ML in Action
Graph Coverage

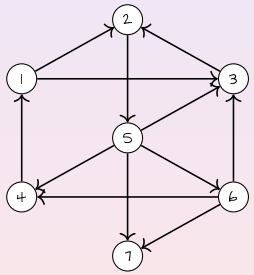
Walter Cazzola

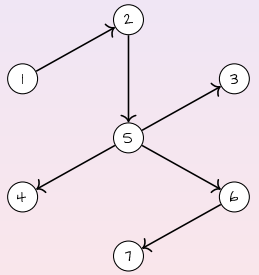
Dipartimento di Informatica
Università degli Studi di Milano
e-mail: cazzola@di.unimi.it
twitter: @w_cazzola

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

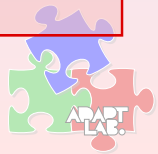
To solve the problem we need:

- a tree datatype to represent the result of the visit

```
type 'a tree = Leaf of 'a | Tree of ('a * 'a tree list);;
```

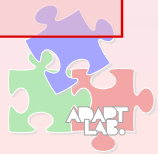
- a graph datatype to support the obvious needing

```
module type GraphADT =
  sig
   type 'a graph

   val empty : unit -> 'a graph

   val add_node : 'a -> 'a graph -> 'a graph
   val add_arc : 'a -> 'a -> 'a graph -> 'a graph
   val adjacents : 'a -> 'a graph -> 'a list
   val node_is_in_graph : 'a -> 'a graph -> bool
   val is_empty : 'a graph -> bool

   exception TheGraphIsEmpty
   exception TheNodeIsNotInGraph
  end;;
```



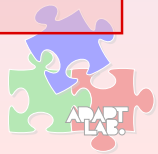
Depth First Search (DFS)
Graph Implementation

```
module Graph : GraphADT =
 struct
    type 'a graph = Graph of ( 'a list ) * ( ( 'a * 'a ) 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 n = function
      Graph(nodes, _) -> is_in_list n nodes

    ...
 end
```



Depth First Search (DFS)
Graph Implementation (Follows)

```
(* adds an element to a list if not present *)
let rec add_in_list ?(res=[]) x = function
  []    -> List.rev x::res
| h::tl when (h=x) -> List.rev_append tl (h::res)
| h::tl            -> add_in_list ~res: (h::res) x tl
(* operations to add new nodes and arcs (with their nodes) to the graph, respectively *)
let add_node n = function
  Graph( [], [] )       -> Graph( [n], [] )
| Graph( nodes, arcs ) -> Graph( (add_in_list n nodes), arcs )
let add_arc s d = function
  Graph(nodes, arcs) ->
     Graph( (add_in_list d (add_in_list s nodes)), (add_in_list (s,d) arcs) )
(* returns the nodes adjacent to the given node *)
let adjacents n =
  let adjacents n l = List.map snd (List.filter (fun x -> ((fst x) = n)) l)
in function
  Graph(_, arcs) -> adjacents n arcs
```

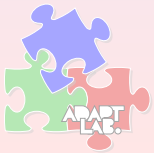


Depth First Search (DFS)
Ancillary Operations on Graphs

```
open Graph
(* transforms a list of arcs in a graph *)
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
(* extract a tree out of acyclic graph with the given node as the root *)
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

```
open Graph
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
```



Depth First Search (DFS)
DFS in Action

```
[18:08]cazzola@surtur:~/lp/ml>ocaml
# #use "tree.ml";;
type 'a tree = Leaf of 'a | Tree of ('a * 'a tree list)
# #use "GraphADT.mli";;
module type GraphADT =
  sig
    type 'a graph
    val empty : unit -> 'a graph
    val add_node : 'a -> 'a graph -> 'a graph
    val add_arc : 'a -> 'a -> 'a graph -> 'a graph
    val adjacents : 'a -> 'a graph -> 'a list
    val node_is_in_graph : 'a -> 'a graph -> bool
    val is_empty : 'a graph -> bool
    exception TheGraphIsEmpty
    exception TheNodeIsNotInGraph
  end
# #use "Graph.ml" ;;
module Graph : GraphADT
# #use "aux.ml" ;;
val arcs_to_graph : ('a * 'a) list -> 'a Graph.graph = <fun>
val graph_to_tree : 'a Graph.graph -> 'a -> 'a tree = <fun>
# #use "dfs.ml" ;;
val dfs : 'a Graph.graph -> 'a -> 'a tree = <fun>
# let g1 = arcs_to_graph [(1,2);(1,3);(4,1);(5,4);(3,2);(2,5);(5,3);(5,6);(5,7);(6,7);(6,3);(6,4)] ;;
val g1 : int Graph.graph = <abstr>
# let g7 = arcs_to_graph([("Algol", "Pascal"); ("Algol", "C"); ("Algol", "Java"); ("C", "Java");
  ("Algol", "Python"); ("Pascal", "Modula 2"); ("C", "C++"); ("Java", "Scala"); ("Lisp", "ML");
  ("Lisp", "Scala"); ("Lisp", "Python"); ("Lisp", "Erlang"); ("ML", "OCaML")]);;
val g7 : string Graph.graph = <abstr>
# dfs g1 1 ;;
- : int tree = Tree (1, [Tree (2, [Tree (5, [Leaf 4; Leaf 3; Tree (6, [Leaf 7])])])])
# dfs g7 "Algol";;
- : string tree = Tree ("Algol", [Tree ("Pascal", [Leaf "Modula 2"]);
   Tree ("C", [Tree ("Java", [Leaf "Scala"]); Leaf "C++"]); Leaf "Python"])
# dfs g7 "Lisp" ;;
- : string tree = Tree ("Lisp", [Tree ("ML", [Leaf "OCaML"]); Leaf "Scala"; Leaf "Python"; Leaf "Erlang"])
```

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.
  Cambridge University Press, 1996.