Programmazione Funzionale

Esercitazione 15 – Grafi: funzioni di base, ricerca di cammini, connettività e ciclicità

In questi esercizi assumiamo di aver definito i grafi come lista di archi;

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

In questo contesto un grafo e anche un lista associativa.

Esercizio 1. Definiamo funzioni di base su un grafo.

1. Definire `add: ('a * 'a) -> 'a graph -> 'a graph` che aggiunge un arco a un grafo.
2. Definire `remove : 'a -> 'a graph -> 'a graph` che toglie un nodo da un grafo – quindi toglie tutti i archi che contengono quel nodo.
3. Definire `transpose : 'a graph -> 'a graph` che inverte tutti i archi di un grafo.
4. Definire una funzione `makeUndirected : 'a graph -> 'a graph` che rende tutti i archi del grafo simmetrici. Cioè se l'arco $(1, 2)$ occorre nel grafo g allora i archi $(1, 2)$ e $(2, 1)$ occorrono in `makeUndirected g`.

   Potete usare;

   - la funzione `transpose`.
   - una funzione `setadd : 'a -> 'a list -> 'a list` che aggiunge a una lista `lst` un elemento `x` se e solo se `x` non occorre in `lst`.

Esercizio 2. Vogliamo definire una funzione che restituisce la lista dei nodi di un grafo.

1. Definire `nodes : 'a graph -> 'a list` una funzione che restituisce la lista – senza ripetizioni – dei nodi di un grafo.
2. Definire un funzione `leftelems : ('a * 'b ) list -> 'a list` che prende una lista di coppie e restituisce la lista dei elementi a sinistra di queste coppie.

   Ad esempio `leftelems [(2,3) ; (1,5) ; (3,3)]` restituisce `[2;1;3]`.
3. Definire un funzione `rightelems : ('a * 'b ) list -> 'a list` che prende una lista di coppie e restituisce la lista dei elementi a destra di queste coppie.

   Ad esempio `rightelems [(2,3) ; (1,5) ; (3,3)]` restituisce `[3;5;3]`.
4. Definire una funzione `list_to_set : 'a list -> 'a list` che prende un lista `lst` e restituisce `lst` in cui ogni elemento di `lst` occorre un unica volta.
5. Mediante le funzione `leftelems`,`rightelems`, e `list_to_set` definire una funzione `getnodes : 'a graph -> 'a list` che prende un grafo e restituisce l'insieme dei nodi del grafo.

Esercizio 3. Vogliamo definire diversi algoritmi di ricerca di un cammino in un grafo.

1. Definire una funzione `search : 'a graph -> 'a -> 'a -> 'a list` che prende un grafo g due elementi `start` e `target` e restituisce un cammino sotto forma di lista di tipo `'a list` nel grafo dal nodo `start` al nodo `target`.

   Se nessun cammino e trovato la funzione solleva un eccezione.
2. Modificare la funzione `search` per definire la funzione `searchCond: int graph -> int -> int -> int -> int list` che prende un grafo di interi `grafo` due elementi `start` e `target` e un intero `value` e restituisce un cammino sotto la forma `int list` da `start` a `value` tale che la somma dei interi della lista vale `value`

   Se nessun cammino e trovato la funzione solleva un eccezione.

   Si puo usare altri funzioni:

   - Una funzione `accept : int list -> int -> bool` che prende una lista e un interi `value` che restituisce `true` quando la somma dei interi vale `value`.
   - Una funzione `reject : int list -> int -> bool`.
   - Definire la funzione `searchCond` con le funzioni `reject` e `accept` mediante backtracking.

Esercizio 4. Usando la funzione che ricerca un cammino in un grafo, vogliamo definire una funzione che verifica se un grafo e ciclico.

1. Definire una funzione `search : 'a graph -> 'a -> 'a -> 'a list` che prende un grafo g due elementi `start` e `target` e restituisce un cammino sotto forma di lista di tipo `'a list` nel grafo dal nodo `start` al nodo `target`.

   Se nessun cammino e trovato la funzione solleva un eccezione.

2. Modificare la funzione `search` per definire una funzione `cycleat : 'a graph -> 'a -> bool` che prende un grafo e un elemento `start` e restituisce `true` se trova un cammino da `start`.
3. Usando la funzione `cycleat` definire una funzione `cycle : 'a graph -> bool` che restituisce `true` se e solo se il grafo contiene un ciclo.

Esercizio 5. Vogliamo definire una funzione che verificà la connetività di un grafo.

Ricordiamo che un grafo e connesso quando per ogni nodo x e y esiste un cammino da x a y. N.B. La funzione che definiremo in questo esercizio non fa una ricerca esplicita di tutti i cammini possibili usando backtracking.

1. Definire una funzione `sons : 'a graph -> 'a -> 'a list` che prende un grafo e un elemento `x` e restituisce la lista dei elementi `a` tale che `(x,a)` e un arco del grafo.
2. Usando la funzione `sons`, definire la funzione `reach : 'a graph -> 'a -> 'a list` che prende un grafo e un elemento `x` e restituisce la lista dei elementi che sono raggiungibili da `x`.

   Attenzione a non creare un loop infinito.
3. Definire la funzione `nodes : 'a graph -> 'a list` che prende un grafo e restituisce la lista dei suoi nodi.
4. Un nodo x in un grafo e *conesso* quando e collegato a tutti i altri nodi del grafo.

   Mediande le funzioni `reach` e `nodes` definire una funzione `connectedNode : 'a -> 'a graph -> bool` che restituisce `true` se l'elemento in entrata e ben collegato nel grafo, altrimenti la fuzione restituisce `false`.
5. Mediande le funzioni `connectedNode` e `nodes`, definire una funzione `connected : 'a graph -> bool` che restituisce `true` se e solo se il grafo in entrata e connesso.

Esercizio 6. Usiamo l'esercizio precedente. Vogliamo definire una altra funzione che verificà la connetività di un grafo.

1. Definire una funzione `undirected: 'a graph -> bool` che restituisce `true` se e solo se il grafo in entrata non e ordinato. Ricordiamo che questo significa che quando un arco `(a,b)` occorre nel grafo allora l'arco `(b,a)` occorre anche.
2. Definire une funzione `connect : 'a -> 'a graph -> bool` che prende un grafo e un nodo `a` e restituisce `true` se e solo se il grafo non e ordinato e l'elemento `a` e connesso.
3. Per qualsiasi grafo `grafo` e ogni nodo `x` del grafo le funzioni `connect x grafo` e `connected grafo` (esercizio precedente) restituiscono lo stesso valore.

   Argomentare perché è cosi, cioè perché `connect x grafo` verificà la connettività del grafo.