Jacques Garrigue, 2006 年 1 月 10 日

V　OCaml応用プログラミング

1　最短経路問題

最短経路問題とは，グラフの中で隣接頂点間の距離が与えられたとき，任意の二つの頂点の間の最短経路を見付ける問題である．これを解くのに有名な Dijkstra のアルゴリズムを使う．

頂点 A と頂点 B の間の最短経路を探すために

1. 頂点 A と隣接する頂点の候補リストを作る．A を既知頂点集合に加える．
2. 候補リストの中から最も近い頂点 C を選ぶ．もしもその頂点が B であればこれで終り．
3. C を候補リストから除き，既知頂点集合に加える．
4. C と隣接する頂点で，既知頂点集合に含まれない頂点を候補リストに加える．既に候補リストに入っていた頂点に関して，短かい方の経路だけを残す．
5. 2 から繰り返す．

このアルゴリズムのポイントは，最も近い隣接頂点を選ぶことにある．最も近いので，それ以外の頂点を経由するもっと短かい経路がないからである．B までの最短経路が見付かるだけでなく，B より近い全ての頂点 (すなわち既知頂点) についても最短経路が計算される．

毎回一つの頂点が既知集合に加えられるので，頂点が N 個あれば，ループが最大で N 回実行される．ループの中で行なわれる処理は全て N ステップ以下でできるので，おおざっぱにいうとアルゴリズム全体が $O(N^2)$ になる．

これを計算する簡単なアルゴリズムを以下に記す．

まず，グラフの表現を決めなければならない．リストのリストが最も書きやすいのでそれを使う．

```
let distances =
  [ "Nagoya", ["Sakae", 2; "Hisaya", 3];
    "Sakae", ["Hisaya", 1; "Imaike", 3];
    "Hisaya", ["Imaike", 3; "Motoyama", 11];
    "Imaike", ["Motoyama", 3]; "Motoyama", ["Daigaku", 1]]
val distances : (string * (string * int) list) list = ...
```

しかし，これでは一方の道しか含まれないので，そのグラフを対称的にしなければならない．

```
let rec add_graph_edge v edge = function
  | [] -> [v, [edge]]
  | (v',edges as line)::graph ->
      if v = v' then (v, edge::edges) :: graph
      else line :: add_graph_edge v edge graph
val add_graph_edge : 'a -> 'b -> ('a * 'b list) list -> ('a * 'b list) list
let symmetric graph =
  List.fold_right
    (fun (v, edges) ->
```

```
      List.fold_right (fun (v',d) -> add_graph_edge v' (v,d)) edges)
    graph graph
val symmetric : ('a * ('a * 'b) list) list -> ('a * ('a * 'b) list) list
let graph = symmetric distances
val graph : (string * (string * int) list) list =
  [("Nagoya", [("Sakae", 2); ("Hisaya", 3)]);
   ("Sakae", [("Nagoya", 2); ("Hisaya", 1); ("Imaike", 3)]); ...]
```

このグラフを使って，最短経路を計算する．

```
(* 最も近い隣接頂点をリストから抜き出し，その頂点と残りのリストを返す *)
val extract_closest : ('a * int) list -> ('a * int) * ('a * int) list
(* ある頂点との最短距離を候補リストに加える．
   既に含まれていれば，短かい方の距離にする *)
val add_edge : 'a * int -> ('a * int) list -> ('a * int) list

(* 前の関数を利用し，隣接頂点を候補リストに加える *)
let add_edges d edges =
  let edges = List.map (fun (v',d') -> (v',d+d')) edges in
  List.fold_right add_edge edges
val add_edges : int -> ('a * int) list -> ('a * int) list -> ('a * int) list

let rec shortest ~graph ~seen ~nexts dest =
  if nexts = [] then failwith "shortest" else
  let (v,d), nexts = extract_closest nexts in
  if v = dest then d else
  let edges = List.assoc v graph in
  let edges = List.filter (fun (v,d) -> not (List.mem v seen)) edges in
  let nexts = add_edges d edges nexts in
  shortest ~graph ~seen:(v::seen) ~nexts dest
val shortest :
  graph:('a * ('a * int) list) list ->
  seen:'a list -> nexts:('a * int) list -> 'a -> int

let shortest_path graph start dest =
  shortest ~graph ~seen:[] ~nexts:[start,0] dest
val shortest_path : ('a * ('a * int) list) list -> 'a -> 'a -> int
```

