Formazione di coalizioni per l'uso intelligente di energia

corso di Ragionamento Automatico

Andrea Zambon (vr092382)

6 marzo 2011

Indice

1 Descrizione del problema

Questo progetto si focalizza sulla risoluzione di un problema di formazione di coalizioni. Si tratta di trovare il migliore modo di coalizzare (cioè partizionare) un insieme di utilizzatori di energia in modo tale da rispettare certi vincoli e da fare in modo che il consumo di energia aggregato di ciascuna coalizione sia il meno variabile possibile. Questa possibilità può essere utile se più utilizzatori di energia vogliono unirsi insieme e pagare solo una fornitura di energia costante nel tempo (bulk) cercando nel contempo di massimizzare l'utilizzazione di tale fornitura di energia. Questo implica che gli utilizzatori di energia dovrebbero essere raggruppati in modo tale che i picchi di consumo di uno coincidano con i momenti di basso consumo degli altri, permettendo così di soddisfare più utenti con la stessa quantità di energia.
Più formalmente, il problema può essere specificato come segue. Sia n il numero di utenti per cui si vuole trovare il raggruppamento ottimo in coalizioni. Per ciascun utente si conosce il profilo di uso dell'energia. Tale profilo d'uso è una funzione specifica per ogni utente che lega l'istante di tempo con il consumo di energia. Indichiamo tali funzioni con $P_i(t)$, con valori positivi che indicano un consumo di energia da parte di un utente. Dato che questi sono utilizzatori di energia, si ha che $\forall i \forall t P_i(t) \geq 0$. Questo significa che non sono ammessi utenti che producono energia (come abitazioni o aziende dotate di tetti fotovoltaici o altro).
Potrebbe essere possibile estendere l'algoritmo per trattare anche il caso di produttori di energia individuando prima tutti gli utenti che hanno anche un solo valore negativo, quindi provando tutti i possibili asseganmenti di questi utenti (con uno schema backtracking puro, senza forward checking o altre ottimizzazioni) e per ciascuno di questi assegnamenti eseguire l'algoritmo utilizzando anche le ottimizzazioni. Questo perchè, se si potesse aggiungere ad una coalizione un utente produttore di energia in qualsiasi momento, non sarebbe più vero che se un utente non può essere aggiunto ad una certa coalizione in un determinato momento, quello stesso utente non possa essere aggiunto a quella coalizione in un momento successivo.
Le variabili del problema possono quindi essere n variabili intere che possono assumere i valori da 1 ad n. Ogni variabile è associata ad un'utenza ed il suo valore rappresenta a quale coalizione tale utenza dovrà appartenere. Ad esempio, se X_1 vale 3, significa che l'utenza 1 farà parte della coalizione 3. Questo schema permette di garantire immediatamente che un utente non possa far parte di più di una coalizione, che un utente non possa comparire più di una volta all'interno della stessa coalizione e che ogni utente debba far parte di almeno una coalizione.
Si conoscono anche il massimo numero di coalizioni utilizzabili, conoscendo

anche per ciascuna di esse il massimo valore di consumo cumulativo ammesso ($C_{max}(j)$).
I vincoli del problema sono limitati al fatto che, in ogni momento, la somma dei consumi delle utenze che fanno parte della stessa coalizione non deve essere superiore al massimo consentito per ciascuna coalizione. In formule:

$\forall j \forall t \sum_{i:X_i=j} P_i(t) \leq C_{max}(j)$

Quindi per ogni coalizione ($\forall j$) ed in ogni istante ($\forall t$) la somma dei consumi degli utenti in quella coalizione ($\sum_{i:X_i=j} P_i(t)$) non deve superare il valore massimo ammesso di consumo per la coalizione ($\leq C_{max}(j)$). Sono quindi presenti fino ad n vincoli n-ari, uno per ogni coalizione, che affermano che il consumo massimo in quella coalizione non può superare il limite fissato.
La funzione obiettivo è la somma dei load diversity factor delle varie coalizioni. Questo valore (il load diversity factor) indica quanto è variabile il consumo dei singoli utenti in rapporto al consumo aggregato ottenuto cumulando i vari consumi. Dato che si vuole raggruppare gli utenti in modo da minimizzare il consumo aggregato, si ha che si dovrà massimizzare la seguente quantità:

massimizzare $\sum_j \frac{\sum_{i:X_i=j} max_t P_i(t)}{max_t \sum_{i:X_i=j} P_i(t)}$

Nel caso di una coalizione vuota, si avrebbe che il rapporto appena descritto per quella coalizione varrebbe $\frac{0}{0}$. Dato che il load diversity factor per una coalizione non vuota è sempre maggiore o uguale ad uno, si stabilisce che il load diversity factor per una coalizione vuota valga 1.

1.1 Dimensione dello spazio di ricerca

Questa rappresentazione presenta un numero di assegnamenti possibili pari a $numero\_di\_coalizioni^n$, dato che ci sono n variabili che possono assumere i valori da 1 al numero di coalizioni presenti. Nel caso peggiore, quindi, si può arrivare a n^n. Notare che il caso di un numero di coalizioni maggiore del numero di utenti non è contemplato, dato che comunque non si potrebbero usare più di n coalizioni distinte per partizionare n utenti. Questo spazio di ricerca di n^n è estremamente ampio, significativamente più grande di 2^n (si ha che $n^n = (2^{log(n)})^n = 2^{nlog(n)}$. Se però si può assumere che tutte le coalizioni abbiano lo stesso limite massimo al consumo di energia, si può ridurre lo spazio di ricerca ad n! con la seguente considerazione.

