

Recursive Search with Backtracking, cont'd

CS 311 Data Structures and Algorithms

Lecture Slides

Monday, February 25, 2008

Glenn G. Chappell

Department of Computer Science
University of Alaska Fairbanks

CHAPPELLG@member.ams.org

© 2005–2008 Glenn G. Chappell

Unit Overview

Recursion & Searching

Major Topics

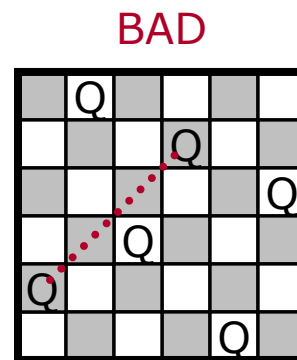
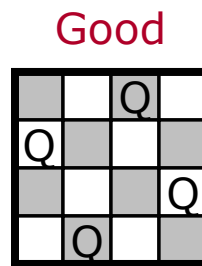
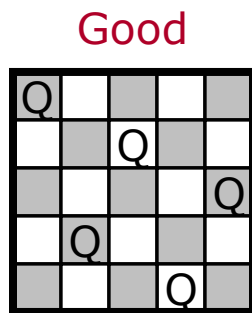
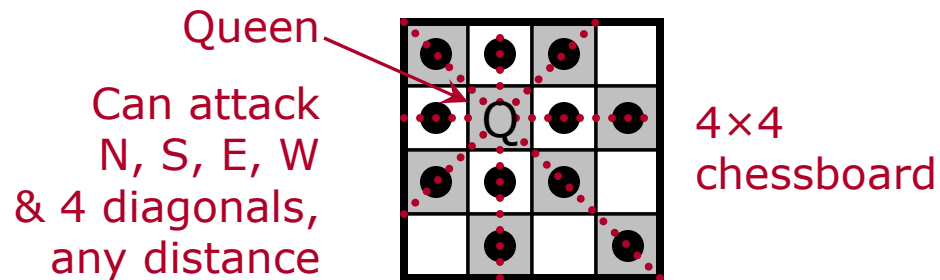
- ✓ • Introduction to Recursion.
- ✓ • Search Algorithms.
- ✓ • Recursion vs. Iteration.
- ✓ • Eliminating Recursion.
- $\frac{1}{2}$ ✓ • Recursive Search with Backtracking.

Review

Recursive Search with Backtracking [1/3]

The n -Queens Problem.

- Place n queens on an $n \times n$ chessboard so that none of them can attack each other.



Review

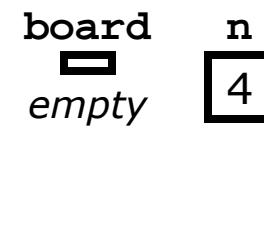
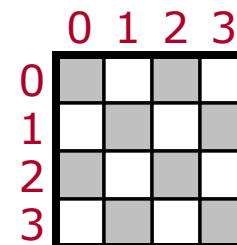
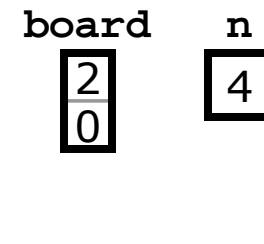
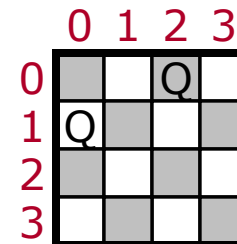
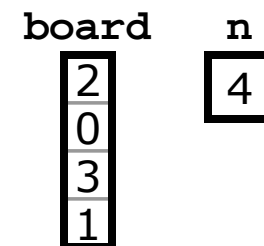
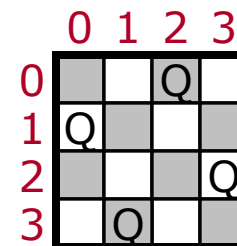
Recursive Search with Backtracking [2/3]

Representing a Partial Solution

- Number rows and columns 0 .. $n-1$.
- Two variables:
 - Variable `board` (vector of `ints`).
 - Variable `n` (`int`).
- Variable `n` holds the number of rows/columns in a full solution.
- Variable `board` holds the columns of the queens already placed, one per row.
- The size of `board` is the number of rows in which queens have been placed.

