Appendice C

CPLEX e Principali Funzioni

Questo appendice è stato pensato come semplice guida relativa all'utilizzo del software IBM ILOG CPLEX all'interno della tesina. I contenuti sono divisi in sezioni, ognuna delle quali cerca di corrispondere ad un modulo del progetto finale. Viene inoltre brevemente descritto un generico output generato dal programma durante il processo di ottimizzazione.
Inoltre saranno illustrate brevemente e in generale le modalità di utilizzo delle varie *routine* di CPLEX, per i dettagli del loro impiego, all'interno di questo lavoro, si rinvia ai relativi capitoli dell'elaborato, invece per quanto riguarda la descrizione riguardante la parametrizzazione dei metodi, che verranno citati, si rimanda alla documentazione ufficiale.

C.1 Creazione del modello

Il modello può essere creato tramite due approcci diversi: mediante un file specifico generato secondo una certa sintassi, o attraverso alcune *routine* di libreria. In entrambi i casi, le operazioni preliminari da eseguire sono le seguenti:

```
int err;
CPXENVptr env;
CPXLPptr lp;

// in err vengono salvati eventuali codici di errore
env = CPXopenCPLEX(&err);
// "tsp" definisce il nome del problema
lp = CPXcreateprob(env, &err, "tsp");
```

La prima operazione da effettuare (prima di qualsiasi altra chiamata alla libreria di CPLEX) è quella alla funzione CPXopenCPLEX, la quale genera il nuovo *environment* in cui verrà creato il modello. A questo punto è necessario definire il problema di programmazione lineare tramite la funzione CPXcreateprob, che ha il compito di crearne uno vuoto (senza vincoli e senza variabili).Come riportato precedentemente, è possibile scegliere di completare il modello a partire da un file di input, dal formato e sintassi specifici (*sav*, *mps* o *lp*), tramite la funzione CPXreadcopyprob. Succesivamente viene riportato un esempio di file lp.

```
\ENCODING=ISO-8859-1
\Problem name: TSP

Minimize
 obj: 408 x(1,2) + 209 x(1,3) + 628 x(1,4) + 585 x(1,5) + 614 x(2,3)
      + 449 x(2,4) + 561 x(2,5) + 811 x(3,4) + 727 x(3,5) + 222 x(4,5)
Subject To
 degree(1): x(1,2) + x(1,3) + x(1,4) + x(1,5) = 2
 degree(2): x(1,2) + x(2,3) + x(2,4) + x(2,5) = 2
 degree(3): x(1,3) + x(2,3) + x(3,4) + x(3,5) = 2
 degree(4): x(1,4) + x(2,4) + x(3,4) + x(4,5) = 2
 degree(5): x(1,5) + x(2,5) + x(3,5) + x(4,5) = 2
Bounds
 0 <= x(1,2) <= 1
 0 <= x(1,3) <= 1
 0 <= x(1,4) <= 1
 0 <= x(1,5) <= 1
 0 <= x(2,3) <= 1
 0 <= x(2,4) <= 1
 0 <= x(2,5) <= 1
 0 <= x(3,4) <= 1
 0 <= x(3,5) <= 1
 0 <= x(4,5) <= 1
Binaries
 x(1,2) x(1,3) x(1,4) x(1,5) x(2,3) x(2,4) x(2,5) x(3,4) x(3,5)
 x(4,5)
End
```

Listing C.1: File location.lp

Come si nota, il file fa riferimento a un problema TSP per cinque nodi, e la sua struttura è divisa in più sezioni: nella prima parte (*Minimize*) viene riportata la funzione obiettivo, e quindi i costi associati a ciascun lato; nella seconda parte (sotto a *Subject to*) vengono elencati i vincoli del problema, ai quali è possibile assegnare un nome per ciascuno (in questo caso *degree(x)*) e che qui corrispondono ai vincoli di grado dei singoli nodi; seguono la parte che definisce il dominio delle variabili e l'insieme delle variabili binarie (rispettivamente indicate da *Bounds* e *Binaries*).

