Convegno Nazionale
Matematica senza Frontiere
Lecce, 5-8 marzo 2003

Un algoritmo euristico per il problema di Set Covering

Giuseppe Reho

giuseppe.reho@libero.it

Sommario

L'importanza rilevante dell'ottimizzazione combinatoria applicata a problematiche aziendali deriva dal fatto che gli attuali sistemi di produzione ed i servizi in generale sono caratterizzati sempre più da un livello di complessità organizzativa e decisionale in continua crescita. Di conseguenza vi è la necessità di organizzare produzione e servizi nell'intento di migliorare i livelli di efficienza e produttività, cercando di contere al minimo costi e spese. In quest'ambito rientra il problema di *Set Covering*, il cui modello matematico è particolarmente utilizzato per l'ottimizzazione dei costi che la "copertura" di un dato servizio richiede.
In quest'articolo si illustrerà la realizzazione e codifica in linguaggio `C` di un elaborato algoritmo euristico per il problema di Set Covering già presentato in letteratura. Infine si riassumeranno i risultati computazionali ottenuti dal programma realizzato per alcuni problemi di Set Covering ottenuti per via telematica.

1 La definizione del problema

Il problema di *Set Covering*, denotato (SCP), è uno dei più rappresentativi problemi di ottimizzazione combinatoria. Gli aspetti principali di questo modello matematico possono essere introdotti attraverso un classico problema (SCP): il problema di localizzione delle caserme dei vigili del fuoco.

Problema di localizzazione. *Una città deve stabilire dove localizzare le caserme dei vigili del fuoco. Essa è suddivisa in un certo numero di distretti da servire (Figura 1) ed ogni caserma situata in un distretto è in grado di servire tutti i distretti confinanti. L'obiettivo è* ***minimizzare il numero delle caserme*** *da situare in modo da "coprire" tutti i distretti.*

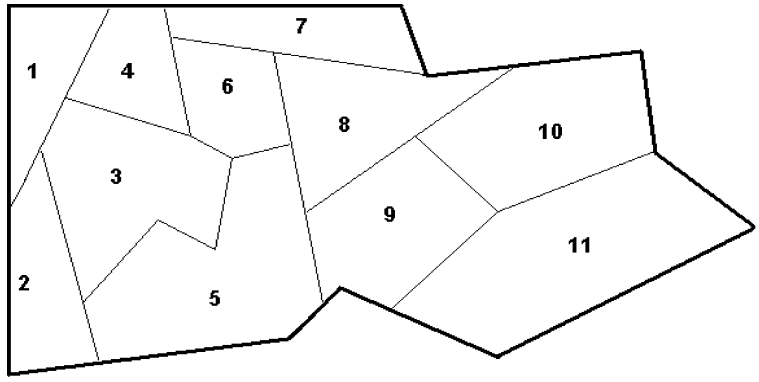

Figura 1: Pianta della città.

Formulazione del problema. Per ogni distretto $j = 1, \ldots, 11$ si associa una variabile logica x_j che vale:

$$x_j = \begin{cases} 1, & \text{se nel distretto } \ j \ \text{ viene situata una caserma} \\ 0, & \text{altrimenti.} \end{cases}$$

Le variabili del problema sono rappresentate dal vettore $\mathbf{x} = (x_1, \ldots, x_{11})$. Poiché l'obiettivo è minimizzare il numero delle caserme da situare, il costo associato ad ogni distretto è $c_j = 1, \ \forall j = 1, \ldots, 11$. La **funzione obiettivo** da minimizzare è così definita:

$$\sum_{j=1}^{11} c_j x_j = \sum_{j=1}^{11} x_j.$$

Si giunge quindi alla seguente formulazione matematica:

$$\begin{cases} \min \ \displaystyle\sum_{j=1}^{11} x_j; \\ x_1 + x_2 + x_3 + x_4 \geq 1 \\ x_1 + x_2 + x_3 + x_5 \geq 1 \\ x_1 + x_2 + x_3 + x_4 + x_5 + x_6 \geq 1 \\ x_1 + x_3 + x_4 + x_6 + x_7 \geq 1 \\ x_2 + x_3 + x_5 + x_6 + x_8 + x_9 \geq 1 \\ x_3 + x_4 + x_5 + x_6 + x_7 + x_8 \geq 1 \\ x_4 + x_6 + x_7 + x_8 \geq 1 \\ x_5 + x_6 + x_7 + x_8 + x_9 + x_{10} \geq 1 \\ x_5 + x_8 + x_9 + x_{10} + x_{11} \geq 1 \\ x_8 + x_9 + x_{10} + x_{11} \geq 1 \\ x_9 + x_{10} + x_{11} \geq 1 \\ x_j \in \{0, 1\}, \ \forall \ j = 1, \ldots, 11 \end{cases}$$

La prima disequazione del sistema lineare indica che il distretto 1 può essere coperto situando una caserma all'interno del suo territorio oppure in uno dei distretti con esso confinanti. Vale la stessa considerazione per le altre disequazioni del sistema. La matrice dei coefficienti $A = (a_{ij})$, $(i = 1, \ldots, 11$ e $j = 1, \ldots, 11)$ del sistema lineare è chiamata **matrice di incidenza** ed è definita in questo modo:

$$a_{ij} = \begin{cases} 1, & \text{se } i \text{ confina con } j \quad \text{oppure} \quad i = j \\ 0, & \text{altrimenti} \end{cases}$$