Partial Solution

Representation



Review

Recursive Search with Backtracking [3/3]

The Code

- **Nonrecursive wrapper function**
 - Creates an empty partial solution.
 - Calls the workhorse function with this partial solution.
- **Recursive workhorse function** is given a partial solution, prints all full solutions that can be made from it.
 - Do we have a full solution?
 - If so, output it.
 - Do we have a clear dead end? } ←
 - If so, simply return.
 - Otherwise:
 - Make a recursive call for each way of extending the partial solution. } ←

Note:

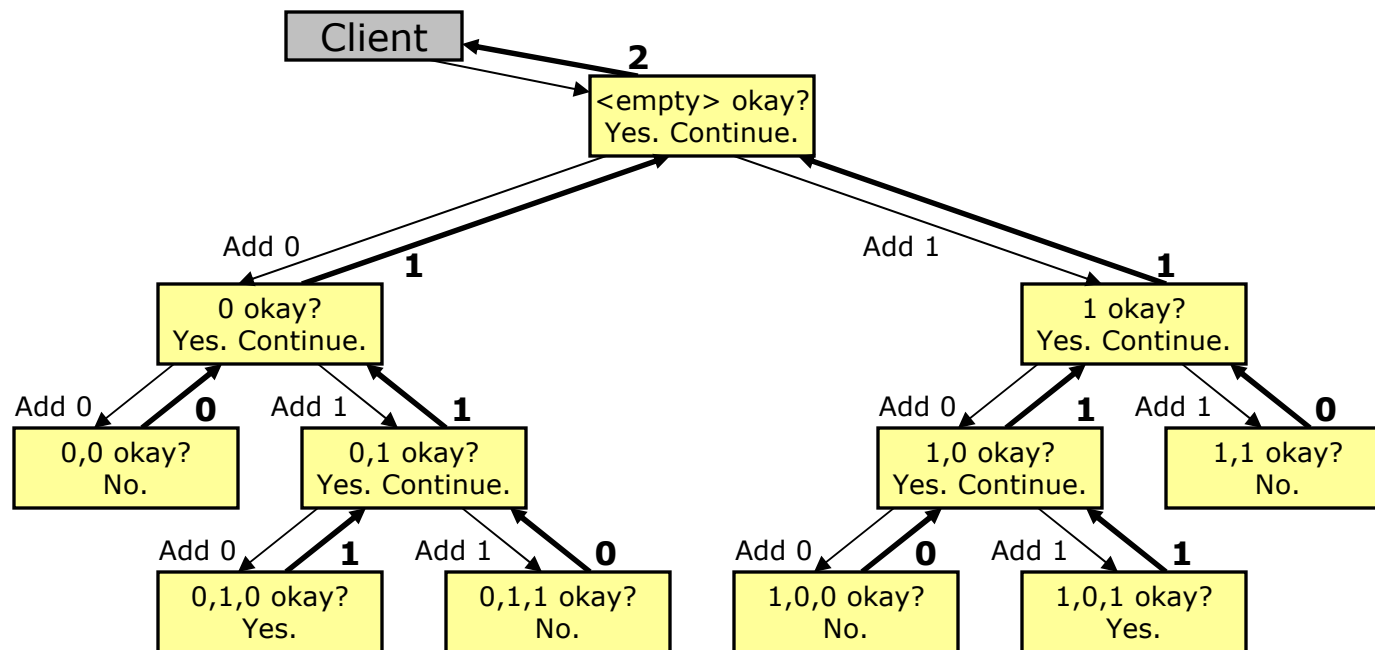
This part **might** not be necessary. Another way to handle dead ends is simply not to make any recursive calls when we get to this part.

Recursive Search with Backtracking, cont'd

Counting Solutions — Diagram

We can use a similar approach to **count solutions**. Each recursive call returns the number of full solutions to be found based on the given partial solution.

- Base Cases
 - "Found a solution" returns 1.
 - "Didn't work" returns 0.
- Recursive Case
 - Make recursive calls, add their return values, and return this.



Recursive Search with Backtracking, cont'd

Counting Solutions — How to Do It

The Code

- **Nonrecursive wrapper function**
 - Creates an empty partial solution.
 - Calls the workhorse function with this partial solution.
 - Returns the return value of the workhorse function.
- **Recursive workhorse function** is given a partial solution, returns the number of full solutions that can be made from it.
 - Do we have a full solution?
 - If so, return 1.
 - Do we have a clear dead end? } ← As before, this **might** be unnecessary.
 - If so, return 0.
 - Otherwise:
 - Set *total* to zero.
 - For each way of extending the current partial solution, make a recursive call, and add its return value to *total*.
 - Return *total*.

Recursive Search with Backtracking, cont'd

Counting Solutions — Try It

TO DO

- Modify the n -queens code to **count** all possible non-attacking arrangements of n queens, instead of printing them.