La seconda opzione, come esposto precedentemente, consiste nel creare il modello da codice; i passaggi da eseguire (non necessariamente nell'ordine) sono essenzialmente due. La funzione obiettivo viene creata aggiungendo al problema una o più variabili per volta, tramite la funzione `CPXnewcols`, con la quale è possibile specificare il dominio e la natura di tali variabili (per esempio, il

fatto che siano binarie). L'inserimento di un nuovo vincolo di grado avviene invece tramite l'uso combinato di due *routine*: con la prima, `CPXnewrows`, viene creato un vincolo nel modello in cui tutti i coefficienti delle variabili sono inizialmente settati a zero, con la seconda, `CPXchgcoef`, è possibile cambiare il valore di tali coefficienti solo per le variabili di interesse (in questo caso, considerando il vincolo di grado sul nodo i-esimo, vengono settati a 1.0 tutti i coefficienti delle variabili corrispondenti agli archi uscenti dal i-esimo nodo)[1].
In qualsiasi momento è possibile reperire il numero corrente di righe (vincoli) o colonne (variabili) del modello, rispettivamente con le chiamate alle funzioni `CPXgetnumrows` e `CPXgetnumcols`.

C.2 Risoluzione del modello e Reperimento dei dati

Una volta che il modello viene creato la chiamata alle *routine* di risoluzione sono banali: la *routine* `CPXmipopt` risolve il modello generato e salva i risultati ottenuti (se ce ne sono) all'interno del suo *environment* associato. A questo punto l'utente può esplorare tale *environment* per ricavare informazioni sulla soluzione ottima, sul vettore della soluzione, sul tempo di esecuzione ecc. Segue un elenco delle principali *routine* utili per estrapolare caratteristiche sul risultato della soluzione di un problema tramite ottimizzazione intera:

- `CPXgetstat`: restituisce un valore intero corrispondente alla stato della soluzione. In generale, per ogni problema di programmazione lineare intera esistono tre esiti possibili:
  - Problema Impossibile (*infeasible*): non esiste una soluzione ammissibile
  - Problema Illimitato (*unbounded*): la funzione obiettivo non è limitata inferiormente
  - Problema risolto all'ottimo: esiste almeno una soluzione amissibile ed essa è stata certificata all'ottimo

  Poichè in `CPLEX` il processo di ottimizzazione può essere controllato e se necessario interrotto secondo vari parametri, i valori possibili di ritorno sono in realtà più dei tre immaginabili: in tabella C.1 sono elencati alcuni dei principali

- `CPXgetobjval`: salva il valore della soluzione in una variabile passata dall'utente come parametro

- `CPXgetx`: la *routine* permette di esplorare il valore assunto dalle singole variabili del modello (anche parzialmente), memorizzando il vettore soluzione all'interno di un array passato come parametro (dopo averlo allocato opportunamente)

[1] In generale, è possibile aggiungere un vincolo al modello tramite la funzione *CPXaddrows*, la quale permette di creare il taglio e specificare i coefficienti con una sola chiamata. La scelta di usare le due funzioni separatamente è dettata unicamente dalla natura del vincolo, per la quale tale modalità è probabilmente più semplice (in termini di costruzione dei parametri da passare alle *routine*).

parametro (valore)	descrizione
CPXMIP_OPTIMAL (101)	Problema risolto all'ottimo
CPXMIP_INFEASIBLE (103)	Problema impossibile
CPXMIP_TIME_LIM_FEAS (107)	Raggiunto *timelimit*, esiste una soluzione intera
CPXMIP_TIME_LIM_INFEAS (108)	Raggiunto *timelimit*, non esiste una soluzione intera
CPXMIP_SOL_LIM (104)	Raggiunto il limite di soluzioni
CPXMIP_OPTIMAL_TOL (102)	Raggiunto il valore di `epgap`

Tabella C.1: Lista dei principali codici di ritorno della funzione `CPXgetstat`

C.3 Metodo Loop

Il metodo *Loop*, come approfondito nel Capitolo 3, è la modalità di risoluzione secondo cui il modello, inizialmente composto solo dai vincoli di grado sui nodi, viene risolto all'ottimo, e iterativamente ri-risolto una volta inseriti i vincoli anticiclo sulle componenti connesse trovate dalla soluzione precedente. La tecnica tende pertanto a utilizzare `CPLEX` come risolutore *"black box"*, e non fa uso di *routine avanzate*.
In questa sezione si farà riferimento alle funzione di settaggio dei parametri, relativi alle tecniche adottate allo scopo di ridurre il tempo di esecuzione di ciascuna iterazione di risoluzione.

