hoping it works out. If we eventually reach our goal from here, we're happy and have no need to consider the paths not taken. However, if this choice didn't work out and eventually leads to nothing but dead ends; when we **backtrack** to this decision point, we try one of the other alternatives.

What's interesting about backtracking is that we back up only as far as needed to reach a previous decision point with an as-yet-unexplored alternative. In general, that will be at the most recent decision point. Eventually, more and more of these decision points will have been fully explored, and we will have to backtrack further and further. If we backtrack all the way to our initial state and have explored all alternatives from there, we can conclude the particular problem is unsolvable. In such a case, we will have done all the work of the exhaustive recursion and known that there is no viable solution possible.

As was the case with recursion, simply discussing the idea doesn't usually make the concepts transparent, it is therefore worthwhile to look at many examples until you begin to see how backtracking can be used to solve problems. This handout contains code for several recursive backtracking examples. The code is short but dense and is somewhat sparsely commented, you should make sure to keep up with the discussion in lecture. The fabulous maze backtracking example is fully covered in the reader as an additional example to study.

Classic exhaustive permutation pattern

First, a procedural recursion example, this one that forms all possible re-arrangements of the letters in a string. It is an example of an exhaustive procedural algorithm. The pseudocode strategy is as follows:

> *If you have no more characters left to rearrange, print current permutation*
> *for (every possible choice among the characters left to rearrange) {*
> *Make a choice and add that character to the permutation so far*
> *Use recursion to rearrange the remaining letters*
> *}*

Here is the code at the heart of that algorithm:

```
void RecPermute(string soFar, string rest)
{
    if (rest.empty()) {
        cout << soFar << endl;
    } else {
        for (int i = 0; i < rest.length(); i++) {
            string remaining = rest.substr(0, i)
                             + rest.substr(i+1);
            RecPermute(soFar + rest[i], remaining);
        }
    }
}
```

In this exhaustive traversal, we try every possible combination. There are n! ways to rearrange the characters in a string of length n and this prints all of them.

This is an important example and worth spending the time to fully understand. The permutation pattern is at the heart of many recursive algorithms— finding anagrams, solving sudoku puzzles, optimally matching classes to classrooms, or scheduling for best efficiency can all be done using an adaptation of the general permutation code.

Classic exhaustive subset pattern

Another of the classic exhaustive recursion problems is listing all the subsets of a given set. The recursive approach is basically the same as the n-choose-k problem we looked at in lecture. At each step, we isolate an element from the remainder and then recursively list those sets that include that element, and recur again to build those sets that don't contain that element. In each case, the set of remaining elements is one smaller and thus represents a slightly easier, smaller version of the same problem. Those recursive calls will eventually lead to the simplest case, that of listing the subsets of an empty set. Here is the pseudocode description:

If there are no more elements remaining,
print current subset
else {
Consider the next element of those remaining
Try adding it to the current subset, and use recursion to build subsets from here
Try not adding it to current subset, and use recursion to build subsets from here
}

In this version of the code, the set elements are represented as characters and a string is used to hold the characters for a set, this allows the code to be very simple.

```
void RecSubsets(string soFar, string rest)
{
   if (rest.empty())
    cout << soFar << endl;
   else {
     RecSubsets(soFar + rest[0], rest.substr(1)); // include first char
     RecSubsets(soFar, rest.substr(1));           // exclude first char
   }
}
```

This is another exhaustive procedural algorithm. It builds every possible combination of the elements. Each call makes two additional recursive calls and continues to a depth of n, thus, the overall algorithm is 2^n. This should match your intuition, because the "power set" is known to be 2^n in size. The in-out subset pattern is another extremely common pattern that has relevance to the algorithms for such tasks as optimally packing a constrained space, partitioning sets, and finding the longest shared subsequence.

Recursive backtracking
Note that in both the permutation and subset examples, we explore every possibility. At each decision point, we make a choice, then unmake it and try the other options, until we have fully exhausted all of our options. This can be expensive, especially if there are a lot of decision points and many choices at each. In backtracking, we will try one choice and only undo that decision if it doesn't work out. Here is some generic pseudocode that shows the structure of a backtracking algorithm:

```
bool Solve(configuration conf)
{
   if (no more choices)     // BASE CASE
         return (conf is goal state);

   for (all available choices) {
     try one choice c;
                              // recursively solve after making choice
     ok = Solve(conf with choice c made);
      if (ok)
          return true;
      else
          unmake choice c;
    }

    return false;   // tried all choices, no soln found
}
```

One great tip for writing a backtracking function is to abstract away the details of managing the configuration (what choices are available, making a choice, checking for success, etc.) into other helper functions so that the body of the recursion itself is as clean as can be. This helps to make sure you have the heart of the algorithm correct and allows the other pieces to be developed, test, and debugged independently.

