4 グラフ・アルゴリズム (続)

訂正 (レコードのフィールド名を変えた)

```
type ('a,'b) graph = {vertices: 'a list; edges: (int*'b) list array}

let rec index x l =
  match l with
    [] -> raise Not_found
  | a::l -> if x = a then 0 else 1 + index x l
val index : 'a -> 'a list -> int

let add_edge g a b w =
  let e = g.edges in
  let i = index a g.vertices and j = index b g.vertices in
  e.(i) <- (j,w) :: e.(i);
  e.(j) <- (i,w) :: e.(j)
val add_edge : ('a, 'b) graph -> 'a -> 'a -> 'b -> unit

let build_graph edgell =
  let len = List.length edgell and names = List.map fst edgell in
  let g = {vertices = names; edges = Array.create len []} in
  List.iter
    (fun (a, neighbours) ->
      List.iter (fun (b, w) -> add_edge g a b w) neighbours)
    edgell;
  g
val build_graph : ('a * ('a * 'b) list) list -> ('a, 'b) graph

let distances =
  [ "Nagoya", ["Sakae", 2; "Hisaya", 3];
    "Sakae", ["Hisaya", 1; "Imaike", 3];
    "Hisaya", ["Imaike", 3; "Motoyama", 11];
    "Imaike", ["Motoyama", 3];
    "Motoyama", ["Daigaku", 1];
    "Daigaku", [] ] ;;
val distances : (string * (string * int) list) list = ...
let graph = build_graph distances ;;
val graph : (string, int) graph =
  {vertices = ["Nagoya"; "Sakae"; "Hisaya"; "Imaike"; "Motoyama"; "Daigaku"];
   edges = [|[(2, 3); (1, 2)]; [(3, 3); (2, 1); (0, 2)]; ... |]}
```

前回問題の答

```
type status = Reached of int * int | Fringe of int * int | Unseen

let visit_neighbour ~status ~fringe ~source ~srcdist (next, dist) =
  match status.(next) with
  | Reached _ -> fringe
  | Fringe (d, src) ->
      let d' = srcdist + dist in
      if d' < d then status.(next) <- Fringe (d', source);
```

```
      fringe
  | Unseen ->
      status.(next) <- Fringe (srcdist + dist, source);
      next :: fringe
val visit_neighbour :
  status:status array ->
  fringe:int list -> source:int -> srcdist:int -> int * int -> int list

let closest_fringe ~status ~fringe =
  match fringe with
    [] -> 0
  | i0 :: fringe ->
      List.fold_left
        (fun closest i ->
          match status.(closest), status.(i) with
          | Fringe (d, _), Fringe (di, _) when di < d -> i
          | _ -> closest)
        i0 fringe
val closest_fringe : status:status array -> fringe:int list -> int

let rec remove x l =
  match l with
    [] -> failwith "remove"
  | a :: l -> if x = a then l else a :: remove x l
val remove : 'a -> 'a list -> 'a list

let rec shortest_rec ~graph ~status ~fringe =
  let next = closest_fringe ~status ~fringe in
  match status.(next) with
    Fringe (d, src) ->
      status.(next) <- Reached (d, src);
      let fringe =
        List.fold_left
          (fun fringe ->
            visit_neighbour ~status ~fringe ~source:next ~srcdist:d)
          (remove next fringe) graph.edges.(next)
      in shortest_rec ~graph ~status ~fringe
  | _ -> ()
val shortest_rec :
  graph:('a, int) graph -> status:status array -> fringe:int list -> unit

let shortest_paths ~graph ~start =
  let status = Array.create (Array.length graph.edges) Unseen in
  let i0 = index start graph.vertices in
  status.(i0) <- Fringe (0, -1);
  shortest_rec ~graph ~status ~fringe:[i0];
  status
val shortest_paths : graph:('a, int) graph -> start:'a -> status array

let read_path ~graph ~status ~dest =
  let rec path v p =
    if v = -1 then p else
    match status.(v) with
    | Reached (_, src) -> path src (List.nth graph.vertices v :: p)
```

```
    | _ -> failwith "unreached"
  in path (index dest graph.vertices) []
val read_path :
  graph:('a, 'b) graph -> status:status array -> dest:'a -> 'a list

let shortest_path ~graph ~start ~dest =
  let status = shortest_paths ~graph ~start in
  let v = index dest graph.vertices in
  match status.(v) with
  | Reached (d, _) -> Some (d, read_path ~graph ~status ~dest)
  | _ -> None ;;
val shortest_path :
  graph:('a, int) graph -> start:'a -> dest:'a -> (int * 'a list) option
shortest_path graph "Nagoya" "Daigaku";;
- : (int * string list) option =
Some (9, ["Nagoya"; "Sakae"; "Imaike"; "Motoyama"; "Daigaku"])
```