Timelimit, gap e sollimit `CPLEX` permette all'utente di modificare decine di valori legati ai suoi processi interni di ottimizzazione; il fissaggio di tali parametri avviene mediante funzioni come (ma non solo) `CPXsetintparam`, `CPXsetdblparam` e `CPXsetlongparam`, che impostano un dato campo del risolutore al valore specificato dall'utente. Di seguito, viene proposto il codice per impostare i parametri utilizzati nella sezione 3.2.1:

```
// imposta il numero massimo di soluzioni intere da trovare, prima di
    terminare l'ottimizzazione, a 3
CPXsetintparam (env, CPX_PARAM_INTSOLLIM, 3);

// imposta il tempo limite massimo per ogni iterazione a 10 secondi; l'
    ottimizzazione si fermerá dopo tale lasso di tempo,
    indipendentemente dal fatto che una soluzione ammissibile sia
    stata trovata o no
CPXsetdblparam (env, CPX_PARAM_TILIM, 10);

// imposta il valore di tolleranza del gap tra miglior soluzione (di un
    nodo valido) e miglior soluzione intera al 5%
CPXsetdblparam (env, CPX_PARAM_EPGAP, 0.05);
```

In tabella C.2 vengono riportati i valori di default dei parametri citati.

parametro	valore di default
CPX_PARAM_INTSOLLIM	2100000000
CPX_PARAM_TILIM	1e+75
CPX_PARAM_EPGAP	1e-04

Tabella C.2: Valori di default dei parametri di *timelimit*, *sollimit* e *gap*

Settaggio del possibile vicinato di un nodo Indipendentemente dal metodo utilizzato per circoscrivere il numero di nodi raggiungibili da ciascun nodo, la *routine* necessaria, a tale scopo, prevede la modifica dell'*upper bound* da 1 a 0, per tutte quelle variabili associate ad archi di collegamento tra un nodo e un altro al di fuori del suo vicinato. Per eseguire questa operazione è possibile utilizzare la funzione `CPXchgbds`, che permette appunto di modificare i *bound* di una o più variabili.

Metodi alternativi per l'aggiunta dei vincoli La funzione normalmente utilizzata per aggiungere un vincolo anticiclo al modello è la `CPXaddrows` (aggiunge staticamente uno o più tagli alla matrice dei vincoli). Sono stati testati altri due metodi per l'inserimento dei suddetti tagli, `CPXaddlazyconstraints` e `CPXaddusercuts`, ma per maggiori informazioni si rimanda alla sezione 3.2.3 dove sono già stati trattati.

C.4 Metodo Callback

A differenza del metodo precedente, le *callback* richiedono maggior impiego della libreria di `CPLEX`. In questa sezione si farà riferimento alle *routine* di installazione delle *callback*, a metodi di riperimento dei dati all'interno dei singoli nodi dell'albero di ricorsione e a qualche parametro accessorio; per qualsiasi riferimento al funzionamento delle *callback* usate, si rimanda al Capitolo 4.

Procedure pre-installazione Le *callback* di `CPLEX` sono strumenti di programmazione avanzati che, se usati nel corretto modo, consentono una maggiore efficienza del processo di ottimizzazione di default, in quanto permettono all'utente di sfruttare le caratteristiche peculiari del problema che si intende risolvere.
Anche per questo motivo, tuttavia, non sempre viene garantito che tali modifiche siano completamente compatibili con l'intero processo di ottimizzazione standard (*preprocessing* e *processing*); per ovviare a tale complicazione, di default l'installazione di una *callback* richiede che alcune procedure di *preprocessing* vengano disattivate. A tale scopo si riporta le chiamate a funzione necessarie:

```
CPXsetintparam (env, CPX_PARAM_MIPCBREDLP, CPX_OFF);

CPXsetintparam (env, CPX_PARAM_PRELINEAR, 0);

CPXsetintparam (env, CPX_PARAM_REDUCE,CPX_PREREDUCE_PRIMALONLY);
```

