UNIVERSITÀ DEGLI STUDI DI PADOVA

Dipartimento di Ingegneria dell'Informazione

Corso di Laurea Triennale in Ingegneria Informatica

Tesi di Laurea

Implementazione di un euristico basato su sub-MIP per la programmazione lineare intera

Relatore

Prof. Domenico Salvagnin

Laureando

Juri Farruku

Anno Accademico 2023/2024

Data di laurea: 26/09/2024

Sommario

Nel seguente elaborato si descrive l'implementazione di un algoritmo euristico per la programmazione lineare (MIP) noto in letteratura come Alternating Criteria Search. Tale euristico è basato sulla risoluzione di una sequenza di sub-MIP, cioè sottoproblemi del problema originario che vengono risolti a scatola chiusa con un risolutore MIP. Vengono presentati risulati sul testset MIPLIB2017 e con il risolutore MIP CPLEX.

Indice

Capitolo 1

Introduzione

La risoluzione di problemi MIP [1] risulta essere molto interessante nel mondo dell'ottimizzazione matematica, sia poichè essi presentano peculiari caratteristiche geometriche, utili all'ottenimento di un risultato ottimo in maniera efficace, sia per la capacità espressiva del suddetto paradigma, il quale è in grado di descrivere molteplici problemi di interesse. All'aumentare dei vincoli e delle variabili utilizzati in un modello MIP, aumenta la complessità, rendendo così più lenta la ricerca di un risultato ottimo. Per questo, in molti casi, si adotta un approccio di tipo euristico, dove non si esamina l'intero insieme delle soluzioni ammissibili per un dato problema, ma se ne cerca una che viene definita *localmente ottima*. In questo elaborato si descrive l'implementazione di un algoritmo euristico, sviluppato tramite l'ausilio del risolutore CPLEX [2], che si basa sulla risoluzione di sub-MIP.

1.1 Mixed Integer Programming

Un problema di programmazione lineare intera è composto da un insieme finito di vincoli lineari, un insieme finito di variabili, alcune delle quali devono poter assumere solo valori interi, e una funzione lineare, detta *funzione obiettivo*, la quale deve essere minimizzata. Tipicamente i problemi MIP vengono rappresentati utilizzando due forme molto comuni: la *forma standard*, dove i vincoli sono solo uguaglianze e le variabili devono essere non negative, e la *forma canonica*, dove le variabili devono sempre essere non negative, ma i vincoli sono solo disuguaglianze di tipo maggiore o uguale. Di seguito si mostra un esempio grafico di un MIP in forma standard:

$$\begin{cases} \min c^T \mathrm{x} \\ \mathrm{Ax} = \mathrm{b} \\ x \geq 0 \\ x_j \in \mathbb{Z} \ j \in J \end{cases} \tag{1.1}$$

Nel caso in cui tutte le variabili del problema fossero intere, si sta trattando un problema di programmazione lineare intera *pura*, al contrario qualora questa condizione non sussistesse si può parlare di programmazione lineare intera *mista*. La condizione di linearità dei vincoli può non sembrare importante, ma essa permette di poter descrivere i problemi MIP attraverso una formulazione geometrica precisa, rendendo così più interessanti e semplici da risolvere modelli

aderenti a tale paradigma. Ogni vincolo di disugualianza individua difatti un *semispazio affine*, ovvero una metà della divisione dello spazio data dall'iperpiano affine, identificato dallo stesso vincolo ponendolo come uguaglianza invece che disuguaglianza. Intersecando tutti questi iperpiani e semispazi si otterrà pertanto un *politopo*, ovvero un poliedro limitato come mostrato in Figura 1.1, al cui interno si trova la soluzione ottima del problema in analisi.
La condizione di interezza complica la ricerca della soluzione ottima rispetto ad un problema di programmazione lineare: infatti in quest'ultima la soluzione ottima è uno dei vertici del politopo individuato, limitando così la ricerca ad essi. In realtà ogni problema MIP può essere formulato affinchè tutti i vertici del politopo in questione siano soluzioni intere, descrivendolo così in una forma definita *Convex Hull*, ma questa nuova rappresentazione può risultare in un aumento esponenziale dei vincoli del problema, rendendo così inutile la nuova formulazione. Molti algoritmi allora, anzichè riformulare totalmente il problema originario, cercano di ottenere anche solo parzialmente questa formulazione ideale tramite diversi espedienti.

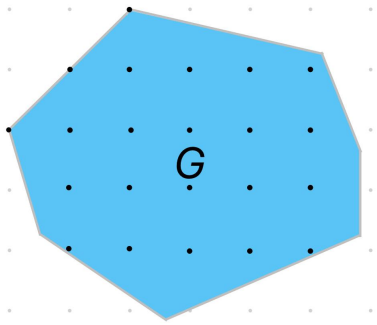

Figura 1.1: Rappresentazione grafica di un politopo associato ad un problema MIP.

1.1.1 Algoritmo Branch & Bound

L'algoritmo Branch & Bound utilizza come base un *rilassamento* del problema MIP originale, ovvero una versione semplificata che presenta meno vincoli, aumentando così la grandezza dell'insieme delle soluzioni ammissibili e rendendo perciò più semplice la ricerca di una di esse. L'algoritmo può essere reso ancora più efficace se si è in possesso di una soluzione ammissibile nota a priori, il cui costo prende il nome di *incumbent*. Tipicamente il vincolo che viene eliminato è quello di interezza, rendendo così possibile l'ottenimento di soluzioni non intere. Una volta risolto il rilassamento lineare del nodo radice si possono presentare tre diversi casi:

1. Si trova una soluzione ottima intera appartenente all'insieme originario delle soluzioni ammissibili o che ha un costo uguale all'incumbent: in questo caso la ricerca si conclude.
2. Il rilassamento è impossibile: allora anche il problema originario è impossibile.
3. Si ottiene una soluzione ottima non intera.