連結成分

```
let rec connected_rec ~graph ~reached i =
  if not reached.(i) then begin
    reached.(i) <- true;
    List.iter (fun (j,_) -> connected_rec ~graph ~reached j) graph.edges.(i)
  end
val connected_rec : graph:('a, 'b) graph -> reached:bool array -> int -> unit

let connected ~graph ~start =
  let reached = Array.create (Array.length graph.edges) false in
  connected_rec ~graph ~reached (index start graph.vertices);
  List.fold_right2
    (fun v r l -> if r then v :: l else l)
    graph.vertices
    (Array.to_list reached)
    [] ;;
val connected : graph:('a, 'b) graph -> start:'a -> 'a list
connected graph "Nagoya";;
- : string list =
["Nagoya"; "Sakae"; "Hisaya"; "Imaike"; "Motoyama"; "Daigaku"]

let remove_edge g a b =
  let e = g.edges in
  let i = index a g.vertices and j = index b g.vertices in
  e.(i) <- List.remove_assoc j e.(i);
  e.(j) <- List.remove_assoc i e.(j);;
val remove_edge : ('a, 'b) graph -> 'a -> 'a -> unit

remove_edge graph "Motoyama" "Daigaku";;
- : unit = ()
connected graph "Imaike";;
- : string list = ["Nagoya"; "Sakae"; "Hisaya"; "Imaike"; "Motoyama"]
connected graph "Daigaku";;
- : string list = ["Daigaku"]
```

5 オブジェクトと隠蔽

隠蔽の必要

自由なデータ構造を使うと，不正なデータを渡される危険性がある．

```
let graph = vertices=["A";"B"]; edges=[|[(1,1)]|];;
val graph : (string, int) graph = ...
connected graph "A";;
Exception: Invalid_argument "index out of bounds".
```

アルゴリズムの中にデータ構造に直接アクセスしているので，データ構造を変えることが難しいという問題もある．

オブジェクト データと手続きをまとめた単位．

```
let counter =
  object                                              (* オブジェクトを作る *)
    val mutable n = 0                                 (* フィールド n が可変である *)
    method next =                                     (* メソッド (手続き)next を定義 *)
      n <- n + 1;                                                  (* n を更新 *)
      n
  end ;;
val counter : < next : int > = <obj>                  (* 型はメソッドの列である *)
                  (* フィールド (変数) は型に現れないので，直接にアクセスできない *)
counter#next;;                                        (* 演算子の # はメソッド呼び出し *)
- : int = 1
counter#next;;
- : int = 2
```

クラス クラスはオブジェクトを生成するための雛形．

```
class counter = object                                (* クラスの定義 *)
  val mutable n = 0
  method next =
    n <- n + 1;
    n
end ;;
class counter : object val mutable n : int method next : int end
let c = new counter ;;                                (* オブジェクトの生成 *)
val c : counter = <obj>                               (* 型がクラス名に省略される *)
c#next;;
- : int = 1
let c' = new counter ;;                               (* 異なるオブジェクト *)
val c' : counter = <obj>
c'#next;;
- : int = 1
```

継承

```
class counter' = object (self)                        (* 自分に self という名前を付ける *)
  inherit counter                                     (* counter から継承 *)
  method last = n
  method skip = ignore (self#next); self#next         (* ignore は結果を無視する *)
end
class counter' :
```

```
  object
    val mutable n : int
    method last : int
    method next : int
    method skip : int
  end
let c' = new counter';;
val c' : counter' = <obj>
c'#skip;;
 - : int = 2
c'#last;;
 - : int = 2
```