Il parametro `CPX_PARAM_MIPCBREDLP` regola l'accesso alle informazioni del modello originale o quello creato da riduzioni o da altre procedure di *preprocessing* (default): alcune *callback* supportano entrambe le modalità, tuttavia, come si può vedere, la `lazyconstraintcallback` utilizza dei metodi al suo interno che rendono la modifica necessaria. Discorso analogo è valido per il parametro `CPX_PARAM_PRELINEAR`, che controlla il livello della complessità delle possibili riduzioni del modello originale. Il terzo parametro modificato, `CPX_PARAM_REDUCE`, indica se durante il *preprocessing* debbano essere eseguite riduzioni a livello di problema duale, primale, entrambi o nessuno dei due; di default l'installazione della `lazyconstraintcallback` impone la limitazione a solo riduzioni sul problema primale.

Installazione Una volta impostati i parametri è possibile procedere con l'installazione delle *callback*. Si riporta di seguito le tre chiamate a funzione adibite a tale scopo (corrispondenti alle tre tipologie di *callback* utilizzate nel progetto):

```
CPXsetlazyconstraintcallbackfunc(env, callback, &data_for_LCC);

CPXsetheuristiccallbackfunc(env, callback, &data_for_HC);

CPXsetusercutcallbackfunc(env, callback, &data_for_UCC);
```

Come si nota, le funzioni presentano tutte la stessa struttura; il parametro `callback` si riferisce al corpo di codice da eseguire ogni volta che verrà eseguita la *callback* (è quindi un puntatore a metodo); tali funzioni dovranno avere pertanto una firma specifica (per tali approfondimenti si rimanda alla documentazione).

Procedure post-installazione L'ultimo accorgimento prima di poter avviare l'ottimizzazione consiste nel correggere le impostazioni di *multithreading*: l'installazione di una qualsiasi *callback* di controllo (ovvero che modifichi in qualche modo l'andamento della ricerca nell'albero di ricorsione, aggiungendo vincoli per esempio) fa in modo infatti che `CPLEX` annulli qualsiasi settaggio precedente a tale operazione riguardo al numero di *thread* concorrenti a disposizione, non potendo garantire la *thread safety* del processo in presenza di codice scritto dall'utente. Se le impostazioni precedenti all'installazione non vengono ristabilite, l'ottimizzazione procede utilizzando un solo *thread*, sequenziando quindi le esecuzioni delle *callback*. Segue un esempio di come impostare il numero di *thread* uguale al numero di processori presenti nella macchina (in accordo con la documentazione, è consigliabile non impostare il parametro a un numero superiore a tale valore).

```
int cores;

// la routine restituisce il numero di processori fisici della macchina
CPXgetnumcores(env, &cores);

CPXsetintparam(env, CPX_PARAM_THREADS, cores);
```

Reperimento dei dati all'interno della callback Una volta che le *callback* vengono installate, non è comunque detto che l'utente voglia che vengano eseguite a prescindere dallo stato dell'albero di ottimizzazione: come discusso nel Capitolo 4, per esempio, l'esecuzione della `UserCutCallback` e `HeuristicCallback` può essere subordinata al fatto che il nodo che deve eseguire la *callback* sia in un livello al di sopra o al di sotto di una certa profondità dell'albero. Per recuperare, durante l'esecuzione di una *callback*, l'informazione sulla profondità del nodo, o altri dettagli, vengono usati i seguenti metodi:

