Depth First Search (DFS)

Problem Definition

Depth First Search

- is an algorithm for traversing graph starting from a given node and exploring as far as possible along each branch before backtracking.

Note,

- DFS depends on how out edges are ordered (in the case above they are sorted by value).
- we focus on acyclic direct graphs

Depth First Search (DFS)

Abstract Datatypes

To solve the problem we need:

- a tree datatype to represent the result of the visit
  ```plaintext
type 'a tree = Leaf of 'a | Tree of ('a * 'a tree list);
```
- a graph datatype to support the obvious needing
  ```plaintext
module type GraphADT =
  sig
    type 'a graph
    let empty() : 'a graph
    let add_node (x : 'a) : 'a graph -> 'a graph
    let add_arc (x : 'a) (y : 'a) : 'a graph -> 'a graph
    let adjacents (x : 'a) (nodes : 'a graph) : 'a list
    let node_is_in_graph (x : 'a) (nodes : 'a graph) : bool
    exception TheGraphIsEmpty
    exception TheNodeIsNotInGraph
  end;
```

Depth First Search (DFS)

Graph Implementation

```plaintext
module Graph : GraphADT =
  struct
    type 'a graph = Graph of ('a list) * ((('a * 'a) list) list)
    let empty() = Graph([], [])
    let is_empty = function
      Graph(nodes, _) -> (nodes = [])
    exception TheGraphIsEmpty
    exception TheNodeIsNotInGraph
    (* checks if an element belongs to the list *)
    let rec is_in_list ?(res= false) x = function
      [] -> res
      | h::tl -> is_in_list ~res: (res || (x = h)) x tl
    (* checks if a node is in the graph *)
    let node_is_in_graph x (nodes : 'a graph) = is_in_list x nodes
    ...
  end
```
Depth First Search (DFS)

Graph Implementation (Follows)

open Graph

let rec add_in_list ?(res=[]) x = function
  | [] -> List.rev x::res
  | h::t when (h=x) -> List.rev_append t (h::res)
  | h::t -> add_in_list ~res: (h::res) x t

(* adds an element to a list if not present *)

let rec add_node n = function
  | Graph( [], [] ) -> Graph( [n], [] )
  | Graph( nodes, arcs ) -> Graph( (add_in_list n nodes), arcs )

let rec add_arc s d = function
  | Graph(nodes, arcs) -> Graph( (add_in_list d (add_in_list s nodes)), (add_in_list (s,d) arcs) )

let adjacents n = function
  | Graph(_, arcs) -> adjacents n arcs

(* returns the nodes adjacent to the given node *)

let dfs g v = let rec dfs g v g' = function
  | [] -> g'
  | hd::tl when (node_is_in_graph hd g') -> dfs g v g' tl
  | hd::tl -> dfs g v (add_arc v hd (dfs g hd (add_node hd g') (adjacents hd g))) tl

in

if is_empty g then raise TheGraphIsEmpty
else if not (node_is_in_graph v g) then raise TheNodeIsNotInGraph
else graph_to_tree (dfs g v (add_node v (empty())) (adjacents v g)) v

let arcs_to_graph arcs = let rec arcs_to_graph g = function
  | [] -> g
  | (s,d)::tl -> arcs_to_graph (add_arc s d g) tl in arcs_to_graph (empty()) arcs

(* transforms a list of arcs in a graph *)

let graph_to_tree g root = let rec make_tree n = function
  | [] -> Leaf n
  | adj_to_n -> Tree n (make_forest adj_to_n)

and make_forest = function
  | [] -> []
  | hd::tl -> (make_tree hd (adjacents hd g))::(make_forest tl)
in

make_tree root (adjacents root g)

Depth First Search (DFS)

DFS Implementation

Depth First Search (DFS)

Ancillary Operations on Graphs
References

- Davide Ancona, Giovanni Lagorio, and Elena Zucca.
  Linguaggi di Programmazione.
  Città Studi Edizioni, 2007.

- Greg Michaelson.
  An Introduction to Functional Programming through λ-Calculus.
  Addison-Wesley, 1989.

- Larry C. Paulson.
  ML for the Working Programmer.