

Remember Java class definitions?

```
public class City
  private LinkedList neighbors;
  private LinkedList coords;
  public City (LinkedList neighbors, LinkedList coords) {
    this.neighbors = neighbors;
    this.coords = coords;
  }
  public LinkedList getNeighbors () {
    return neighbors;
  }
  public LinkedList getCoords () {
    return coords;
  }
  public void setNeighbors (LinkedList newNeighbors) {
    neighbors = newNeighbors;
  }
  public void setCoords (LinkedList newCoords) {
    coords = newCoords;
  }
}
```

Creating object classes with defstruct

defstruct defines a new *structure*:

```
(defstruct city (neighbors nil) (coords nil))
```

Constructor, reader and setter functions are defined automatically:

```
> (setf s (make-city :neighbors `(a d)
                  :coords `(0.5 2.75)))
#S(CITY :NEIGHBORS (A D) :COORDS (0.5 2.75))
> (city-neighbors s)
(A D)
> (city-coords s)
(-0.5 2.75)
> (setf (city-neighbors s) `(a b d))
(A B D)
> (setf (city-coords s) `(0.0 0.0))
(0.0 0.0)
```

Formatted printing

format allows formatted printing of *strings*

```
> (format t "Hello!")
Hello!
NIL

> (defun print-eec ()
  (format t "hist whist ~%little ghostthings")
  (format t "~%tip-toe ~%twinkle-toe"))

> (print-eec)
hist whist
little ghostthings
tip-toe
twinkle-toe
> (format t "The winner is ~a! Congrats!" 'Susie)
The winner is SUSIE! Congrats!
NIL
> (format t "Course: ~10a ~4a ~a" 'CS 232 'AI)
Course: CS          232 AI
NIL
```

search.lisp

```
(defun general-search (start goal search-method
  &optional (queue (list (list start)))
  (paths-extended 0))
  (cond ((endp queue) nil)
        ((goalp (first (first queue)) goal)
         (format t "~%A successful path was found after extending
                   ~a paths:~%~a"
                   paths-extended (reverse (first queue)))
         (reverse (first queue)))
        (t (general-search start goal search-method
          (funcall search-method (extend (first queue))
            (rest queue) goal)
          (1+ paths-extended))))))

(defun extend (path)
  (format t "~%Extending the path ~a." (reverse path))
  (mapcar #'(lambda (new-state) (cons new-state path))
    (remove-if #'(lambda (new-state)
      (member new-state path :test #'same-state))
      (get-next-states (first path)))))
```

Specific search methods:

```
(defun depth-first (start goal)
  (general-search start goal #'queue-depth-first))

(defun queue-depth-first (new old goal)
  (declare (ignore goal))
  (append new old))

(defun breadth-first (start goal)
  (general-search start goal #'queue-breadth-first))

(defun queue-breadth-first (new old goal)
  (declare (ignore goal))
  (append old new))
```

Hill-climbing:

```
(defun hill-climb (start goal)
  (general-search start goal #'queue-hill-climb))

(defun queue-hill-climb (new old goal)
  (append (sort new
                #'(lambda (p1 p2)
                    (closerp p1 p2 goal)))
          old))

(defun closerp (path1 path2 goal)
  (< (distance (first path1) goal)
     (distance (first path2) goal)))
```

distance is defined in a separate application-specific code file

Branch-and-Bound:

```
(defun branch-and-bound (start goal)
  (general-search start goal #'queue-branch-and-bound))

(defun queue-branch-bound (new old goal)
  (declare (ignore goal))
  (mapcar #'second
    (sort (mapcar #'(lambda (path)
                    (list (path-length path) path))
                  (append new old))
          #'(lambda (x y) (< (first x) (first y))))))
```

path-length is defined in a separate application-specific code file

Searching a network of cities in `cities.lisp`

```
(defstruct city (neighbors nil) (coords nil))

(defvar S (make-city :neighbors '(A D) :coords '(-0.5 2.75)))
(defvar A (make-city :neighbors '(S B D) :coords '(1.0 5.0)))
(defvar B (make-city :neighbors '(A C E) :coords '(5.0 5.0)))
(defvar C (make-city :neighbors '(B) :coords '(9.0 5.0)))
...

(defun search-cities (search-method)
  (funcall search-method 'S 'G))

(defun goalp (state goal) (equal state goal))

(defun same-state (state1 state2) (equal state1 state2))

(defun get-next-states (state)
  (city-neighbors (eval state)))

(defun distance (state1 state2)
  (let ((coords1 (city-coords (eval state1)))
        (coords2 (city-coords (eval state2))))
    (sqrt (+ (expt (- (first coords1) (first coords2)) 2)
             (expt (- (second coords1) (second coords2)) 2)))))
```

Search results...

(depth-first `S` `G)

Extending the path (S).

Extending the path (S A).

Extending the path (S A B).

Extending the path (S A B C).

Extending the path (S A B E).

Extending the path (S A B E D).

Extending the path (S A B E F).

A successful path was found after extending 7 paths:

(S A B E F G)

(S A B E F G)