Supponiamo di avere un ordine sulle variabili (indicato con degli indici X_1, X_2, X_3...) e di utilizzare tale ordine per un algoritmo di tipo backtracking. Quando si va ad assegnare la prima variabile, essa dovrà essere obbligatoriamente messa in una coalizione vuota (dato che inizialmente sono tutte vuote). Dato che tutte le coalizioni hanno lo stesso limite massimo al consumo di energia, non ha importanza ai fini della soluzione in quale coalizione viene messa X_1, quindi si può supporre che essa sia sempre nella coalizione 1 (se si trovasse una soluzione con $X_1 = 2$, sarebbe possibile trasformarla in una soluzione con $X_1 = 1$ semplicemente scambiando le coalizioni 1 e 2, quindi imporre che X_1 valga 1 non impedisce di trovare la soluzione ottima, se questa esiste).
Consideraimo ora X_2. Ci sono solo due casi possibili: o X_2 viene messa nella stessa coalizione di X_1, oppure viene messa in una coalizione nuova. In questo secondo caso, come sopra, non conta in quale coalizione viene messo l'utente rappresentato da X_2, basta che sia una coalizione diversa da quella di X_1, cioè da 1. Quindi non è restrittivo imporre che se X_2 deve essere in una coalizione diversa da quella di X_1, questa sia la coalizione 2. Quindi per X_2 ci sono solo due valori possibili: 1 se questa variabile deve appartenere alla stessa coalizione di X_1 o 2 se deve far parte di una coalizione nuova.
Per X_3 il discorso è simile: ci sono solo tre casi possibili: o questa deve far parte della coalizione a cui appartiene X_1, o deve far parte della coalizione a cui appartiene X_2, o deve far parte di una coalizione nuova (che, come sopra, non è restrittivo assumere sia la coalizione 3). Se poi X_1 ed X_2 hanno lo stesso valore, cioè 1, allora X_3 potra valere solo 1 o 2 (dato che già la coalizione 2 è vuota in quanto X_1 ed X_2 fanno entrambi parte della coalizione 1).
In generale si ha che ciascuna variabile nell'ordine potrà assumere solo i valori che rispettano la seguente formula:

$$1 \leq X_i \leq 1 + max_{1 \leq j \leq i-i} X_j$$

Considerando il caso peggiore, ogni variabile deve essere messa in una nuova coalizione, quindi si ha che la formula si riduce a:

$$1 \leq X_i \leq i$$

Lo spazio di ricerca è dato dalla produttoria del numero di valori possibili per ogni variabile, da cui deriva che, nel caso pessimo, tale spazio avrà una dimensione pari a n!.

2 L'algoritmo di risoluzione

2.1 Scelta del tipo di algoritmo

Per risolvere problemi di ottimizzazione come quello appena descritto ci sono due strade possibili: algoritmi di ricerca di tipo backtracking e algoritmi derivanti da tecniche di programmazione dinamica. Nella prima categoria c'è l'algoritmo branch and bound, che usa una funzione di upper bound (o lower bound, a seconda che il problema preveda di massimizzare o minimizzare una funzione obiettivo) per evitare di esplorare delle soluzioni che non possono portare a migliorare la migliore soluzione trovata finora. Nella seconda categoria è presente l'algoritmo di bucket elimination, che sostanzialmente cerca di isolare il contributo del valore di ciascuna variabile per quanto riguarda il valore della funzione obiettivo. Per fare questo, l'algoritmo trova il valore ottimo da assegnare ad una variabile per ottenere il migliore valore possibile della funzione obiettivo in funzione di ogni possibile valore assegnato alle altre variabili del problema. Se non sono presenti vincoli di arietà elevata e se la funzione obiettivo può essere decomposta in una serie di componenti che dipendono ciascuna da poche variabili, l'algoritmo di bucket elimination può essere estremamente efficiente.
Nel caso del problema considerato, si ha che sono presenti dei vincoli n-ari. Se si applicasse un algoritmo di bucket elimination al problema, si dovrebbe minimizzare il valore di una variabile in funzione dei valori delle altre n-1 variabili, effettivamente riducendo l'algoritmo di bucket elimination alle prestazioni di una semplice ricerca per forza bruta su tutti gli n^n assegnamenti possibili. Pertanto si è deciso di utilizzare un algoritmo di branch and bound.