La colonna j-esima della matrice di incidenza rappresenta l'insieme dei distretti che l'eventuale caserma situata nel distretto j può servire. L'obiettivo è selezionare il minor numero di distretti dove situare le caserme. Ogni distretto deve essere servito **almeno** da una caserma. Una soluzione ottima per questo tipo di problema facilmente individuabile è rappresentata dai valori $x_3 = x_8 = x_9 = 1$ ed il resto delle variabili x_j uguali a 0. Si può decidere quindi di situare una caserma nei distretti 3, 8 e 9. Il vettore

$$\mathbf{x} = (0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0)$$

rappresenta la soluzione trovata e si definisce **vettore di incidenza**.

L'esempio riportato è un problema di *Set Covering* ed è caratterizzato dalla definizione di variabili binarie ($x_j \in \{0, 1\}$), da vincoli di disuguaglianza di tipo '≥ 1' ($A\mathbf{x} \geq \mathbf{1}$) e da una funzione obiettivo lineare da minimizzare. Il problema di Set Covering è quindi un **problema di minimizzazione**

Dato un insieme di base di m elementi $i \in M = \{1, 2, 3, \ldots, m\}$, una famiglia di n sottoinsiemi $S_j \subseteq M$, $j \in N = \{1, 2, 3, \ldots, n\}$ ed i costi $c_j > 0$ di ogni sottoinsieme S_j, il problema di Set Covering ha come obiettivo l'individuazione di una famiglia di sottoinsiemi S_j con il minimo costo totale e con il vincolo che ogni elemento $i \in M$ sia contenuto da almeno un sottoinsieme. La formulazione generale del problema di Set Covering è:

$$\text{(SCP)} \quad \begin{cases} \min \quad cost(\mathbf{x}) = \sum_{j \in N} c_j x_j; \\ \sum_{j \in N} a_{ij} x_j \geq 1, \quad \forall i \in M; \\ x_j \in \{0, 1\}, \quad \forall j \in N; \end{cases}$$

dove

$$a_{ij} = \begin{cases} 1, & \text{se } i \in S_j; \\ 0, & \text{altrimenti.} \end{cases}$$

$S = \left\{ \mathbf{x} = (x_1, \ldots, x_n) : x_j \in \{0, 1\}, \ \forall j \in N, \ \sum_{j \in N} a_{ij} x_j \geq 1, \quad \forall i \in M \right\}$ è definita **regione ammissibile** di (SCP) ed un vettore $\mathbf{x} \in S$ è detto

soluzione ammissibile del problema. Il vettore

$$\mathbf{x}^* = \arg\min\{ \sum_{j \in N} c_j x_j \ : \ \mathbf{x} = (x_1, x_2, \ldots, x_n) \in S \ \}$$

è detto **soluzione ottima** di (SCP).

2 Un algoritmo euristico per (SCP)

Una grande innovazione nell'ambito dei metodi risolutivi per problemi di ottimizzazione combinatoria è stata apportata dai **metodi euristici** che ricercano soluzioni approssimate. Infatti, il metodo euristico non è in grado di dimostrare l'ottimalità delle soluzioni trovate. L'algoritmo presentato da Yagiura, Kishida ed Ibaraki [1] fa parte di questa classe di metodi ed è un algoritmo di *ricerca locale*. Scopo di questa sezione è di dare le idee generali su cui tale euristica è basata.

2.1 Le idee generali dell'euristica

Quando si utilizza un approccio di ricerca locale per la soluzione di problemi di ottimizzazione combinatoria è basilare definire l'intorno di una soluzione corrente rappresentata dal vettore $\mathbf{x}$. L'idea che sta alla base dell'algoritmo citato è la definizione dell'intorno "3-flip" di una soluzione che può essere ammissibile o non ammissibile. Con il termine "flip" si indica uno scambio di valori tra le componenti di un vettore. Trattandosi di componenti binarie, la tecnica consiste appunto nello scambiare il valore di una o più componenti, opportunamente individuate, da 0 ad 1 o viceversa.

Definizione. Sia r un numero intero positivo e $\mathbf{x} \in \{0,1\}^n$ un vettore di n componenti binarie. Sia ora $\mathbf{x}' \in \{0,1\}^n$

$\mathbf{x}' \in NB_r(\mathbf{x}) \ \Leftrightarrow \ \mathbf{x}'$ è ottenuto da $\mathbf{x}$ scambiando al più r componenti.

In quest'algoritmo $r = 1, 2, 3$. Dato un vettore $\mathbf{x} \in \{0,1\}^n$ lo scambio di componenti avviene selezionando un insieme di indici $J \subseteq N$ e definendo il vettore

$$\mathbf{x} \updownarrow J = \{x_1', x_2', \ldots, x_n'\} \iff x_j' = \begin{cases} 1 - x_j, & \text{se } j \in J \\ x_j, & \text{altrimenti} \end{cases}$$

ottenuto da $\mathbf{x}$ scambiando le variabili in J.