Nell'ultimo caso, l'ottenimento di una soluzione non intera viene utilizzato per adottare una regola di *branching*, ovvero si divide l'insieme delle soluzioni ammissibili del problema in due, dove in una metà viene aggiunto un vincolo aggiuntivo del tipo $x_j \leq \lfloor x_j \rfloor$ mentre nell'altra il vincolo aggiuntivo sarà $x_j \geq \lceil x_j \rceil$, dove in entrambi i casi x_j sarà una variabile che presenta un valore non intero. Questo metodo permette la divisione e il conseguente restringemento dello spazio di ricerca, come si può evincere dalla Figura 1.2.

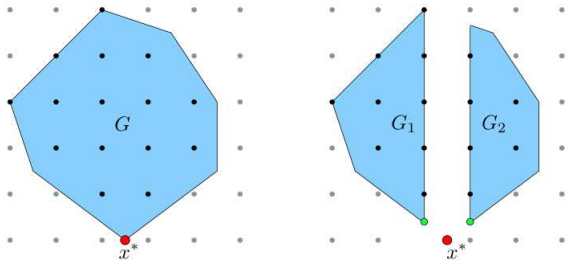

Figura 1.2: Rappresentazione grafica di un caso di branching, dove a sinistra si ha il politopo iniziale mentre a destra si ottengono i due spazi di ricerca separati in seguito ad un branching sulla variabile x^*.

E' doveroso notare come una volta ottenuta una soluzione ottima intera di uno dei sottoalberi ottenuti dal branching, gli altri nodi dell'albero vadano comunque visitati e risolti in quanto potrebbero contenere una soluzione ottima migliore di quella precedentemente trovata. Se questa soluzione dovesse avere un costo migliore dell'incumbent, essa sostituirà il suddetto, altrimenti se il nodo non dovesse presentare soluzioni o la sua soluzione ottima dovesse avere costo maggiore dell'attuale incumbent, l'intero sottoalbero verrà scartato. Nel caso in cui il nodo in analisi non possa essere scartato, si procede con un altro branching come descritto precedentemente. Il procedimento continua in maniera iterativa fino a che il valore dell'incumbent non coincide con il valore del miglior rilassamento globlamente trovato. Se la procedura dovesse essere interrotta in anticipo, la differenza tra questi due valore prende il nome di *optimality gap*. Il processo dell'algoritmo può essere osservato nella Figura 1.3.

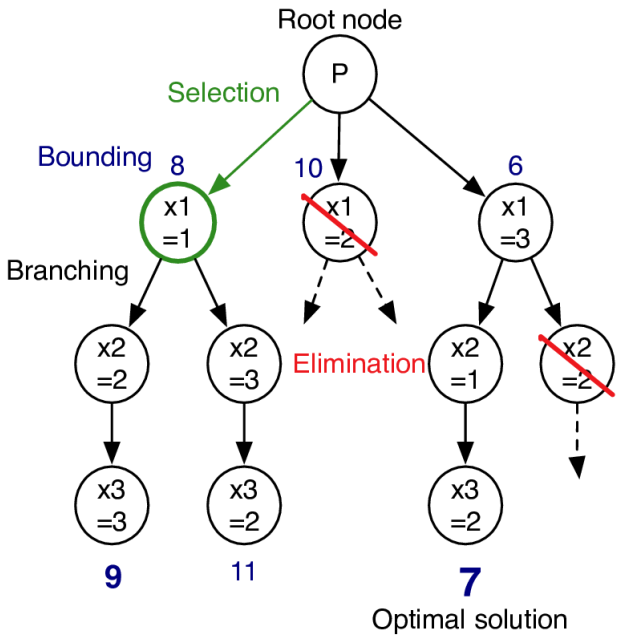

Figura 1.3: Caso di ricerca attuato dall'algoritmo Branch & Bound.

1.1.2 Algoritmo Branch & Cut

L'algoritmo Branch & Cut è un ampliamento dell'algoritmo descritto precedentemente: infatti esegue le stesse operazioni dell'algoritmo Branch & Bound ma con l'aggiunta dei *piani di taglio* ad ogni rilassamento lineare, come si può osservare nella Figura 1.4. Essi non sono altro che ulteriori

vincoli utilizzati per restringere in maniera ancora maggiore lo spazio delle soluzioni ammissibili del problema in analisi. Ciò permette di ottenere soluzioni intere in maniera più facile ma anche di diminuire il numero di branching necessario per ottenere una soluzione ottima.

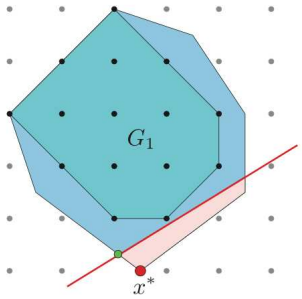

Figura 1.4: Esempio di utilizzo di un piano di taglio per ottenere un rafforzamento del rilassamento lineare.

1.2 Alternating Criteria Search

L'algoritmo di *Alternating Criteria Search* (ACS) [3] negli ultimi anni ha rappresentato una grande innovazione all'interno del panorama dell'ottimizzazione matematica. Questo tipo di approccio è largamente utilizzato in campi specifici, dove la presenza di molteplici vincoli e variabili rende la ricerca di una soluzione ottima molto lunga e costosa temporalmente. Infatti, grazie a questo tipo di algoritmo, è possibile ridurre notevolmente il tempo richiesto per ottenere una soluzione, mantenendo tuttavia un' ottima qualità per quanto riguarda la soluzione ottenuta. Tipicamente per ridurre ulteriormente il costo della computazione, questo tipo di algoritmi risolve un numero elevato di sub-MIP in parallelo. Oltre al parallelismo l'ACS fa uso di un' altra importante tecnica, ovvero la *Large Neighborhood Search* [4].

