Jacques Garrigue, 2006 年 1 月 18 日

12 NP 問題

これまで見たアルゴリズムは最悪でも $O(n^3)$ の計算量だった．ある k が存在し，計算量が $O(n^k)$ のアルゴリズムを多項式アルゴリズムという．計算量クラス P に属するともいう．しかし，多項式アルゴリズムが見付かっていない問題も多くある．その中で特に有名なのは，非決定性多項式アルゴリズムである (non-deterministice polynomial = NP)．すなわち，無限に並列性が使えるコンピュータでは多項式時間で実行できる．

巡回セールスマン問題

廻らなければ町のリストとその間の距離を与えられ，全ての町を一回だけ通りながら出発点に戻る最短経路を求める．通常の最短経路問題と違い，多項式アルゴリズムが知られていない．

```
let rec last l =
  match l with
    [] -> invalid_arg "last"
  | [a] -> a
  | _ :: l -> last l
val last : 'a list -> 'a

let rec search graph (visited : string list) next dist =
  if List.length visited = List.length graph then
    if next = last visited then [dist, List.rev visited]
    else []
  else if List.mem next visited then [] else
  let search' = search graph (next :: visited) in
  let nexts = List.assoc next graph in
  let sols = List.map (fun (next,d) -> search' next (dist+d)) nexts in
  List.flatten sols
val search : (string * (string * int) list) list ->
             string list -> string -> int -> (int * string list) list

let distances : (string * (string * int) list) list =
  [ "Nagoya", ["Sakae", 2; "Hisaya", 3; "Kanayama", 1];
    "Sakae", ["Nagoya", 2; "Hisaya", 1; "Imaike", 3; "Kanayama", 4];
    "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; "Kanayama", 11];
    "Kanayama", ["Nagoya", 1; "Sakae", 4] ]
val distances : (string * (string * int) list) list = ...
# search distances [] "Nagoya" 0;;
- : (int * string list) list =
[(22, ["Nagoya";"Sakae";"Hisaya";"Imaike";"Motoyama";"Daigaku";"Kanayama"]);
 (32, ["Nagoya";"Sakae";"Imaike";"Hisaya";"Motoyama";"Daigaku";"Kanayama"]);
 (23, ["Nagoya";"Hisaya";"Sakae";"Imaike";"Motoyama";"Daigaku";"Kanayama"]);
 (27, ["Nagoya";"Hisaya";"Imaike";"Motoyama";"Daigaku";"Kanayama";"Sakae"])]
```

search は，並列性と見做せる List.map と List.flatten の利用を計算から外せば，$O(n^2)$ で計算できる．もしもリストより効率のよいデータ構造に変えれば，$O(n \log n)$ でできる．しかし，

現在のコンピュータでは無限な配列実行はできないので，実際には指数的時間 $O(e^n)$ になる．こういうアルゴリズムを NP という．

3-SAT

3 項論理標準形の充足問題も NP 問題である．

3 項論理標準形は以下の形式をいう．

$$\bigwedge_{i=1}^{n}(L_{i1} \vee L_{i2} \vee L_{i3})$$

そこで L_{ij} は x または $\neg x$ という形である．

```
type formula = (string * bool) list list

let check (env : (string * bool) list) (f : formula) =
  List.for_all (List.exists (fun (x,b) -> List.assoc x env = b)) f
val check : (string * bool) list -> formula -> bool
let rec satisfy (vars : string list) env f =
  match vars with
    [] -> check env f
  | x :: rem ->
      let sat_x b = satisfy rem ((x,b)::env) f in
      sat_x true || sat_x false
val satisfy : string list -> (string * bool) list -> formula -> bool

let rec union accu l =
  match l with
    [] -> accu
  | a :: l -> if List.mem a accu then union accu l else union (a::accu) l
val union : 'a list -> 'a list -> 'a list
let sat3 f =
  let all_vars = List.flatten (List.map (List.map fst) f) in
  let vars = union [] all_vars in
  satisfy vars [] f
val sat3 : formula -> bool

# sat3 [["a",true; "b",false]; ["a",false; "b",false; "c", true];
        ["a",false; "c",false]; ["b",true; "a",true]];;
- : bool = true
# sat3 [["a",true; "b",false]; ["a",false; "b",false; "c", true];
        ["a",false; "c",false]; ["b",true]];;
- : bool = false
```

