Introduction to AI

Thursday September 13, 2018

Class Outline

- Programming Assignment 0 Questions
- Programming Assignment 1: Search
- Discussion on Constraints and Optimization

Programming Assignment 0

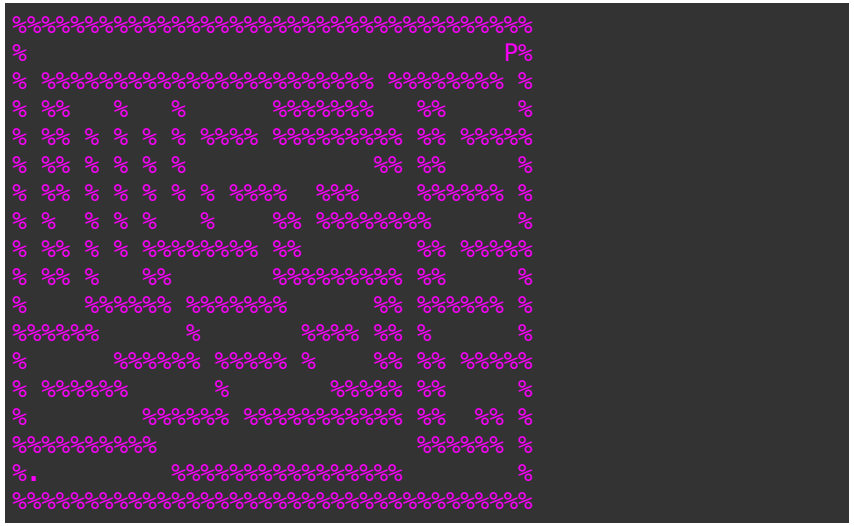
Questions?

Programming Assignment 1: Search

For this assignment you will be writing search algorithms to help Pacman eat food in a maze.

