CS900 Lab - Day 07

July 8th, 2003

Instructor: Thomas Hofmann

New Topics Covered

Recursion – Lecture: Day 7, King, Section 9.6
Combinatorial Search – Lecture: Day 7

Installing

Copy all files from the directory /course/cs900/day07/src by typing

```
cd cs900
mkdir day07
cd day07
cp /course/cs900/day07/src/* .
```

This should include the files euclid.c, power.c, maze.c, queens.c and knight.c (as well as our friends g2.h g2_X11.h and libg2.a).

Problem 1 - Recursive Functions: GCD

The Greatest Common Divisor (GCD) of two numbers is the largest number that will divide both numbers with no remainder. For example, the GCD of 12 and 8 is 4, because 12 / 4 = 3 and 8 / 4 = 2 and 4 is the largest number with that property. Euclid's approach for computing the GCD is based on the following observation:

Let a and b be two numbers. If $b|a$ (read "b divides a," meaning the remainder of a / b is 0), then gcd(a, b) = b. Otherwise, $a = tb + r$ ($a / b = t$, with a remainder of r), and gcd(a, b) = gcd(b, r).

So the base case can be expressed in C as

```
if (a%b==0) return b;
```

whereas the inductive is given by

```
return gcd(b,a%b);
```

Implement a recursive function in your euclid.c file. Write some code in `main()` that asks the user to enter two numbers and then prints out their GCD.

Problem 2 - Computing Powers

Write a program with a function `power` that computes the `n`-th power (`n` being an integer) of a real-valued number `x`. Use a recursion scheme that reduces the exponent by (at least) one half with every recursion level. For example, in order to compute x^{16} the function should compute x^{8} on the next recursion level (and not, for example x^{15}). Moreover, the function `power` should have exactly two calls to itself in the function body (see hint).

Implement the `power` function in your file power.c. Write a simple `main` program that requests a number `x` and an exponent `n` and outputs the result, `x` raised to the `n`-th power.

Hint: You need to account for positive, negative, zero, odd, and even powers. For simplicity's sake, don't bother with fractions.

Problem 3 – The Great Escape

Modify the recursive search program to compute the knight's tour we discussed in class to find a path (not necessarily the shortest one) from a start location to the exit through a maze of walls. The raw data defining the maze is included in your maze.c file and represented by a global character array. There are also helper functions which check whether a location specified by row and column is occupied (a wall) or that start or end location. The return value is a Boolean variable (0/1):

```
int isOccupied(int row, int col);
int isStart   (int row, int col);
int isExit    (int row, int col);
```

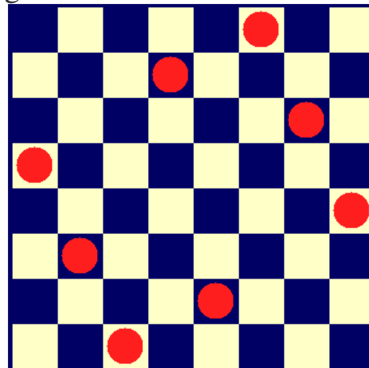
You should stick to the design presented in class and simply modify the relevant functions and constants appropriately. This should be amazingly simple!

Hint: Think of the walls as being squares that are always occupied, while the path should avoid cycles and hence it should not visit squares that have previously been visited (similar to the knight's tour).

There is also a file xmaze.c available that offers some graphics functionality for those interested in these kinds of things.

Problem 4 – Queens

Write a program using a recursive function `queens` to compute a solution to the n queens problem. The problem can be described as follows: On a n-by-n chess board, a queen can move as far as she pleases, horizontally, vertically, or diagonally. The queen's problem asks how to place n queens on the board so that none of them can hit any other in one move. The following figure shows a solution for the case of an ordinary chess board with n=8. Follow the example discussed in class for the knight's tour in designing this function.



Your program should be able to compute solutions to the n-queens problem for arbitrary n. We recommend that you represent the board by global variables on which functions `MakeMove` and `UnmakeMove` operate. Your program should also implement a simple print function that displays the solution using simple `printf` statements.

Investigate up to what n your program is still able to compute a solution within a couple of seconds.

Hint: Think of the act of putting an additional queen on the board as a "move". In the first move, put a queen in the first column, in the second move put a queen in the second column, etc. At some point you may not be able to put a queen in the m-th column so that it does not attack any of the queens already on the board, in which case the program will have to backtrack.