1.2.1 Large Neighborhood Search

La Large Neighborhood Search è di vitale importanza all'interno dell'algoritmo ACS. Grazie a questa tecnica si è in grado di esplorare un ampio spazio delle soluzioni modificando in maniera significativa la soluzione corrente, permettendo così di riuscire ad evitare di rimanere bloccati su un ottimo locale, ma al contempo di ottenere una soluzione di ottima qualità. Attraverso questo metodo si riesce ad esplorare lo spazio delle soluzioni più in profondità, analizzando anche punti che altrimenti verrebbero normalmente tralasciati. L' analisi del grande spazio di ricerca viene talvolta limitato in modo da non rendere eccessivamente dispensiosa la ricerca. Ciò è possibile grazie a diverse tecniche che filtrano alcune soluzioni ammissibili, a seconda del dominio che si sta analizzando.

1.2.2 Prestazioni

Recenti studi hanno dimostrato come l'ACS sia in grado di ottenere soluzioni di qualità migliori e in tempo minore rispetto al risolutore CPLEX se adattato a specifici domini, in quanto è possibile modificare i diversi criteri di ricerca in base al problema sotto analisi. Queste caratteristiche lo rendono un importante algortimo, adattabile a diversi problemi del mondo reale.

Capitolo 2

Sviluppo dell'algoritmo

2.1 Descrizione

La complessità di un dato problema MIP è strettamente correlato al numero di vincoli e variabili che esso presenta, pertanto una massiccia presenza di questi elementi può comportare un aumento del tempo richiesto per ottenere una soluzione ottima del modello. Per questo motivo, spesso e volentieri, il risolutore non viene applicato direttamente all' istanza originaria del modello, ma si adottano diverse strategie risolutive. Nel caso in analisi, l'algoritmo di Alternating Criteria Search sviluppato ha come punto di partenza una soluzione iniziale che, attraverso differenti criteri, viene migliorata tramite un processo iterativo. Questo tipo di approccio comporta un notevole risparmio di tempo, in quanto non viene esplorato tutto l'insieme delle soluzioni ammissibili, ma si restringe il campo ad un sottoinsieme più piccolo detto *neighborhood*, ottenendo così una soluzione che è localmente ottima, ma non globalmente. La soluzione localmente ottima viene utilizzata come nuovo punto di partenza e successivamente si ripete iterativamente il procedimento descritto precedentemente, cercando una soluzione sempre migliore e spostandosi all'interno dello spazio di ricerca, come si può osservare nella Figura 2.1.
Nel caso specifico si è deciso di risolvere i problemi MIP attraverso la risoluzione di una sequenza di sub-MIP, ovvero modelli che rappresentano un sottoinsieme del problema originale, ottenuti fissando un certo numero di variabili del modello. Per poter studiare l'effettiva efficacia dell'algoritmo, ad ogni iterazione è stato calcolato il miglioramento della soluzione ottenuta come gap *chiuso*, ovvero come diminuizione del primo valore di gap, calcolato grazie alla soluzione trovata al nodo radice. Il limite massimo di iterazioni che è stato scelto di far eseguire all'algoritmo è 10 e si è inoltre aggiunto un tempo limite globale di 30 minuti.

2.2 Variabili

Lo sviluppo dell'algoritmo si appoggia a diverse variabili utilizzate all'interno dello stesso. I due puntatori `CPXENVptr env` e `CPXLPptr mip` sono utilizzati per creare l'ambiente del modello MIP su cui verranno utilizzate le varie funzioni del software CPLEX. La variabile `status` viene utilizzata per ottenere il valore di ritorno delle funzioni di CPLEX, le quali restituiscono 1 in caso di errore e 0 in caso contrario. I due array `gaps` e `times` vengono adoperati per salvare rispettivamente i valori del gap di ottimalità e dei tempi di ogni iterazione, utilizzati in seguito per

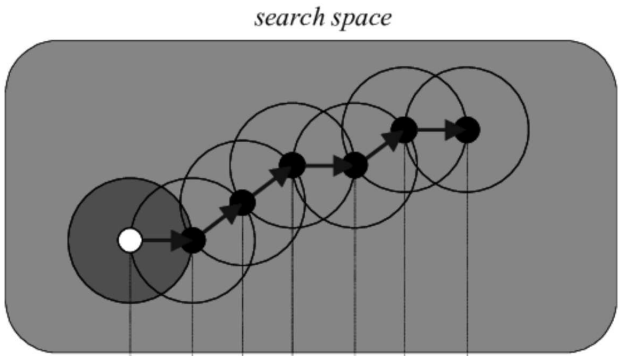

Figura 2.1: Rappresentazione grafica del cammino percorso da un algoritmo di ACS con tecnica di Large Neighborhood Search all'interno di un insieme di soluzioni ammissibili.

la composizione di due grafici. I puntatori a caratteri `set_bounds`, `set_ub` e `set_lb` contengono rispettivamente i caratteri 'B', 'U' e 'L' in ogni loro cella, e vengono allocati dinamicamente durante l'esecuzione, a seconda del numero di variabili che si è scelto di fissare. Essi risultano essere importanti poichè tramite la funzione `CPXchgbds` sarà possibile fissare e sfissare le variabili del modello. Infine, vi sono altri 4 importanti puntatori utilizzati all'interno del programma: `set_values`, `old_ub`, `old_lb` e `total_variables`: il primo conterrà il valore delle variabili che verranno fissate, il secondo e il terzo conterranno i valori dei lowerbound e upperbound iniziali del modello, così da poter sfissare le variabili in qualsiasi momento, mentre l'ultimo sarà un puntatore dove ogni elemento rappresenta l'indice di una variabile del problema, in ordine crescente.

