Università di Perugia
Dipartimento di Matematica e Informatica

Artificial Intelligent Systems: Intelligen Application

Correzione Stringa a Stringa (Ricerca in ampiezza)

Professore
Prof. Stefano Marcugini

Studente
Tommaso Romani

Indice

1 Obiettivo

L'obiettivo del progetto è quello di fornire un implementazione basata sulla ricerca in ampiezza del seguente problema:

> Date due stringhe x e y, quale è, se esiste il numero minore di passi per ottenere la stringa y dalla stringa x utilizzando una sequenza di operazioni dei seguenti 2 tipi:
>
> 1. Cancellazione di Simboli
> 2. Scambio simboli adiacenti

Il tutto implementato del lignauggio di programmazione OCaml.
Il codice può essere trovato nella seguente repository di GitHub:Code Link

2 Natura del Problema

Questo problema risulta avere, data la sua natura, un branching factor molto elevato, in quanto, per ogni singolo livello, andranno generati $n-1$ figli per sottostringhe col caratteri adiacenti scambiati a due a due, e n figli per le sottostringhe con caratteri cancellate.

Data ad esempio la parola `CIAO`, espandendo la sua frontiera avremo:

4 Cancellazioni	**3 Scambi**
`CIA_`	`ICAO`
`CI_O`	`CAIO`
`C_AO`	`CIOA`
`_IAO`	

Questo comporta un costo esponenziale del tipo $O(b^{(2n-1)})$, con $(2n-1)$ come branching factor e b come profondità della soluzione.
Dalla seguente tabella possiamo vedere alcuni esempi di costo, in funzione della dimensione della parola e della profondità della soluzione:

n	b	Costo
4	4	16384
8	5	3.5×10^{10}
10	6	2.6×10^{16}

Questo costo rende impossibile risolvere problemi che coinvolgono stringhe lunghe che necessitano di molti scambi per essere corrette.

3 Struttura di Ricerca

Per rappresentare i vari nodi dell'albero di ricerca ho utilizzato una serie di liste, contenenti in ogni posizione i singoli elementi delle stringhe di input, così facendo è stato possibile controllare ad ogni profondità, se il prossimo nodo da espandere x fosse uguale al goal y, utilizzando così la possibilità di early goal test fornita dalla BFS.
Per facilitare il passaggio da rappresentazione interna a stringhe ho definito le seguenti funzioni:

```
(* Converte una stringa in una lista di caratteri *)
(* string -> char list*)
let string_to_list str = List.of_seq (String.to_seq str);;

(* Converte una lista di caratteri in una stringa *)
(* char list -> string*)
let list_to_string lst = String.of_seq (List.to_seq lst);;
```

4 Implementazione

Per analizzare come è stato implementato l'agloritmo prima analizzerò come vengono generati i nodi di frontiera, e successivamente la logica implementativa della ricerca in ampiezza.

4.1 Generazione della frontiera

Come già detto in precedenza la frontiera viene espansa a seguito di 2 possibili operazioni, la cancellazione di un carattere o lo scambio di due caratteri adiacenti.

La generazione dei nodi contenenti i risultati della prima operazione(**rimozione di caratteri**) è stata implementata come segue:

```
(* Genera tutte le stringhe possibili rimuovendo un carattere alla volta,
  generando quindi delle possibili stirghe che andranno aggiunte alla frontiera
  per essere espanse *)

(* int -> 'a list -> 'a list *)
let rec cancella_a len = function
    | [] -> []
    | h :: t -> if len = 0
                then t
                else h :: cancella_a (len-1) t

(* 'a list -> 'a list list *)
let cancellazioni lst =
  let rec aux acc len = function
    | [] -> acc
    | h :: t -> if len = 0
                then aux acc (len-1) t
                else aux ((cancella_a (len-1) lst) :: acc) (len-1) t
in aux [] (List.length lst) lst
```

Listing 1: Funzioni per la cancellazione di caratteri

Sono utilizzate una funzione principale `cancellazioni`, e una funzione di supporto `cancella_a`.
La prima prende una lista, su cui grazie ad un altra funzione ricorsiva di supporto interna `aux`, che aggiunge un accumulatore per salvare tutte le possibili cancellazioni, viene effettuato un pattern matching, e:

- se la lista è vuota allora `aux` viene valutata con l'accumulatore
- se la lista ha un `h` seguito da una lista `t`, allora controlla:

  - se `len = 0`, allora richiama se stessa per effettuare l'operazione di rimozione sull'ultimo elemento della lista di input `lst`
  - altrimenti richiama se stessa con `acc = cancella_a (len-1) lst` , `len = len -1` e il resto della lista `t`

La seconda funzione, `cancella_a` serve a cancellare un elemento dalla lista, data una specifica posizione.
Viene sempre effettuato un pattern matching che controllando la lista:

- se se la lista è vuota, viene valutato con la lista vuota
- altrimenti controlla len e:
  - se `len = 0` ritorna la coda della lista, cancellando il primo elemento
  - altrimenti richiama se stessa diminuendo `len` di 1, cercando così l'elemento da eliminare.

Quindi riassumendo scorre la lista utilizzando len, e per ogni possibile posizione aggiunge all'accumulatore la lista originale con carattere cancellato in quella posizione.

