**Esercizi proposti – 10**

In questo gruppo di esercizi assumiamo, dove non sia specificato diversamente, di rappresentare i grafi mediante liste di archi, con il tipo di dati così dichiarato:

```
type 'a graph = ('a * 'a) list
```

Quando si risolve un esercizio, è ovviamente consentito utilizzare funzioni già definite in esercizi precedenti, e non se ne deve ripetere la definizione (nemmeno se la funzione utilizzata è la soluzione di un esercizio di un gruppo precedente).

---

**Suggerimento generale:** Quando affrontate un problema sui grafi, chiedetevi se l'algoritmo migliore da adottare per risolvere il problema sia quello di visita (in ampiezza o profondità) o quello per la ricerca di cammini. Gli algoritmi di visita vanno bene ogni volta che non interessa quale strada si fa. Non vanno bene quando si vuole in uscita il cammino o si deve in qualche modo tener conto di caratteristiche del cammino (ad esempio la lunghezza).

Quando si può adottare un algoritmo di visita, conviene adattarne il codice al problema specifico, dato che è più semplice del codice per la ricerca di cammini.

Ricordate poi che, nella maggior parte dei casi, la prima cosa da fare è definire una funzione che, dato un grafo e un nodo, riporti tutti i successori del nodo (o i suoi vicini, se si tratta di un grafo non orientato).

---

1. Scrivere una funzione `test_connessi: 'a graph -> 'a -> 'a -> bool` che, dato un grafo orientato G e due nodi N e M, determini se esiste un cammino da N a M. La funzione riporterà un booleano (non un cammino).

2. Scrivere una funzione `esiste_ciclo: 'a graph -> 'a -> bool` che, dato un grafo orientato G e un nodo N, determini se esiste un ciclo su N (cioè un cammino da N a N che contenga almeno un arco). La funzione riporterà un booleano (non un cammino).

3. Scrivere una funzione `ciclo: 'a graph -> 'a -> 'a list` che, dato un grafo orientato G e un nodo N, riporti, se esiste, un ciclo su N, altrimenti sollevi un'eccezione (in questo caso la funzione deve riportare una lista di nodi).

4. Un grafo non orientato è connesso se, per ogni coppia di nodi distinti N e M, esiste un cammino da N a M. Si definisca un tipo di dati `'a graph` (diverso da quello dato all'inizio di questo gruppo di esercizi) per la rappresentazione di grafi mediante due componenti: lista di nodi e lista di archi, e scrivere una funzione `grafo_connesso: 'a graph -> bool` che, dato un grafo non orientato G, determini se G è connesso.

   Si noti che, per controllare se un grafo non orientato con nodi `[n1;n2;....nk]` è connesso, basta controllare se `n1` è connesso a `n2`, poi se `n2` è connesso a `n3` (quindi `n1` sarà connesso anche a `n3`), poi se `n3` è connesso a `n4`, ecc. Oppure, equivalentemente, si può controllare se `n1` è connesso a `n2`, a `n3`, a `n4`,... e basta.

5. Si consideri il problema dei missionari e cannibali dell'esercizio 4 del Gruppo 7. La soluzione al problema si può ridurre a un problema di ricerca di un cammino a partire dal nodo che rappresenta la situazione iniziale (`initial`) nel grafo la cui funzione "successori" è la funzione `from_sit`.

   Nella ricerca, il test "nodo già visitato" non dovrebbe utilizzare `List.mem`, in quanto l'ordine degli elementi nelle due liste di una situazione deve essere ignorato: ad esempio, `([Miss;Cann;Barca],[Cann;Cann;Miss;Miss])` e `([Cann;Barca;Miss],[Miss;Cann;Miss;Cann])` rappresentano la stessa situazione.

   Con tali modifiche, risolvere il problema dei missionari e cannibali, scrivendo una funzione `goal: situazione -> bool`, che determina se una situazione è l'obiettivo, e una funzione `miss_cann: unit -> situazione list` che riporti una lista delle situazioni rappresentante una possibile soluzione al problema.

   Se si vuole invece avere come risultato non una lista di situazioni, ma una lista di azioni, occorre modificare la funzione `from_sit`, in modo che, applicata a una situazione `sit`, riporti una `(azione * situazione) list` (tutte le coppie `(a,s)` dove `a` è un'azione applicabile a `sit` e `s` la situazione che ne risulta). Con questa modifica, adattare il codice di `miss_cann` in modo che venga riportata una `azione list`. Suggerimento: trattare a parte la situazione iniziale, e far sì che la funzione ausiliaria per la ricerca a partire da una singola situazione abbia come argomento (oltre alla lista delle situazioni già "visitate") una coppia `(act,sit)`, di tipo `azione * situazione`, dove `act` è l'azione che ha portato alla situazione `sit`.