- `CPXgetcallbacknodeinfo`
- `CPXgetcallbackinfo`

Le due funzioni ricevono, tra i vari parametri, un valore intero che specifica il dato che si desidera ottenere: la prima è utilizzata principalmente per informazioni riguardo al nodo da cui viene chiamata, come numero seriale o profondità del nodo; la seconda per ricavare informazioni riguardo allo stato globale dell'ottimizzazione (*incumbent*, migliore soluzione disponibile, numero di iterazioni, *timestamp*, tempo di esecuzione ecc.), all'albero di ricorsione (es. numero di nodi processati e ancora da processare) o riguardo alla *callback* stessa (es. id del *thread* che la esegue). Per esempio, le seguenti righe di codice hanno il compito di ottenere i dati sul *thread* e sulla profondità del nodo corrente:

```
int thread, node_depth;

// env, cbdata e wherefrom sono parametri che fanno riferimento agli
    stessi parametri passati dalla callback. Il quarto parametro,
    settato a 0, indica che il nodo al quale di sta facendo
    riferimento é quello corrente
CPXgetcallbacknodeinfo(env, cbdata, wherefrom, 0,
    CPX_CALLBACK_INFO_NODE_DEPTH, &node_depth);
CPXgetcallbackinfo(env, cbdata, wherefrom,
    CPX_CALLBACK_INFO_MY_THREAD_NUM, &thread);
```

All'interno di una *callback* è ovviamente anche possibile richiedere il valore della soluzione al nodo corrente, unitamente al vettore delle variabili:

```
double sol;
double *sol_vector;

// env, cbdata e wherefrom sono parametri che fanno riferimento agli
    stessi parametri passati dalla callback.
CPXgetcallbacknodeobjval(env, cbdata, wherefrom, &sol;
// [0, cols - 1] rappresenta l'intervallo di interesse all'interno del
    vettore delle variabili
CPXgetcallbacknodex(env, cbdata, wherefrom, sol_vector, 0, cols - 1);
```

Soluzione iniziale In questa modalità di risoluzione ha anche senso fornire al risolutore una soluzione *incumbent* di partenza: invece nel Metodo *Loop*, dare una soluzione iniziale ha un effetto molto marginale sull'ottimizzazione, in quanto molto probabilmente tale soluzione (un circuito ammissibile per le specifiche del problema) sarà peggiore delle soluzioni trovate nelle prime iterazioni (soluzioni disconnesse, in quanto molti vincoli anticiclo non sono ancora stati trovati) e quindi scartata.
La soluzione trovata, in questo caso attraverso i diversi algoritmi euristici (e in aggiunta opzionalmente una ridefinizione matheuristica), viene fornita a `CPLEX` attraverso la *routine* `CPXaddmipstarts`.

Aggiunta dei tagli Infine, si cita il metodo per inserire correttamente i tagli trovati all'interno delle due *callback* `LazyConstraintCallback` e `UserCutCallback`. Si ricorda che esse cercano rispettivamente vincoli anticiclo mediante la funzione di `SecSeparation` e vincoli di `mincut`, o meglio capacità violata. Una volta che tali vincoli vengono trovati, l'aggiunta al modello avviene attraverso il metodo `CPXcutcallbackadd`, che aggiunge i tagli nel problema a livello globale (usando la *routine* `CPXcutcallbackaddlocal` è possibile invece aggiungere i tagli solo a livello del nodo corrente, e quindi del suo sottoalbero).

C.5 Output

`CPLEX` offre la possibilità all'utente di visionare l'andamento dell'ottimizzazione attraverso alcune stampe real time sul flusso di output specificato. In questa sezione verrà brevemente descritto come vengono strutturate e interpretate queste stampe, sottolineando le informazioni più importanti o utili.
Innanzitutto, è necessario attivare la modalità "verbosa" legata al realtivo parametro interno:

```
CPXsetintparam(env, CPX_PARAM_SCRIND, CPX_ON);
```

`CPLEX` a questo punto stamperà i progressi riguardo ciascuna fase di ottimizzazione, dal *preprocessing* alla risoluzione dei singoli nodi, fornendo informazioni sul modello e sull'albero di ricorsione. Dal momento che `CPLEX` implementa l'algoritmo *branch-and-cut*, è in grado di gestire una coda di nodi, all'interno della quale sono contenuti quelli che devono essere ancora elaborati. Inizialmente la coda ha zero nodi. Al termine del *preprocessing* il primo nodo a essere eseguito è il nodo radice, in cui verrà risolto il rilassamento continuo del problema originale. Si osserva che in queste prime fasi può darsi che il risolutore non trovi rapidamente una soluzione intera ammissibile, pertanto l'unico valore a disposizione, come riferimento per l'ottimalità della soluzione, è quello della miglior soluzione frazionaria disponibile. Una volta trovato il primo *incumbent*, il risolutore continuerà ad espandere l'albero di *branching* aggiornando i due valori (*incumbent* e miglior soluzione), fino a quando la differenza tra questi non diventa accettabile (se non specificato diversamente, fino a quando tale differenza raggiunge un valore $\simeq 0$).
Nell'esempio in Figura C.1 viene riportato uno *screenshot* di *report* di un'esecuzione di `CPLEX`.