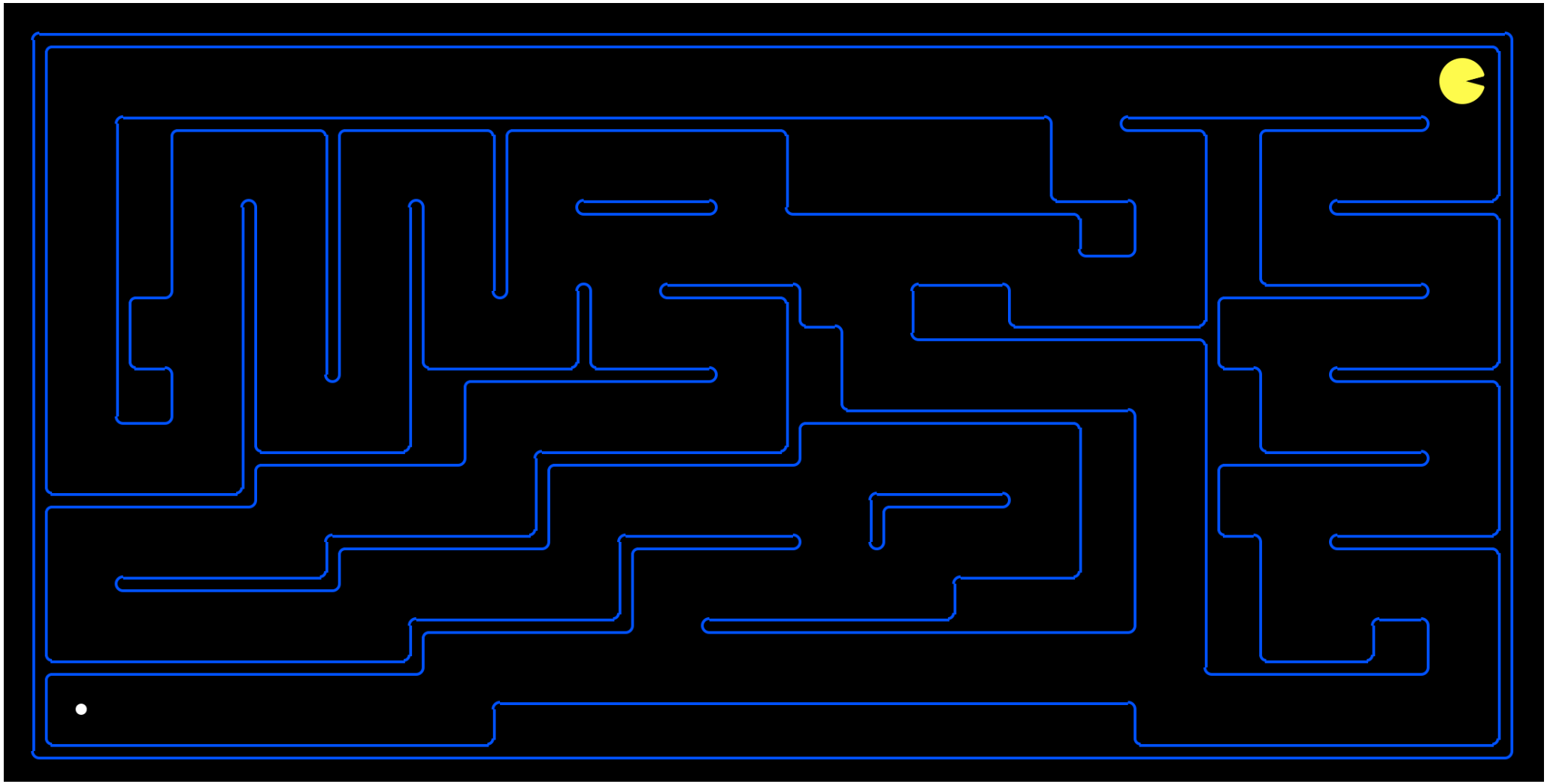
Layout Files

This is the text of mediumMaze.lay in the layouts folder.



Visualized

Here is what it looks like when visualized using the provided python visualization libraries.



Search Review

- DFS
- BFS
- A*

Utils

Functions for

- data types
- finding successor states
- testing for goal states
- heuristics for A*

Stacks

The code in utils for stacks

```
(define (make-stack)
  (list))

(define (push-stack x st)
      (cons x st))

(define (pop-stack st)
  (values (car st) (cdr st)))

(define (stack-empty? st)
  (empty? st))
```

Example Stack Usage

What are the values of `elem`, `st` and `old-st` after running this code?

```
(define st (make-stack))
(define elem null)
(set! st (push-stack 1 st))
(set! st (push-stack 2 st))
(set! st (push-stack 3 st))

(define old-st st)

(set!-values (elem st) (pop-stack st))
```

Example Stack with While Loop

What will this print?

```
(let ([st (make-stack)]
      [elem null])
  ;; put some stuff on the stack
  (set! st (push-stack 1 st))
  (set! st (push-stack 2 st))
  (set! st (push-stack 3 st))

  ;; pop some stuff off using a while loop
  (for ([i (in-naturals 0)]
    #:break (stack-empty? st))
    (set!-values (elem st) (pop-stack st))
    (print elem)
    ))
```

Queues

```
(provide make-queue)

(define (push-queue x queue)
      (enqueue! queue x)
  queue)

(define (pop-queue queue)
  (let ([output (dequeue! queue)])
    (values output queue)))

(provide queue-empty?)

;; some more functions for viewing queues
(define (length-queue queue)
  (length (queue->list queue)))

(provide queue->list)
```

Example Queue with While Loop

What will this print?

```
(let ([q (make-queue)]
      [elem null])
  ;; put some stuff in the queue
  (set! q (push-queue 1 q))
  (set! q (push-queue 2 q))
  (set! q (push-queue 3 q))

  ;; pop some stuff off using a while loop
  (for ([i (in-naturals 0)]
    #:break (queue-empty? q))
    (set!-values (elem q) (pop-queue q))
    (print elem)
    ))
```

Priority Queues

A priority queue must come accompanied by a priority

```
(define (make-priority-queue priority-func)
  (make-heap (lambda (x y)
               (<= (priority-func x) (priority-func y)))))

(define (push-priority-queue x queue)
      (heap-add! queue x)
  queue)

(define (pop-priority-queue queue)
  (let ([output (heap-min queue)])
    (heap-remove-min! queue)
    (values output queue)))

(define (priority-queue-empty? queue)
  (equal? (heap-count queue) 0))
```

Sets

Sets are useful for keeping track of previously visited states. If you don't prune out these states, your frontier will grow too large, and racket will throw a stack overflow exception.

Here are some useful functions for sets provided in `utils.rkt`.

```
(define (make-set)
        (list))

(define (ismember? elm lst)
  (ormap [lambda (val) (equal? val elm)] lst))

(def (push-set elm lst)
  (cons elm lst))
```

Reading and Interpreting Layouts

```
;; to read in the maze
(define (layout-path-parser path)
  (file->lines path #:mode 'text #:line-mode 'linefeed))

;; some functions to find things in the maze
(define (get-pos maze row col)
  (string-ref (list-ref maze row) col))

(define (find-pacman maze)
  (let ([output (index-of-not-false (map-maze maze (string PACMAN)))])
    (if (equal? output #f)
        (let ([output-alt (index-of-not-false (map-maze maze (string PACMAN-ALT)))])
          (if (equal? output-alt #f)
              (raise "Couldn't find Pacman")
              output-alt))
        output)))

(define (find-first-food maze)
  (index-of-not-false (map-maze maze (string FOOD))))

(define (map-maze maze elm)
  (map (lambda (row)
         (string-contains row elm)) maze))

(define (index-of-not-false lst)
  (let loop ((lst lst)
             (idx 0))
    (cond ((empty? lst) #f)
          ((not (equal? (first lst) #f)) (list idx (first lst)))
          (else (loop (rest lst) (add1 idx))))))
```