2.3 Funzioni utilizzate

Sono state usate diverse funzioni appartenenti al software CPLEX per poter sviluppare l'algoritmo. La funzione `CPXgettime` è stata utilizzata per ottenere il tempo trascorso in diversi punti dell'esecuzione, per controllare che il tempo limite non venisse superato. Tramite `CPXgetub` e `CPXgetlb` è stato possibile ottenere gli upperbound e i lowerbound delle variabili del modello, mentre per ottenere il numero di queste ultime è stato necessario usare la funzione *CPXgetnumcols*. Fondamentali in molti casi sono state le funzioni `CPXsetintparam` e `CPXsetdblparam`, le quali riescono ad impostare certi particolari valori del risolutore. Ad esempio, la prima è stata utilizzata per impostare il numero di nodi massimo da poter risolvere dal risolutore di CPLEX, invocato tramite la funzione `CPXmipopt`. Per elaborare le soluzioni ottenute dal risolutore è stato necessario utilizzare altre tre diverse funzioni: `CPXgetobjval`, la quale restituisce il valore della soluzione ottima trovata, `CPXgetbestobjval`, per ottenere il dual bound e `CPXgetx` per ottenere i valori dei parametri una volta trovata una soluzione del modello.

2.4 Fase di set-up

Nella fase di set-up dell'algoritmo viene creato l'*environment* del problema tramite alcune funzioni del software CPLEX che permettono ad esempio di leggere direttamente da file i modelli del testset MIPLIB2017 [5], i quali sono stati scritti adottando le regole del formato *MPS*. Per maggiore leggibilità nei seguenti frammenti di codice sono stati omessi i controlli di errore, i quali rimandano alla label `TERMINATE`.

```
//Crea l'ambiente per il problema MIP, in caso di errore va a TERMINATE
env = CPXopenCPLEX (&status);

//Ottiene il tempo all'avvio dell'ambiente
CPXgettime(env, &start_time);

//Crea il problema usando il nome del file, in caso di errore va a TERMINATE
mip = CPXcreateprob(env, &status, argv[1]);

//Legge il problema dal file, tramite il nome passato da riga di comando
status = CPXreadcopyprob(env, mip, argv[1], NULL);
```

Listing 2.1: Fase di set-up.

Nel frammento di codice appena mostrato nel Listing 2.1 nella variabile `mip` viene creato il modello del problema con tutte le sue variabili e vincoli, mentre nella variabile `start_time` viene salvato il tempo iniziale, utilizzato poi per fermare la parte iterativa dell'algoritmo nel caso in cui superasse i 30 minuti. La label `TERMINATE` indica una parte di codice adibita per la deallocazione dei vari puntatori del programma in caso di errore segnalato all'interno della variabile status o una volta terminata l'esecuzione dell'algoritmo.

In seguito, come mostrato nel Listing 2.2, per ottenere una percentuale di variabili da dover fissare che sia consona per ogni diverso problema, si ottiene il numero di parametri del modello in analisi tramite le funzioni adibite e si crea la variabile `rate_fixed_variables`, dove si moltiplica il numero ottenuto precedentemente per 0.5, ottenendo così una percentuale del 50%.

```
//Ottiene il numero di colonne, ovvero di variabili, e crea l'array dove
    verranno   salvati i valori delle variabili una volta ottenuta una soluzione
int num_cols = CPXgetnumcols (env, mip);
int rate_fixed_variables = num_cols * 0.5;
double * x = (double *) malloc (num_cols * sizeof(double));
```

Listing 2.2: Creazione della percentuale di variabili fissate.

Per ottenere una soluzione ammissibile da dover poi migliorare tramite un approccio iterativo, viene impostato un numero di nodi adeguato al problema MIP da risolvere dal risolutore. Nel caso in questione questo numero si limita al solo nodo radice, rendendo così la prima risoluzione meno laboriosa computazionalmente. Si ottengono inoltre i *lowerbound* e *upperbound* originari delle variabili del problema, i quali verranno utilizzati in seguito per sfissare esse nel caso in cui nell'iterazione precedente fossero state fissate al valore della soluzione ottima trovata. Per la risoluzione del nodo radice viene settato un tempo massimo, calcolato grazie all'ausilio della funzione `count_remaining_time` mostrata nel Listing 2.4, in modo da non oltrepassare il tempo limite.

```
//Setta la sola soluzione del problema MIP al solo nodo radice
status = CPXsetintparam (env, CPXPARAM_MIP_Limits_Nodes, 0);

//Ottiene gli upperbound originali delle variabili del problema
old_ub = (double *) malloc (num_cols * sizeof(double));
```

```
6  status = CPXgetub(env, mip, old_ub, 0, num_cols-1);
7
8  //Ottiene i lowerbound originali delle variabili del problema
9  old_lb = (double *) malloc (num_cols * sizeof(double));
10 status = CPXgetlb(env, mip, old_lb, 0, num_cols-1);
11
12 CPXgettime(env, &current_time);
13
14 //Setta il tempo per la risoluzione al nodo radice
15 status = CPXsetdblparam(env, CPX_PARAM_TILIM, count_remaining_time(start_time,
       current_time));
```

Listing 2.3: Ottenimento di lower ed upperbound.

```
1
2  double count_remaining_time (double start_time, double current_time)
3  {
4     double remaining = 1800.0 - (current_time - start_time);
5     return remaining;
6  }
```

Listing 2.4: Funzione che calcola il tempo rimanente dai 30 minuti impostati come limite globale.