L'altra idea su cui si basa questo algoritmo euristico di ricerca locale è un meccanismo chiamato "oscillazione strategica" tra la regione ammissibile e la regione inammissibile in modo alternato, e cioè la ricerca della soluzione ottima intorno alla frontiera della regione ammissibile. Una soluzione ammissibile è detta *minimale* se diventa inammissibile assegnando il valore 0 a

qualche componente (variabile) x_j con valore 1. In effetti, una soluzione ottima del problema di Set Covering è sempre minimale e l'oscillazione strategica può essere considerata un efficace strumento di ricerca di soluzioni minimali. In quest'algoritmo essa viene realizzata attraverso una *funzione costo penalizzato* definita attraverso dei coefficienti di penalità relativi alle righe della matrice di incidenza A. Questi coefficienti vengono variati, sulla base di determinati criteri, ogni volta che si è in presenza di un minimo locale, definendo una nuova funzione costo penalizzato per realizzare l'oscillazione. Per ogni riga si definisce il coefficiente di penalità $p_i > 0$. Sia ora $\mathbf{x} \in \{0, 1\}^n$ una soluzione ammissibile o non ammissibile. La funzione

$$pcost(\mathbf{x}) = \sum_{j \in N} c_j x_j + \sum_{i \in I_s(\mathbf{x})} p_i$$

con

$$I_s(\mathbf{x}) = \{i \in M : \ i \text{ riga non "coperta" da } \mathbf{x}\}$$

è chiamata *funzione costo penalizzato* e dipende anche dai valori che assumono le penalità.

Un altro aspetto fondamentale dell'euristica citata è la riduzione della dimensione del problema. Tale riduzione diventa una necessità soprattutto per problemi con un gran numero di vincoli e variabili. Per questo motivo nell'algoritmo la dimensione del problema è ridotta euristicamente fissando alcune variabili $x_j = 0$ ed altre $x_j = 1$. Le variabili non ancora assegnate costituiranno un problema di Set Covering di dimensione ridotta chiamato *core problem* sul quale viene condotta la ricerca locale. L'insieme delle variabili fissate è modificato dinamicamente definendo una partizione degli indici delle colonne della matrice di incidenza del problema. Fissando di volta in volta gli indici si costruisce un nuovo core problem.

2.2 Le fasi dell'algoritmo

L'algoritmo si divide in tre fasi principali:

- **Inizializzazione** (Definizione del primo ***core problem***)
- **Ricerca locale**
- **Definizione di un nuovo *core problem***

Le fasi di ricerca locale e di definizione di un nuovo core problem vengono alternativamente ripetute dopo la fase di inizializzazione fino a quando un criterio di arresto non viene soddisfatto. Nella fase di inizializzazione si calcola una prima soluzione ammissibile $\mathbf{x}^*$ (soluzione incombente) del problema mediante una procedura di tipo GREEDY e si inizializzano i coefficienti di penalità p_i per ogni riga. Successivamente si utilizza il *metodo del subgradiente* per il calcolo di un limite inferiore LB (*lower bound*) per l'ottimo di (SCP) per il quale risulta $LB \leq cost(\mathbf{x}), \ \forall \ \mathbf{x} \in S$.

La fase di ricerca locale è realizzata dalle procedure 1-FLIP($\mathbf{x}$), 2-FLIP($\mathbf{x}$) e 3-FLIP($\mathbf{x}$) che, a partite da una soluzione corrente $\mathbf{x}$, ricercano soluzioni migliori di $\mathbf{x}$ nel sistema di intorni $NB_r(\mathbf{x})$ con $r = 1, 2, 3$. Tali procedure ricercano un insieme di indici delle componenti di $\mathbf{x}$, in modo tale da migliorare la soluzione corrente attraverso lo scambio (*flip*) delle relative componenti. Se le procedure FLIP non individuano nessun insieme di indici, esse ritornano la soluzione corrente ricevuta in input. La funzione che all'interno di queste procedure di ricerca locale valuta la qualità delle soluzioni trovate è la funzione costo penalizzato *pcost* precedentemente definita. Si può allora riassumere che, data una soluzione $\mathbf{x}$, la ricerca locale viene effettuata mediante le tre procedure:

1-FLIP($\mathbf{x}$), che ricerca un indice $j_1 \in N$ tale che:

$$pcost\big(\mathbf{x} \updownarrow \{j_1\}\big) < pcost\big(\mathbf{x}\big) \quad \text{e pone} \quad \mathbf{x} := \mathbf{x} \updownarrow \{j_1\},$$

2-FLIP($\mathbf{x}$), che ricerca una coppia $\{j_1, j_2\} \subset N$ tale che:

$$pcost\big(\mathbf{x} \updownarrow \{j_1, j_2\}\big) < pcost\big(\mathbf{x}\big) \quad \text{e pone} \quad \mathbf{x} := \mathbf{x} \updownarrow \{j_1, j_2\},$$

3-FLIP($\mathbf{x}$), che ricerca una terna $\{j_1, j_2, j_3\} \subset N$ tale che:

$$pcost\big(\mathbf{x} \updownarrow \{j_1, j_2, j_3\}\big) < pcost\big(\mathbf{x}\big) \quad \text{e pone} \quad \mathbf{x} := \mathbf{x} \updownarrow \{j_1, j_2, j_3\}.$$

