it quickly) the calculated distance between directly or indirectly connected nodes makes a single search through the network much faster. The hashing of precomputed distances greatly speeds up subsequent searches through the network. Our example makes use of both of these tricks.

11.2 A* Network Search Program

The function *A*search* is called with the names of the starting and goal nodes for the search. Function *A*search* uses the local list *possible-paths* to cache the Cartesian distances between nodes in the network as they are calculated. The helper function *a*-helper* performs a breadth-first search by starting with the first possible path (this list is kept sorted so that the possible paths with an ending node closest to the goal node are kept first on the list) and fanning the search out preferentially towards the goal node. Note that this search stops when the first path segment on the possible path list contains the goal node.

The function *init-network* precalculates the lengths of all path segments in the network. The function *init-path-list* puts calculated path lists as a property value on each node. By caching precalculated path costs for each node, the example program avoids recalculating any path sequence. For example, if after calculating the path between nodes *n1* and *n11* in Figure 11.1, function *A*search* is then asked to calculate the best path between nodes *n2* and *n11*, no real search is necessary since the path sequence from *n2* to *n11* will already be cached at node *n2*.

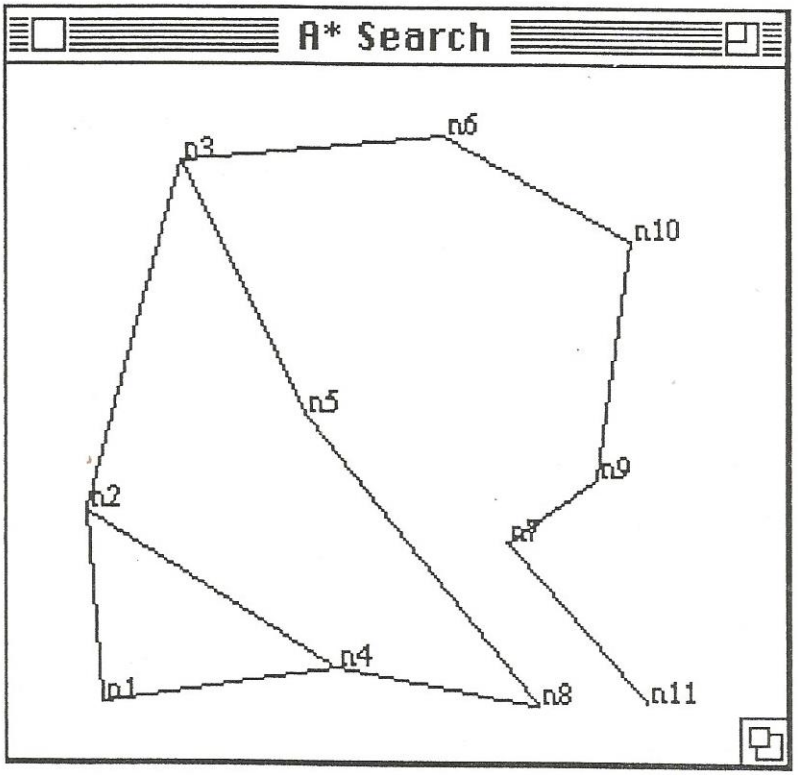

Figure 11.1. Sample Network with 11 Nodes and 12 Edges

The function *enumerate-children* collects all path segments from a specified starting node to the goal node. If a stored path is found cached on the starting node, function *consider-best-path* uses the stored information; if a stored path is not found cached in the starting node, function *consider-all-nodes* is called to search for a best path.

The function *Collect-ascending-search-list-order* places the possible paths fanning out from a node in a "best-first" order using previously calculated direct distances ("as the crow flies") from each node to the goal node.

The function *test1* in the sample program demonstrates the initialization of data for the A* search program. Two variables define the network used by the example program: *nodes* and *paths*. In *test1*, eleven test nodes are created as global data items with values of Cartesian coordinates. For example, node *n1* is defined to be at the location specified by the coordinates (30, 201) in some arbitrary coordinate system. The global variable *paths* is set to a list of node pairs defining the connections in the network.