First, let's just take the exhaustive permutation code and change it into a backtracking algorithm. We want to take a string of letters and attempt to rearrange it into a valid word (as found in our lexicon). It seems like permutations is the right start, but also, you can agree that once we find a word, we can stop, no need to keep looking. So we start with the standard permutation code and change it to return a result that is either the word that was found or the empty string on failure. The base case does the check in the lexicon to know whether a given permutation is a word or not. Each time we make a recursive call, we check the result to see if it was successful, and if so, stop here. Only if it fails do we continue exploring the other choices. The empty string at the bottom of the function is what triggers the backtracking from here.

```
string FindWord(string soFar, string rest, Lexicon &lex)
{
  if (rest.empty()) {
    return (lex.containsWord(soFar)? soFar : "");
  } else {
    for (int i = 0; i < rest.length(); i++) {
      string remain = rest.substr(0, i) + rest.substr(i+1);
      string found = FindWord(soFar + rest[i], remain, lex);
      if (!found.empty()) return found;
    }
  }

  return "";   // empty string indicates failure
}
```

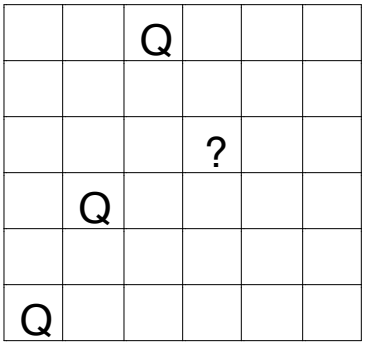
A nice heuristic we can employ here is to prune paths that are known to be dead-ends. For example, if permuting the input "zicquzcal", once you have assembled "zc" as the leading characters, a quick check in the lexicon will tell you that there are no English words that start with that prefix, so there is no reason to further explore the permutations from here. This means inserting an additional base case that returns the empty string right away when **`soFar`** is not a valid prefix.

The Venerable 8-Queens

This one is a classic in computer science. The goal is to assign eight queens to eight positions on an 8x8 chessboard so that no queen, according to the rules of normal chess play, can attack any other queen on the board. In pseudocode, our strategy will be:

Start in the leftmost columm

If all queens are placed, return true
for (every possible choice among the rows in this column)
if the queen can be placed safely there,
make that choice and then recursively try to place the rest of the queens
if recursion successful, return true
if !successful, remove queen and try another row in this column
if all rows have been tried and nothing worked, return false to trigger backtracking




```
/*
 * File: queens.cpp
 * ----------------
 * This program implements a graphical search for a solution to the N
 * N queens problem, utilizing a recursive backtracking approach. See
 * comments for Solve function for more details on the algorithm.
 */
#include "chessGraphics.h"    // routines to draw chessboard & queens

const int NUM_QUEENS = 8;

bool Solve(Grid<bool> &board, int col);
void PlaceQueen(Grid<bool> &board, int row, int col);
void RemoveQueen(Grid<bool> &board, int row, int col);
bool RowIsClear(Grid<bool> &board, int queenRow, int queenCol);
bool UpperDiagIsClear(Grid<bool> &board, int queenRow, int queenCol);
bool LowerDiagIsClear(Grid<bool> &board, int queenRow, int queenCol);
bool IsSafe(Grid<bool> &board, int row, int col);
void ClearBoard(Grid<bool> &board);

int main()
{
   Grid<bool> board(NUM_QUEENS, NUM_QUEENS);

   InitGraphics();
   ClearBoard(board);                // Set all board positions to false
   DrawChessboard(board.numRows()); // Draw empty chessboard
   Solve(board,0);                   // Attempt to solve the puzzle
   return 0;
}
```

```
/*
 * Function: Solve
 * ---------------
 * This function is the main entry in solving the N queens problem. It takes
 * the partially filled board and the column we are trying to place queen in.
 * It will return a boolean value which indicates whether or not we found a
 * successful arrangement starting from this configuration.
 *
 * Base case:  if there are no more queens to place, we have succeeded!
 *
 * Otherwise, we find a safe row in this column, place a queen at (row,col)
 * recursively call Solve starting at the next column using this new board
 * configuration. If that Solve call fails, we remove that queen from (row,col)
 * and try again with the next safe row within the column.  If we have tried all
 * rows in this column and haven't found a solution, we return false from this
 * invocation, which will force backtracking from this unsolvable configuration.
 *
 * The starting call to Solve has an empty board and places a queen in col 0:
 *       Solve(board, 0);
 */
bool Solve(Grid<bool> &board, int col)
{
   if (col >= board.numCols()) return true;     // base case

   for (int rowToTry = 0; rowToTry < board.numRows(); rowToTry++)  {
      if (IsSafe(board, rowToTry, col)) {
         PlaceQueen(board, rowToTry, col);          // try queen here
         if (Solve(board, col + 1)) return true;    // recur to place rest
         RemoveQueen(board, rowToTry, col);         // failed, remove, try again
      }
   }
   return false;
}


/*
 * Function: PlaceQueen
 * --------------------
 * Places a queen in (row,col) of the board by setting value in
 * grid to true and drawing a 'Q' in that square on the displayed chessboard.
 */
void PlaceQueen(Grid<bool> &board, int row, int col)
{
   board(row, col) = true;
   DrawLetterAtPosition('Q', row, col, "Black");
}

/*
 * Function: RemoveQueen
 * ---------------------
 * Removes a queen from (row,col) of the board by setting value in grid to
 * false and erasing the 'Q' from that square on the displayed chessboard.
 */
void RemoveQueen(Grid<bool> &board, int row, int col)
{
    board(row, col) = false;
    EraseLetterAtPosition(row, col);
}
```

```
/*
 * Function: IsSafe
 * ----------------
 * Given a partially filled board and location (row,col), returns boolean
 * which indicates whether that position is safe (i.e. not threatened by
 * another queen already on the board.)
 */
bool IsSafe(Grid<bool> &board, int row, int col)
{
    return (LowerDiagIsClear(board, row, col) &&
            RowIsClear(board, row, col) &&
            UpperDiagIsClear(board, row, col));
}

/*
 * Function: RowIsClear
 * --------------------
 * Given a partially filled board and location (queenRow, queenCol), checks
 * that there is no queen in the row queenRow.
 */
bool RowIsClear(Grid<bool> &board, int queenRow, int queenCol)
{
   for (int col = 0; col < queenCol; col++) {
      if (board(queenRow, col))     // there is already a queen in this row!
         return false;
   }
   return true;
}


/*
 * Function: UpperDiagIsClear
 * --------------------------
 * Given a partially filled board and location (queenRow, queenCol), checks
 * there is no queen along northwest diagonal through (queenRow, queenCol).
 */
bool UpperDiagIsClear(Grid<bool> &board, int queenRow, int queenCol)
{
   int row, col;
   for (row = queenRow, col = queenCol; col >= 0 && row < board.numRows();
                                                        row++, col--) {
      if (board(row, col))    // there is already a queen along this diagonal!
         return false;
   }
   return true;
}


/*
 * Function: LowerDiagIsClear
 * --------------------------
 * Given a partially filled board and (queenRow, queenCol), checks that there
 * is no queen along southwest diagonal through (queenRow, queenCol).
 */
bool LowerDiagIsClear(Grid<bool> &board, int queenRow, int queenCol)
{
   int row, col;
   for (row = queenRow, col = queenCol; row >= 0 && col >= 0; row--, col--) {
```

```
      if (board(row, col))    // there is already a queen along this diagonal!
         return false;
   }
   return true;
}


/*
 * Function: ClearBoard
 * --------------------
 * Simply initializes the board to be empty, ie no queen on any square.
 */
void ClearBoard(Grid<bool> &board)
{
   for (int row = 0; row < board.numRows(); row++)
      for (int col = 0; col < board.numCols(); col++)
         board(row, col) = false;
}
```

Solving Sudoku puzzles

Everyone who's everyone is crazy for this little logic puzzle that involving filling numbers into grid. The goal of Sudoku is to assign digits to the empty cells so that every row, column, and subgrid contains exactly one instance of the digits from 1 to 9. The starting cells are assigned to constrain the puzzle such that there is only one way to finish it. Sudoku solvers pride themselves on the fact that there is no need to "guess" to solve the puzzle, that careful application of logic will lead you to the solution. However, a computer solver can make and unmake guesses fast enough to not care, so let's just throw some recursive backtracking at it!