2.1.1 Upper bound e lower bound

Per utilizzare un algoritmo di branch and bound occorre avere una funzione di upper bound (nel caso di un problema di massimizzazione). Questa funzione deve, dato un assegnamento parziale alle variabili del problema, determinare un upper bound al migliore valore della funzione obiettivo ottenibile estendendo tale assegnamento. Questo upper bound deve essere sempre maggiore o uguale al vero valore ottimo ottenibile estendendo l'assegnamento parziale, deve essere il più stretto possibile e deve essere calcolabile velocemente. Le caratteristiche della funzione di upper bound utilizzata sono il fattore in assoluto più importante per determinare le prestazioni dell'algoritmo.
Dopo molto tempo impiegato per cercare di trovare una funzione di upper

bound più stretta possibile, si è trovata la funzione descritta di seguito. Per ottenere un upper bound, si possono calcolare i LDF delle varie coalizioni considerando solo le variabili già istanziate. Quindi si cerca di stimare (per eccesso) quale può essere il contributo delle variabili non istanziate all'incremento del valore della funzione obiettivo. Non essendo semplice trovare un upper bound all'incremento complessivo ottenibile, si considerano le variabili indipendentemente. Si ha che l'incremento complessivo delle variabili non assegnate è non maggiore della somma degli incrementi massimi di ogni variabile non assegnata presa singolarmente. Quindi si ha che una stima per eccesso dell'incremento del valore della funzione obiettivo dopo aver aggiunto X_i alla coalizione generica non vuota j può essere:

$$\frac{max_t P_i(t)+\sum_{k:X_k=j} max_t P_k(t)}{\max_t\left(\left(\sum_{k:X_k=j} P_k(t)\right)+P_i(t)\right)} - \frac{\sum_{k:X_k=j} max_t P_k(t)}{max_t \sum_{k:X_k=j} P_k(t)} \leq$$

$$\leq \frac{max_t P_i(t)+\sum_{k:X_k=j} max_t P_k(t)}{max\left(min_t P_i(t)+max_t \sum_{k:X_k=j} P_k(t); max_t P_i(t)\right)} - \frac{\sum_{k:X_k=j} max_t P_k(t)}{max_t \sum_{k:X_k=j} P_k(t)} =$$

(facendo un'apposizione per semplicità)

$$= \frac{max_t P_i(t)+n(j)}{max(min_t P_i(t)+d(j); max_t P_i(t))} - \frac{n(j)}{d(j)} =$$

$$= \frac{d(j)*max_t P_i(t)+d(j)*n(j)-n(j)*max(min_t P_i(t)+d(j); max_t P_i(t))}{d(j)*(max(min_t P_i(t)+d(j); max_t P_i(t)))} \leq$$

$$\leq \frac{d(j)*max_t P_i(t)+d(j)*n(j)-n(j)*(min_t P_i(t)+d(j))}{d(j)*(max(min_t P_i(t)+d(j); max_t P_i(t)))} =$$

$$= \frac{d(j)*max_t P_i(t)+d(j)*n(j)-n(j)*min_t P_i(t)-n(j)*d(j)}{d(j)*(max(min_t P_i(t)+d(j); max_t P_i(t)))} =$$

$$= \frac{d(j)*max_t P_i(t)-n(j)*min_t P_i(t)}{d(j)*(max(min_t P_i(t)+d(j); max_t P_i(t)))} = \frac{max_t P_i(t)-\frac{n(j)}{d(j)}*min_t P_i(t)}{max(min_t P_i(t)+d(j); max_t P_i(t))} \leq$$

$$\leq \frac{max_t P_i(t)-LB\left(\frac{n(j)}{d(j)}\right)*min_t P_i(t)}{LB(max(min_t P_i(t)+d(j); max_t P_i(t)))}$$

Questo significa che, trovando dei lower bound per le quantità $\frac{n(j)}{d(j)}$ e $max\left(min_t P_i(t)+d(j); max_t P_i(t)\right)$, si può ottenere un upper bound per l'incremento del valore della funzione obiettivo dovuto ad X_i. Per la prima quantità un lower bound banale è 1, dato che la frazione non può mai avere un numeratore minore del denominatore. Per la seconda quantità si può notare che un lower bound è ottenibile trovando il più piccolo valore possibile per d(j). Quindi si possono semplicemente considerare tutte le varie coalizioni (eventualmente limitandosi a quelle a cui la variabile X_i può essere aggiunta senza violare qualche vincolo) e trovare quella con il valore

minimo di $d(j) = max_t \sum_{k:X_k=j} P_k(t)$. Questo è corretto in quanto, se si aggiunge una variabile ad una coalizione j, è garantito che il valore d(j) di quella coalizione non possa diminuire.
Nel caso poi siano presenti delle coalizioni vuote, non si può usare direttamente il metodo sopra descritto per calcolare un upper bound. Infatti, aggiungendo una variabile ad una colaizione vuota, l'incremento ottenibile è pari a 0, in quanto il LDF per una coalizione vuota vale 1, così come quello per una colaizione con una sola variabile. Tuttavia, se si suppone di aggiungere una variabile ad una coalizione vuota, si può avere la situazione in cui il valore di d(j) per quella coalizione con una sola variabile sia minore del valore di d(j) di tutte le altre coalizioni. Quindi, se sono presenti delle coalizioni vuote, occorre anche tenere conto di quale potrebbe essere il minimo valore di d(j) per quelle coalizioni aggiungendovi una variabile. Il minimo valore di d(j) che una coalizione vuota può assumere inserendovi una variabile è pari al minimo picco di consumo massimo di una variabile, poichè $d(j) = max_t \sum_{k:X_k=j} P_k(t) = max_t P_i(t)$. Quindi per ottenere un lower bound per d(j) se sono presenti delle coalizioni vuote si calcola il minimo tra i valori di d(j) per le coalizioni non vuote e il minimo picco di consumo massimo per una variabile non assegnata.
L'algoritmo necessita anche di un lower bound, cioè un valore che viene aggiornato ogni volta che si trova una nuova soluzione migliore della migliore trovata fino ad allora. Questo lower bound viene usato per tagliare delle parti dell'albero di ricerca che non possono portare a trovare delle soluzioni migliori della migliore già trovata. Dato che il load diversity factor di una coalizione non può mai essere minore di 1, inizialmente si può assumere come lower bound il valore che si ottiene assumendo che il LDF di ogni coalizione valga 1. Per evitare che, nel caso in cui l'unica soluzione sia quella che mette un solo utente in ogni coalizione, l'algoritmo si rifiuti di esplorare l'unica soluzione possibile perchè il suo upper bound non è migliore del lower bound, si decrementa il lower bound iniziale di una unità. Se si dispone di una soluzione del problema (anche non ottimale), si può calcolare il valore della funzione obiettivo per quella soluzione ed usare tale valore come lower bound iniziale.

