Notes on Uninformed Search in Scheme for 16.410 and 16.413

1 Trees

The following code is how you might set up a binary tree on real numbers.

```
(define (init_tree) '())

(define (root tree) (car tree))

(define (left_child tree)
  (cond ((null? tree) '())
        ((null? (cdr tree)) '())
        (else (cadr tree))))

(define (right_child tree)
  (cond ((null? tree) '())
        ((null? (cdr tree)) '())
        ((null? (cddr tree)) '())
        (else (caddr tree))))

(define (insert_elem x tree)
  (cond ((null? tree) (list x))
        ((< x (root tree)) (list (root tree)
                                 (insert_elem x (left_child tree))
                                 (right_child tree)))
        (else (list (root tree) (left_child tree)
                    (insert_elem x (right_child tree))))))

(define (build_tree l tree)
  (if (null? l) tree
      (insert_elem (car l) (build_tree (cdr l) tree))))

(define (generate_list length)
  (if (> length 0) (cons (random 100) (generate_list (- length 1))) '()))

(define (sort tree)
  (cond ((null? tree) '())
        (else (append (sort (left_child tree)) (list (car tree))
                      (sort (right_child tree))))))
```

2 Building the Maze Data Structure

The following is how you might set up a data structure to describe a route map.

```
(define (search maze-entry k l)
  (cond ((null? maze-entry) '())
        ((and (= (caddr maze-entry) k)
              (= (cadddr maze-entry) l)) #t)
        (else (search (list-tail maze-entry 4) k l))))

(define (edge_exists i j k l x edge_list)
  (search (car (list-tail edge_list (+ (* i x) j))) k l))

(define (maze-neighbour i j k x y l)
  (cond ((and (= k 0) (= i (- x 1))) '())
        ((and (= k 1) (= j (- y 1))) '())
        ((and (= k 2) (= i 0)) '())
        ((and (= k 3) (= j 0)) '())
        ((and (= k 2) (edge_exists (- i 1) j i j x l))
         (list i j (- i 1) j))
        ((and (= k 3) (edge_exists i (- j 1) i j x l))
         (list i j i (- j 1)))
        ((> (random 3) 1) '())
        ((= k 0) (list i j (+ i 1) j))
        ((= k 1) (list i j i (+ 1 j)))
        (else '())))

(define (maze-entry i j k x y l)
  (cond ((= k 4) '())
        (else (append (maze-neighbour i j k x y l)
                      (maze-entry i j (+ k 1) x y l)))))

(define (maze-helper i j x y l)
  (cond ((and (= i (- x 1)) (= j y)) l)
        ((= j y) (maze-helper (+ i 1) 0 x y l))
        (else (maze-helper i (+ j 1) x y
                           (append l (list (maze-entry i j 0 x y l)))))))

(define (maze size) (maze-helper 0 0 size size '()))
```

3 Breadth-first Search

The following is pseudo-code for breadth-first search. `Q` is an abstact data type for a first-in, first-out queue.

```
push Q, start
while ! empty(Q)
  current <- pop Q
  if (current contains goal)
     return current
  push Q, children(current)
```

The following code is how you might carry out breadth-first search on a particular kind of data structure.

```
(define (children-helper l size)
  (cond ((null? l) '())
        (else (cons (+ (* (caddr l) size) (cadddr l))
                    (children-helper (list-tail l 4) size)))))

(define (children x maze)
  (children-helper (car (list-tail maze x)) (sqrt (length maze))))

(define (child-paths path maze)
  (map (lambda (x) (if (member x path) '() (cons x path)))
               (children (car path) maze)))

(define (bfs-helper goal queue maze)
  (cond ((null? queue) '())
        ((null? (car queue)) (bfs-helper goal (cdr queue) maze))
        ((equal? (caar queue) goal) (car queue))
        (else (bfs-helper goal (append (cdr queue)
                                (child-paths (car queue) maze)) maze))))

(define (bfs x y maze)
  (reverse (bfs-helper y (list (list x)) maze)))

(define m (maze 10))
```

4 Mutable Data Structures

The following is how you might set up a persistent queue.

```
(define (init_queue) (list '(0)))

(define (enqueue item q)
  (set-car! q (list (+ (caar q) 1)))
  (set-cdr! q (append (cdr q) (list item)))
  (caar q)
)

(define (dequeue q)
  (let ((value (cadr q)))
    (set-car! q (list (- (caar q) 1)))
    (set-cdr! q (cddr q))
    value))
```

YOU DO NOT NEED TO UNDERSTAND THE FOLLOWING CODE TO DO THE ASSIGNMENTS.

But, it is included because I used it in class, and you might be curious.

```
(define xwin (make-graphics-device 'x))

(define (scale x size) (/ (- x (/ size 2)) (/ size 2)))

(define (draw-entry entry size)
  (cond ((null? entry) '())
        (else
         (graphics-draw-line xwin (scale (car entry) size)
                             (scale (cadr entry) size)
                             (scale (caddr entry) size)
                             (scale (cadddr entry) size))
         (draw-entry (cddddr entry) size))))

(define (draw-maze-helper maze size)
  (cond ((null? maze) '())
        (else (draw-entry (car maze) size) (draw-maze-helper
                                            (cdr maze) size))))

(define (draw-maze maze)
  (graphics-clear xwin)
  (draw-maze-helper maze (sqrt (length maze))))

(draw-maze m)
```