型推論と記入

```
let rec genlist c n =
  if n = 0 then [] else c#next :: genlist c (n-1) ;;
val genlist : < next : 'a; .. > -> int -> 'a list
genlist c 3;;
- : int list = [4; 3; 2]
genlist c' 3;;
- : int list = [5; 4; 3]
let rec genlist (c : counter) n =
  if n = 0 then [] else c#next :: genlist c (n-1) ;;
val genlist : counter -> int -> int list
genlist c 3;;
- : int list = [7; 6; 5]
genlist c' 3;;
Error: This expression has type counter' but is here used with type counter
       The second object type has no method last
let rec genlist (c : #counter) n =
  if n = 0 then [] else c#next :: genlist c (n-1) ;;
val genlist : #counter -> int -> int list
genlist c 3;;
- : int list = [10; 9; 8]
genlist c' 3;;
- : int list = [8; 7; 6]
```

クラスと多相性

```
class cell x = object
  val mutable x = x
  method set y = x <- y
  method get = x
end
Error: Some type variables are unbound in this type:
         class cell : 'a ->
           object
             val mutable x : 'a
             method get : 'a
             method set : 'a -> unit
           end
       The method get has type 'a where 'a is unbound
class ['a] cell (x : 'a) = object                (* 型引数と明示的に束縛する *)
```

```
  val mutable x = x
  method set y = x <- y
  method get = x
end ;;
class ['a] cell :
  'a -> object val mutable x : 'a method get : 'a method set : 'a -> unit end
let c = new cell 3 ;;
val c : int cell = <obj>                          (* オブジェクトは普通の多相型 *)
c#set 4 ;;
- : unit = ()
```

グラフの直接的な定義

```
class ['a,'b] graph vertices = object (self)
  val edges = Array.create (List.length vertices) []
  method vertices : 'a list = vertices
  method nvertices = Array.length edges
  method index v = index v vertices          (* vertices の基本操作を与える *)
  method vertex i = List.nth vertices i
  method edges i = edges.(i)                       (* edges の配列を見せない *)
  method add_edge a b (w : 'b) =
    let i = self#index a and j = self#index b in
    edges.(i) <- (j,w) :: edges.(i);
    edges.(j) <- (i,w) :: edges.(j)
end
class ['a, 'b] graph : 'a list ->
  object
    val edges : (int * 'b) list array
    method add_edge : 'a -> 'a -> 'b -> unit
    method edges : int -> (int * 'b) list
    method index : 'a -> int
    method nvertices : int
    method vertex : int -> 'a
    method vertices : 'a list
  end
```

edges が隠蔽されているので，不正な値が入れられないが，拡張性がない．

グラフの段階的な定義

```
class ['a,'b] directed_graph (vertices : 'a list) = object (self)
  val edges = Array.create (List.length vertices) []
  val vertices = vertices        (* 「関数的更新」のためにフィールドを定義する *)
  method vertices = vertices
  method nvertices = Array.length edges
  method index v = index v vertices
  method vertex i = List.nth vertices i
  method edges i = edges.(i)
  method private add_edge_int i j (w : 'b) =      (* クラスの外から使えない *)
    edges.(i) <- (j,w) :: edges.(i)
  method add_edge a b w =
    let i = self#index a and j = self#index b in
    self#add_edge_int i j w
end
class ['a, 'b] directed_graph : 'a list ->
```

```
  object
    val edges : (int * 'b) list array
    val vertices : 'a list
    method add_edge : 'a -> 'a -> 'b -> unit
    method private add_edge_int : int -> int -> 'b -> unit
    method edges : int -> (int * 'b) list
    method index : 'a -> int
    method nvertices : int
    method vertex : int -> 'a
    method vertices : 'a list
  end
class ['a,'b] directed_graph_init ll = object (self)
  inherit ['a,'b] directed_graph (List.map fst ll)
  initializer                                          (* 初期化のためのコード *)
    List.iter
      (fun (a, neighbours) ->
        List.iter (fun (b, w) -> self#add_edge a b w) neighbours)
      ll
end
class ['a, 'b] directed_graph_init : ('a * ('a * 'b) list) list ->
  object ... (* directed_graphと同じ *) end
class ['a,'b] graph ll = object
  inherit ['a,'b] directed_graph_init ll as super   (* 継承したメソッドを束縛 *)
  method private add_edge_int i j w =
    super#add_edge_int i j w;                       (* 継承したメソッドを呼ぶ *)
    super#add_edge_int j i w
end
class ['a, 'b] graph : ('a * ('a * 'b) list) list ->
  object ... (* また同じ *) end

let graph = new graph distances ;;
val graph : (string, int) graph = <obj>
graph#vertices;;
- : string list = ["Nagoya"; "Sakae"; "Hisaya"; "Imaike"; "Motoyama"; "Daigaku"]
graph#edges (graph#index "Imaike");;
- : (int * int) list = [(4, 3); (2, 3); (1, 3)]

let dgraph = new directed_graph_init distances;;
val dgraph : (string, int) directed_graph_init = <obj>
dgraph#edges (dgraph#index "Imaike");;
- : (int * int) list = [(4, 3)]
```