Function *test1* calls function *init-network* to calculate and cache the path lengths of all path segments in the network. Function *test1* then plots the nodes in a graphics window (assuming the graphics library from chapter 2 has been loaded).

Function *test2* calls the function *A*search* which returns a path list for the best route from node *n1* to node *n11*. The best path is plotted in a graphics window.

```
;;
; File A*.lisp
;
; Perform a heuristic A* search between the start and goal nodes:
;
; Copyright 1990 by Mark Watson
; ;

(defun A*search (start goal &aux possible-paths best)
 (setq possible-paths
     (list
      (list
       (dist-between-points
         (eval start)
         (eval goal))
        0
        (list start))))
 (defun a*-helper ()  ;; Note: this function is nested inside of function A*
  (if possible-paths  ;;          so its lexical scoping makes it invisible outside A*
   (let ()
     (setq best (car possible-paths))
     (setq possible-paths (cdr possible-paths))
     (if (equal (first (caddr best)) goal)
       best
       (let ()
```

```
        (dolist (child (enumerate-children best goal))
          (setq possible-paths
              (Collect-ascending-search-list-order
                child possible-paths)))
        (a*-helper))))))
  (a*-helper))

;;
; Define memonics to fetch x,y coordinates from nodes:
;;

(defun Y-coord (x) (truncate (cadr x)))
(defun X-coord (x) (truncate (car x)))

;;
; Initialize the network by calculating path lengths
; of all path segments and caching this information
; at each node for future use:
;;

(defun init-network ()
  (setq paths (init-lengths paths))
  (init-path-list nodes paths))

;;
; Add the length of the path to each path in the path list:
;;

(defun init-lengths (pathlist)
  (let (new-path-list pathlength path-with-length)
    (dolist (path pathlist)
      (setq pathlength (slow-path-length path))
      (setq path-with-length (append path (list pathlength)))
      (setq new-path-list (cons path-with-length new-path-list)))
    new-path-list))


;;
; Place the path-list property on each node :
;;

(defun init-path-list (nodes paths)
  (dolist (node nodes)
        (put node 'path-list
            ;; let returns all paths connected to node:
```

```
            (let (path-list)
              (dolist (path paths)
                (if (equal node (start-node-name path))
                  (setq path-list
                      (cons (list (end-node-name path)
                                  (path-length path))
                            path-list))
                  (if (equal node (end-node-name path))
                    (setq path-list (cons (list (start-node-name path)
                                                (path-length path))
                                          path-list)))))
              path-list ))))

;;
; "As the crow flies" distance between
;  the starting and ending nodes on a path:
;;

(defun slow-path-length (path)
  (dist-between-points (start-node path) (end-node path)))


;;
; Define mnemonic for getting path-length from a node:
;;

(defun path-length (x) (caddr x))

;;
; Calculate the Cartesian distance between points:
;;

(defun dist-between-points (point1 point2)
  (let ((x-dif (- (X-coord point2) (X-coord point1)))
        (y-dif (- (Y-coord point2) (Y-coord point1))))
  (sqrt (+ (* x-dif x-dif)  (* y-dif y-dif)))))

;;
; Define memonics for accessing data in lists:
;;

(defun start-node (path) (eval (car path)))
(defun end-node (path) (eval (cadr path)))
(defun start-node-name (x) (car x))
(defun end-node-name (x) (cadr x))
```

```
(defun first-on-path (x) (caddr x))
(defun goal-node (x) (car x))
(defun distance-to-that-node (x) (cadr x))

;;
 ; Generate all child paths from a node to the goal:
 ;;

(defun enumerate-children (node goal)
 (let* (
        (start-to-lead-node-dist (cadr node)) ;; distance already calculated
         (path (caddr node))
         (lead-node (car path)))
   (if (get-stored-path lead-node goal)
     (consider-best-path lead-node goal path start-to-lead-node-dist)
