Jacques Garrigue, 2006 年 1 月 11 日

11　続・グラフアルゴリズム

添字による表現

「メモリ上」表現の変型として，配列による表現が考えられる．各ノードを名前で指すのではなく，配列の何番目かを利用するので，直接表現に近い効率を保ちながら，データの表示が簡単になる．

```
let index (x : 'a) (arr : 'a array) =                          (* 添字を返す *)
  let len = Array.length arr in
  let rec find n =
    if n >= len then raise Not_found else
    if x = arr.(n) then n else find (n+1)
  in find 0
val index : 'a -> 'a array -> int

type edge = {next: int; dist: int}

let array_of_graph ll =                                  (* グラフを配列表現に変える *)
  let data = Array.of_list ll in
  let names = Array.map fst data in
  let graph =
    Array.map
      (fun (_, l) ->
        List.map (fun (n,d) -> {next = index n names; dist = d}) l)
      data
  in (names, graph)
val array_of_graph : ('a * ('a * int) list) list -> 'a array * edge list array

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

# let (names, dist_array) = array_of_graph distances ;;
val names : string array =
  [|"Nagoya"; "Sakae"; "Hisaya"; "Imaike"; "Motoyama"; "Daigaku"|]
val dist_array : edge list array =
  [|[{next = 1; dist = 2}; {next = 2; dist = 3}];
    [{next = 0; dist = 2}; {next = 2; dist = 1}; {next = 3; dist = 3}];
    [{next = 0; dist = 3}; {next = 1; dist = 1}; {next = 3; dist = 3};
     {next = 4; dist = 11}];
    [{next = 4; dist = 3}; {next = 1; dist = 3}; {next = 2; dist = 3}];
    [{next = 3; dist = 3}; {next = 2; dist = 11}; {next = 5; dist = 1}];
    [{next = 4; dist = 1}]|]
```

到達可能性

直感的なアルゴリズムでもうまくいくので，簡単.

```
let rec visit arr seen i =                                    (* 深さ優先でたどる *)
  if not seen.(i) then begin
    seen.(i) <- true;
    List.iter (fun {next=j} -> visit arr seen j) arr.(i)
  end
val visit : edge list array -> bool array -> int -> unit

let accessible arr start =                                    (* 配列の初期化 *)
  let seen = Array.create (Array.length arr) false in
  visit arr seen start;
  seen
val accessible : edge list array -> int -> bool array
```

同じ辺を一回しか通らないので，このアルゴリズムの計算量は $O(m)$ である．もしも，ノードを考えた場合，辺の数が最大であれば $O(n^2)$ になる.

最短経路問題

Dijkstra のアルゴリズムによる解.

```
type edge' = {prev: int; dist': int}

type status = Reached of edge' | Fringe of edge' | Unseen

let rec closest seen p0 fringe =                              (* 最も近いノードを探す *)
  match fringe with
    [] -> p0
  | n1 :: rem ->
      match seen.(n1) with
        Fringe p1 ->
          let p' =
            if p1.dist' < p0.dist then {next=n1; dist=p1.dist'} else p0 in
          closest seen p' rem
      | _ -> failwith "closest"
val closest : status array -> edge -> int list -> edge

let rec remove (n : int) fringe =                             (* たどったノードを外す *)
  match fringe with
    [] -> failwith "remove"
  | n' :: rem -> if n = n' then rem else n' :: remove n rem
val remove : int -> int list -> int list

let rec add_to_fringe seen p steps fringe =
  match steps with
    [] -> fringe
  | e :: rem ->
      let n = e.next in
      match seen.(n) with
        Unseen ->                                 (* まだ見ていないので，リストに追加 *)
          seen.(n) <- Fringe {prev = p.next; dist' = p.dist + e.dist};
```

```
          add_to_fringe seen p rem (n :: fringe)
      | Fringe p' ->                                    (* 距離の更新だけを行う *)
          let d = p.dist + e.dist in
          if d < p'.dist' then seen.(n) <- Fringe {prev = p.next; dist' = d};
          add_to_fringe seen p rem fringe
      | Reached _ ->                                    (* 何もすることがない *)
          add_to_fringe seen p rem fringe
val add_to_fringe : status array -> edge -> edge list -> int list -> int list

let get_fringe seen n =                          (* Fringeだけを読み出す補助関数 *)
  match seen.(n) with
    Fringe p -> p
  | _ -> failwith "get_fringe"
val get_fringe : status array -> int -> edge'

let rec dijkstra arr seen fringe =                              (* アルゴリズム *)
  match fringe with
    [] -> ()
  | n0 :: rem ->
      let p0 = {next = n0; dist = (get_fringe seen n0).dist'} in
      let p = closest seen p0 rem in              (* 原点に最も近いノードを選ぶ *)
      let n = p.next in
      seen.(n) <- Reached (get_fringe seen n);                (* 距離を固定する *)
      let fringe = add_to_fringe seen p arr.(n) (remove n fringe) in
      dijkstra arr seen fringe
val dijkstra : edge list array -> status array -> int list -> unit

let shortest_paths arr n0 =                                     (* 配列の初期化 *)
  let seen = Array.create (Array.length arr) Unseen in
  seen.(n0) <- Fringe {prev = -1; dist' = 0};
  dijkstra arr seen [n0];
  seen
val shortest_paths : edge list array -> int -> status array
```

最悪の場合の計算量を考える．最初から全てのノードが fringe に入っていると見る．毎回最も近いものを選ばなければならないので，そのために毎回 $O(n)$ かかる．そこから行けるところを fringe に追加するために，最大でさらに $O(n)$ かかる．それを n 回くりかえすので，最終的な計算量は $O(n^2)$ である．

実習課題 (11 回目)

1. 上の Dijkstra のアルゴリズムを利用して，距離と実際の最短経路を返す関数を書きなさい．

   ```
   val shortest_path : edge list array -> int -> int -> int * int list
   # shortest_path dist_array 0 5;;
   - : int * int list = (9, [0; 1; 3; 4; 5])
   ```

2. グラフのノードを近いもの順の lazylist として返す関数を定義せよ．
   `val closests :  edge list array -> int -> edge lazylist`
   後部を参照したときに初めて計算が起きるようにせよ．

3. `lazylist` の型定義を多相型`'a lazylist` に変更せよ．
   ファイルの中身を行の `lazylist` として返す関数を定義せよ．`End_of_file` が起こされた

ら，ファイルを閉じて Nil を返す.

```
val read_lazy : string -> string lazylist
```