2.2 Caratteristiche dell'algoritmo

L'algoritmo implementa un sistema di forward checking per propagare gli effetti dell'assegnamento di una variabile ai domini delle variabili non ancora assegnate. Per la forma dei vincoli, quando si assegna un valore ad una nuova variabile X_i, l'unico valore che può diventare proibito per le variabili

non assegnate è lo stesso valore che viene assegnato ad X_i. Questo perchè i vincoli dicono semplicemente che il consumo cumulativo in ogni coalizione non può superare un massimo stabilito. Se una variabile X_k può essere inserita nella coalizione j senza che sia violato nessun vincolo, assegnando l ad X_i (con $j \neq l$) si potrà ancora assegnare j ad X_k.
Per tenere traccia di quali valori sono stati proibiti da ciascun assegnamento di variabili si usa un insieme di mappe di bit, una per ogni livello dell'albero di backtracking. Consideriamo il livello k e sia $X_i = j$ l'assegnamento corrente a questo livello. La mappa di bit al livello k ricorderà solo quali variabili hanno avuto il valore j proibito al livello k (cioè se il valore j era già proibito per un assegnamento ad un livello superiore, la mappa di bit al livello k non lo memorizza). Questo permette di creare una mappa complessiva dei valori proibiti per ciascuna variabile non assegnata, ma permette anche di ripristinare lo stato dei domini di ciascuna variabile quando si deve effettuare un backtracking.
Ogni volta che riesce ad instanziare con successo una variabile, l'algoritmo sceglie dinamicamente la prossima variabile da istanziare. Con il forward checking si tiene traccia per ogni variabile non istanziata di tutti i valori proibiti a causa degli assegnamenti delle variabili già istanziate. Quando si deve scegliere la successiva variabile da istanziare, si sceglie la variabile con il più alto numero di valori proibiti. Questo fa si che se un assegnamento parziale impedisce di trovare dei valori per un insieme di variabili, è garantito che una di tali variabili venga scelta come prossima variabile da istanziare, permettendo dunque di accorgersi immediatamente dell'inconsistenza. Questo perchè ovviamente una variabile con tutti i valori proibiti avrà un numero di valori proibiti maggiore di quello di qualsiasi variabile che ha almeno un valore ammissibile. Questa euristica "most constrained variable" cerca anche di spingere le ramificazioni dell'albero di ricerca più in basso, verso le foglie dell'albero. Questo non riduce il numero di foglie, ma almeno cerca di ridurre il numero di nodi interni dell'albero.
É anche presente un sistema di riordino dei valori da assegnare a ciascuna variabile. Quando si sceglie la successiva variabile da istanziare, si estraggono tutti i valori non proibiti per quella variabile, per ciascuno di essi si calcola un upper bound e si salvano solo quelli con un upper bound migliore del lower bound corrente. Quindi si riordinano i valori secondo il relativo upper bound in ordine decrescente. In questo modo l'algoritmo rimane di tipo depth-first, ma ad ogni nodo può scegliere di esporare i figli nell'ordine più promettente. Quando si deve scegliere il successivo valore da assegnare ad una variabile, si recupera l'indice del valore vecchio, lo si incrementa, e si verifica se l'upper bound del nuovo valore è ancora valido o se nel frattempo il lower bound è aumentato. Nel primo caso si procede con l'assegnamento,

nel secondo caso si esegue un backtracking, dato che anche tutti i valori successivi avrebbero un upper bound inferiore al lower bound corrente.
Nell'algoritmo non sono stati implementati meccanismi di backjumping a causa della presenza di vincoli n-ari. Si è notato infatti che la presenza di tali vincoli, unitamente al riordinamento dinamico delle variabili, avrebbe reso inutile un algoritmo di backjumping. Infatti tale algoritmo non avrebbe mai effettuato salti all'indietro di più di una variabile, risultando effettivamente equivalente ad un semplice backtracking. Il Gashing's backjumping, infatti, genererebbe come conflict set l'insieme di tutte le variabili assegnate prima della variabile di fallimento, effettivamente saltando alla variabile immediatamente precedente. Il graph based backjumping, invece, marcherebbe come parenti di ogni variabile tutte le variabili antenati del nodo di fallimento nell'albero, finendo ancora per saltare alla variabile immediatamente precedente al nodo di fallimento.
L'algoritmo è stato creato facendo attenzione ad evitare di utilizzare l'allocazione di memoria durante il ciclo di elaborazione principale e tutte le procedure invocate da questo (salvo quando si deve comunicare con un'eccezione che la computazione è terminata). In questo modo si elimina l'overhead per l'allocazione della memoria durante l'esecuzione e si favorisce la possibilità che tutti gli oggetti usati dall'algoritmo possano essere mantenuti in cache, migliorando le prestazioni.

