Commento al Laboratorio n. 4

Esercizio n.1: **Vertex cover**

Si crea una `struct arco` con 2 campi interi `u` e `v` per memorizzare un arco tramite i 2 vertici su cui esso insiste e un vettore di archi `a` di `E` elementi.

Il problema richiede di generare tutti i sottoinsiemi degli `N` vertici e di verificare se tutti gli archi del grafo hanno almeno uno dei vertici su cui insistono nel sottoinsieme corrente. Il modello per generare tutti i sottoinsiemi è quello del powerset, l'implementazione con divide et impera, disposizioni ripetute o combinazioni semplici è indifferente. Se si fosse dovuto risolvere un problema di ottimizzazione, ad esempio ritornare il vertex cover di cardinalità minima, allora l'implementazione del powerset con combinazioni semplici sarebbe stata conveniente in quanto, grazie all'iterazione che fa crescere la dimensione della combinazione corrente, ci si sarebbe potuti fermare alla prima soluzione valida, che era per costruzione anche a cardinalità minima.

Visto che le scelte sono i vertici e visto che i vertici sono identificati da interi nell'intervallo 0...`N`-1, non è necessario registrare le scelte in un vettore `val`. Si presentano 2 soluzioni:

1. powerset costruito con le disposizioni ripetute: la soluzione `sol` è un vettore di N elementi, ciascuno dei quali indica se il vertice corrispondente all'indice fa o no parte della soluzione. Nella condizione di terminazione si chiama una funzione `check` per validare la soluzione corrente: essa è scartata se e solo se per almeno uno degli archi del grafo entrambi i vertici non appartengono al sottoinsieme corrente
2. powerset costruito con le combinazioni semplici: nel main si itera la chiamata alla funzione delle combinazioni semplici con dimensione `k` crescente da 1 a `N`. La soluzione `sol` è un vettore di `k` elementi, ciascuno dei quali è un vertice della soluzione. Una funzione `check` verifica se per ogni arco esiste nella soluzione corrente almeno un vertice su cui l'arco insiste.

Esercizio n.2: **Anagrafica con liste**

Menu: come nell'es. 2 del Lab. 1, invece di utilizzare un tipo definito per enumerazione per i comandi, nella soluzione proposta si utilizzano esplicitamente gli interi come indici di un vettore di stringhe che li contiene.

Dati: si presentano 2 soluzioni per il tipo `Item`:

1. versione con `Item` con dati composti per valore in cui i campi stringa sono vettori di caratteri sovrallocati. Per gestire i casi particolari si introduce un `Item` vuoto (caratterizzato da codice vuoto), creato dalla funzione `ItemSetVoid` e riconosciuto dalla funzione `ItemCheckVoid`. La funzione `leggiItem` chiama al suo interno la `scomponiData` per trasformare la data da stringa a `struct` con campi interi peri giorno, mese ed anno. Tutte le funzioni che ricevono o ritornano Item lo passano per valore, facendo sempre quindi copie dei dati. Alcune funzioni, ad esempio quella di stampa, potrebbero in alternativa ricevere puntatori ad `Item`.
2. versione con `Item` con dati composti per riferimento in cui i campi stringa sono vettori di caratteri allocati dinamicamente. All'item si accede unicamente tramite puntatore. La funzione `ItemNew` alloca un nuovo item e lo inizializza con i dati passati come parametri. La funzione `leggiItem` alloca un item mediante `ItemNew` e ne trasferisce il possesso al programma chiamante, che si occupa di deallocarlo quando non più necessario mediante

la `ItemFree`. Non è più necessario gestire con `ItemSetVoid` e `ItemCheckVoid` il caso di item vuoto, in quanto basta usare puntatori `NULL`.

Vista la semplicità dell'item non si introducono funzioni di accesso o confronto sulle chiavi.

Collezione di dati: il nodo della lista è definito secondo la modalità 3 (*Puntatori e strutture dati dinamiche 4.1.1*). La creazione di un nuovo nodo è fatta come in *Puntatori e strutture dati dinamiche 4.1.3*. La funzione `insertOrdinato` (*Puntatori e strutture dati dinamiche 4.1.3*) trasferisce alla lista la proprietà dell'item ricevuto. La ricerca `ricercaCodice`, essendo per codice mentre l'ordinamento è per data, è una ricerca su lista non ordinata e quindi non può sfruttare l'ordinamento per un'interruzione anticipata in caso negativo. Le funzioni di eliminazione `elimina e eliminaTraDate` ritornano l'item al programma chiamante, cui viene trasferita la proprietà e la responsabilità di deallocazione.

