Programming Abstractions

Lecture 16: Backtracking continued

Stephen Checkoway

Announcements

Due dates for homework 5, 6, 7, and 8 changed!

New dates:
- Homework 5: Friday, November 19
- Homework 6: Friday, December 03
- Homework 7: Friday, December 17
- Homework 8: Friday, January 07

Backtracking in Racket

```
; sofar is the list of steps so far in reverse order
; curr is the current value to try
(define (backtrack params sofar curr)
  (cond [⟨sofar is a complete solution⟩ (reverse sofar)]
        [⟨curr is out of the range of possible values⟩ #f]
        [(feasible sofar curr)
         (let ([res (backtrack params
                                (cons curr sofar)
                                ⟨first value for next step⟩))])
           (if res
               res
               (backtrack params sofar ⟨value after curr⟩)))]
        [else (backtrack params sofar ⟨value after curr⟩)]))
```

Using backtrack

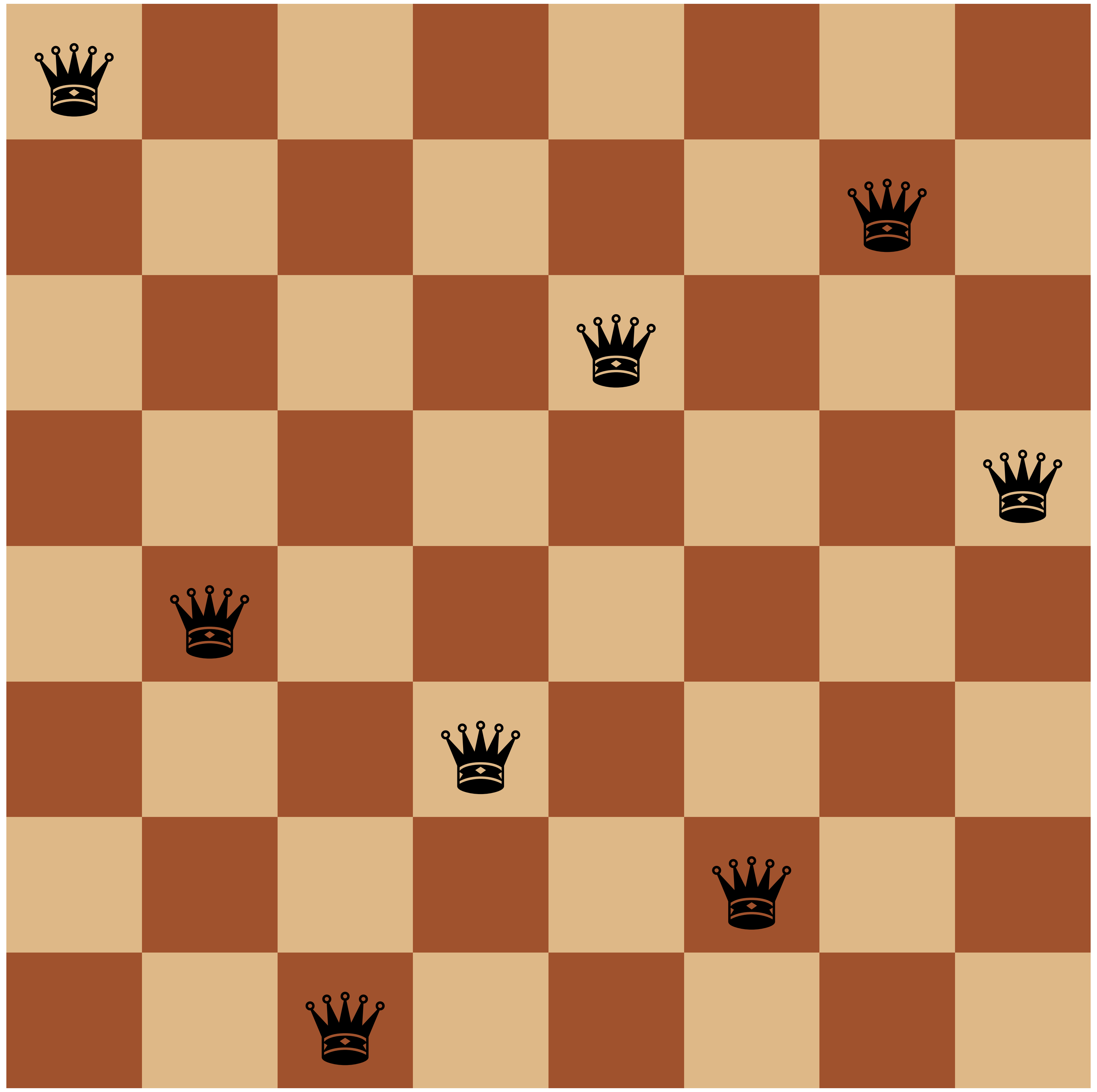
(Of course, you'll write specific backtrack and feasible functions for each problem)

```
(backtrack params empty ⟨first value for first step⟩)
```

