

OCamlによる無限グラフにも使える ダイクストラ法の実装

ML Day #2

@fetburner

2018年9月16日

競プロで使うのでダイクストラ法を実装

実用上の要請：

- 疎なグラフに対して高速に
- ある始点からの最短距離を一度に計算

趣味：

- 無限グラフにも使えるようにしてみる

アウトライン

- ① ダイクストラ法とは
- ② 雑実装
- ③ Map をヒープとして使う
- ④ 純粹関数型に
- ⑤ 無限グラフに使えるように
- ⑥ OCaml は手続き型言語

ダイクストラ法

非負の重みが付いた有向グラフに対して、
単一始点最短経路問題を解くアルゴリズム

1. V を頂点全ての集合，始点への最短距離を 0，それ以外への最短距離を無限大とする
2. V が空集合ならば手続きを終了
3. 最短距離が最小となる $v \in V$ を V から削除
4. v から伸びる辺で最短距離を更新
5. 2へ

線形探索による実装 (1/2)

```
let rec dijkstra_aux : int list ->
  (int -> (int * float) list) ->
  float array -> unit = fun q e d ->
  match q with
  | [] -> ()
  | v :: q ->
    let v, q =
      List.fold_left (fun (u, q) v ->
        if d.(u) < d.(v)
        then (u, v :: q)
        else (v, u :: q)) (v, []) q in
    List.iter (fun (u, c) ->
      d.(u) <- min d.(u) (c +. d.(v))) (e v);
    dijkstra_aux q e d
```

線形探索による実装 (2/2)

```
let dijkstra : int ->  
  (int -> (int * float) list) ->  
  int -> int -> float = fun n e s ->  
    let d = Array.make n infinity in  
    d.(s) <- 0.;  
    dijkstra_aux (List.init n (fun v -> v)) e d;  
    fun v -> d.(v)
```

- 実装を隠蔽するため，関数を返す
- 最も近い頂点を線形探索しているため，疎なグラフに対しては遅い ($O(V^2)$)

Mapをヒープとして使う(1/2)

```
let rec dijkstra_aux : float array ->
  (int -> (int * float) list) ->
  int list FloatMap.t -> unit = fun d e q ->
  match FloatMap.min_binding q with
  | exception Not_found -> ()
  | (w, us) -> dijkstra_aux d e @@
    List.fold_left (fun q u ->
      if d.(u) < w then q
      else List.fold_left (fun q (v, c) ->
        if d.(v) <= w +. c then q
        else begin
          d.(v) <- w +. c;
          FloatMap.add (w +. c)
            (v :: try FloatMap.find (w +. c) q
              with Not_found -> []) q
        end) q (e u)) (FloatMap.remove w q) us
```

Mapをヒープとして使う (2/2)

```
let dijkstra : int ->  
  (int -> (int * float) list) ->  
  int -> int -> float = fun n e s ->  
  let d = Array.make n infinity in  
  d.(s) <- 0.;  
  dijkstra_aux d e (FloatMap.singleton 0. [s]);  
  fun v -> d.(v)
```

- 標準ライブラリのMapは $O(\log N)$ で
最小値を取り出せる
 - 計算量は $O(E \log V)$ に

最短距離も Map で管理してみる (1/2)

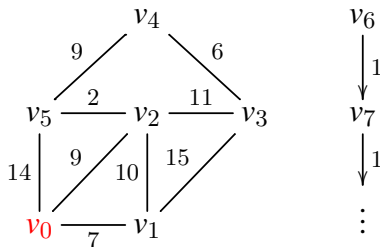
```
let rec dijkstra_aux : (int ->(int*float)list)->
  float IntMap.t * int list FloatMap.t->
  float IntMap.t = fun e (d, q) ->
  match FloatMap.min_binding q with
  | exception Not_found -> d
  | (w, us) -> dijkstra_aux e @@
    List.fold_left (fun (d, q) u ->
      if IntMap.find u d < w then (d, q)
      else List.fold_left (fun (d, q) (v, c)->
        if try IntMap.find v d <= w +. c
          with Not_found -> false then (d, q)
        else IntMap.add v (w +. c) d,
          FloatMap.add (w +. c)
            (v :: try FloatMap.find (w +. c) q
              with Not_found -> [])) q)
        (d, q) (e u))(d, FloatMap.remove w q) us
```

最短距離も Map で管理してみる (2/2)

```
let rec dijkstra :  
  (int -> (int * float) list) ->  
  int -> int -> float = fun e s ->  
    let d = dijkstra_aux e (IntMap.singleton s 0.,  
      FloatMap.singleton 0. [s]) in  
    fun v -> try IntMap.find v d  
              with Not_found -> infinity
```

- 破壊的代入が不要に
 - ・ 計算量は $O(E \log V)$ のまま
- 頂点数を与える必要がなくなった
- ファンクタを使えば辺の重みだけでなく、頂点の型も自由に選べる

