## A* for graphs

- Re-expands a node if it find a better path to the node
- Finds optimal solutions even if the heuristic is inconsistent

```
function A*( problem) returns a solution, or failure
    closed ← an empty set
    fringe ← a list containing MAKE-NODE(INITIAL-STATE[problem])
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST[problem] applied to STATE(node) succeeds return node
        insert node into closed
        for each node $n \in$ EXPAND(node, problem) do
            if there is a node $m \in$ closed $\cup$ fringe such that
                STATE(m) = STATE(n) and $f(m) \leq f(n)$
            then do nothing
            else
                insert n into fringe after the last node m such that $f(m) \leq f(n)$
    end
```

```lisp
;;; Each node is a structure whose fields include the current state, a pointer
;;; to the parent node, and the heuristic-function values. The DEFSTRUCT
;;; command automatically creates the access operators for these fields.
(defstruct (node) state parent g f)

;;; ----------
;;; FIND-PATH and its helper functions.
;;; ----------

;;; This is a direct translation of the A* pseudocode from my lecture
(defun find-path (start boundary finish heuristic)
  (let (closed fringe current-node new-g new-h new-f)
    ;; compute START's heuristic info, and put START into fringe
    (setq new-h (funcall heuristic start boundary finish))
    (push (make-node
           :state start
           :parent nil
           :g 0
           :f new-h)
          fringe)
    (loop
(loop
 (if (not fringe) (return 'failure))
 (setq current-node (pop fringe))   ; node with lowest f-value

 ;; If edge intersects finish line, return the path (sequence of states)
 ;; There's a subtle error here; see if you can figure out what it is!
 (if (and (node-parent current-node) ; start node => no parent => no edge
          (intersectp (get-edge current-node) finish))
     (return (reverse (path-to-root current-node))))

 ;; Otherwise generate and test the current node's successors
 (push current-node closed)
 (dolist (new-state (successors (node-state current-node) boundary))
   (setq new-g (+ 1 (node-g current-node))) ; new state's g-value
   (setq new-h (funcall heuristic new-state boundary finish))
   (setq new-f (+ new-g new-h))             ; new state's f-value
   ;; If same state already exists same or better f-value, do nothing.
   ;; Else add new state to fringe.
   (or (already-in new-state new-f closed)
       (already-in new-state new-f fringe)
       (setq fringe
             (insert-node
              (make-node :state new-state
                         :parent current-node
                         :g new-g
                         :f new-f)
              fringe))))))
```

```
(defstruct (node) state parent g f)
```

```
;;; the edge to NODE from its parent
(defun get-edge (node)
  (let (loc1 loc2)
    (setq loc2 (first (node-state node)))
    (setq loc1 (first (node-state (node-parent node))))
    (list loc1 loc2)))

;;; find the path from a node back to the root
(defun path-to-root (node)
  (cond ((null (node-parent node))       ; at the root node
         (list (node-state node)))
        (t (cons (node-state node) (path-to-root (node-parent node))))))

;;; test whether STATE already exists in NODELIST with same or better F vaue
(defun already-in (state f nodelist)
  (let ((found-node (find state nodelist :key #'node-state :test #'equal)))
    (and found-node (<= (node-f found-node) f))))

;;; easiest way to insert NODE into FRINGE in order of F values
(defun insert-node (node fringe)
  (sort (cons node fringe) #'< :key #'node-f))
```

```lisp
;;;----------
;;; successors and crashp
;;;----------

(defun successors (state boundary)
  (let ((answers nil)
        (location (first state))
        (velocity (second state))
        v w x y edge)
    ;; check all 9 possibilities for new velocity
    (dolist (v-change '(-1 0 1))
      (dolist (w-change '(-1 0 1))
        (setq v (+ v-change (first velocity)))
        (setq w (+ w-change (second velocity)))
        (setq x (+ v (first location)))
        (setq y (+ w (second location)))
        ;; see for crash on edge from current state to new location
        (setq edge (list (first state) (list x y)))
        (if (not (crashp edge boundary))
            (push (list (list x y) (list v w)) answers))))
    answers))

(defun crashp (edge boundary)
  ;; crash iff EDGE intersects an edge in BOUNDARY
  (dolist (b boundary nil)
    (if (intersectp edge b) (return t))))
```