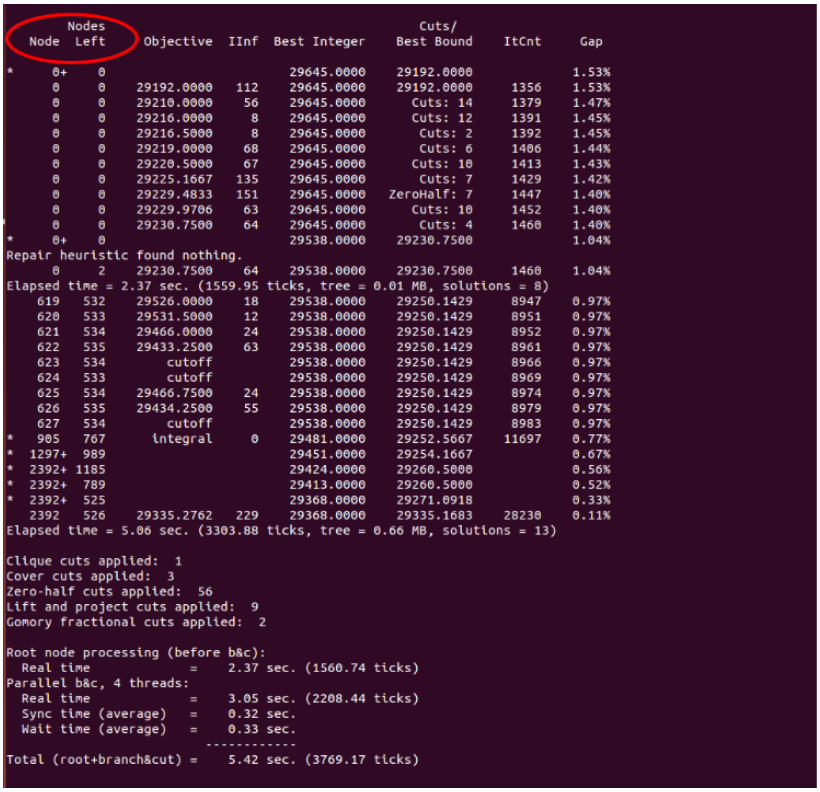

Figura C.1: *Screenshot* di *report* di un'esecuzione di `CPLEX`: viene mostrato all'utente l'andamento dello sviluppo del *branch-and-cut*.

In particolare, si deduce che l'ottimizzazione abbia coinvolto circa 2400 nodi, e abbia impiegato 5.42 secondi, di cui 2.37 di *processing* al nodo radice e 3.05 di *branch-and-cut*. Come si nota, il *report* mostra diversi dati relativi alla risoluzione di ciascun nodo:

- `node`: l'indice del nodo. Un valore uguale a 0 indica una fase di *processing* a livello della radice. Prima di tale valore può essere presente un asterisco (*), indicante che il valore *incumbent* è stato aggiornato
- `Nodes Left`: il numero di nodi ancora da processare
- `Objective`: valore della funzione obiettivo al nodo, o nel caso in cui un nodo e il suo albero vengono scartati, la ragione del *pruning*
- `Best Integer`: se non vuoto, indica il valore della miglior soluzione intera trovata fino a questo momento
- `Cuts/Best Bound`: il miglior `objective` trovato fino a questo momento. In caso di generazione di tagli di un unico genere, nella colonna viene scritto `Cuts` o il nome della loro famiglia
- `ItCnt`: somma del numero di iterazioni dell'algoritmo del simplesso, nei nodi eseguiti fino ad ora
- `Gap`: dissimilarità percentuale tra `Best Bound` e `Best Integer`

Appendice D

Concorde

Poichè, rispetto a CPLEX, la documentazione disponibile online per Concorde è meno abbondante, si conclude commentando brevemente le due funzioni di libreria utilizzate all'interno del progetto. Le funzioni di Concorde impiegate sono state la CCcut_connect_components e la CCcut_violated_cuts.