*Done. See `nqueencount.cpp`,
on the web page.*

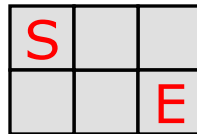
Recursive Search with Backtracking, cont'd

Spider Tours — Problem Description [1/4]

Now we look at the problem you are to solve in Assignment 4.

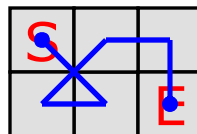
Consider a rectangle divided into squares. One square is “start”, and another is “end”. We call the result a **board**.

- An example 3×2 board is shown below.



Now place a “spider” on the board.

- The spider begins on the start square.
- At each step, the spider goes north, south, east, west, or any of the four diagonal directions, to an adjacent square.
- We want the spider to finish on the “end” square, having visited every square exactly once. A path that the spider can follow, in order to accomplish this, is a **spider tour**. An example spider tour of the above board is shown below.

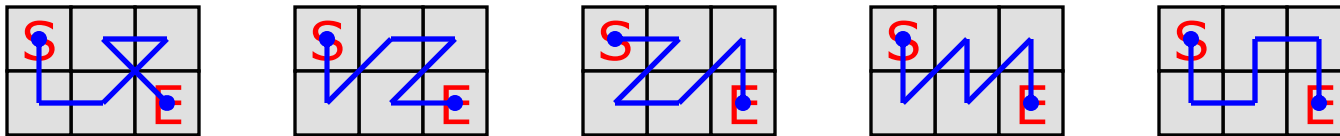


Recursive Search with Backtracking, cont'd

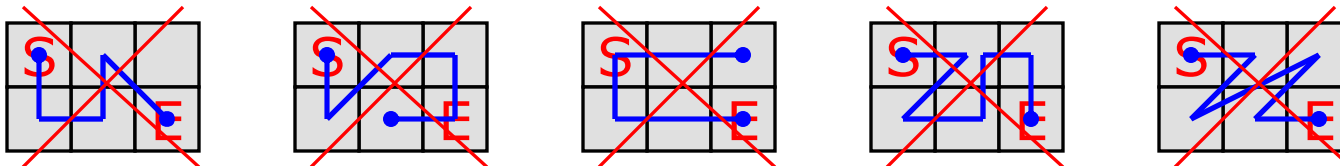
Spider Tours — Problem Description [2/4]

How many different spider tours does this board have?

- The answer turns out to be six: the one on the previous slide, and the five below.



Here are some paths that are **not** spider tours.

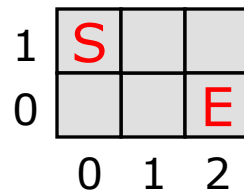


Recursive Search with Backtracking, cont'd

Spider Tours — Problem Description [3/4]

Place a coordinate system on a board, as shown below.
Specify squares as x, y .

- Giving the horizontal coordinate first, the start square is $(0, 1)$, and the end square is $(2, 0)$.



You are to write a function `countSpiderTours` that takes a board size, start square, and end square, each specified as x, y , and returns the number of spider tours the board has.

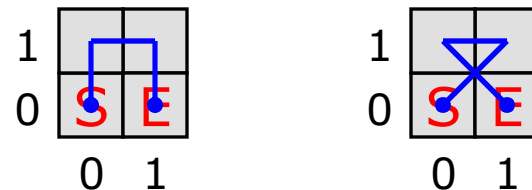
- So `countSpiderTours(3, 2, 0, 1, 2, 0)` should return 6.

Recursive Search with Backtracking, cont'd

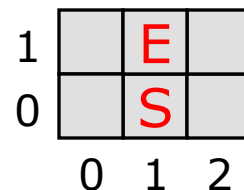
Spider Tours — Problem Description [4/4]

Other sizes and start/end squares are possible.

- Here is a 2×2 board.



- The above are the only possible spider tours on this board. Thus, `countSpiderTours(2, 2, 0, 0, 1, 0)` should return 2.
- Here is a 3×2 board with no possible spider tours at all.



- Thus, `countSpiderTours(3, 2, 1, 0, 1, 1)` should return 0.

Recursive Search with Backtracking, cont'd Spider Tours — Writing It [1/7]

We consider how to write `countSpiderTours` using our recursive methods.

In the following slides I will offer some *suggestions* on how to do this. However, the requirements of the assignment allow for quite a bit of variation. You do not need to follow my suggestions.