Esercizio n. 3: **Collane e pietre preziose**

Trattasi di problema di ottimizzazione. Una volta letti i dati (numero di zaffiri, rubini, topazi e smeraldi) è calcolabile la lunghezza massima della collana `maxlun`. Il `main`, mediante un ciclo, esplora i problemi di lunghezza `k` crescente tra 1 e `maxlun` e registra in `bestlun` il massimo valore di `k` per cui si è trovata una soluzione accettabile. Questo soddisfa la richiesta di trovare una soluzione ottima, quindi collana a lunghezza massima. Il `main` opera iterativamente su `numtestset` problemi: letta da un file di ingresso la quaterna che rappresenta il problema corrente, calcola la lunghezza massima possibile della collana e poi per tutte le lunghezze `k` tra 1 e la massima risolve il problema. Si ipotizza per il file in ingresso un formato con la prima riga che contiene `numtestset` (numero di problemi, cioè di quaterne), seguita dalle quaterne che descrivono ciascun problema. Vengono proposti diversi file di prova di difficoltà variabile.

Il modello del Calcolo Combinatorio è quello delle disposizioni ripetute di `N` oggetti presi a `k` a `k`. Si presentano 4 soluzioni: da un file di ingresso

1. versione 0: la verifica dell'accettabilità di una soluzione di lunghezza `k` è fatta nella condizione di terminazione. La funzione `check`:
   - calcola in `usGemme` il numero di occorrenze di ciascuna gemma nella soluzione corrente. Se tale numero eccede la disponibilità registrata nel vettore `numGemme`, la soluzione è scartata
   - verifica le regole di composizione: scorrendo la soluzione `sol`, in base alla gemma scelta in posizione `i`-1 si verifica che quella in posizione `i` sia conforme alla regola, altrimenti si scarta la soluzione.

   Non essendo prevista alcuna forma di pruning, questa soluzione è accettabile solo per lunghezze massime di collane molto piccole
2. versione 1: si introduce una prima forma di pruning: nella condizione di terminazione si verificano solo le regole di composizione, mentre la discesa ricorsiva è subordinata alla verifica della disponibilità di gemme. Sperimentalmente si osserva un discreto miglioramento nella capacità di trattare in tempi ragionevoli lunghezze massime maggiori
3. versione 2: la condizione di terminazione non prevede verifica di accettabilità, in quanto anche la verifica di regole di composizione è usata per condizionare la ricerca ricorsiva. Sperimentalmente si verifica la capacità di trattare in tempi ragionevoli lunghezze massime notevoli.

La versione 2 viene modificata a livello di `main` nel ciclo che itera sulle catene:

- versione 3: il ciclo avviene per lunghezze decrescenti delle catene, nell'ipotesi di interromperlo non appena giunti ad una soluzione all'iterazione con lunghezza `k`, in quanto le iterazioni successive possono portare solo a lunghezze minori
- versione 4: si seleziona `k` in maniera dicotomica (a metà della catena). Se si trova una soluzione di lunghezza `k` si procede per lunghezze da `k`+1 a `N`, altrimenti per lunghezze da 1 a `k`-1.

Esercizio n. 4: Collane e pietre preziose (versione 2)

Trattasi di problema di ottimizzazione dove si chiede di massimizzare il valore della collana nel rispetto delle regole di composizione. Si segue la strategia dell'esercizio precedente con il `main` che opera iterativamente su `numtestset` problemi, acquisendo per ciascuno da un file di ingresso i dati sul numero di gemme, sul loro valore e sul numero massimo di ripetizioni consecutive. Si ipotizza per il file in ingresso un formato con la prima riga che contiene `numtestset` (numero di problemi), seguita dalle n-uple di 9 dati che descrivono ciascun problema. Una volta letto il numero di zaffiri, rubini, topazi e smeraldi è calcolabile la lunghezza massima della collana `maxlun`.

La funzione wrapper `solve` alloca le strutture dati per la funzione ricorsiva di risoluzione: i vettori `sol` e `bestSol` di significato evidente, il vettore `usGemme` per tener traccia del numero di gemme di ogni tipo usate nella soluzione corrente, il vettore `ripGemme` per tener conto del numero di ripetizioni consecutive di una gemma nella soluzione corrente, gli interi passati per riferimento `bestval` e `bestlun` per tener traccia del valore e della lunghezza migliore stimati, l'intero `prec` per ricordare la gemma decisa al passo precedente di ricorsione.

Il modello del Calcolo Combinatorio è anche in questo esercizio quello delle disposizioni ripetute di `N` oggetti presi a `k` a `k`. La disponibilità di gemme, i valori consecutivi ripetuti e le regole di composizione sono utilizzate per condizionare la discesa ricorsiva. Il vincolo su zaffiri e smeraldi è invece verificato nella condizione di terminazione per non precludere l'esplorazione di tutto lo spazio utile. Il vettore `ripGemme` serve per registrare per ogni gemma il numero di occorrenze consecutive. Esso viene assegnato in fase di decisione su di una gemma e ripristinato nella configurazione precedente in fase di backtrack.