The 8-queens problem

CS 5010 Program Design Paradigms
“Bootcamp”
Lesson 9.3
Introduction

• In this lesson, we'll consider another application of graph searching: the eight queens problem.
• We'll study this as an example of searching in a graph.
• Along the way we'll learn something more about \textit{layered design}. 
Layered Design

• In layered design, we write a data design and a set of procedures for each data type.
• We try to manipulate the values of the type only through the procedures.
• We already did this once— we hooked things up so that our graph programs (reachables and path?) didn't care how the graphs were represented, so long as we had a successor function that gave right answers.
• In general, we start with the lowest-level pieces and work our way up.
The problem for this lesson: 8-queens

• Find a placement of 8 queens on a chessboard so that no queen can capture another queen.

• Here's one solution:
What can a queen capture?

- A queen can move any number of spaces horizontally, vertically, or diagonally
What can a queen capture?

• If the queen is at row $r$ and column $c$, then it can attack any square $(r', c')$ such that
  • $r' = r$ (horizontal movement)
  • $c' = c$ (vertical movement)
  • $r' + c' = r + c$ (northwest-southeast movement)
  • $r' - c' = r - c$ (northeast-southwest movement)
Of course, we'll generalize to boards of other sizes

• and our data representation should be independent of board size.

• If we need information about the board size, we'll put that in an invariant.
Data Design for Queen

;;; Queens:
(define-struct queen (row col))
;;; A Queen is a (make-queen PosInt PosInt)

;;; Queen Queen -> Boolean
(define (threatens? q1 q2)
  (or
   (= (queen-row q1) (queen-row q2))
   (= (queen-col q1) (queen-col q2))
   (=+
     (+ (queen-row q1) (queen-col q1))
     (+ (queen-row q2) (queen-col q2)))
   (=+
     (- (queen-row q1) (queen-col q1))
     (- (queen-row q2) (queen-col q2))))

;;; Queen ListOf<Queens> -> Boolean
(define (threatens-any? this-queen other-queens)
  (ormap
   (lambda (other-queen) (threatens? this-queen other-queen))
   other-queens))
N-queens as a graph problem

• Nodes (first approximation): a configuration of queens that can't attack each other.
• Since no two queens can occupy the same row, we'll search row-wise. So we'll say:
• Nodes: a configuration of queens of the form
  \{(1,c_1), \ldots, (k, c_k)\}
  for some k.
N-queens as a graph problem (2)

• Edges will look like this:

\[
((1, c_1), \ldots, (k, c_k)) \rightarrow ((1, c_1), \ldots, (k, c_k), (k+1, c_{(k+1)})
\]

• We have an edge from a configuration \(c\) to each legal configuration that extends it by placing a queen in row \((k+1)\).
N-queens as a graph problem (3)

• Question: starting from the empty configuration, find a reachable configuration of size N.
Manipulating Configurations

;; Configurations:
;; A Config is a ListOf<Queen>
;; WHERE: the queens are listed in decreasing row order, eg
;; ((k, c_k), (k-1, c_k-1), ... (1, c1))
;; AND WHERE: no two of the queens threaten each other.

;; : -> Config
(define empty-config empty)
Growing a configuration

;; cons-config : PosInt Config -> Maybe<Config>
;; GIVEN: a column col and a config ((k, c_k), (k-1, c_k-1), ... (1, c1))
;; RETURNS: the configuration ((k+1, col) (k, c_k), (k-1, c_k-1), ... (1, c1)) if that is a legal configuration, otherwise false.
;; Strategy: function composition
;; Algorithm: Check to see if the new queen threatens any of the existing queens.

(define (cons-config col config)
  (if (empty? config)
      ;; if the configuration is empty, then the first move is in row 1
      ;; and is always legal
      (list (make-queen 1 col))
      (local
        ((define next-row (+ 1 (queen-row (first config))))
          (define new-queen (make-queen next-row col)))
        (if
          (threatens-any? new-queen config)
          false
          (cons new-queen config))))

None of the old queens threaten each other, so we only have to see if the new queen threatens any of the old queens.
Now we can write successors.

;;; Config PosInt -> ListOf<Config>
;;; GIVEN: a configuration config and a board size
;;; RETURNS: the set of all the successors of config with columns in
;;; [1,size]
;;; STRATEGY: function composition
(define (successors config size)
  (filter
   (lambda (v) (not (false? v)))
   (map
    (lambda (col) (cons-config col config))
    (build-columns size))))

;;; PosInt -> ListOf<PosInt>
;;; GIVEN: PosInt n
;;; RETURNS: (list 1 .. n)
(define (build-columns n)
  (build-list n (lambda (i) (+ i 1)))))
We're almost ready to write our program

- We'll start with the final path? program.
- In place of `graph` we'll use the size of the board.
- In place of `src`, we'll use the empty configuration.
- In place of `tgt`, we'll just test to see if we ever get a configuration whose length is size.
- We'll never get a duplication of configurations, so we can eliminate the `set-diff`, and just keep track of `newest`. 
queen-dfs