練習問題 1.1 関数 extract_closest および add_edge を定義せよ．

経路記録

上のやりかたでは経路が記録されなくて，最短距離だけが分かる．経路も知りたければ，距離と一緒に記録していかなければならない．

各関数の型が以下のように変わる．

```
type path = { dest : string; length : int; path : string list }
val extract_closest : path list -> path * path list
val add_edge : path -> path list -> path list
val add_edges : path -> (string * int) list -> path list -> path list
val shortest :
  graph:(string * (string * int) list) list ->
  seen:string list -> nexts:path list -> string -> path
```

```
val shortest_path :
  (string * (string * int) list) list -> string -> string -> path
```

修正された関数は以下のような働きをする．

```
# shortest_path graph "Nagoya" "Daigaku";;
- : path = {dest = "Daigaku"; length = 9;
            path = ["Motoyama"; "Imaike"; "Sakae"; "Nagoya"]}
```

途中の駅はリストの頭から加えていくので，経路が逆の順番で出力される．

練習問題 1.2 経路を記録するコードを書きなさい．

ライブラリーのデータ構造

アルゴリズムの計算量がおおざっぱに $O(N^2)$ とはいえ，それはグラフ情報へのアクセスを含まない．前記のように全ての処理にリストを使うと，ループ内の処理が $O(N)$ ではなく $O(N^2)$ になるので，全体が $O(N^3)$ になるわけだ．

それを避けるために，リストより効率のよいデータ構造を使わなければならい．配列にするというアプローチもあるが，頂点を整数にしないといけないのでアルゴリズムが読みにくくなる．ここでは，標準ライブラリーに含まれる Set および Map を使う．どちらもファンクターとして提供されている．

内部的に要素の順序を利用する木構造が使われるので，ファンクターの引数は `OrderedType` である．ほとんどの操作が $O(\log N)$ で実行できる．

```
module type OrderedType =
  sig
    type t                                              (* 要素の型 *)
    val compare : t -> t -> int                         (* 比較関数 *)
  end
module Set.Make :
  functor (Ord : OrderedType) ->
    sig
      type elt = Ord.t
      type t
      val empty : t
      val add : elt -> t -> t
      val remove : elt -> t -> t
      val mem : elt -> t -> bool
      val fold : (elt -> 'a -> 'a) -> t -> 'a -> 'a
      ...
    end
module Map.Make :
  functor (Ord : OrderedType) ->
    sig
      type key = Ord.t
      type 'a t
      val empty : 'a t
      val add : key -> 'a -> 'a t -> 'a t
      val find : key -> 'a t -> 'a
      val remove : key -> 'a t -> 'a t
      val mem : key -> 'a t -> bool
      val fold : (key -> 'a -> 'b -> 'b) -> 'a t -> 'b -> 'b
```

```
    ...
  end
```

それらを使うとこんなプログラムになる.

```
type path = {length: float; path: string list}
module VMap = Map.Make(String)          (* Stringモジュールを引数として渡す *)
module VSet = Set.Make(String)

let find_closest m =
  VMap.fold
    (fun v p -> function
      | Some (_,p') as vp' when p.length >= p'.length -> vp'
      | _ -> Some (v,p))
    m None
val find_closest : path VMap.t -> (VMap.key * path) option

let add_edge v p nexts =
  try
    let p' = VMap.find v nexts in                          (* O(log N) *)
    if p.length < p'.length then raise Not_found else nexts
  with Not_found ->
    VMap.add v p nexts
val add_edge : VMap.key -> path -> path VMap.t -> path VMap.t

let rec shortest ~graph ~seen ~nexts dest =
  match find_closest nexts with
  | None -> failwith "shortest"
  | Some (v,p) ->
      if v = dest then p else
      let nexts = VMap.remove v nexts in
      let edges = List.assoc v graph in
      let nexts =
        List.fold_right
          (fun (v', d) nexts ->
            if VSet.mem v' seen then nexts else            (* O(log N) *)
            add_edge v' {length = p.length +. d; path = v::p.path} nexts)
          edges nexts in
      shortest ~graph ~seen:(VSet.add v seen) ~nexts dest
val shortest :
  graph:(VMap.key * (VSet.elt * float) list) list ->
  seen:VSet.t -> nexts:path VMap.t -> VMap.key -> path

let shortest_path graph start dest =
  let nexts = VMap.add start {length = 0.; path = []} VMap.empty in
  shortest ~graph ~seen:VSet.empty ~nexts dest
val shortest_path :
  (VMap.key * (VSet.elt * float) list) list -> VMap.key -> VMap.key -> path
```