今度も，変数の数を m とすれば，check は $O(n)$ で，satisfy の中の||を並列に実行できれば，$O(m+mn)$ になるので，変数の数は最大でも $3n$ なので，全体が非決定性 $O(n^2)$ になる．すなわち NP である．

判定問題と翻訳による還元

3-SAT のように，問題の結果が真か偽であるものを判定問題という．

もしもある判定問題 A の入力を，結果を保ちながら，別の判定問題 B の入力に翻訳する機械的な方法があれば，A が B に還元されたという．

NP と NP 完全性

3-SAT に翻訳することによって計算できる問題は NP である．3-SAT をある問題に翻訳することが可能であれば，その問題は NP-Hard である．お互いに翻訳できる場合では NP 完全という．

(距離を気にしない) 巡回セールスマン問題を 3-SAT に還元する．各町の対 (i,j) に変数 x_{ij} を割り振る．もしも $x_{i,j}$ が真ならば，i を先に通る，偽ならば j を先に通るという意味で使う．1 に戻ったときを区別するために，それを $n+1$ とする．空の式か始めて，$j \neq 1$ なる全ての (i,j,k) について，$(\neg x_{ij} \vee \neg x_{jk} \vee x_{ik})$ を式に追加する．すなわち，i が k より先でなければ，x_{ij} か x_{jk} が偽でなければならない．さらに，全ての j について，(x_{1j}) および $(x_{j(n+1)})$ を式に追加する．すなわち，経路は 1 に始まり，1 に終る．そして，逆戻りを禁止する．全ての異なる (i,j) について，$(\neg x_{ij}, \neg x_{ji})$ を式に追加する．これだけやれば，経路が完全順序であることが分かる．今度は，各ステップが辺を通るという条件を導入する．まず辺でつながっている全ての (i,j) について (y_{ij}) を追加し，それ以外のものについて $(\neg y_{ij})$ を追加する．そして，全ての (i,j) に対して，$(\neg x_{ij}, y_{ij}, t_{ij2}, \ldots, t_{ij(n-1)})$ を追加する．この式は 4 個以上になるが，変数を増やせば 3 個のものに分割できる．t_{ijk} は i から k に直接行けて，さらに k から j は 1 ステップ以上で行けることを表している．そのために全ての (i,j,k) について $(\neg t_{ijk}, y_{ik}, x_{kj})$ を追加する．こういうように得られた論理式の充足可能性がちょうど巡回セールスマン問題の解の存在と同値である．

逆に 3-SAT を巡回セールスマン問題に還元できるので，NP 完全性が証明される．

NP に含まれない問題

NP は非決定性ながら多項式でなければならないので，当然 NP にも入らない問題もある．例えば，ある論理式について，変数の割り当てが何通りあるかという問題は，指数的な足し算が必要になるので NP ではない．

NP と P

NP に含まれる問題は全て SAT に翻訳可能である．しかし，SAT が P に含まれないという証明はいまだになされていない．もしも多項式アルゴリズムが見付かれば，$P = NP$ ということになる．逆に NP を並列なしに計算するのに指数的な時間が必要と分かれば，$P \neq NP$ になる．

実習課題 (12 回目)

1. `search` と同様に，`sat3` が各変数の真偽値を返すようにプログラムを変更せよ．
   `val sat3 : formula -> (string * bool) list`

2. `search` や `sat3` では，`map` の結果に対して `flatten` を使っている．同じ働きを効率よく行う関数 `flat_map` を定義せよ．
   `val flat_map : ('a -> 'b list) -> 'a list -> 'b list`

プログラミング課題

レポートを出した人に返事のメールを書きましたので，届いていなければ連絡を下さい．