# *Commento all'es. 1*

**Dati:**

- **quasi ADT `Item`:** serve a memorizzare i dati relativi a ciascuna sala. Su di esso si definiscono funzioni per estrarre ciascuna delle 3 chiavi (nome della sala, numero di monete e punti-ferita) e per creare un nuovo item (cfr "Puntatori e strutture dati dinamiche, 6.2.1, opzione 3 pag. 166)
- **ADT di I classe `ST`:** tabella di simboli per gli item implementata come vettore con ordinato con funzioni di inizializzazione (`STinit`), inserzione di item (`STinsert`), ricerca per nome della stanza (`STsearch`), estrazione di ciascuna delle 3 chiavi (`STretrieveKey1, STretrieveKey2, STretrieveKey3`)
- **ADT di I classe `Graph`:** grafo non orientato e non pesato, implementato come matrice delle adiacenze con funzioni di lettura da file (`GRAPHread`), inizializzazione (`GRAPHinit`) e ricerca di cammino (`GRAPHpath`). La funzione di lettura legge i nvertici, poi i vertici (item), inserendoli nella tabella di simboli, ed infine gli archi, popolando la matrice delle adiacenze.

**Algoritmo:** la funzione `GRAPHpath` è un wrapper per risolvere un problema di ottimizzazione che consta nel calcolare il cammino da ingresso a uscita semplice e con massimo numero di monete compatibile con i punti-ferita. Ingresso e uscita sono nodi i cui indici sono noti a priori (0 e numero di vertici – 1). I nodi che compongono il cammino (corrente e ottimo) sono memorizzati in 2 vettori di interi (`currpath` e `bestpath`) la cui lunghezza è memorizzata in `len`. Il vettore `visited` serve per forzare la condizione di cammino semplice.
Il calcolo del cammino ottimo è fatto dalla funzione ricorsiva `pathR`. Essa ha come modello il principio di moltiplicazione, dove le scelte sono condizionate dal soddisfacimento della condizione sui punti-ferita e, dinamicamente, dal fare parte di un cammino semplice. La condizione di terminazione comporta essere arrivati al nodo di uscita. La verifica di ottimalità non presenta modifiche significative rispetto a quella standard.

# *Commento all'es. 2*

Si tratta di un problema di partizionamento. Poiché è richiesta una sola soluzione e non vi sono vincoli su eventuali simmetrie, si utilizza come modello quello delle disposizioni ripetute. Il vettore `occupa` serve a registrare quante persone sono state al momento piazzate in ciascuna delle macchine. Il test sul numero di passeggeri compatibile con il numero di posti in macchina permette una prima forma di pruning. La condizione `if (pos == 0 && i > 0) break;` è un'ulteriore forma di pruning che permette di rompere un caso semplice di simmetria delle soluzioni: generate tutte le possibili partizioni in cui l'amico 0 è fisso in un'auto, tutte le altre assegnazioni sono una variante simmetrica di quelle già considerate (amico 0 sempre in auto0, o in auto1 e così via...) La funzione `check` determina quante macchine sono state effettivamente riempite (il numero di blocchi nel partizionamento) come `max` (massimo indice di blocco) + 1, poi calcola l'umore medio di tutte le macchine e lo restituisce per verificare la condizione di accettabilità della soluzione.

# *Commento all'es. 3*

Si tratta di un problema di partizionamento con ottimizzazione. Poiché è richiesto di non generare soluzioni simmetriche, si utilizza l'algoritmo di Er. Essendo un problema di ottimizzazione, si generano tutte le soluzioni e si determina quella che soddisfa il criterio.