3 Input e output

L'algoritmo necessita in input di un insieme di utenti di cui sia noto il profilo di consumo dell'energia, del numero massimo di coalizioni utilizzabili e del massimo consumo aggregato di energia per ciascuna coalizione. I dati degli utenti vengono caricati da dei file CSV in cui sono presenti valori di uso dell'energia in funzione del tempo per numerosi utenti reali. Questi dati di uso sono accompagnati dal numero di serie dell'utenza e dal giorno a cui corrisponde ciascun insieme di dati.
Si è quindi creato un oggetto capace di aprire un file CSV e leggere da questo il numero di utenti specificato, copiandone quindi i dati all'interno di oggetti appositi. Una difficoltà nel realizzare questa operazione è dovuta al fatto che alcuni dati di consumo sono errati, riportando o il valore 0 oppure dei valori decisamente esagerati. Pertanto si è aggiunto del codice per eliminare quegli insiemi di dati che presentano o dei valori a 0, oppure dei salti eccessivi tra due valori successivi. Quando si trova una linea di dati che non presenta errori, si usa tale linea per creare i dati di un utente e quindi si ignorano tutti i dati finchè non si raggiunge un diverso numero di serie dell'utenza nel file CSV. Questo per evitare di creare vari utenti usando i dati dello stesso utente reale in differenti giorni.
Il numero di utenti da caricare e il percorso del file da cui caricare i dati vengono prelevati da un'interfaccia grafica, che permette anche di specificare il numero massimo di coalizioni e il massimo consumo aggregato per queste. L'interfaccia grafica permette di navigare graficamente nel file system fino a trovare il file .CSV da cui caricare i dati. A tale scopo la finestra di selezione dei file di default nasconde tutti i file con estensione diversa da .CSV.
Man mano che gli utenti vengono caricati, il sistema assegna a ciascuno un colore progressivo. Questo viene utilizzato per rappresentare l'utente in un grafico a barre che mostra per ciascuna coalizione la somma dei consumi in ogni istante di tempo. É possibile visualizzare l'identificatore a cui corrisponde ciascun colore ed anche modificare manualmente il colore di ciascun utente prima dell'inizio dell'esecuzione dell'algoritmo. Questo può essere utile per avere un'idea immediata di dove viene allocato un particolare utente già nella rappresentazione grafica della soluzione che viene aggiornata durante l'esecuzione dell'algoritmo.
Durante l'esecuzione dell'algoritmo, la migliore soluzione trovata fino a quel momento viene visualizzata nella finestra dell'interfaccia grafica. Questa presenta una fascia verticale sul margine destro in cui sono presenti tanti rettangoli quanti sono le coalizioni utilizzabili dall'algoritmo. In ciascun rettangolo vengono rappresentati i consumi cumulativi per quella coalizione ad ogni istante di tempo tramite delle barre verticali di vari colori. La lunghez-

za di ogni colore rappresenta il contributo di ciascun utente al consumo in quell'istante. Se non è possibile trovare nessuna soluzione, il programma visualizza un messaggio corrispondente nella zona della finestra corrispondente alla migliore soluzione trovata. Notare che i segmenti di barre verticali usati per rappresentare i consumi dei vari utenti vengono calcolati come valori floating point, ma poi per essere visualizzati devono essere convertiti a valori interi. Quindi si può verificare una perdita di precisione nella visualizzazione su schermo delle soluzioni, in particolare nel caso di utenze con valori di consumo molto piccoli.
L'interfaccia grafica mostra anche una barra che rappresenta l'avanzamento del processo di computazione per trovare la soluzione ottimale. Questa barra è aggiornata come segue. Si sceglie a priori un livello dell'albero di backtracking a cui si vuole aggiornare il valore visualizzato dalla barra. Se questa quantità ha un valore piccolo, la barra verrà aggiornata solo quando si cambia il valore di una variabile molto vicino alla radice dell'albero di ricerca, cosa che significa che la barra non verrà aggiornata molto spesso. Viceversa se si sceglie un valore troppo elevato, la barra verrà aggiornata troppo di frequente, causando una potenziale perdita di prestazioni. Si è pertanto deciso di scegliere un valore pari a 6 per questa soglia.
Per calcolare il valore da visualizzare sulla barra, ogni volta che si assegna un nuovo valore alla variabile al livello corrispondente a quello scelto per aggiornare la barra di progresso, si calcola il contributo di tutte le variabili assegnate a livelli minori od uguali a quello corrente. Questo schema permette di fornire alcune indicazioni all'utente per quanto riguarda il progresso della computazione, ma in certi casi non è molto utile. A volte succede infatti che, con problemi con un certo numero di utenti, si raggiunge il livello a cui la barra viene aggiornata con tutte le variabili che valgono 0, cosa che fa sì che la lunghezza della barra sia anch'essa 0. Quindi l'algoritmo inizia a lavorare con variabili a livelli maggiori (più lontane dalla radice) senza mai modificare i valori delle variabili in cima all'albero. Quando poi si finisce di lavorare con le variabili a livelli superiori, si esegue un backtracking e si verifica che non è possibile assegnare in nessun'altro modo le variabili più vicine alla radice dell'albero senza violare dei vincoli e quindi l'algoritmo termina senza mai aggiornare la barra. Questo problema può essere arginato aumentando il livello a cui si aggiorna il valore della barra, avendo però anche un corrispondente peggiornamento delle prestazioni dovuto all'overhead di dover aggiornare più spesso l'interfaccia grafica. Inoltre, dato che spesso l'algoritmo passa molto tempo prima di visualizzare un valore maggiore di 0 sulla barra, si è deciso di visualizzare la barra nello stato indeterminato finchè questa non viene aggiornata con un valore maggiore di 0 per la prima volta.