L'allocazione dei puntatori a caratteri chiamati `set_lb` e `set_ub` descritti precedentemente è stata realizzata tramite una funzione adibita, descritta nel Listing 2.6. L'array `total_variables` è stato direttamente allocato nel `main`, vista la brevità di scrittura.

```
1
2  //Crea un array utilizzato per specificare che i valori vengono utilizzati come
       lowerbound
3  set_lb = set_char_array(set_lb, num_cols, 'L');
4
5
6  //Crea un array utilizzato per specificare che i valori vengono utilizzati come
       upperbound
7  set_ub = set_char_array(set_ub, num_cols, 'U');
8
9  //Crea un array dove gli indici sono il numero di variabili del problema
10 total_variables = (int *) malloc (num_cols * sizeof(int));
11 while (j < num_cols)
12 {
13     total_variables[j] = j;
14     j++;
15 }
16 j = 0;
```

Listing 2.5: Allocazione degli array utilizzati per settare upperbound e lowerbound.

```
1
2  char * set_char_array (char * a, int size, char bound)
3  {
4     free (a);
5     int j = 0;
6     a = (char *) malloc (size * sizeof(char));
```

```
7
8     while (j < size)
9     {
10        a[j] = bound;
11        j++;
12    }
13
14    return a;
15 }
```

Listing 2.6: Funzione che alloca un array di char.

Infine viene risolto il problema al nodo radice tramite l'utilizzo del risolutore CPLEX. Si verifica se si è riusciti ad ottenere una soluzione osservando il numero di soluzioni trovate nel *pool*: in caso negativo si termina l'algoritmo, in caso contrario invece si salva la soluzione ottenuta come *bound primale*, si ottiene il *bound duale* e si calcola il *gap di ottimalità iniziale* tramite la funzione `calc_gap` mostrata nel Listing 2.8. Esso verrà in seguito calcolato ad ogni nuova iterazione, per osservare l'andamento dell'algoritmo e valutarne l'efficacia. Una volta ottenuti tutti questi valori, viene calcolato il tempo rimanente, si setta il numero di nodi massimi risolvibili ad ogni invocazione del risolutore, ovvero 200, e si comincia la fase iterativa dell'algoritmo. Viene inoltre salvato all'interno della variabile `first_node_time` il tempo impiegato dal risolutore per risolvere il problema al nodo radice. Questo valore risulterà importante per la creazione di uno dei due grafici utilizzati per analizzare l'algoritmo.

```
1
2  CPXgettime(env, &current_time);
3
4  //Setta il tempo per la risoluzione al nodo radice
5  status = CPXsetdblparam(env, CPX_PARAM_TILIM, count_remaining_time(start_time,
       current_time));
6
7  //Risolve il problema al nodo radice
8  status = CPXmipopt(env, mip);
9
10 //Calcola il tempo necessario a risolvere il mip al nodo radice
11 CPXgettime(env, &temp_time);
12 first_node_time = temp_time - current_time;
13
14 //Ottiene il numero di soluzioni trovate, se non ve ne sono termina il programma
15 number_solutions = CPXgetsolnpoolnumsolns(env, mip);
16
17 if (!number_solutions)
18     printf("Impossibile risolvere il problema al nodo radice\n");
19
20 //Salva il risultato del primo risultato ottenuto, in caso di fallimento termina
       il   programma
21 status = CPXgetobjval(env, mip, &first_result);
22
23 primal_bound = first_result;
24
25 //Trovo il dual bound
26 status = CPXgetbestobjval(env, mip, &dual_bound);
```

```
27
28 gaps[0] = calc_gap(primal_bound, dual_bound);
29
30 //Setta il  numero di nodi da risolvere a 200
31 status = CPXsetintparam (env, CPXPARAM_MIP_Limits_Nodes, 200);
32
33 //Calcola il tempo trascorso sino ad ora
34 CPXgettime(env, &current_time);
35 elapsed_time = current_time - start_time;
36
37 //Ottengo il tempo prima di iterare per 10 volte
38 CPXgettime(env, &temp_time);
```

Listing 2.7: Risoluzione del nodo radice ed ottenimento del primo gap di ottimalità.

```
1
2 double calc_gap(double primal_bound, double dual_bound)
3 {
4         double abs_primal = fabs(primal_bound);
5         double abs_dual = fabs(dual_bound);
6
7         if (abs_primal < 1e-6 && abs_dual < 1e-6)
8                 return 0.0;
9
10        if(primal_bound * dual_bound < 0)
11                return 1.0;
12
13        if(abs_primal > abs_dual)
14                return (fabs(primal_bound - dual_bound)/abs_primal);
15        else
16                return (fabs(primal_bound - dual_bound)/abs_dual);
17 }
```

Listing 2.8: Funzione che calcola il gap relativo.

2.5 Fase iterativa

La fase iterativa rappresenta la parte dell'algoritmo che sfrutta la risoluzione di diversi sub-MIP per poter ottenere ad ogni iterazione una soluzione localmente migliore di quella precedentemente ottenuta. Ciò si può facilmente ottenere tramite un ciclo *while*, il quale itera per un massimo di 10 volte e di 30 minuti totali. All'inizio di ogni iterazione si ottengono il tempo, per calcolare in seguito il tempo di ogni iterazione, e i valori dei parametri in prossimità della soluzione precedentemente trovata, per poi fissare alcune variabili casuali del problema a questi valori.

```
1 //Itera per 10 volte e per un massimo di 30 minuti
2 while (elapsed_time < 1800 && cont < 10)
3 {
4     //Ottengo il tempo ad inizio di ogni iterazione
5     CPXgettime(env, &init_loop_time);
6
7     //In caso fosse la prima volta che itera fissa il 50% per cento delle
          variabili
```

```
8       if (!cont)
9       {
10          set_bounds = set_char_array(set_bounds, rate_fixed_variables, 'B');
11          set_values = (double *) malloc ((rate_fixed_variables) * (sizeof(double)
                ));
12      }
13      //Si scrivono in x i valori delle variabili una volta ottenuta la soluzione
14      status = CPXgetx(env, mip, x, 0, num_cols-1);
15
16      //Si scrivono in x i valori delle variabili una volta ottenuta la soluzione
17      status = CPXgetx(env, mip, x, 0, num_cols-1);
18
19      random_variables = random_indexes(random_variables, rate_fixed_variables,
            num_cols);
```

Listing 2.9: Ottenimento dei valori delle variabili in corrispondenza di una soluzione ottenuta precedemente e creazioni di indici randomici per successivamente fissare dei parametri del modello.