La generazione dei nodi contenenti i risultati dello **scambio** vengono generati nel seguente modo:

```
(* Genera tutte le stringhe possibili scambiando coppie di caratteri adiacenti,
  generando quindi delle possibili stirghe che andranno aggiunte alla frontiera
  per essere espanse *)

(* int -> 'a list -> 'a list *)
let rec scambia_a len = function
    | [] | [_] -> []
    | a :: b :: t -> if len = 0
                     then b :: a :: t
                     else a :: scambia_a (len-1) (b :: t)

(* 'a list -> 'a list list *)
let scambi lst =
  let rec aux acc len = function
    | [] | [_] -> acc
    | h :: t -> if len = 0
                then aux acc (len-1) t
                else aux ((scambia_a (len-1) lst) :: acc) (len-1) t
  in
  aux [] (List.length lst -1) lst
```

Listing 2: Funzioni per lo scambio di caratteri adiacenti

Come per le precedenti funzioni indicate in 1 la logica delle operazioni è simile, con alcuni adattamenti:

- la condizione terminale del matching di `scambi` non controlla solo liste vuote `[]`, ma anche liste con un solo carattere `[_]`.
- la condizione terminale in `scambia_a` è analoga a quella in `scambi`, inoltre il secondo case del matching controlla 2 elementi della lista `a::b::t` e:
  - se la lunghezza `len = 0` allora ha raggiunto la posizione da scambiare, e quindi viene valutato a `b::a::t`.
  - altrimenti, come per 1 richiama se stessa decrementando `len` per cercare la posizione su cui effettuare lo scambio, e passa alla chiamata ricorsiva la lista `b::t`.

4.2 Ricerca in Ampiezza

La ricerca in ampiezza viene gestita da due funzioni, la prima che va a richiamare le funzioni 1 e 2 per espandere la frontiera, mentre la seconda implementa la logica di esplorazione.
Il loro funzionamento è il seguente.

```
(* Espande la frontiera con tutte le possibili stringhe generate *)
(* 'a list list -> 'a list -> 'a list list -> 'a list list *)
let espandi frontiera x visti =
  List.fold_left (fun acc next ->
                    if List.mem next visti
                    then acc
                    else next :: acc)
  frontiera (cancellazioni x @ scambi x);;
```

Listing 3: Funzione che espande la frontiera

In questa funzione viene utilizzato la funzione `fold_left` del modulo `List`. Questa funzione è una funzione di ordine superiore che prende 3 argomenti in input: una funzione `f`, un accumulatore `acc` e una lista, e funziona applicando la funzione f ricorsivamente, prima al primo elemento e all'accumulatore, poi applica f al risultato e al secondo elemento e così via.
Nel nostro caso la funzione usata è una funzione anonima che controlla se, data la nostra attuale frontiera, quando andiamo ad aggiungere i nuovi nodi generati dalla concatenazione dei risultati di `cancellazioni` e `scambi`, vengono aggiunti nodi già visitati, e in caso positivo non li aggiunge, evitando così alla funzione principale che ora descriveremo di espandere più volte lo stesso nodo.

La ricerca viene gestita dalla segente funzione:

```
(* BFS per trovare il numero minimo di passi per trasformare x in y *)
(* char list -> char list -> int *)
let correggi_stringa x y =
  let rec bfs frontiera prossima_frontiera visti passi espansi =
  match frontiera with
  | [] ->
    (match prossima_frontiera with
     | [] ->
       Printf.printf "Trasformazione non trovata\n";
       Printf.printf "Numero di nodi espansi: %d\n" espansi;
       raise NotFound
     | _ -> bfs prossima_frontiera [] visti (passi + 1) espansi)
  | h :: t when h = y -> passi
  | h :: t ->
    let nuovi_visti = h :: visti in
    let nuova_frontiera = espandi prossima_frontiera h nuovi_visti in
    bfs t nuova_frontiera nuovi_visti passi (espansi + List.length nuova_frontiera)
  in bfs [x] [] [] 0 1;;
```

Listing 4: Implementazione della ricerca in ampiezza

La funzione `correggi_stringa` prende le due stringhe x e y convertite a liste di `char`, e poi utilizza la helper function ricorsiva `bfs` per effettuare la ricerca in ampiezza nel seguente modo:

- Come primo match controlla se la frontiera è vuota, che in caso positivo inizia un altro match in cui prova a vedere se la frontiera alla prossima iterazione è vuota, quindi serve a far partire l'algoritmo allo step iniziale, quando ancora non ci sarà la frontiera, e in caso positivo significa che è stato esaurito lo spazio di ricerca, quindi non è possibile convertire x in y.
- altrimenti controlla se il valore che sta attualmente analizzando nella frontiera è uguale a y e in caso positivo si ferma.
- infine chiama la funzione vista in precedenza `espandi` e genera cosi la nuova frontiera

5 Test dell'Algoritmo

Di seguito viene riportata una tabella contenente alcuni test per mostrare il funzionamento dell'algoritmo.

x	**y**	**n. operazioni**	**nodi espansi**
ioca	ciao	3	2361
iocaacc	ciao	6	241858509
macmeoll	cammello	5	447995265
maaclomelee	cammello	non termina	non termina