仮想クラス　そのままオブジェクトが作れないが，継承して仮想メンバーを埋めれば完成する．仮想メンバーは型のみの宣言になる．

```
class virtual ['a,'b] graph_remove = object (self)
  val virtual edges : (int * 'b) list array              (* 仮想フィールド *)
  method virtual index : 'a -> int                        (* 仮想メソッド *)
  method private remove_edge_int i j =                (* 取り敢えず方向付き用 *)
    edges.(i) <- List.remove_assoc j edges.(i)
  method remove_edge a b =
    let i = self#index a and j = self#index b in
    self#remove_edge_int i j
end
```

```
class virtual ['a, 'b] graph_remove :
  object
    val virtual edges : (int * 'b) list array
    method virtual index : 'a -> int
    method remove_edge : 'a -> 'a -> unit
    method private remove_edge_int : int -> int -> unit
  end
class ['a,'b] graph2 ll = object
  inherit ['a,'b] graph ll
  inherit ['a,'b] graph_remove as super
  method private remove_edge_int i j =            (* 方向なしのために上書き *)
    super#remove_edge_int i j;
    super#remove_edge_int j i
end
class ['a, 'b] graph2 : ('a * ('a * 'b) list) list ->
  object
    ...
    method remove_edge : 'a -> 'a -> unit
    method private remove_edge_int : int -> int -> unit
  end
```

頂点の追加　関数的更新によって，可変でないフィールドの値が変えられる．可変ではないので，元のオブジェクトは変更されずに，新しいオブジェクトが生成される．

```
class ['a,'b] graph3 ll = object (self)
  inherit ['a,'b] graph2 ll
  method add_vertices l =            (* 関数的更新で新しいオブジェクトを返す *)
    {< vertices = vertices @ l;
       edges = Array.append edges (Array.create (List.length l) []) >}
end
class ['a, 'b] graph3 : ('a * ('a * 'b) list) list ->
  object ('c)                                          (* 'c は自身の型 *)
    ...
    method add_vertices : 'a list -> 'c                (* 自身の型を返す *)
  end
let graph = new graph3 distances;;
val graph : (string, int) graph3 = <obj>
let graph' = graph#add_vertices ["Gokiso"; "Yagoto"];;
val graph' : (string, int) graph3 = <obj>
graph'#add_edge "Imaike" "Gokiso" 2;;
graph'#add_edge "Gokiso" "Yagoto" 3;;
graph'#add_edge "Yagoto" "Daigaku" 2;;
```

add_vertices の返り値は継承で拡張されると変わるので，graph3 ではなく，実際に生成されたオブジェクトの型になる．

練習問題 5.1　*1.* 連結成分アルゴリズムがグラフオブジェクトを使うように修正せよ．グラフは直接的な定義でいい．

2. 最短経路アルゴリズムがグラフオブジェクトを使うように修正せよ．

3. add_vertices を独立した仮想クラスで定義せよ．

4. 最短経路アルゴリズムを使い，連結成分の直径 *(*最も離れている二つの頂点の距離*)* を計算せよ．最悪の場合の計算量を調べよ．