Per ottenere degli indici randomici che rappresentano variabili casuali, è stata creata una funzione apposita, chiamata `random_indexes`.

```
1
2   int * random_indexes (int * a, int size, int max_index)
3   {
4      free(a);
5      int j = 0;
6      a = (int *) malloc (size * sizeof(int));
7
8      while (j < size)
9         {
10           int index = rand()%(max_index);
11           int presente = 0;
12           int i = 0;
13           while (i < j)
14           {
15              if(a[i] == index)
16              {
17                 presente = 1;
18                 break;
19              }
20              i++;
21           }
22           if(!presente)
23           {
24              a[j] = index;
25              j++;
26           }
27        }
28     return a;
29  }
```

Listing 2.10: Funzione che alloca un array di int e crea 'size' indici casuali tra 0 e `max_index`-1 tutti diversi tra loro.

Nel caso in cui l'algoritmo stia eseguendo un'iterazione diversa dalla prima, si sfissano le variabili fissate precedentemente, utilizzando i valori di upperbound e di lowerbound ottenuti precedentemente nella fase di set-up. In seguito si fissano nuove variabili, si setta il tempo di risoluzione massimo del risolutore e si risolve il nuovo sub-MIP con queste nuove impostazioni, ottenendo così una nuova soluzione.

```

//Si ottengo i valori delle variabili che verranno fissate
while (j < rate_fixed_variables)
{
set_values[j] = x[random_variables[j]];
j++;
}
j = 0;

//Si sfissano le variabili precedentemente fissate nel caso in cui questa sia una iterazione diversa alla prima
if (cont)
{
status = CPXchgbds(env, mip, num_cols, total_variables, set_lb, old_lb );
status = CPXchgbds(env, mip, num_cols, total_variables, set_ub, old_ub );
}

//Se ne fissano di nuove
status = CPXchgbds(env, mip, rate_fixed_variables, random_variables, set_bounds, set_values );

//Calcolo il tempo attuale
CPXgettime(env, &current_time);

//Setto il tempo ti risoluzione con il tempo rimanente dai 30 minuti, quindi 30 - (current_time - start_time)
status = CPXsetdblparam(env, CPX_PARAM_TILIM, count_remaining_time(start_time, current_time));


//Si risolve il subMIP con il limite di 200 nodi ed il tempo limite
status = CPXmipopt(env, mip);
```

Listing 2.11: Costruzione del sub-MIP fissando le variabili e risoluzione di essi.

L'ultima soluzione ottenuta verrà usata come nuovo primal bound e verrà utilizzata per ottenere il nuovo gap relativo. Sottraendo questo ultimo al valore precedentemente trovato si avrà a disposizione il miglioramento della soluzione in percentuale. Viene inoltre calcolato il tempo dell'iterazione. Tutte queste informazioni vengono scritte sul file di testo che verrà poi utilizzato per condurre l'analisi.

```

//Salva il risultato nella variabile result, in caso di errore va a TERMINATE
status = CPXgetobjval(env, mip, &result);

```

```
5  //Calcolo il miglioramento della soluzione in percentuale
6  gaps[i] = calc_gap(result, dual_bound);
7  fprintf(fp, "%f\n", gaps[i-1] - gaps[i]);
8
9
10 //Si ottiene da quanto tempo si sta iterando
11 CPXgettime(env, &current_time);
12 elapsed_time = current_time - start_time;
13
14 times[cont] = current_time - init_loop_time;
15
16 cont++;
17 i++;
18
19 for (j = 0; j < 10; j++)
20 {
21         fprintf(fp, "%f\n", times[j]);
22 }
23
24 goto TERMINATE;
25
26 }
```

Listing 2.12: Ottenimento del gap relativo e del tempo della iterazione con successiva stampa su file di testo.

Capitolo 3

Analisi dei risultati

3.1 Ottenimento dei risultati

Ogni istanza della MIPLIB2017 è stata sottoposta all'algoritmo 5 volte per ottenere una maggiore precisione e accuratezza. Per analizzare tutte queste istanze di benchmark è stato utilizzato il cluster del dipartimento. Il gruppo di ricerca operativa ha due cluster dedicati: il cluster *razor* ed il cluster *arrow*. L'ottenimento dei risultati è stato possibile grazie all'utilizzo di quest'ultimo, il quale è composto da 15 lame, ciasciuna dotata delle seguenti specifiche:

- Intel(R) Xeon(R) CPU E5-2623 v3 @ 3.00GHz
- 16 GB di RAM

Lo scheduler deputato alla gestione dei vari cluster è *SLURM* [6]. Per poter sottomettere i vari job allo scheduler, che li distribuirà poi in maniera autonoma tra le varie lame, è necessario utilizzare un comando particolare, visibile nel Listing 3.1.

```
sbatch --wckey=rop --requeue
```

Listing 3.1: Comando per sottomere un job allo scheduler SLURM.

Per ogni job è stato creato uno *script shell* diverso, la cui struttura però rimane la stessa e si può osservare di seguito:

```
#!/bin/bash
#SBATCH --job-name=prova_1
#SBATCH --partition=arrow
#SBATCH --ntasks=1
#SBATCH --mem=14GB
# warm up processors
sudo cpupower frequency-set -g performance
sleep 0.1
stress-ng -c 4 --cpu-ops=100
# set limits
ulimit -v 16777216
######################
./prova2 /home/farrukujur/tesi/file_test/30n20b8.mps.gz
```

```
14 ######################
15 # back to power saving mode
16 sudo cpupower frequency-set -g powersave
```

Listing 3.2: Script shell per un job da sottomettere a SLURM.