;; placement-reachable-from?:
;;   ListOfNodes ListOfNodes PosInt -> Maybe<Config>
;; GIVEN:
;; 1. The list of nodes 'newest' whose successors we haven't taken
;; 2. The list 'nodes' of all the nodes we've seen
;; 3. The size of the chessboard
;; RETURNS: a complete configuration of the queens, if one exists,
;; otherwise false.
;; INVARIANT: newest is a subset of nodes
;; AND:
;;   (there is a some complete placement of queens)
;; iff (there is some complete placement that extends one of the
;; nodes in 'newest')
;; HALTING MEASURE: the number of graph nodes _not_ in 'nodes'
(define (placement-reachable-from? newest nodes size)
  (cond
    [(empty? newest) false]
    [(any-complete? size newest)
      (first-complete size newest)]
    [else (local
      ((define candidates (successors (first newest) size)))
      (cond
        [(empty? candidates)
          (placement-reachable-from? (rest newest) nodes size)]
        [else (placement-reachable-from?
          [else (append candidates (rest newest))
            (append candidates nodes)
            size))])])))

(define (queens-dfs size)
  (placement-reachable-from?
    (list empty-config) (list empty-config) size))
Help functions

(define (complete? size config)
  (= size (length config)))

(define (any-complete? size configs)
  (ormap
   (lambda (c) (complete? size c))
   configs))

;; PosInt ListOf<Config> -> Config
;; WHERE: one of the configs is guaranteed to be complete.
(define (first-complete size configs)
  (first
   (filter
    (lambda (c) (complete? size c))
    configs)))
Try this

• Try this with arguments ranging from 1 to 12 or so.
A Simpler Approach

• We've presented this problem as an application of graph search.
• But the graph is acyclic, so we can fall back on plain old general recursion.
• The problem is to find some completion of the empty configuration.
• So the obvious thing is to generalize away from the empty configuration:
Generalized Problem

;; generalized-queens-dfs
;;   : Config PosInt-> Maybe<Config>
;; GIVEN: a configuration
;; RETURNS: a completion of the configuration, 
;; if there is one, otherwise false.
Algorithm

• If \( c \) is already complete, it is its own completion.

• Otherwise, look at each of the successors of \( c \) in turn, and choose the first completion.
Function Definition

;; STRATEGY: generative recursion
;; HALTING MEASURE: (- size (length c))
(define (generalized-queens-dfs c size)
  (if (complete? size c)
      c
      (first-success
       (lambda (new-c)
         (generalized-queens-dfs new-c size))
       (successors c size))))

The successors of c are all of length (+ 1 (length c)), so the halting measure will decrease at each recursive call.
Help Function

;;; first-success is similar to ormap, but it returns the first
;;; non-false value of (f x_i)

;;; (X -> Maybe<Y>) ListOf<X> -> Maybe<Y>
;;; GIVEN: lst = (list x_1 x_2 ...) 
;;; RETURNS: The first non-false value of (f x_1), (f x_2), etc.
;;; Algorithm: compute (f x_1), etc. from left to right. Stop
;;; and return the first non-false value– don't continue computing.
(define (first-success fn lst)
  (cond
    [(empty? lst) false]
    [else (local
      ((define firstval (fn (first lst))))
      (if (false? firstval)
        (first-success fn (rest lst))
        firstval))])))
Top-level function

(define (queens-dfs.v3 size)
  (generalized-queens-dfs empty-config size))
Layered Design

• We designed our system in 4 layers:
  1. Queens. The operations were make-queen, queen-row, and threatens?
  2. Configurations. The operations were empty-config, cons-config, and complete?
  3. Graphs. The only operation was successors.
  4. Search. This was the top-level function queens-dfs.
Information-Hiding

• At each level, we could have referred to the implementation details of the lower layers, but we didn't need to.
• We only needed to refer to the procedures that manipulated the values in the lower layers.
• So when we code the higher layers, we don't need to worry about the details of the lower layers.
Information-Hiding (2)

• We could have written 4 files: queens.rkt, configs.rkt, graphs.rkt, and search.rkt, with each file **provide**-ing just those few procedures.

• In larger systems this is a must. It is the major topic of Managing System Design (aka Bootcamp 2)
Information-Hiding (3)

• These procedures form an *interface* to the values in question.

• If you continue along this line of analysis, you will be led to objects and classes (next week's topic!).
Information-Hiding (4)

- You use information-hiding every day.
- Example: do you know how Racket *really* represents numbers? Do you care? Ans: No, so long as the arithmetic functions give the right answer.
- Similarly for file system, etc: so long as fopen, fclose, etc. do the right thing, you don't care how files are actually implemented. 

Except for performance, of course.
Summary

• In this lesson, we wrote two solutions to the N-queens problem.
  – one solution turned it into a graph problem
  – the other did it by pure generative recursion.

• We constructed our solution in layers
  – At each layer, we got to forget about the details of the layers below
  – This enables us to control complexity: to solve our problem while juggling less stuff in our brains.