3		6	5		8	4		
5	2							
	8	7					3	1
		3		1			8	
9			8	6	3			5
	5			9		6		
1	3					2	5	
							7	4
		5	2		6	3		

In pseudocode, our strategy is:

Find row, col of an unassigned cell
If there is none, return true

For digits from 1 to 9
if there is no conflict for digit at row,col
assign digit to row,col and recursively try fill in rest of grid
if recursion successful, return true
if !successful, remove digit and try another
if all digits have been tried and nothing worked, return false to trigger backtracking

And below is the C++ code for the solver. Note it code (not including comments) is just a page long. As with the queens solution, by moving the slightly messy problem-specific details (finding unassigned, determining conflicts, etc.) to helper functions, the Solve function becomes just a clean restatement of the idiomatic recursive backtracking pattern.

```
/*
 * Function: SolveSudoku
 * ---------------------
 * Takes a partially filled-in grid and attempts to assign values to all
 * unassigned locations in such a way to meet the requirements for Sudoku
 * solution (non-duplication across rows, columns, and boxes). The function
 * operates via recursive backtracking: it finds an unassigned location with
 * the grid and then considers all digits from 1 to 9 in a loop. If a digit
 * is found that has no existing conflicts, tentatively assign it and recur
 * to attempt to fill in rest of grid. If this was successful, the puzzle is
 * solved. If not, unmake that decision and try again.  If all digits have
 * been examined and none worked out, return false to backtrack to previous
 * decision point.
 */
bool SolveSudoku(Grid<int> &grid)
{
   int row, col;

   if (!FindUnassignedLocation(grid, row, col)) return true; // success!

   for (int num = 1; num <= 9; num++) {         // consider digits 1 to 9
      if (NoConflicts(grid, row, col, num)) {   // if looks promising,
         grid(row, col) = num;                  // make tentative assignment
         if (SolveSudoku(grid)) return true;    // recur, if success, yay!
         grid(row, col) = UNASSIGNED;           // failure, unmake & try again
      }
   }
   return false;        // this triggers backtracking
}

/*
 * Function: FindUnassignedLocation
 * --------------------------------
 * Searches the grid to find an entry that is still unassigned. If found,
 * the reference parameters row, col will be set the location that is
 * unassigned, and true is returned. If no unassigned entries remain, false
 * is returned.
 */
bool FindUnassignedLocation(Grid<int> &grid, int &row, int &col)
{
   for (row = 0; row < grid.numRows(); row++)
      for (col = 0; col < grid.numCols(); col++)
         if (grid(row, col) == UNASSIGNED) return true;
   return false;
}

/*
 * Function: NoConflicts
 * ---------------------
 * Returns a boolean which indicates whether it will be legal to assign
 * num to the given row,col location. As assignment is legal if it that
 * number is not already used in the row, col, or box.
 */
```

```
bool NoConflicts(Grid<int> &grid, int row, int col, int num)
{
   return !UsedInRow(grid, row, num) && !UsedInCol(grid, col, num)
         && !UsedInBox(grid, row - row%3 , col - col%3, num);
}

/*
 * Function: UsedInRow
 * -------------------
 * Returns a boolean which indicates whether any assigned entry
 * in the specified row matches the given number.
 */
bool UsedInRow(Grid<int> &grid, int row, int num)
{
   for (int col = 0; col < grid.numCols(); col++)
      if (grid(row, col) == num) return true;
   return false;
}

/*
 * Function: UsedInCol
 * -------------------
 * Returns a boolean which indicates whether any assigned entry
 * in the specified column matches the given number.
 */
bool UsedInCol(Grid<int> &grid, int col, int num)
{
   for (int row = 0; row < grid.numRows(); row++)
      if (grid(row, col) == num) return true;
   return false;
}

/*
 * Function: UsedInBox
 * -------------------
 * Returns a boolean which indicates whether any assigned entry
 * within the specified 3x3 box matches the given number.
 */
bool UsedInBox(Grid<int> &grid, int boxStartRow, int boxStartCol, int num)
{
   for (int row = 0; row < 3; row++)
      for (int col = 0; col < 3; col++)
      if (grid(row+boxStartRow, col+boxStartCol) == num)
         return true;
   return false;
}
```

Solving cryptarithmetic puzzles
Newspapers and magazines often have cryptarithmetic puzzles of the form:

```
  SEND
+ MORE
 MONEY
```

The goal here is to assign each letter a digit from 0 to 9 so that the arithmetic works out correctly. The rules are that all occurrences of a letter must be assigned the same digit, and no digit can be assigned to more than one letter. First, I will show you a workable, but not very efficient strategy and then improve on it.