Nell'esempio appena proposto si può osservare come le prime righe, ovvero quelle la cui prima keyword è *#SBATCH*, servano ad impostare alcuni dei parametri SLURM, come ad esempio il nome del job o la partizione su cui si lavorerà, in questo caso *arrow*. Le successive invece servono per preparare il processore ad operare in modalità *performance*, sottoponendo la CPU in questione ad un piccolo job. Infine verrà scritto il comando vero e proprio da eseguire, con il conseguente ritorno alla modalità *power-saving* del processore.

Data la grande quantità di istanze da analizzare, è stato creato sia un semplice programma *Python* che creasse tutti gli script necessari, sia uno script per poter lanciare tutti e 240 i jobs in maniera immediata, descritto nel Listing 3.3.

```
1
2 #!/bin/bash
3
4 job_folder="/home/farrukujur/tesi/job_files"
5
6 job_name="job_array"
7
8 sbatch_options="--wckey=rop --requeue"
9
10 for index in {1..240}; do
11         sbatch $sbatch_options $job_folder/job_${index}.sh
12 done
```

Listing 3.3: Script shell per lanciare tutti i 240 jobs.

In seguito, ogni singolo dataset ottenuto è stato analizzato, creando per ognuno di essi un grafico diverso. Come si vedrà in seguito, in molti casi sono presenti iterazioni spurie causate da alcune delle istanze risolte, che possono contaminare l'analisi dell'andamento dell'algoritmo nella sua totalità. Perciò è stato creato un ulteriore grafico, il quale rappresenta la media di tutti e 5 i valori contenuti nei dataset, in modo da poter mitigare eventuali valori che possano compromettere l'analisi dell'andamento dell'algoritmo.

3.2 Risultati ottenuti

I risultati ottenuti tramite l'applicazione dell'algoritmo sulle 240 istanze della MIPLIB sono stati analizzati secondo due diversi punti di vista. Nel primo caso di interesse è stato calcolato il miglioramento medio del gap relativo ad ogni diversa iterazione. Nel secondo invece si è stati interessati a osservare questo miglioramento a seconda di diversi tempi *nodi radice*, ovvero il tempo trascorso è stato normalizzato utilizzando il tempo iniziale impiegato dall'algoritmo per risolvere il problema al nodo radice, usando perciò quest'ultimo come unità di misura temporale. Entrambi questi metodi sono stati adottati per studiare l'efficacia dell'algoritmo creato, difatti

queste analisi sono state condotte per capire per quanto tempo l'algoritmo possa riportare risultati interessanti. Le istanze che non sono state risolte al nodo radice sono state filtrate e scartate, non influendo così nello studio finale.

3.3 Grafici

Il grafico corrispondente all'analisi del miglioramento del gap relativo in base alle diverse iterazioni si può osservare in Figura 3.1. Come si può notare in questo caso, il grafico in questione presenta un valore particolarmente elevato alla settima iterazione, seppur nelle precedenti si possa osservare un andamento decrescente.

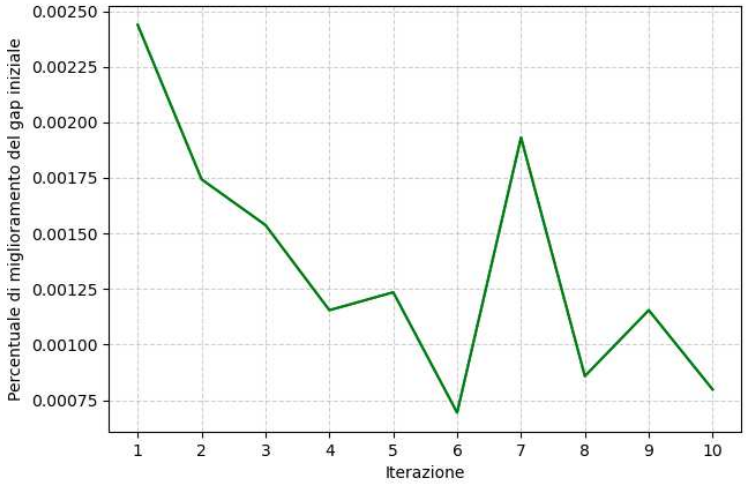

Figura 3.1: Grafico che rappresenta l'andamento medio del miglioramento del gap relativo in base all'iterazione per un solo dataset.

Nel grafico che espone la media dei valori ottenuti attraverso 5 diversi dataset, mostrato in Figura 3.2, si può notare invece come non vi siano iterazioni spurie. Infatti dalla figura si evince un andamento discendente abbastanza marcato, con le prime iterazioni più efficaci e le ultime meno.

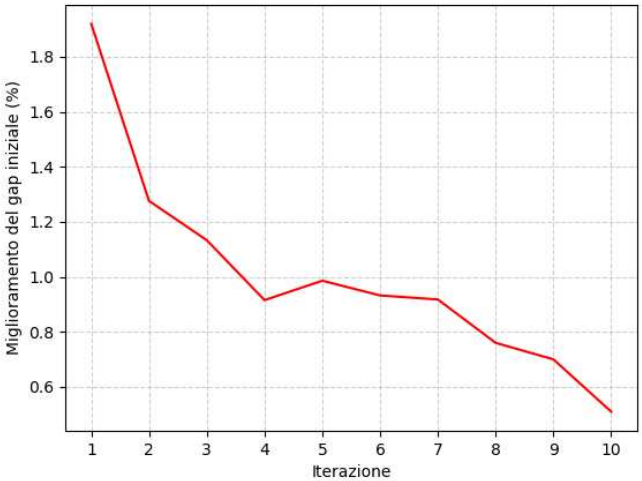

Figura 3.2: Grafico che rappresenta l'andamento medio del miglioramento del gap relativo in base all'iterazione per tutti e 5 i dataset.