- Requirements in a nutshell:
 - Wrapper function `countSpiderTours`.
 - Recursive workhorse function `countSpiderTours_recurse`, is given a partial solution, returns number of full solutions based on this partial solution.
- Note: If you have trouble with this sort of thing, then *follow my suggestions!*

Recursive Search with Backtracking, cont'd

Spider Tours — Writing It [2/7]

How can we write `countSpiderTours` using our recursive methods?

- We need a recursive workhorse function: `countSpiderTours_recurse`.
 - In fact, the assignment requires this.

What **information** do we need to maintain? Ideas:

- A board is a vector of `ints`. Each item corresponds to a square. An item is `1` if the spider has been to the square, otherwise `0`.

```
typedef std::vector<int> Board;
```

- We also need to know:
 - The board size: `x` and `y`.
 - The spider's current position: `x` and `y`.
 - The end square: `x` and `y`.
 - The number of squares covered so far.
 - Not strictly necessary, but helpful (I will call this "`numCovered`").

We do *not* need to remember the start square.

← Wrap these up in an object, **if you want**. (I did not.)

Recursive Search with Backtracking, cont'd

Spider Tours — Writing It [3/7]

Conceptually, a board is a 2-D array. However, I suggest storing it in a 1-D (smart) array.

Why: 2-D smart (or dynamic) arrays in C++ are a serious pain to declare and initialize. I suggest avoiding them.

How: You can simulate a 2-D array using a 1-D array in the same way that C++ does it internally.

- In order to simulate a 2-D `int` array with dimensions `x` and `y` (think `"int array[x][y];"`), we can use a 1-D vector with dimension `x*y`.

```
typedef std::vector<int> Board;
Board b(x*y); // size = x*y
```

- To look up item `i, j` (think `"array[i][j]"`) use the subscript `i*y+j`.

```
b[i*y+j] = 1; // item i, j in conceptual 2-D array
```

Recursive Search with Backtracking, cont'd

Spider Tours — Writing It [4/7]

What is a **partial solution**?

- Rules. A partial solution:
 - Starts at the start square.
 - Makes legal moves (N, S, E, W, or diagonal, staying on board).
 - Never hits a square twice.
- Therefore, we have a *full* solution if:
 - The spider is on the end square.
 - The number of squares covered (`numCovered`) equals the number of squares on the board.

Recursive Search with Backtracking, cont'd

Spider Tours — Writing It [5/7]

In an empty solution:

- The board is all 0's, except for the start square, which is 1.
- The spider is on the start square.
- The number of squares covered is 1.

Procedure for workhorse function:

- Check for a full solution (see previous slide).
 - If so, return 1.
- Check for a dead end (e.g., spider on end, but not all squares covered).
 - If so, return 0.
- Set *total* to zero.
- For each of the eight squares adjacent to the spider's current position:
 - Check if this (1) lies on the board and (2) is not-yet-visited.
 - If so:
 - Move current position.
 - Mark square visited.
 - Increment number of squares covered.
 - Make recursive call.
 - Add return value to *total*.
 - Restore all changes (except change to *total*).
- Return *total*.

Recursive Search with Backtracking, cont'd

Spider Tours — Writing It [6/7]

More Suggestions

- If you are careful to leave the board in the same state when `countSpiderTours_recurse` ends as when it began, then you can pass the board by reference, avoiding the copy.

```
int countSpiderTours_recurse(Board & b, ...
```

- Why not pass by reference-to-const?
- You may find formulas for the number of spider tours in special situations (or in general). Such formulas are certainly of interest, but do not use them in your code; they will not help you meet the requirements of the assignment.

Recursive Search with Backtracking, cont'd

Spider Tours — Writing It [7/7]

Final Notes

- Again, you do **not** have to write your code the way I have outlined. Any code that meets the requirements of the assignment is acceptable.
 - Function `countSpiderTours`: prototyped as required.
 - Function `countSpiderTours_recurse`: recursive, takes partial solution and counts final solutions, does the bulk of the work.
- There will be a test program. Test your code!
- If you make use of someone else's code (like mine?), give credit!

Unit Overview

Algorithmic Efficiency & Sorting

Next we begin a unit on Algorithmic Efficiency and Sorting.

- Topics will include:
 - Introduction to Analysis of Algorithms
 - Introduction to Sorting
 - Sorting Algorithms
 - Lots of them!
 - More on Big- O
 - The Limits of Sorting
 - Divide-and-Conquer
 - Practical Sorting
- We will (mostly) follow the text.
 - Efficiency and sorting are in Chapter 9.

After the Midterm Exam, we will look at Data Abstraction.

- Chapter 3.