Get Successors

The `get-succ` function gets successor states to a given state and returns a list of valid successor states with the action character needed to get there.

```
;; get successor states
(define (get-succ maze)
  (let ([pac-pos (find-pacman maze)]
        [no-pacman-maze (delete-pacman maze)])
    (map (lambda (move)
           (let ([move-name (first move)]
                 [pac-row (+ (first pac-pos) (first (second move)))]
                 [pac-col (+ (second pac-pos) (second (second move)))]
                 )
             (list move-name (add-pacman no-pacman-maze pac-row pac-col))
             ))
         (filter (lambda (move)
                   (let ([move-name (first move)]
                         [pac-row (+ (first pac-pos) (first (second move)))]
                         [pac-col (+ (second pac-pos) (second (second move)))]
                         )
                     (not (equal? (get-pos maze pac-row pac-col) WALL))))
                 '(
                   (#\W (0 -1))
                   (#\E (0 1))
                   (#\N (-1 0))
                   (#\S (1 0))
                   ))
         )))
```

Some helper functions for get successors...

```
(define (delete-pacman maze)
  (map (lambda (row)
         (if (string-contains? row (string PACMAN))
             (list->string (map (lambda (char) (if (equal? char PACMAN) SPACE char)) (string->list row)))
             (if (string-contains? row (string PACMAN-ALT))
                 (list->string (map (lambda (char) (if (equal? char PACMAN-ALT) SPACE char)) (string->list row)))
                 row)))
       maze))

(define (add-pacman maze  pac-row pac-col)
  (map (lambda (row-index)
         (if (equal? row-index pac-row)
             (list->string
              (map (lambda (col-index)
                     (if (equal? pac-col col-index)
                         PACMAN
                         (get-pos maze row-index col-index)))
                   (stream->list (in-range (string-length (list-ref maze row-index))))))
             (list-ref maze row-index)))
       (stream->list (in-range (length maze)))))
```

Get Successors in Action

```
(define maze (layout-path-parser "../layouts/tinyMaze.lay"))

(print (get-succ maze))
```

Prints this output:

```
'((#W ("%%%%%%%%" "%   P %" "% %%% %" "%  %  %" "%%   %%" "%. %%%%" "%%%%%%%%")) (#S ("%%%%%%%%" "%     %" "% %%%P%" "%  %  %" "%%   %%" "%. %%%%" "%%%%%%%%")))
```

Get Successors with prettier printing

```
(for ([succ (get-succ maze)])
  (display-succ succ))
```

Prints this:

Check for Goal State

This function checks if a state is a goal state for Pacman (Pacman's goal is to eat all the food).

```
;; check if this is a goal state by checking if the amount of food is equal to zero
(define (is-goal maze)
  (equal?
   (length
    (filter
     (lambda (row)
       (string-contains row (string FOOD)))
     maze))
   0)
  )
```

Note: Make sure to use this function to test if your code is complete, since we will be using other goals during grading.

H and G Functions

We give you a g function to use with the priority queue and some heuristic functions.

Note that the g function takes a function as an arguement and returns a function as the output

```
;; a g-func to use with a priority queue for A*
;; this assumes each node is stored on the frontier as tuple of (<path to state>, state)
;; thus the g funciton for a node is (+ (length path) (heuristic state))
(define (g-func heurisitc-fun)
  (lambda (node)
    (+ (length (first node)) (heurisitc-fun (second node)))))

;; some heuristics for A-star to use
(define (count-food maze)
  (apply +
         (map (lambda (row)
                (length
                 (filter
                  (lambda (char) (equal? char FOOD))
                  (string->list row))))
              maze)))

(define (distance-to-food maze)
  (let ([pac-pos (find-pacman maze)]
        [food-pos (find-first-food maze)])
    (if (equal? food-pos #f)
        0
        (+ (abs (- (first pac-pos) (first food-pos))) (abs (- (second pac-pos) (second food-pos)))))))

(define (null-heuristic maze) 0)
```

Example of How the G Function Works with a Priority Queue

What will this print?

```
(let ([q (make-priority-queue (g-func null-heuristic))]
      [elem null])
  (set! q (push-priority-queue (list '(#\W #\E #\S #\N) "bad maze") q))
  (set! q (push-priority-queue (list '(#\S #\N) "worse maze") q))
  (set! q (push-priority-queue (list '(#\E) "worst maze") q))
  (for ([i (in-naturals 0)]
        #:break (priority-queue-empty? q))
    (set!-values (elem q) (pop-priority-queue q))
    (print elem)
    ))
```