La fase di definizione del core problem è rappresentata dalle procedure FIRST-FIXING, che realizza il primo core problem, e dalla procedura MODIFY-FIXING che crea un nuovo core problem per la ricerca di soluzioni migliori di quella corrente. In queste procedure si realizza una partizione di tutti gli indici ($j \in N$) delle variabili dell'intero problema rappresentata dagli insiemi N_0, N_1, N_{free}. Una volta individuata questa partizione si pone: $\forall j \in N_0,\ x_j = 0$ e $\forall j \in N_1,\ x_j = 1$. L'insieme N_{free} rappresenta gli indici delle variabili non ancora assegnate e quindi "libere". Data, allora, una partizione definita dalla terna di insiemi (N_1, N_0, N_{free}), viene definito **core problem** il seguente problema ridotto:

$$\begin{cases} \min \quad cost(\mathbf{x}) = \displaystyle\sum_{j \in N_{\text{free}}} c_j x_j \\ \displaystyle\sum_{j \in N_{\text{free}}} a_{ij} x_j \geq 1,\ \forall i \in M \setminus \left\{ i' \in M : \sum_{j \in N_1} a_{i'j} \geq 1 \right\} \\ x_j \in \{0, 1\}, \quad \forall j \in N_{\text{free}} \end{cases}$$

Esso viene passato come argomento alla fase di **ricerca locale**.

2.3 La struttura dell'intero algoritmo

Descriviamo ora l'algoritmo nella sua intera struttura. Il parametro di tempo massimo di esecuzione `time_lim` è un limite specifico che dipende dalla dimensione del problema che si intende risolvere. La soluzione $\mathbf{x}^*$ è la

soluzione incombente, cioé quella che restituisce l'algoritmo una volta che il tempo computazionale supera il parametro `time_lim`. La soluzione $\mathbf{x}$ è la soluzione corrente, argomento delle procedure di ricerca locale. Il parametro `minitr_ls` = 100 è il numero massimo di iterazioni della fase di ricerca locale su di uno stesso core problem.

- **Passo 1.** Poni $\mathbf{x}^* :=$ GREEDY, $UB := cost(\mathbf{x}^*)$. Calcola LB con il metodo del subgradiente.

- **Passo 2.** Inizializza $(N_1,\ N_0,\ N_{\text{free}})$, poni `counter` := 0 e $\mathbf{x} := 0$. Inizializza $p_i := \min\{c_j\ : a_{ij} \neq 0\}\ ,\ \forall\ i \in M$.

- **Passo 3.** Se il tempo di calcolo ha superato `time_lim`, emetti $\mathbf{x}^*$ e STOP.

- **Passo 4.** Poni $\tilde{\mathbf{x}} :=$ 1-FLIP($\mathbf{x}$). Se $\tilde{\mathbf{x}} \neq \mathbf{x}$, poni $\mathbf{x} := \tilde{\mathbf{x}}$ e torna al Passo 4.

- **Passo 5.** Poni $\tilde{\mathbf{x}} :=$ 2-FLIP($\mathbf{x}$). Se $\tilde{\mathbf{x}} \neq \mathbf{x}$, poni $\mathbf{x} := \tilde{\mathbf{x}}$ e torna al Passo 4.

- **Passo 6.** Se $cost(\mathbf{x}) \leq LB$, vai al Passo 7, altrimenti poni $\tilde{\mathbf{x}} :=$ 3-FLIP($\mathbf{x}$). Se $\tilde{\mathbf{x}} \neq \mathbf{x}$, poni $\mathbf{x} := \tilde{\mathbf{x}}$ e torna al Passo 4.

- **Passo 7.** Poni `counter` := `counter` + 1. Se sono state trovate soluzioni ammissibili nei Passi **4**, **5** e **6**, sia $\mathbf{x}^+$ quella con il minimo $cost(\mathbf{x})$. e $cost(\mathbf{x}^+) < UB$, poni $\mathbf{x}^* := \mathbf{x}^+$, $UB := cost(\mathbf{x}^*)$ e `counter`:=0.

- **Passo 8.** Aggiorna le penalità p_i. Se le penalità sono incrementate o `counter` < `minitr_ls`, torna al Passo 3.

- **Passo 9.** Modifica $(N_1,\ N_0,\ N_{\text{free}})$. Poni `counter` := 0 e torna al Passo 3.

In quest'algoritmo, il Passo 1 è la fase di inizializzazione, i Passi 2 e 9 rappresentano la fase in cui si fissano le variabili (FIRST-FIXING) e successivamente si modificano (MODIFY-FIXING). La ricerca locale viene effettuata dal Passo 3 al Passo 8 insieme all'aggiornamento delle penalità. Il parametro `counter` è un contatore ed indica il numero di iterazioni della ricerca locale eseguita dopo che la soluzione incombente $\mathbf{x}^*$ è stata aggiornata o dopo che si ha un nuovo aggiornamento delle variabili fissate. Da osservare che un'iterazione della ricerca locale trova una soluzione il cui valore *pcost* non può essere migliorato dal Passo 4 al Passo 6, a meno che le penalità p_i non sono state aggiornate nel Passo 8.