n-queens
(single solution)

First, how should we represent a solution?

- A list of row–column pairs like
  `'((0 0) (4 1) (7 2) (5 3)`
  `(2 4) (6 5) (1 6) (3 7))`
- A list of rows like `'(0 4 7 5 2 6 1 3)`



Either works and we can easily convert from one to the other

- `(map list list-of-rows (range n))`
- `(map first list-of-pairs)`
  The list must be sorted by column first

Let's use a list of rows

Careful!

Our normal procedure for constructing the list of steps prepends the current step to our partial solution

- `(bt (cons curr sofar) initial)`

This means our partial solution will be in reverse order which means we need to

- reverse our final result so it's in the correct order; and
- write our `(feasible? sofar curr)` procedure keeping this in mind

n-queens

```
(define (bt n sofar curr)
  (cond [(is-complete? sofar) (reverse sofar)]
        [(out-of-range? curr) #f]
        [(feasible? sofar curr)
         (let ([res (bt n (cons curr sofar) initial)])
           (if res
               res
               (bt n sofar (next curr))))]
        [else (bt n sofar (next curr))]))

(define (n-queens n)
  (bt n empty initial))
```

What's our initial value?

```
(define (bt n sofar curr)
  (cond [(is-complete? sofar) (reverse sofar)]
        [(out-of-range? curr) #f]
        [(feasible? sofar curr)
         (let ([res (bt n (cons curr sofar) initial)])
           (if res
               res
               (bt n sofar (next curr))))]
        [else (bt n sofar (next curr))]))

(define (n-queens n)
  (bt n empty initial))
```

A. 0

B. 1

C. n

D. n-1

E. n+1

What's our `(next curr)` procedure?

```
(define (bt n sofar curr)
  (cond [(is-complete? sofar) (reverse sofar)]
        [(out-of-range? curr) #f]
        [(feasible? sofar curr)
         (let ([res (bt n (cons curr sofar) initial)])
           (if res
               res
               (bt n sofar (next curr))))]
        [else (bt n sofar (next curr))]))

(define (n-queens n)
  (bt n empty initial))
```

A. `(add1 curr)`

B. `(add1 (modulo curr n))`

C. `(modulo (add1 curr) n)`

D. `(modulo (add1 curr) (add1 n))`

E. More than one of the above

What's our `(is-complete? sofar)` procedure?

```
(define (bt n sofar curr)
  (cond [(is-complete? sofar) (reverse sofar)]
        [(out-of-range? curr) #f]
        [(feasible? sofar curr)
         (let ([res (bt n (cons curr sofar) initial)])
           (if res
               res
               (bt n sofar (next curr))))]
        [else (bt n sofar (next curr))]))

(define (n-queens n)
  (bt n empty initial))
```

A. `(feasible? sofar null)`

B. `(= (length sofar) n)`

C. `(= (length sofar) (add1 n))`

D. `(= (length sofar) (sub1 n))`

E. More than one of the above

What's our `(out-of-range? curr)` procedure?

```
(define (bt n sofar curr)
  (cond [(is-complete? sofar) (reverse sofar)]
        [(out-of-range? curr) #f]
        [(feasible? sofar curr)
         (let ([res (bt n (cons curr sofar) initial)])
           (if res
               res
               (bt n sofar (next curr))))]
        [else (bt n sofar (next curr))]))

(define (n-queens n)
  (bt n empty initial))
```

A. `(< curr n)`

B. `(= curr n)`

C. `(> curr n)`

D. `(< n 0)`

E. `(not (integer? curr))`

feasible?