     (consider-all-nodes lead-node goal path start-to-lead-node-dist))))

;;
 ; Set "best-first" search towards next node:
 ;;

(defun consider-best-path (lead-node goal path distance-to-here)
 (let* (
         (stored-path (get-stored-path lead-node goal))
         (stored-dist (distance-to-that-node stored-path))
         (first-node (get-first-node-in-path lead-node goal))
         (dist-to-first (+ distance-to-here
                           (get-stored-dist lead-node first-node)))
         (total-estimate (+ distance-to-here
                            (get-stored-dist lead-node goal)))
         (new-path (cons first-node path)))
 (list (list total-estimate dist-to-first new-path))))

;;
 ; Return optimum path between two nodes (if available):
 ;;

(defun get-stored-path (start goal)
 (if (equal start goal)
    (list start 0)
    (assoc goal (get start 'path-list))))
```

```
;;
; If optimum path between nodes has not been calculated
; already, consider all nodes connected to lead-node:
;;

(defun consider-all-nodes (lead-node goal path start-to-lead-node-dist)
 (let (dist-to-first total-estimate new-path new-nodes)
  (dolist (node (collect-linked-nodes lead-node))
   (if (node-not-in-path node path)
    (let ()
     (setq dist-to-first (+ start-to-lead-node-dist
                            (get-stored-dist lead-node node)))
     (setq total-estimate (+ dist-to-first
                             (dist-between-points
                              (eval node)
                              (eval  goal))))
     (setq new-path (cons node path))
     (setq new-nodes (cons (list total-estimate
                                 dist-to-first
                                 new-path)
                           new-nodes)))))
  new-nodes))


;;
; Collect all nodes connected to node:
;;

(defun collect-linked-nodes (node)
 (let (links)
  (dolist (link (get node 'path-list))
   (if (null (first-on-path link))
    (setq links (cons (goal-node link) links))))
  links))

;;
; Check to see if a node is on a given path:
;;

(defun node-not-in-path (node path)
 (if (null path)
  t
  (if (equal node (car path))
   nil
   (node-not-in-path node (cdr path)))))
```

```
;;
; Retrieve the distance between nodes:
;;

(defun get-stored-dist (node1 node2)
  (distance-to-that-node (get-stored-path node1 node2)))

;;
; Order the paths leading out of a node in "best-first" order:
;;

(defun Collect-ascending-search-list-order (a l)
  (if (null l)
    (list a)
    (if (< (car a) (caar l))
      (cons a l)
      (cons (car l) (Collect-ascending-search-list-order a (cdr l))))))


;;
; Return the first node in a path towards the goal node:
;;

(defun get-first-node-in-path (start goal)
  (let (first-node)
    (setq first-node  (first-on-path (get-stored-path start goal)))
    (if first-node first-node goal)))

;;
;     Throw away test code:   Execute (test1) then (test2) to display
;     a network and performs an A* search between two nodes:
;;

(defun test1 ()
  (setq
      n1 '(30 201)
      n2 '(25 140)
      n3 '(55 30)
      n4 '(105 190)
      n5 '(95 110)
      n6 '(140 22)
      n7 '(160 150)
      n8 '(170 202)
      n9 '(189 130)
      n10 '(200 55)
```

```
      n11 '(205 201))
  (setq nodes
      '(n1 n2 n3 n4 n5 n6 n7 n8 n9 n10 n11))
  (setq paths
      '((n1 n2) (n2 n3) (n3 n5) (n3 n6) (n6 n10)
        (n9 n10) (n7 n9) (n1 n4) (n4 n2) (n5 n8)
        (n8 n4) (n7 n11)))
  (init-network)
  (init-plot)
  (dolist (p paths)
    (let ((from (eval (car p)))
          (to (eval (cadr p))))
      (plot-line
        (x-coord from)
        (y-coord from)
        (x-coord to)
        (y-coord to))))
  (dolist (n nodes)
    (let ((n-val (eval n)))
      (plot-string
        (+ 2 (x-coord n-val))
        (y-coord n-val)
        (princ-to-string n)))))

(defun test2 ()
  (defun draw-path (pl)
    (print pl)
    (pen-width 3)
    (let ((node1 (car pl)))
      (setq pl (cdr pl))
      (dolist (p pl)
        (plot-line (x-coord (eval node1)) (y-coord (eval node1))
                   (x-coord (eval p)) (y-coord (eval p)))
        (setq node1 p))))
  (draw-path (caddr (A*search 'n1 'n11))))
```