D.1 Funzioni utilizzate

CCcut_connect_components La funzione CCcut_connect_components è stata utilizzata come funzione di SecSeparation all'interno del Metodo *Loop* e all'interno della LazyConstraintCallback per il riconoscimento delle componenti connesse in una soluzione intera. Essa riceve i seguenti sette parametri (i primi quattro di input e gli ultimi tre di output):

- int ncount: numero di nodi del grafo n
- int ecount: numero di nodi del grafo, rispettivamente pari a $\frac{n*(n-1)}{2}$
- int *elist: è un vettore di lunghezza ecount che specifica i vertici di ciascun lato. Al momento della creazione di tale vettore è necessario tenere presente come è stato generato il vettore delle variabili della soluzione (x), per far corrispondere ogni lato al valore rispettivo
- double *x: la soluzione di cui calcolare le componenti
- int *ncomp: il numero di componenti connesse
- int **compscount: un vettore di vettori contenenti il numero di nodi per ciascuna componente connessa, in modo che compscount[i] contenga il numero di nodi presenti nell'i-esima componente connessa. Non è necessario che l'utente costruisca tale vettore da passare come parametro, sarà il metodo stesso a crearlo efficientemente a *run time* [1]
- int **comps: un vettore di vettori contenenti gli indici dei nodi presenti all'interno delle componenti; valgono le stesse considerazioni esposte precedentemente.

[1] Poichè il numero di componenti connesse non è noto a run time, se ncessario, l'utente dovrà stimare sia tale valore che il numero di nodi per componente: ponendo ragionevolmente che una componente sia formata almeno da tre nodi, si avrà che il numero massimo di componenti sarà $\frac{n}{3}$ e il numero massimo di nodi per componente $n-3$.

CCcut_violated_cuts La funzione `CCcut_violated_cuts` è stata utilizzata come funzione di ricerca di sezioni di capacità inferiore a una certa soglia all'interno della `UserCutCallback`. Essa riceve i seguenti sette parametri:

- `int ncount`, `int ecount`, `int *elist`: definiti ugualmente a quanto spiegato per `CCcut_connect_components`
- `double *dlen`: lista delle capacità dei lati, ovvero il vettore soluzione
- `double cutoff`: valore di soglia per determinare se un taglio è violato o meno: nel caso del problema del TSP, tale valore è impostato a (2 - EPSILON) dove EPSILON deve essere pensato in modo che un taglio violato risulti effettivamente tale durante sia l'esecuzione del metodo che, una volta aggiunto al modello (durante l'ottimizzazione (tipicamente EPSILON = 0.1)). Per EPSILON troppo piccoli, è possibile che la funzione rilevi il taglio violato, ma che venga poi ignorato (e quindi possibilmente non soddisfi la nuova soluzione) dal risolutore, poichè la sua soglia di tolleranza interna è maggiore di tale valore: la funzione all'iterazione successiva riscontrerà nuovamente il medesimo taglio e il sistema sarebbe soggetto a un loop
- `int (*doit_fn) (double, int,int *, void *)`: funzione *callback* da eseguire ogni qualvolta venga trovato un taglio violato; in particolare la *callback* passerà come parametri il valore del taglio, il numero di nodi, l'indice dei nodi e la struttura dati fornita in input dall'utente in `pass_param`
- `void *pass_param`: struttura dati passata dall'utente, tipicamente la stessa o una simile a quella utilizzata per memorizzare i vincoli trovati da una funzione di `SecSeparation`

Quindi il funzionamente della *routine* prevede che ogni volta che un taglio viene trovato, venga passato all'utente tramite la *callback* `doit_fn`, che a sua volta lo potrà inserire nel nuovo insieme di tagli da aggiungere al modello. Poichè durante la sua esecuzione il metodo suppone che la soluzione rappresenti un grafo connesso, è necessario effettuare tale controllo prima di eseguire la *routine* (per esempio eseguendo `CCcut_connect_components` e verificando il valore `ncomp`), o gestire il caso di un errore di ritorno[2].