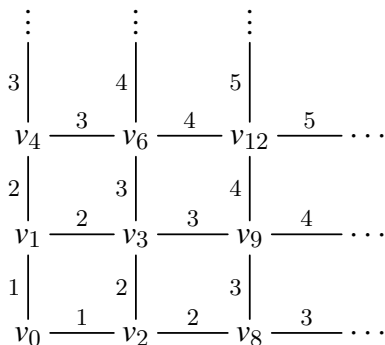
実際に試してみる (1/2)



```
# List.init 6 (dijkstra (function ... ) 0)
- : float list = [0.; 7.; 9.; 20.; 20.; 11.]
```

始点から到達可能な頂点が有限ならば，
有限時間で最短距離を返す

実際に使ってみる (2/2)



```
# dijkstra (function ... ) 0 12  
^CInterrupted.
```

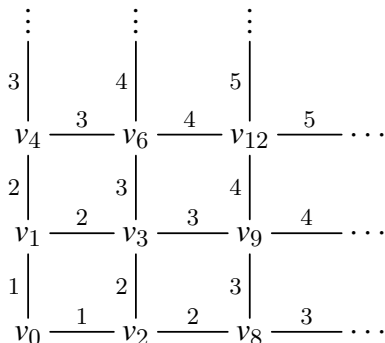
始点から到達可能な頂点が無限だと無理

改善策：計算の切り上げ

```
let rec dijkstra : (int ->(int*float)list)->
  int -> float IntMap.t * int list FloatMap.t->
  float = fun e t (d, q) ->
  match FloatMap.min_binding q with
  | exception Not_found -> infinity
  | (w, us) ->
    if List.exists (( = ) t) us then w
    else dijkstra e t @@
      List.fold_left (
        ...
      ) (d, FloatMap.remove w q) us

let dijkstra e s t = dijkstra e t
  (IntMap.singleton s 0.,
   FloatMap.singleton 0. [s])
```

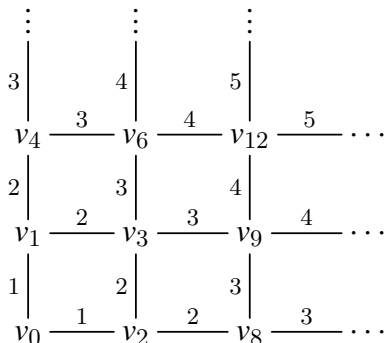
使用例 (1/2)



```
# dijkstra (function ... ) 0 12  
- : int = 10
```

終点までの距離が分かった時点で切り上げればよい

使用例 (2/2)



```
# List.init 10000 (dijkstra (function ... ) 0)  
^CInterrupted.
```

計算の途中経過が捨てられるため，とても遅い

OCaml は手続き型言語 (1/2)

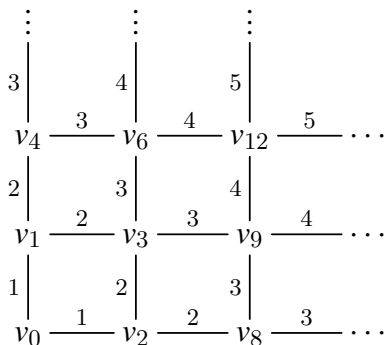
- 計算の途中経過を，クロージャの中に参照として覚えておく

```
let dijkstra e s =  
  let d = ref @@ IntMap.singleton s 0. in  
  let q = ref @@ FloatMap.singleton 0. [s] in  
  let rec dijkstra_aux t =  
    let ans = try IntMap.find t !d  
              with Not_found -> infinity in  
    match FloatMap.min_binding !q with  
    | exception Not_found -> ans  
    | (w, us) ->  
      if ans <= w then ans  
      else begin
```


OCaml は手続き型言語 (2/2)

```
q := FloatMap.remove w !q;  
List.iter (fun u ->  
  if w <= IntMap.find u !d then  
    List.iter (fun (v, c) ->  
      if  
        try w +. c < IntMap.find v !d  
        with Not_found -> true  
      then begin  
        d := IntMap.add v (w +. c) !d;  
        q := FloatMap.add (w +. c)  
          (v :: try FloatMap.find (w +. c) !q  
              with Not_found -> [])) !q  
      end) (e u)) us;  
dijkstra_aux t  
end in dijkstra_aux
```

解決！



```
# List.init 10000 (dijkstra (function ... ) 0)
- : float list =
[0.; 1.; 1.; 3.; 3.; 6.; 6.; 10.; 3.; 6.; ...]
```

計算の途中経過が再利用され，高速になった

まとめ

- OCaml により, 様々な好ましい性質を持つダイクストラ法の実装を与えた
 - ・ 疎なグラフに対して高速
 - ・ ある始点からの最短距離を一度に計算
 - ・ 無限グラフにも対応可

元ネタのソースコード : <https://github.com/fetburner/compelib/blob/master/graph/dijkstra.ml>

AtCoder での使用例 : <https://beta.atcoder.jp/contests/abc012/submissions/3198374>