

```

; A solution to Question 4.

; Transform the representation of an undirected graph
; from a set of edges to a set of adjacency lists.
; Nodes is a list of nodes, and Edges is a list of edges (pairs of
; nodes). The output is a list of the form (A1 A2 ... An), where each
; Ai is an adjacency list for a node. ie, the first element of Ai is a
; node, and the remaining elements are its neighbours.

(define (transform Nodes Edges)
  (map (lambda (Node)
        (cons Node (adjList Node Edges)))
       Nodes))

; Given a list of Edges, return a list of the neighbours of Node.

(define (adjList Node Edges)
  (mapAppend (lambda (Edge) ; Apply this lambda function to each
edge.
              (let ((N1 (car Edge))
                    (N2 (cadr Edge)))
                (cond ((equal? N1 Node) (list N2))
                      ((equal? N2 Node) (list N1))
                      (#t '()))))
            Edges))

; Apply function F to each element of List, and append the results
together.

(define (mapAppend F List)
  (cond ((null? List) '())
        (#t (append (F (car List))
                     (mapAppend F (cdr List))))))

; Find a Hamiltonian path in RoadMap starting at City.
; If such a path does not exist, then return #f.

(define (path RoadMap City)
  (let ((Cities (car RoadMap))
        (Roads (cadr RoadMap)))
    (let ((answer (choose (list City) ; Look for a
                          Hamiltonian path
                          '()
                          (transform Cities Roads)
                          (length Cities))))
      (if (list? answer) (reverse answer) #f))) ; If the
answer is a list, then reverse it

```

```
; Path is a prefix of a Hamiltonian path. N is the number of cities
; that have to be added to it to make it a complete Hamiltonian path.
; Roads is a representation of the roads in adjacency-list form. The
; Choose function tries to extend Path by choosing a city from the
; Cities list, adding it to Path, and then recursing. If this fails to
; lead to a Hamiltonian path, then a different city is chosen.
```

```
(define (choose Cities Path Roads N)
  (cond ((zero? N) Path)
        ((null? Cities) #f) ; no more cities left to choose from
        (#t (or (extend (car Cities) Path Roads N)
                 (choose (cdr Cities) Path Roads N)))))
```

```
; Try to extend Path by adding City to it,
; and then choose a neighbour of City to extend Path even further.
```

```
(define (extend City Path Roads N)
  (cond ((member City Path) #f) ; the city is already on the path
        (#t (choose (neighbours City Roads)
                     (cons City Path)
                     Roads
                     (- N 1)))))
```

```
; Given a City, return a list of its neighbouring cities,
; i.e., the cities to which it is connected by a road.
; Assume Roads is in adjacency-list form.
```

```
(define (neighbours City Roads)
  (let ((Cities (car Roads)))
    (cond ((equal? City (car Cities)) (cdr Cities))
          (#t (neighbours City (cdr Roads))))))
```