Iterazione	Miglioramento per 1 dataset	Miglioramento medio per 5 dataset
Prima	0.0024386	1.9199272
Seconda	0.0017435	1.2763050
Terza	0.0015377	1.1332923
Quarta	0.0011547	0.9153575
Quinta	0.0012352	0.9858984
Sesta	0.0006936	0.9323438
Settima	0.0019318	0.9178280
Ottava	0.0008581	0.7628432
Nona	0.0011548	0.7001267
Decima	0.0007978	0.5097452

Tabella 3.1: Tabella contenente i valori di miglioramento del gap per un dataset ed il valore medio per 5 dataset.

Di seguito invece si può osservare in Figura 3.3 un grafico dove sull'asse delle ascisse è indicato il tempo normalizzato in nodi radice. Anche in questo caso si può notare in maniera abbastanza netta come l'algoritmo sia molto più efficace nei primi momenti di esecuzione, mentre con l'aumentare del tempo la sua proficuità cali in maniera notevole.

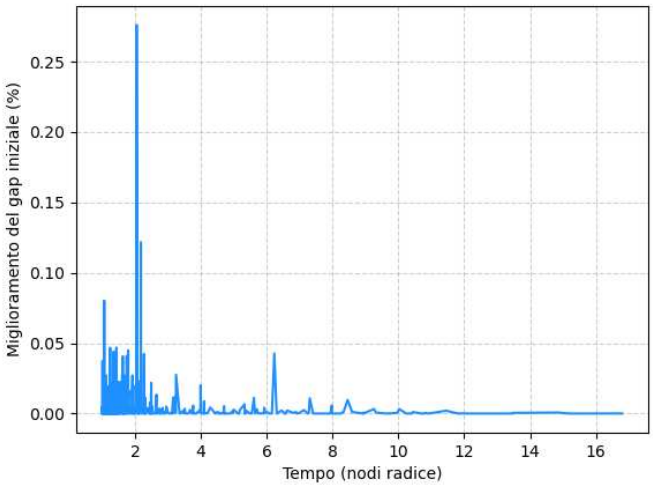

Figura 3.3: Grafico che rappresenta l'andamento medio del miglioramento del gap relativo in base al tempo.

Nella Tabella 3.1 è possibile osservare i risultati numerici ottenuti durante l'esperimento.

3.4 Possibili miglioramenti e approfondimenti

Per studiare in maniera più approfondita l'efficacia dell'algoritmo si potrebbero condurre diversi test, modificandone i parametri principali. Un maggiore numero di variabili fissate risulterebbe in un neighborhood più piccolo e perciò meno soluzioni ammissibili da sondare, al contrario si riscontrerebbe uno spazio di ricerca più ampio con un conseguente aumento di tempo richiesto ma con più possibilità di trovare soluzioni migliori. Una percentuale di variabili fissate che varia in base all'esecuzione dell'istanza potrebbe risultare utile nei casi in cui l'algoritmo per molte iterazioni incappasse nella stessa soluzione, aumentando così il gap relativo migliorato ad ogni iterazione. Incrementare il numero di iterazioni e il tempo limite massimo potrebbero

essere ulteriori strategie interessanti da analizzare per osservare se l'andamento riscontrato con 10 iterazioni e 30 minuti venga mantenuto anche con periodi diversi.

Capitolo 4

Conclusioni

Attraverso l'analisi dei diversi grafici si è potuto constatare come la maggior parte dell'efficacia dell'algoritmo risieda nelle sue prime iterazioni. Infatti all'inizio della procedura il miglioramento del gap iniziale ha valori nettamente più elevati rispetto a iterazioni successive o a tempi più elevati. Questo potrebbe indicare come, per aumentare l'utilità dell'algoritmo sviluppato in questo caso, non sia vantaggioso incrementare in maniera troppo drastica il numero di iterazioni o il tempo limite massimo in quanto questi ultimi non andrebbero ad inficiare in maniera sufficientemente netta sulla qualità della soluzione finale trovata. Un espediente più interessante da attuare invece potrebbe essere l'aumentare e diminuire il numero di variabili fissate, rendendo questo valore variabile in esecuzione e non fissato preventivamente.

Bibliografia

[1] D. Salvagnin. «Cenni di Programmazione Lineare Intera». (2024), indirizzo: https://www.dei.unipd.it/~salvagni/didattica/mip.

[2] IBM. «IBM ILOG CPLEX Optimization Studio». (2024), indirizzo: https://www.ibm.com/products/ilog-cplex-optimization-studio.

[3] Y. Shao, L.-M. Munguìa, S. Ahmed, D. A. Bader e G. L. Nemhauser, «Alternating criteria search: a parallel large neighborhood search algorithm for mixed integer programs.», *Computational Optimization and Applications*, 2018. indirizzo: https://doi.org/10.1007/s10589-017-9934-5.

[4] D. Pisinger e S. Røpke, «Large Neighborhood Search», *Handbook of Metaheuristics*, 2010. indirizzo: https://backend.orbit.dtu.dk/ws/portalfiles/portal/5293785/Pisinger.pdf.

[5] M. Miltenberger, D. G. Espinoza, S. Heinz et al., *MIPLIB 2017: Mixed-Integer Programming Library*, 2017. indirizzo: http://miplib.zib.de.

[6] SchedMD, *Simple Linux Utility for Resource Management*, SchedMD, 2023. indirizzo: https://slurm.schedmd.com/documentation.html.