In pseudocode, our first strategy will be:

First, create a list of all the characters that need assigning to pass to Solve

If all characters are assigned, return true if puzzle is solved, false otherwise
Otherwise, consider the first unassigned character
for (every possible choice among the digits not in use)
make that choice and then recursively try to assign the rest of the characters
if recursion sucessful, return true
if !successful, unmake assignment and try another digit
if all digits have been tried and nothing worked, return false to trigger backtracking

And here is the code at the heart of the recursive program (other code was excluded for clarity):

```
/* ExhaustiveSolve
 * ---------------
 * This is the "not-very-smart" version of cryptarithmetic solver. It takes
 * the puzzle itself (with the 3 strings for the two addends and sum) and a
 * string of letters as yet unassigned. If no more letters to assign
 * then we've hit a base-case, if the current letter-to-digit mapping solves
 * the puzzle, we're done, otherwise we return false to trigger backtracking
 * If we have letters to assign, we take the first letter from that list, and
 * try assigning it the digits from 0 to 9 and then recursively working
 * through solving puzzle from here. If we manage to make a good assignment
 * that works, we've succeeded, else we need to unassign that choice and try
 * another digit. This version is easy to write, since it uses a simple
 * approach (quite similar to permutations if you think about it) but it is
 * not so smart because it doesn't take into account the structure of the
 * puzzle constraints (for example, once the two digits for the addends have
 * been assigned, there is no reason to try anything other than the correct
 * digit for the sum) yet it tries a lot of useless combos regardless.
 */
bool ExhaustiveSolve(puzzleT puzzle, string lettersToAssign)
{
   if (lettersToAssign.empty()) // no more choices to make
       return PuzzleSolved(puzzle); // checks arithmetic to see if works
   for (int digit = 0; digit <= 9; digit++) { // try all digits
       if (AssignLetterToDigit(lettersToAssign[0], digit)) {
           if (ExhaustiveSolve(puzzle, lettersToAssign.substr(1)))
               return true;
           UnassignLetterFromDigit(lettersToAssign[0], digit);
       }
   }
   return false;  // nothing worked, need to backtrack
}
```

The algorithm above actually has a lot in common with the permutations algorithm, it pretty much just creates all arrangements of the mapping from characters to digits and tries each until one works or all have been unsuccessfully tried. For a large puzzle, this could take a while.

A smarter algorithm could take into account the structure of the puzzle and avoid going down dead-end paths. For example, if we assign the characters starting from the ones place and moving to the left, at each stage, we can verify the correctness of what we have so far before we continue onwards. This definitely complicates the code but leads to a tremendous improvement in efficiency, making it much more feasible to solve large puzzles.

Our pseudocode in this case has more special cases, but the same general design:

```
Start by examining the rightmost digit of the topmost row, with a carry of 0
If we are beyond the leftmost digit of the puzzle, return true if no carry, false otherwise

If we are currently trying to assign a char in one of the addends
   If char already assigned, just recur on row beneath this one, adding value into sum
   If not assigned, then
        for (every possible choice among the digits not in use)
              make that choice and then on row beneath this one, if  successful, return true
             if !successful, unmake assignment and try another digit
       return false if no assignment worked to trigger backtracking
Else if trying to assign a char in the sum
   If char assigned & matches correct,
         recur on next column to the left with carry, if success return true
   If char assigned & doesn't match, return false
   If char unassigned & correct digit already used, return false
   If char unassigned & correct digit unused,
         assign it and recur on next column to left with carry, if success return true
 return false to trigger backtracking
```

Path finding

Chapter 6 of the reader does an excellent job working through another recursive backtracking example, finding a path through a maze. I won't repeat the maze discussion here, but it's another illustrative problem worth studying and mastering. For assignment 2, you wrote an iterative breadth-first solver. The version in chapter 6 is a recursive depth-first solver. It is interesting to think about how the two strategies differ, in terms of coding ease, solving power, efficiency, and so on.