D.2 Esempio di utilizzo

Si riportano in questa sezione dei brevi *code snippets* al fine di fornire un semplice esempio di possibile utilizzo delle funzioni presentate precedentemente.
Per prima cosa, si precisa che `Concorde` è resa disponibile tramite una libreria statica. Tali tipi di librerie rappresentano cataloghi di moduli oggetto collezionati in un unico contenitore e vengono installati direttamente nell'eseguibile del programma (senza richiedere la ricompilazione dei loro sorgenti).

[2] É necessario effettuare il controllo anche nel caso in cui alla funzione venga sempre passato il grafo completo, in quanto i lati con capacità nulla non vengono considerati.

Di seguito si riporta un esempio di possibile integrazione della libreria `Concorde` nella compilazione dell'eseguibile all'interno di un `MakeFile`:

```
CONCORDELIB =/path/to/concorde
...
CCDIR =/path/to/concorde
...
CFLAGS=-I$(IDIR) -I${CPLEX_HOME}/include/ilcplex -I${CCDIR}
...
LIBS = -L${CPLEX_HOME}/lib/x86-64_linux/static_pic
        -L. -lcplex -lm -lpthread ${CONCORDELIB}/concorde.a
...
main: $(OBJ)
gcc -o $@ $^ $(CFLAGS) $(LIBS)
```

All'interno del codice, per utilizzare le funzioni di tutti i moduli della libreria `Concorde` è necessario includerla tramite
`#include <concorde.h>`.
In questo caso, dato che le funzioni di interesse appartengono a un modulo solo, basterà
`#include <cut.h>`.
La funzione `CCcut_connect_components` è utilizzata per la ricerca di componenti connesse, e viene anche usata all'interno della `UserCutCallback` (vedi Listing D.1) allo scopo di determinare se una soluzione sia connessa o no, come descritto precedentemente.

```
...
if(CCcut_connect_components (inst->nnodes, nedge, elist, xstar, &ncomp,
     &compscount, &comps))
    print_error("Error during concorde connect comps algortihm!");
...
if(ncomp == 1)
    if(CCcut_violated_cuts(inst->nnodes, nedge, elist, xstar, 2.0 -
        EPSILON, doit_fn_concorde, (void*) &in))
        print_error(" error in CCcut_violated_cuts");
...
```

Listing D.1: Porzione di codice della *callback* `UserCutCallback`

Per quanto riguarda i parametri passati alle funzioni, possono essere ricavati nel seguente modo:

```
// numero di archi
int nedge = inst->nnodes * (inst->nnodes - 1) / 2;

// struttura elist
```

```
int elist[nedge * 2];
int i, j;
int loader = 0;
for(i = 0; i < inst->nnodes; ++i)
   for(j = i + 1; j < inst->nnodes; ++j){
      elist[loader++] = i;
      elist[loader++] = j;
   }

// strutture per le componenti
int ncomp = 0;
int *comps = (int*) malloc(inst->nnodes * sizeof(int));
int *compscount = (int*) malloc(inst->nnodes * sizeof(int));
```

La `doit_fn_concorde` è stata instanziata come segue:

```
int doit_fn_concorde(double cutval, int cutcount, int *cut, void *
    inParam)
```

Al suo interno, il taglio generato verrà creato ed aggiunto al modello di `CPLEX`.
É utile a tal proposito costruire una struttura dati appositamente creata per l'integrazione con tale *callback*:

```
    typedef struct{
        instance *inst;
        CPXCENVptr env;
        void *cbdata;
        int wherefrom;
        int *useraction_p;
    } input;
```

Come si nota, si tratta principalmente di impacchettare e passare gli stessi parametri necessari alla `UserCutCallback` di `CPLEX` (`env`, `cbdata`, `wherefrom`, `useraction_p`), includendo le informazioni sull'istanza (`inst`).