3 La realizzazione in Linguaggio C

In questa sezione viene presentata la realizzazione dell'algoritmo euristico descritto precedentemente. Esso è stato codificato in linguaggio C in un progetto chiamato scp_ls (*Set Covering problem local search*). Il programma legge i dati da *files* esterni e scrive i risultati ottenuti su un file di testo chiamato log_file.txt. Il progetto scp_ls è stato organizzato nei moduli:

- scpls.h
- main.c
- read_instance.c
- initialization.c
- core_problem.c
- local_search.c
- update_variables.c

Diamo ora una decrizione generale del progetto.

3.1 La lettura dei dati e la loro rappresentazione

Il modulo scpls.h chiamato *header file* è la parte di programma in cui sono state definite le strutture dei dati e delle variabili del problema, le costanti che il programma utilizza e le dichiarazioni delle funzioni principali che lo caratterizzano.
La struttura principale in cui vengono memorizzati i dati di una generica istanza di un problema di Set Covering è chiamata Tmatrix ed è una particolare rappresentazione della matrice booleana che definisce i vincoli del problema. Tale tipo di struttura è anche utilizzata per definire la matrice che rappresenta il core problem. Tutte le informazioni riguardanti la matrice vengono memorizzate in questo tipo di struttura: numero di righe nrows, numero di colonne ncolumns, gli elementi non nulli delle righe nel vettore cols_ind[ ] e quelli delle colonne in rows_ind[ ], le penalità delle righe pweight[ ], i differenziali dei costi penalizzati delle colonne delta_pcost[ ] ed i costi ridotti red_cost[ ] relativi al vettore dei moltiplicatori di lagrange u_mult[ ] per la procedura del subgradiente.
La struttura Nset rappresenta la partizione delle colonne della matrice costituita dagli insiemi N_1 (set_one[ ]), N_0 (set_zero[ ]) ed N_{free}(set_free[ ]). Tale struttura è argomento principale di first_fix e modify_fix, che realizzano la partizione delle colonne dell'intero problema e che consentono di definire il core problem.

Nella struttura `Solutions` vengono memorizzate le informazioni inerenti alle soluzioni ottenute dall'algoritmo durante la fase di ricerca locale. La soluzione incombente è definita dal vettore `inc_sol[ ]` in cui vengono memorizzati gli indici delle colonne appartenenti ad essa. Poiché la stampa della soluzione finale sul file di testo dovrà riportare il costo totale della soluzione, gli indici delle colonne selezionate ed i relativi costi, oltre al valore `inc_tot_cost` (costo totale) è stato definito il vettore `inc_cost[ ]` in cui il generico elemento di posto j è il costo del corrispondente elemento (colonna) di posto j di `inc_sol[ ]`. La soluzione corrente è definita in modo analogo dalla variabile `curr_tot_cost`, dai vettori `curr_cost[ ]` , `curr_sol[ ]` ed inoltre è caratterizzata dal vettore di incidenza `curr_vect[ ]` riferito alle colonne della matrice originaria, dalla variabile booleana `curr_sol_feasible` il cui valore indica l'ammissibilità della soluzione e dal costo penalizzato `pcost`. Il vettore di contatori `cover_count[ ]` è riferito alle righe della matrice che viene passata come argomento nella fase di ricerca locale. Il valore del generico elemento di posto i indica il numero di colonne della soluzione alle quali la riga i appartiene. Infine, la struttura `Lagr_par` è costituita dai parametri computazionali utilizzati nella procedura del subgradiente.
Per ogni tipo di struttura è stato definito un puntatore alla struttura stessa. Infatti, le funzioni del progetto `scp_ls` accedono alle strutture dei dati e delle variabili mediante puntatori a strutture. Essi vengono allocati dinamicamente nel modulo principale `main.c` e sono:

1) `*orig_matr` (puntatore alla struttura matrice di origine);

2) `*core_matr` (puntatore alla struttura matrice del core problem);

3) `*sol` (puntatore alla struttura delle soluzioni);

4) `*part` (puntatore alla struttura partizione);

5) `*lagr` (puntatore alla struttura dei parametri lagrangiani).

Il modulo `read_instance.c` legge i dati da files esterni e crea mediante le funzioni `readscp` e `transpose` la matrice del problema originario puntata da `orig_matr`. La matrice booleana dei vincoli del problema viene rappresentata in una forma compatta memorizzando nel vettore `rows_ind[ ]` gli elementi non nulli di ogni colonna j ossia gli indici delle righe appartenenti ad ogni colonna. Per poter stabilire quali indici del vettore `rows_ind[ ]` appartengono alla generica colonna j viene utilizzato il vettore di puntatori `*col[ ]` nel quale l'elemento di posto j punta alla locazione di `rows_ind[ ]` dove è allocato il primo elemento della colonna j considerata. Il numero degli elementi della generica colonna j è dato dalla differenza `col[`j`+1]-col[`j`]`. La funzione `transpose`, sulla base dei dati già acquisiti da `readscp`, definisce il vettore `cols_ind[ ]` inserendo in esso gli indici non nulli delle colonne alle quali la generica riga i appartiene ed inoltre definisce il vettore di puntatori alle righe `*row[ ]` avente le stesse

caratteristiche di quello visto per le colonne. In questo modo la matrice è completamente rappresentata. Il formato in cui sono stati memorizzati i dati consente di mantenere in memoria soltanto i valori significativi della matrice dei vincoli (quelli non nulli) permettendo in tal modo un notevole risparmio della memoria del calcolatore.