11.3 Executing the A* Search Program

Executing function *test1* generates the plot seen in Figure 11.1 showing 11 nodes in a sparsely connected network. The graphics library from chapter 2 must be loaded before running function *test1*.

Executing function *test2* in the sample program calculates the best path between nodes N1 and N11 and hilights the path in the graphics display as seen in Figure 11.2.

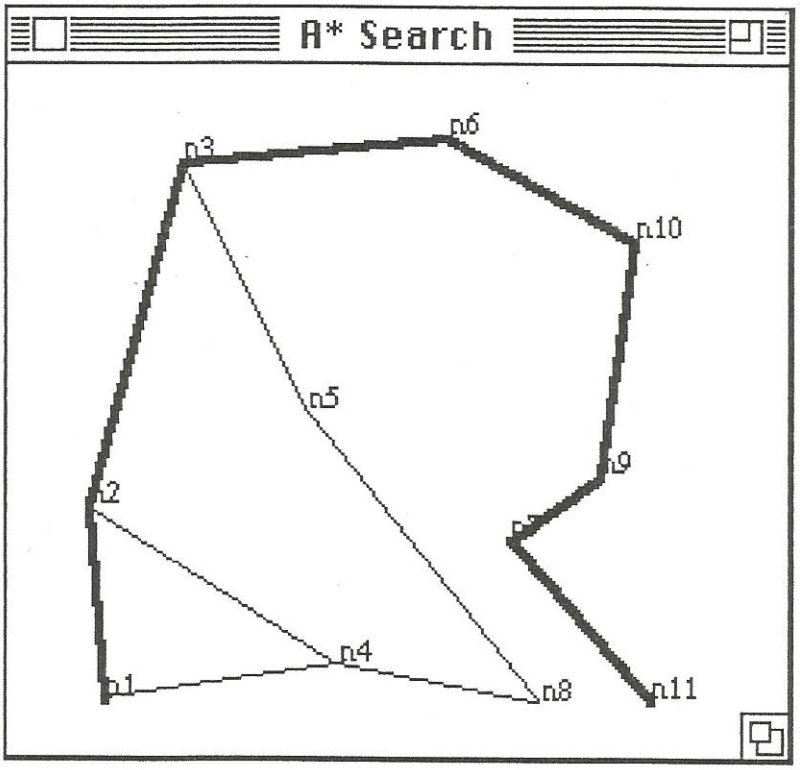

Figure 11.2. The Best Path between Nodes n1 and n11

The program presented in this chapter can search a network with a 100 nodes for a path in a few seconds on a Mac II running Macintosh Common LISP. Notice that in our example, the heuristically directed breadth-first search algorithm is "misled" rather badly since this cost function will cause the search to proceed first to node N4 and then to node N8; the program still finds the correct shortest path, but the search is more expensive computationally.

11.4 Projects

1. Write a program to calculate best paths between cities in the USA. Superimpose a Cartesian grid over a map with units in miles. Then enter the location of each major city as a point (x, y). Enter a list of roads where each road is defined as a city pair. Using this data, plot the cities in the graphics window. Your program should accept two cities as input (use the mouse pointing device, or require typing in the names) and plot out the best path between the cities using the available roads. Reuse the test functions *test1* and *test2* , changing the data initialization code.

2. Modify the example program to allow each path to have a speed-cost attribute. The cost function *dist-between-points* in the program in section 11.2 uses the