**HINT:** Suppose $e = (\,(x_1 \;\; y_1) \quad (x_2 \;\; y_2)\,)$. Then for every point $(x \;\; y)$ in $e$, there is a number $0 \le t \le 1$ such that

$$x = x_1 + t(x_2 - x_1);$$
$$y = y_1 + t(y_2 - y_1).$$

Similarly, suppose $e' = (\,(x'_1 \;\; y'_1) \quad (x'_2 \;\; y'_2)\,)$. Then for every point $(x' \;\; y')$ in $e'$, there is a number $0 \le t' \le 1$ such that

$$x' = x'_1 + t'(x'_2 - x'_1);$$
$$y' = y'_1 + t'(y'_2 - y'_1).$$

Thus $e$ and $e'$ intersect iff there are numbers $0 \le t \le 1$ and $0 \le t' \le 1$ such that

$$x_1 + t(x_2 - x_1) = x'_1 + t'(x'_2 - x'_1); \tag{1}$$
$$y_1 + t(y_2 - y_1) = y'_1 + t'(y'_2 - y'_1). \tag{2}$$

From Equations 1 and 2, you should be able to derive formulas for $t$ and $t'$.

```
(defun intersectp (e eprime)
  ;; E goes from (x1,y1) to (x2,y2); EPRIME goes from (x1p,y1p) to (x2p,y2p)
  (let ((x1 (first (first e)))
        (y1 (second (first e)))
        (x2 (first (second e)))
        (y2 (second (second e)))
        (x1p (first (first eprime)))
        (y1p (second (first eprime)))
        (x2p (first (second eprime)))
        (y2p (second (second eprime)))
        bottom tp-top t-top)
    ;; denominator (x2p - x1p) (y2 - y1) - (y2p - y1p) (x2 - x1)
    (setq bottom (- (* (- x2p x1p) (- y2 y1)) (* (- y2p y1p) (- x2 x1))))
    ;; t' numerator (y1p - y1) (x2 - x1) - (x1p - x1) (y2 - y1)
    (setq tp-top (- (* (- y1p y1) (- x2 x1)) (* (- x1p x1) (- y2 y1))))
    ;; t numerator (y2p - y1p) (x1 - x1p) - (y1 - y1p) (x2p - x1p)
    (setq t-top (- (* (- y2p y1p) (- x1 x1p)) (* (- y1 y1p) (- x2p x1p))))
    (cond ((not (= bottom 0))
           ;; normal case; check whether t and tp are both in [0,1]
           (and (<= 0 (/ tp-top bottom) 1)
                (<= 0 (/ t-top bottom) 1)))
          ((and (= tp-top 0) (= t-top 0))
            ;; both numerators are 0, so edges are collinear
            ;; check whether either edge has an endpoint inside the other edge
            (or (collinear-point-in-edge-p x1 y1 x1p y1p x2p y2p)
                (collinear-point-in-edge-p x2 y2 x1p y1p x2p y2p)
                (collinear-point-in-edge-p x1p y1p x1 y1 x2 y2)
                (collinear-point-in-edge-p x2p y2p x1 y1 x2 y2)))
          ;; otherwise edges are parallel but not collinear, so don't intersect
          ;; this clause is unnecessary but I'm including it for clarity
          (t nil))))
```

```lisp
    (cond ((not (= bottom 0))
           ;; normal case; check whether t and tp are both in [0,1]
           (and (<= 0 (/ tp-top bottom) 1)
                (<= 0 (/ t-top bottom) 1)))
          ((and (= tp-top 0) (= t-top 0))
            ;; both numerators are 0, so edges are collinear
            ;; check whether either edge has an endpoint inside the other edge
            (or (collinear-point-in-edge-p x1 y1 x1p y1p x2p y2p)
                (collinear-point-in-edge-p x2 y2 x1p y1p x2p y2p)
                (collinear-point-in-edge-p x1p y1p x1 y1 x2 y2)
                (collinear-point-in-edge-p x2p y2p x1 y1 x2 y2)))
          ;; otherwise edges are parallel but not collinear, so don't intersect
          ;; this clause is unnecessary but I'm including it for clarity
          (t nil))))

;;; if (x,y) is collinear with the edge from (x1,y1) to (x2,y2),
;;; then (x,y) is in the edge if x is in [x1,x2] or [x2,x1]
;;; and y is in [y1,y2] or [y2,y1] (redundant except when line is vertical)
(defun collinear-point-in-edge-p (x y x1 y1 x2 y2)
  (and (<= (min x1 x2) x (max x1 x2))
       ;; Next line is unneeded except when the line is vertical
       (<= (min y1 y2) y (max y1 y2))))
```

Final homework assignment:
16.2
17.4
17.9
17.13
18.4
5 problems, 10 points each, 50 points total.

Remaining class sessions:
Thurs April 29
Tues May 6
Thurs May 6
Tues May 11

Current due date for project 3: May 2, late date May 4

Option 1
Project due May 4, late date May 6
Homework due May 6, late date May 11 (last class)

Option 2
Homework due May 4, late date May 6
Project due May 7, late date May 9