3.2 La fase di inizializzazione e di selezione del core problem

La fase iniziale dell'algoritmo in cui si inizializzano le variabili è stata realizzata nel modulo `initialization.c`. La funzione `greedy_algorithm` inizializza la soluzione incombente calcolando una prima soluzione euristica del problema mediante la procedura greedy descritta nel capitolo precedente ed ha come argomenti il puntatore alla matrice originaria ed il puntatore alla struttura delle soluzioni. Il costo totale della soluzione greedy viene utilizzato successivamente nel metodo del subgradiente realizzato dalla funzione `subgrad` la quale calcola un limite inferiore, il vettore dei moltiplicatori di lagrange `u_mult[]` ed i relativi costi ridotti di tutte le colonne di `orig_matr`. Utilizzando i risultati ottenuti dal subgradiente viene definito il core problem. La selezione delle colonne da inserire nella matrice ridotta, implementata nel modulo `core_problem.c`, è realizzata dalle funzioni `first_fix`, `modify_fix` e `make_minor`, le quali hanno come argomento principale la struttura partizione. La funzione `first_fix` viene utilizzata soltanto nella fase di inizializzazione in quanto crea una prima partizione delle colonne della matrice originaria. La matrice che rappresenta il core problem puntata da `core_matr` viene creata dalla funzione `make_minor`. Essa ha come argomenti le matrici puntate da `orig_matr`, `core_matr` e la struttura partizione puntata da `part`. Infatti, utilizzando le informazioni contenute in quest'ultima, la funzione `make_minor` copia in `core_matr` soltanto le caratteristiche delle righe e delle colonne che definiscono il core problem: il loro numero, le penalità, i differenziali dei costi penalizzati. Per non perdere la corrispondenza tra elementi della matrice ridotta ed elementi di quella originaria, vengono memorizzati nel vettore `orig_col[]` ed in `orig_rows[]` gli indici originari delle colonne e delle righe ai quali corrispondono gli indici delle colonne e delle righe del core problem. In questo modo, quando la matrice ridotta viene passata come argomento alle funzioni che realizzano la fase della ricerca locale, se una colonna di indice j viene selezionata, allora viene memorizzato nella soluzione corrente l'indice `orig_col[`j`]` che è quello di origine. La funzione `modify_fix` modifica la struttura partizione selezionando un nuovo insieme di righe e colonne per definire un nuovo core problem creato dalla successiva chiamata di `make_minor`.

3.3 La fase di ricerca locale e l'aggiornamento delle penalità

Il modulo `local_search.c` è costituito dalle funzioni `one_flip`, `two_flip` e `three_flip` che realizzano la fase di ricerca locale e dalla funzione `update` che aggiorna il vettore dei differenziali dei costi penalizzati relativi a tutte le colonne del core problem corrente. La funzione `update` aggiorna anche la soluzione corrente `curr_sol[]` e la somma dei costi `curr_tot_cost`. Se l'algoritmo individua una soluzione ammissibile con un costo complessivo minore del costo dell'ultimo aggiornamento della soluzione incombente, quest'ultima viene aggiornata mediante la funzione `update_inc_sol` presente nel modulo `update_variables.c`. La funzione principale del modulo è `update_penalty` che aggiorna le penalità delle righe della matrice. Una volta modificate le penalità, viene aggiornato il costo penalizzato della soluzione corrente attraverso la funzione `update_pcost_sol`, i differenziali dei costi penalizzati delle colonne e, se il core problem viene modificato, la funzione `update_cover_count` aggiorna il vettore di contatori delle righe della nuova matrice ridotta.

4 Risultati computazionali

L'algoritmo di ricerca locale codificato nel progetto `scp_ls` è stato sperimentato su problemi di Set-Covering ottenibili per via telematica dalla collezione OR-Library nel sito [5]. Gli esperimenti computazionali sono stati effettuati utilizzando un Personal Computer con un processore AMD Athlon(TM)XP 1800+, 1,53 GHz e 256 MB di RAM.

4.1 Le caratteristiche dei problemi test

Tra i problemi di set covering presenti nella collezione OR-Library sono stati considerati i seguenti tipi: 4, 5, 6, A, B, C, D, E, F, G, H e RAIL. Ogni tipo di problema ha diverse istanze.
Il tipo di problemi 4 e 5 presentano dieci istanze, mentre i tipi 6 e quelli A-H ne comprendono cinque. Essi sono stati generati con dati casuali. Il numero di vincoli, denotato con m, è compreso fra 200 e 1000, mentre il numero delle variabili, indicato con n è compreso fra 1000 e 10000. Il costo della j-esima variabile c_j, anch'esso generato casualmente, è un intero appartenente all'insieme chiuso [1, 100]. Indicato con M l'insieme dei vincoli (righe) e con N l'insieme delle variabili (colonne), per ogni istanza viene definita densità il valore $d = \sum_{i \in M} \sum_{j \in N} a_i j/mn$. Essa è compresa tra 0.02 e 0.2. Le soluzioni ottime di tutte le istanze sono note, eccetto quelle dei problemi di tipo E-H.
Il tipo RAIL comprende istanze che rappresentano problemi di turnazione di equipaggi (*crew scheduling problem*) nelle ferrovie italiane. Queste istanze rappresentano quindi problemi reali e sono caratterizzate da un gran numero

di vincoli e variabili. I costi delle variabili c_j assumono i valori 1 oppure 2 e la densità degli elementi non nulli della matrice dei vincoli è molto bassa. I valori ottimi dei problemi RAIL sono noti soltanto per alcune istanze. La **Tabella 1** riassume le caratteristiche di tutti i problemi considerati.