There are three conditions

- No two queens share the same column
  - Easy, we're picking one queen per column so this is always satisfied
- No two queens share the same row
  - We'll need to check that `sofar` doesn't already contain `curr`
- No two queens share the same diagonal
  - Two diagonals to check: up-left from `curr` and down-left from `curr`
  - Lots of ways to do this, here's one: move left through columns; up through rows

```
(define (up-left-ok? queen-rows row)
  (cond [(empty? queen-rows) #t]
        [(= (first queen-rows) row) #f]
        [else (up-left-ok? (rest queen-rows) (sub1 row))]))
(up-left-ok? sofar (sub1 curr))
```

feasible?

There are three conditions

- No two queens share the same column
  - Easy, we're picking one queen per column so this is always satisfied
- No two queens share the same row
  - We'll need to check that `sofar` doesn't already contain `curr`
- No two queens share the same diagonal
  - Two diagonals to check: up-left from `curr` and down-left from `curr`
  - Lots of ways to do this, here's one: move left through columns; up through rows

Move left through reversed columns

```
(define (up-left-ok? queen-rows row)
  (cond [(empty? queen-rows) #t]
        [(= (first queen-rows) row) #f]
        [else (up-left-ok? (rest queen-rows) (sub1 row))]))
(up-left-ok? sofar (sub1 curr))
```

feasible?

There are three conditions

- No two queens share the same column
  - Easy, we're picking one queen per column so this is always satisfied
- No two queens share the same row
  - We'll need to check that `sofar` doesn't already contain `curr`
- No two queens share the same diagonal
  - Two diagonals to check: up-left from `curr` and down-left from `curr`
  - Lots of ways to do this, here's one: move left through columns; up through rows

Move left through reversed columns

Move up through rows

```
(define (up-left-ok? queen-rows row)
  (cond [(empty? queen-rows) #t]
        [(= (first queen-rows) row) #f]
        [else (up-left-ok? (rest queen-rows) (sub1 row))]))
(up-left-ok? sofar (sub1 curr))
```

At various points, the backtracking algorithm needs to choose the next value to try for the current step or it needs to backtrack to a previous step.

When does it need to backtrack to a previous step?

A. It backtracks each time it encounters a partial solution that isn't feasible

B. It backtracks whenever there are no more choices for the current step

C. It backtracks when the choice it makes for the final step leads to an invalid solution

D. It backtracks after each invalid choice

E. All of the above

One common variant: all solutions

Rather than using `#f` to signal failure, we'll use `empty` to indicate the set of solutions is empty

Key differences

- Rather than stopping after a single solution is found, keep going
- Each call will return a list of solutions
- When we have a feasible solution, we need to get all the solutions both using the feasible one and not

All solutions in Racket

```
(define (all-sol params sofar curr)
  (cond [⟨sofar is a complete solution⟩ (list (reverse sofar))]
        [⟨curr is out of the range of possible values⟩ '()]
        [(feasible sofar curr)
         (let ([res1 (all-sol params
                              (cons curr sofar)
                              ⟨first value for next step⟩))]
               [res2 (all-sol params sofar ⟨value after curr⟩)])
           (append res1 res2))]
        [else (all-sol params sofar ⟨value after curr⟩)]))

(all-sol params empty ⟨first value for first step⟩)
```

Permutations of {0, 1, ..., n-1}

(Not the most efficient way)

Let's compute all permutations of {0, 1, ..., n-1} using backtracking

```
(define (bt n sofar curr)
  (cond [(is-complete? sofar) (list sofar)]
        [(out-of-range? curr) empty]
        [(feasible? sofar curr)
         (let ([with-curr (bt n (cons curr sofar) initial)]
               [without-curr (bt n sofar (next curr))])
           (append with-curr without-curr))]
        [else (bt n sofar (next curr))]))

(define (all-perms n)
  (bt n empty initial))
```

We just need to deal with the problem-specific parts

n-queens all solutions

No harder than getting one solution, we just need to plug in the usual parts

```
(define (bt n sofar curr)
  (cond [(is-complete? sofar) (list (reverse sofar))]
        [(out-of-range? curr) empty]
        [(feasible? sofar curr)
         (let ([with-curr (bt n (cons curr sofar) initial)]
               [without-curr (bt n sofar (next curr))])
           (append with-curr without-curr))]
        [else (bt n sofar (next curr))]))

(define (all-queens n)
  (bt n empty initial))
```