L'applicazione attualmente non esporta la soluzione trovata (se non visualizzandola sullo schermo in maniera grafica), tuttavia la soluzione viene salvata internamente sotto forma di un vettore di coalizioni contenenti gli utenti suddivisi nella maniera ottima trovata. Se si volesse esportare tale soluzione, sarebbe relativamente facile estrarre questa soluzione e scriverla su un file o salvarla in qualche altra maniera.

4 Risultati ottenuti

Dalle prove effettuate, l'algoritmo riesce a risolvere in tempi ragionevoli istanze con fino a circa 20 utenti, indipendentemente dal numero e dalla dimensione massima delle coalizioni disponibili. Con dei numeri di utenti maggiori di questi, il numero e la capacità delle coalizioni diventano determinanti per il tempo di esecuzione.
I problemi a risolvere problemi di dimensioni maggiori derivano molto probabilmente dal fatto che la funzione di upper bound trovata non è molto stretta. La forma della funzione obiettivo, infatti, rende difficile ottenere un upper bound stretto, obbligando a ripiegare su degli upper bound molto larghi. Riuscendo ad ottenere un miglioramento in questo ambito, non sarebbe difficile ipotizzare prestazioni accettabili anche con un numero di utenti maggiore. In effetti, se fosse possibile avere una funzione di upper bound capace di fornire sempre il valore esatto della migliore soluzione ottenibile con l'assegnamento parziale considerato, allora sarebbe possibile trovare la soluzione ottima del problema in tempo lineare nel numero delle variabili. Si comprende quindi che la qualità della funzione di upper bound è determinante ai fini delle prestazioni.
Una cosa che è stata osservata è che l'interfaccia grafica non sembra riuscire a gestire il rateo con cui l'algoritmo invia le soluzioni da visualizzare man mano che queste vengono individuate. In effetti i thread che gestiscono internamente l'interfaccia grafica lanciano periodicamente delle eccezioni, probabilmente derivanti da qualche race condition indotta dal rapido aggiornamento degli oggetti che costituiscono i grafici a barre che rappresentano la migliore soluzione trovata.
Per cercare di stabilire quale sia la qualità delle soluzioni trovate da questo algoritmo, si è confrontato l'output dell'algoritmo ottimo descritto in questa relazione con l'output di due algoritmi non ottimi: il disporre casualmente gli utenti nelle coalizioni e l'algoritmo che mette ogni utente in una coalizione separata. Per ogni dato mostrato nei grafici sono state eseguite dieci misurazioni con il programma, annotando tutti i risultati in una tabella. Da questa sono poi stati generati i grafici calcolando media e varianza di ogni serie di misurazioni. Il valore minimo di 10 utenti è stato scelto per cercare di evitare di generare dati per campioni di utenti troppo piccoli, che potrebbero fornire valori anomali; il valore massimo di 18 utenti è stato dettato dal tempo di esecuzione dell'algoritmo, che per valori maggiori di 18 rendeva difficile eseguire dieci misurazioni in tempi ragionevoli.

I primi due grafici mostrano il miglioramento del valore del load diversity factor passando da una soluzione non ottima a quella ottima. In questi test

il numero di coalizioni disponibili per l'algoritmo è pari al numero di utenti (dimensione delle coalizioni costante e pari a 5). Si nota come la soluzione che mette ogni utente in una coalizione separata sia peggiore della soluzione ottenuta disponendo gli utenti casualmente nelle varie coalizioni, in quanto l'incremento del valore del load diversity factor passando alla soluzione ottima è maggiore. Questo perchè se si dispongono gli utenti in maniera casuale, si può comunque riuscire ad incastrare i dati di consumo degli utenti in modo da aumentare il load diversity factor della soluzione, mentre se gli utenti sono disposti tutti in coalizioni separate, il load diversity factor della solzione sarà pari al numero delle coalizioni.
La varianza degli incrementi risulta molto contenuta, non superando mai l'1% circa. Nel caso della soluzione con ogni utente in una coalizione separata, poi, la varianza è particolarmente limitata. Questo potrebbe essere utile se si volesse usare un algoritmo non ottimo, in quanto permetterebbe di avere un'idea abbastanza precisa di quanto non ottimale possa essere la soluzione trovata.