Tabella 1: **Caratteristiche dei problemi test**

Problemi	m	n	Densità	Insieme Costi
tipo 4	200	1000	2%	[1, 100]
tipo 5	200	2000	2%	[1, 100]
tipo 6	200	1000	5%	[1, 100]
tipo A	300	3000	2%	[1, 100]
tipo B	300	3000	5%	[1, 100]
tipo C	400	4000	2%	[1, 100]
tipo D	400	4000	5%	[1, 100]
tipo E	500	5000	10%	[1, 100]
tipo F	500	5000	20%	[1, 100]
tipo G	1000	10000	2%	[1, 100]
tipo H	1000	10000	5%	[1, 100]
RAIL507	507	63009	1.2%	[1, 2]
RAIL516	516	47311	1.3%	[1, 2]
RAIL582	582	55515	1.2%	[1, 2]
RAIL2586	2586	920683	0.4%	[1, 2]

4.2 I risultati ottenuti

Il progetto `scp_ls` è stato sperimentato inizialmente sui problemi di set covering di tipo 4-6 e A-D caratterizzati da istanze aventi una dimensione minore rispetto ai problemi di tipo E-H e RAIL. Ad ogni esecuzione del programma viene inserito sulla linea di comando un valore numerico scelto a caso chiamato "seme" (*seed*), attraverso il quale la funzione `random` genera iterativamente una successione pseudo-casuale di numeri reali nell'intervallo $(0, 1)$. Tale successione viene utilizzata dall'algoritmo per effettuare quelle scelte casuali previste nella fase di ricerca locale e di definizione del core problem. Ogni istanza viene risolta dieci volte e ad ogni esecuzione viene inserito un seme diverso per ottenere diversi numeri casuali. Un buon risultato in un tempo relativamente breve dipende anche dalle operazioni di scelta casuale che l'algoritmo compie. Per questo motivo nella **Tabella 2** e nella **Tabella 3** è stato riportato per ogni istanza risolta il tempo minimo (Min), medio (Med) e massimo (Max) impiegato dal programma per ottenere il valore ottimo. L'unità di misura considerata per esprimere i

valori dei tempi è il millisecondo, che corrisponde effettivamente alla sensibilità delle misure effettuate dal calcolatore.

Tabella 2: **Tempo computazionale espresso in millisecondi impiegato per trovare la soluzione ottima per i problemi test di tipo 4, 5, 6.**

Problema	val. ottimo	Min	Med	Max
4.1	429	16	16.0	16
4.2	512	15	15.0	15
4.3	516	15	15.0	15
4.4	494	16	16.0	16
4.5	512	16	16.0	16
4.6	560	47	54.5	62
4.7	430	15	15.5	16
4.8	492	266	1523.5	2781
4.9	641	16	31.5	47
4.10	514	16	16.0	16
5.1	253	94	375.0	656
5.2	302	125	211.0	297
5.3	226	15	15.0	15
5.4	242	15	23.0	31
5.5	211	141	274.0	407
5.6	213	16	16.0	16
5.7	293	172	398.5	625
5.8	288	16	16.0	16
5.9	279	15	15.5	16
5.10	265	16	16.0	16
6.1	138	16	16.0	16
6.2	146	15	312.0	609
6.3	145	16	16.0	16
6.4	131	16	16.0	16
6.5	161	32	55.0	78

Dall'analisi dei risultati ottenuti possiamo affermare che per le istanze di SCP di tipo 4-6 e di tipo A-D l'algoritmo impiega pochi secondi per il calcolo di ciascuna soluzione. Considerando la **Tabella 2** e la colonna relativa ai tempi medi, si può notare come la maggioranza di essi si attestano intorno ai 15, 16 millisecondi, mentre per i problemi 4.8, 5.1, 5.2, 5.5, 5.7 e 6.2 il tempo medio di calcolo aumenta fino a 1500 millisecondi. Un'altra osservazione può essere fatta in merito ai risultati ottenuti per i problemi

Tabella 3: **Tempo computazionale espresso in millisecondi impiegato per trovare la soluzione ottima per i problemi test A, B, C, D.**

Problema	val. ottimo	Min	Med	Max
A.1	253	296	4054.5	7813
A.2	252	47	1297.0	2547
A.3	232	31	2109.0	4187
A.4	234	32	328.0	625
A.5	236	421	726.0	1031
B.1	69	125	312.5	500
B.2	76	234	617.0	1000
B.3	80	234	796.5	1359
B.4	79	156	1219.0	1141
B.5	72	15	46.5	78
C.1	227	687	773.0	859
C.2	219	1015	1906.0	2797
C.3	243	312	8304.0	16296
C.4	219	1563	2531.5	3500
C.5	215	204	524.0	844
D.1	60	250	1109.5	1969
D.2	66	187	1249.5	2312
D.3	72	62	523.5	985
D.4	62	235	844.0	1453
D.5	61	63	70.5	78

A-D e riportati nella **Tabella 3**. Non si riscontra un'omogeneità tra i valori medi dei tempi. Infatti, per le istanze A.1 e C.3 il tempo medio di calcolo del valore ottimo aumenta considerevolmente rispetto a quello impiegato per altre aventi medesime caratteristiche. Ciò è dovuto sia al diverso grado di difficoltà tra istanze appartenenti alla stessa classe, sia alla natura probabilistica dell'algoritmo e cioè alla probabilità di passare come argomento un seme iniziale che determini "buone" scelte casuali per ottenere in pochi secondi, almeno per questo tipo di problemi, la soluzione ottima.