6. (Dal compito d'esame di febbraio 2009). Sia data la seguente definizione di tipo per rappresentare grafi orientati:

   ```
   type 'a graph = 'a list * ('a * 'a) list
   ```

   In altre parole, un grafo (orientato) è rappresentato da una coppia: il primo elemento è una lista che rappresenta l'insieme dei nodi del grafo, il secondo elemento è una lista di coppie, ciascuna delle quali rappresenta un arco del grafo. Si assume che la lista dei nodi sia senza ripetizioni.

   (a) Scrivere una funzione `cammino: 'a graph -> 'a list -> 'a -> 'a -> 'a list` che, dato un grafo G, una lista L senza ripetizioni e due nodi $n$ e $m$ di G, riporti, se esiste, un cammino da $n$ a $m$ che passi solo per nodi contenuti in L e per ciascuno di essi esattamente una volta. Se un tale cammino non esiste, la funzione solleverà un'eccezione.

   Suggerimento: adattare l'algoritmo di ricerca di un cammino in modo tale che dalla lista L vengano via via eliminati i nodi già incontrati. Si noti che in tal modo non è necessario memorizzare i nodi già visitati, dato che la lista L stessa serve ad evitare i cicli.

   (b) Un ciclo in un grafo è detto *hamiltoniano* se esso tocca tutti i nodi del grafo esattamente una volta (eccetto il primo e l'ultimo nodo, che sono, evidentemente, uguali). Scrivere una funzione `hamiltoniano: 'a graph -> 'a list` che, dato un grafo orientato $G$, determini se in $G$ esiste un ciclo hamiltoniano e riporti un tale ciclo, se esiste, un errore altrimenti.

Si ricordi che un cammino ciclico è una sequenza di nodi $n_1, n_2, ..., n_k$, con $k > 1$ e $n_k = n_1$, tale che, per ogni $i = 1, ..., k-1$, esiste un arco da $n_i$ a $n_{i+1}$.

Si noti che, se $x$ e $y$ sono due nodi di un grafo, esiste un ciclo hamiltoniano su $x$ se e solo se esiste un ciclo hamiltoniano su $y$. Quindi per controllare se in un grafo c'è un ciclo hamiltoniano basta prendere un nodo qualsiasi $x$ e vedere se esiste un ciclo su $x$ che passa esattamente una volta per tutti i nodi del grafo.

7. (Dal compito d'esame di luglio 2009). Si considerino le seguenti dichiarazioni di tipo, per la rappresentazione di colori e associazioni di colori:

   ```
   type col = Rosso | Giallo | Verde | Blu
   type 'a col_assoc = (col * 'a list) list
   ```

   Scrivere un programma con una funzione `colori_alterni: 'a graph -> 'a col_assoc -> 'a -> 'a -> 'a list` che, dato un grafo orientato, un'associazione di colori e due nodi del grafo `start` e `goal`, riporti – se esiste – un cammino a colori alterni da `start` a `goal`, nel grafo dato. Se un tale cammino non esiste, verrà sollevata un'eccezione. (Si vedano gli esercizi 14 del Gruppo 8 e l'esercizio 12 del Gruppo 11, che propongono lo stesso problema per gli alberi binari ed n-ari).

8. (Dal compito d'esame di settembre 2009). Scrivere un programma con una funzione `connessi_in_glist: 'a graph list -> 'a -> 'a -> bool` che, data una lista di grafi orientati [$G_1$,$G_2$,...,$G_n$] e due elementi `b` e `c`, determini se `b` e `c` sono nodi diversi tali che, per qualche grafo $G_i$ appartenente alla lista [$G_1$,$G_2$,...,$G_n$], esiste un cammino da `b` a `c` o viceversa (da `c` a `b`).

9. (Dal compito d'esame di febbraio 2010). Scrivere un programma con una funzione `cammino_con_nodi: 'a graph -> 'a -> 'a list -> 'a list` che, dato un grafo orientato $G$, un nodo $N$ di $G$ e una lista $L$ senza ripetizioni, restituisca, se esiste, un cammino senza cicli che, partendo da $N$, contenga tutti i nodi di $L$ (in qualsiasi ordine) ed eventualmente anche altri nodi. Se un tale cammino non esiste, il programma solleverà un'eccezione.