L'ultimo grafico mostra il miglioramento del valore del load diversity factor passando dalla soluzione casuale a quella ottima se si mantiene costante il numero di coalizioni (fissato a 5) e si incrementa la dimensione di queste in maniera proporzionale al numero di utenti presenti. In questa situazione si può notare come l'incremento relativo del valore della soluzione sia significativamente maggiore di quello ottenuto con un numero di coalizioni variabile. Questo probabilmente è dovuto al fatto che, con poche coalizioni a disposizione, è difficile che una disposizione casuale degli utenti riesca a produrre una soluzione con un load diversity factor particolarmente elevato. Notare che non è possibile effettuare il test con il numero di coalizioni costante per l'algoritmo che mette ogni utente in una coalizione diversa perchè tale algoritmo richiede che sia presente un numero di coalizioni almeno uguale al numero di utenti.
Da questi grafici si nota quindi come il miglioramento del valore del load diversity factor sia perlopiù compreso tra il 15% e il 30% rispetto ad una soluzione non ottima e calcolabile in tempi molto brevi. In una situazione reale, occorre quindi valutare caso per caso se convenga accettare la qualità inferiore di una soluzione non ottima o se sia necessario accettare il carico computazionale richiesto da una soluzione ottima.

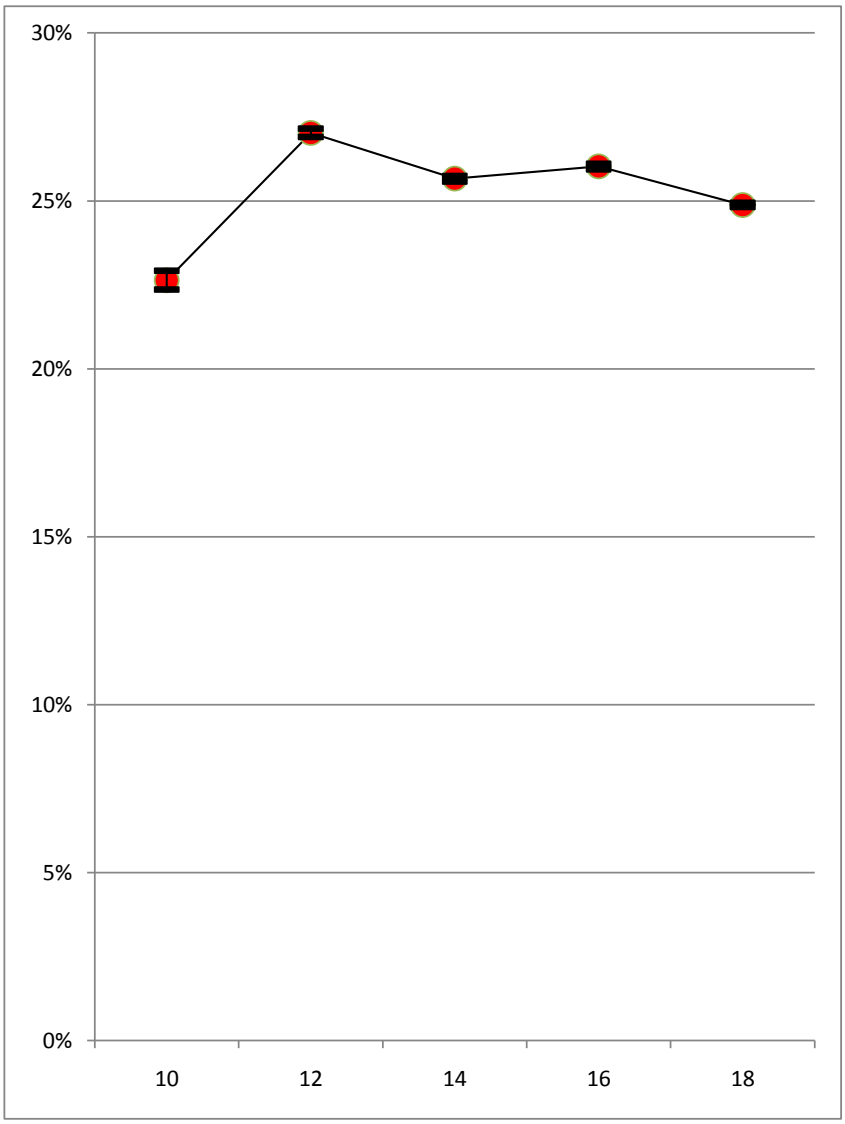

Figura 1: Miglioramento relativo del load diversity factor passando da un assegnamento che mette ogni utente in una coalizione separata all'assegnamento ottimo