Ottenuti questi primi risultati sui problemi di set covering per i quali sono note le rispettive soluzioni ottime, il programma `scp_ls` è stato eseguito sui problemi E-H e RAIL. Anche per questi tipi di problemi sono state effettuate dieci prove per ognuno di essi ed ogni volta con un seme iniziale diverso. I risultati ottenuti sono riassunti nella **Tabella 4**. Il tempo massimo di esecuzione è rappresentato dal valore del parametro `time_lim`

oltre il quale il programma restituisce la migliore soluzione euristica trovata. Si è deciso di porre `time_lim` = 900 secondi per i problemi di tipo E-H, `time_lim` = 3600 secondi per i problemi RAIL 507, RAIL 516 e RAIL 582 ed infine `time_lim` = 18000 secondi per il problema RAIL 2586.

Tabella 4: **Risultati per i problemi E-H e tipo RAIL.**

Problemi	LB	MVC	V_min	V_max	Min	Med	Max
E1	22	29	∗29	29	371	7917	15092
E2	22	30	∗30	31	121605	–	–
E3	21	27	∗27	27	471	121075	241678
E4	21	28	∗28	28	1723	93851	185978
E5	21	28	∗28	28	1342	6910	12478
F1	9	14	∗14	14	4297	12391	20484
F2	10	15	∗15	15	250	7422	14594
F3	9	14	∗14	14	11437	22664	33891
F4	9	14	∗14	14	547	7993	15438
F5	8	13	∗13	13	22156	351921	681687
G1	159	176	∗176	176	875	25625	50375
G2	142	154	∗154	154	12578	431828	851078
G3	148	166	∗166	167	26218	–	–
G4	148	168	∗168	168	114328	404742	695156
G5	148	168	∗168	168	38500	413226	787953
H1	48	63	∗63	64	24453	–	–
H2	48	63	∗63	64	519266	–	–
H3	45	59	∗59	60	156219	–	–
H4	44	58	∗58	58	11938	95501	179063
H5	42	55	∗55	55	35657	–	–
RAIL 507	173	174	∗174	175	886750	–	–
RAIL 516	182	182	∗182	182	11375	248039	484703
RAIL 582	210	211	∗211	211	7016	274331	541646
RAIL 2586	936	945	950	955	–	–	–

Nella **Tabella 4** sono riportati per ogni istanza il limite inferiore (LB), la migliore soluzione conosciuta (MVC), il valore minimo (V_min) e massimo (V_max) della soluzione euristica trovata dal programma entro il tempo limite stabilito ed infine il tempo minimo (Min), medio (Med) e massimo (Max) entro il quale il programma è riuscito a calcolare la soluzione migliore. Nelle colonne in cui sono riportate le soluzioni il segno '∗' indica che il programma ha calcolato una soluzione che coincide con la migliore conosciuta (MVC). Nella colonna relativa al tempo massimo il segno '–' indica che il programma non è riuscito a calcolare in tutte le dieci prove

effettuate la miglior soluzione entro il limite stabilito. Si può osservare che comunque la colonna delle soluzioni conosciute (MVC) e la colonna in cui si riportano le soluzioni minime calcolate (V_min) coincidono eccetto per l'istanza RAIL 2586. Possiamo quindi concludere che il programma è riuscito a calcolare sempre la soluzione ottima per quanto riguarda le istanze in corrispondenza delle quali sono stati riportati sia il tempo minimo, che quelli medio e massimo di calcolo, mentre per le istanze E.2, G.3, H.1, H.2, H.3, H.5 e RAIL 507 il programma ha restituito in alcune prove una soluzione che differisce da quella ottima di una unità. Per quanto riguarda i risultati ottenuti per l'istanza RAIL 2586 il programma ha restituito come miglior soluzione 950. Valgono quindi le stesse considerazioni fatte in precedenza riguardanti il grado di difficoltà nel risolvere una data istanza e riguardo la natura probabilistica dell'algoritmo di ricerca locale che si è implementato.

Bibliografia

[1] M. Yagiura, M. Kishida e T. Ibaraki, "A 3-Flip Neighborhood Local Search for the Set Covering Problem", (Sottoposto per la pubblicazione)

[2] A. Sassano, "Modelli e Algoritmi della Ricerca Operativa", Franco Angeli, Milano (1999)

[3] B.W.Kernighan e D.M.Ritchie, "The C programming language, second edition", *Prentice-Hall, Englewood Cliffs*, New Jersey, USA (1989)

[4] Herbert Schildt, "C - The Complete Reference", *McGraw-Hill*, (1990)

[5] J.E. Beasley OR-Library http://www.ms.ic.ac.uk/jeb/pub.html