   Ad esempio, se $G$ è il grafo rappresentato da `[(1, 2); (1, 3); (1, 4); (2, 6); (3, 5); (4, 6); (6, 5); (6, 7); (5, 4)]` e $L$ è la lista `[2; 5]`, la ricerca a partire dal nodo 1 restituirà il cammino `[1; 2; 6; 5]`. . Se la lista $L$ è `[2; 6; 3]`, la ricerca a partire dal nodo 1 fallirà perché non esistono cammini in $G$ che a partire da 1 tocchino tutti i nodi di $L$.

10. (Dal compito d'esame di giugno 2010). La prima parte del compito è stata proposta nell'esercizio 3 del Gruppo 7. Qui proseguiamo:

    (a) Definire una funzione `nodi: int -> cassaforte list` che, applicata a un intero $N$ riporti la lista con tutte le possibili configurazioni di una cassaforte con $N$ chiavi (si veda l'esercizio 1k del Gruppo 6).

    (b) Definire una funzione `archi: int -> (cassaforte * cassaforte list) list`, che, applicata a $N$ riporta la lista con tutte le coppie `(clist,successori clist)`, per ogni possibile configurazione `clist` di una cassaforte con $N$ chiavi.

(c) Definire una funzione `start: int -> cassaforte` che costruisce la configurazione iniziale di una cassaforte con $N$ chiavi, in cui tutte sono chiuse, e una funzione `aperta: cassaforte -> bool`, che determina se tutte le chiavi della cassaforte sono aperte.

(d) Il problema dell'apertura di una cassaforte con $N$ chiavi si può ridurre al problema della **ricerca di un cammino nel grafo** $\langle V, A\rangle$, dove $V$=`nodi(`$N$`)` e $A$=`archi(`$N$`)`, a partire dalla configurazione iniziale in cui tutte le $N$ chiavi sono chiuse (`start` $N$), fino alla configurazione in cui le $N$ chiavi sono tutte aperte.

Definire una funzione `apri: int -> cassaforte list`, che risolva in questo modo il problema dell'apertura di una cassaforte con $N$ chiavi.

11. (Dal compito d'esame di settembre 2010). Scrivere una funzione

```
cammino_di_primi: int graph -> int-> int -> int list
```

che, applicata a un grafo (orientato) di interi $g$ e interi *start* e *goal* riporti, se esiste, un cammino in $g$ da *start* a *goal* costituito soltanto da numeri primi.

12. (Dal compito d'esame di febbraio 2011). Le formule proposizionali costruite utilizzando soltanto gli operatori $\neg$, $\wedge$ e $\vee$ si possono rappresentare mediante il tipo così definito:

```
type form = Prop of string | Not of form
        | And of form * form | Or of form * form
```

Se `g` è un `form graph` (cioè un grafo i cui nodi sono etichettati da formule del tipo `form` sopra definito), un cammino in `g` si dice contraddittorio se contiene una formula e la sua negazione.

Scrivere una funzione `non_contradictory_path: form graph -> form -> form -> form list` che, dato un grafo (orientato) di formule e due formule `start` e `goal` riporti, se esiste, un cammino *non* contraddittorio da `start` a `goal` nel grafo dato.

Si noti che se $L$ è una lista di formule non contraddittorie, l'aggiunta di una formula $F$ a $L$ produce una lista contraddittoria se e solo se vale uno di questi due casi:

- $L$ contiene $\neg F$;
- $F$ ha la forma $\neg G$ e $L$ contiene $G$.

13. (Dal compito d'esame di giugno 2011). Definire una funzione `path_n_p: 'a graph -> ('a -> bool) -> int -> 'a -> 'a list`, che, applicata a un grafo orientato `g`, un predicato `p: 'a -> bool`, un intero *non negativo* `n` e un nodo `start`, riporti, se esiste, un cammino non ciclico da `start` fino a un nodo `x` che soddisfa `p` e che contenga esattamente `n` nodi che soddisfano `p` (incluso `x`). La funzione solleverà un'eccezione se un tale cammino non esiste.

Ad esempio, sia `pari: int -> bool` definito in modo tale che `pari x` è vero se e solo se `x` è pari e `g = [(1,3);(2,6);(3,4);(3,5);(3,6);(4,2);(4,5);(5,4);(6,5)]`. Allora, `path_n_p g pari 2 1` può avere uno dei valori seguenti: `[1;3;4;2]`, `[1;3;5;4;2]`, `[1;3;6;5;4]`.