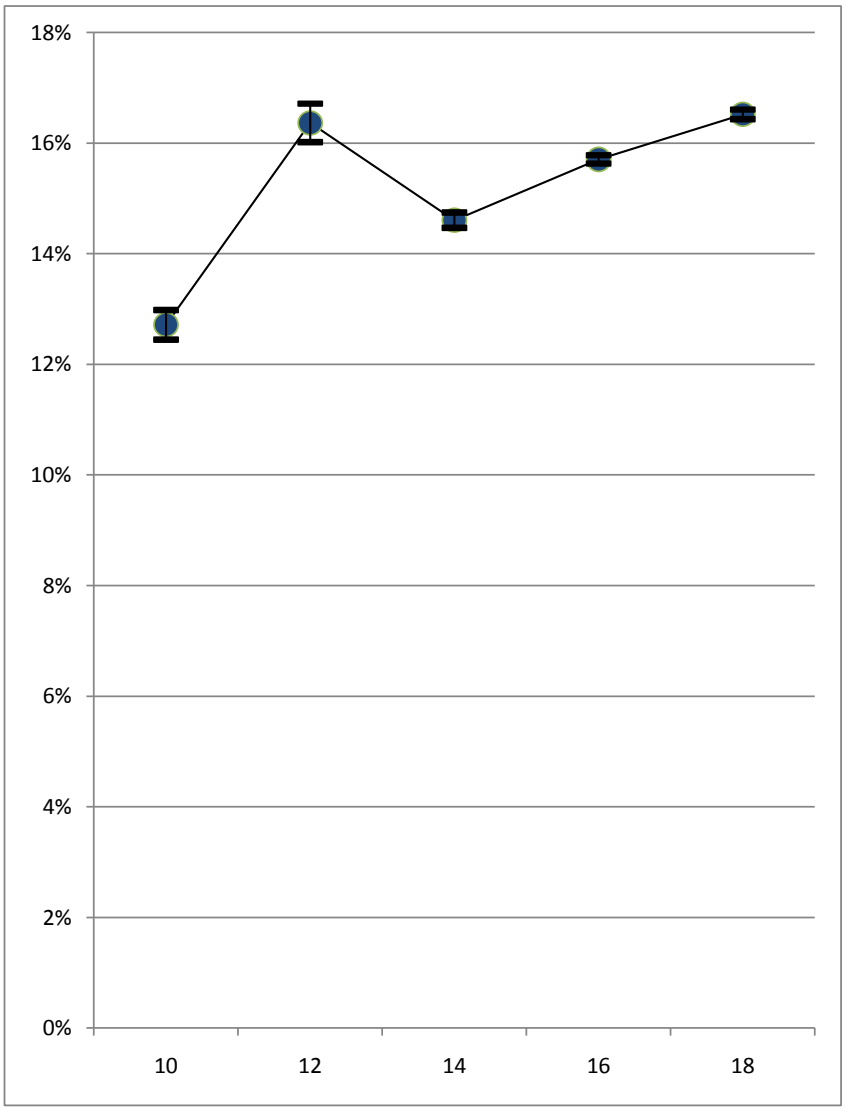

Figura 2: Miglioramento relativo del load diversity factor passando da un assegnamento casuale degli utenti all'assegnamento ottimo

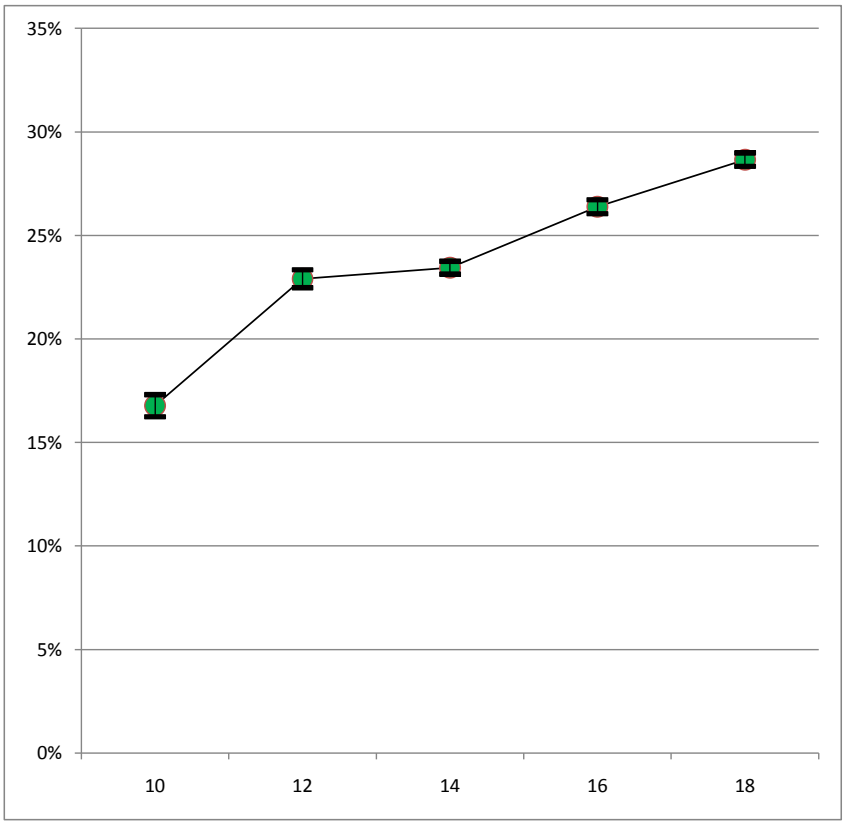

Figura 3: Miglioramento relativo del load diversity factor passando da un assegnamento casuale degli utenti all'assegnamento ottimo con un numero di coalizioni costante