REALIZZAZIONE DI UNA CLASSE WRAPPER C# PER I SOLVER COINMP E CPLEX

Elaborato per il corso di Algoritmi per applicazioni aziendali

Anno accademico 2008-2009

Realizzato da

Andrea Pierantoni

SOMMARIO

1. INTRODUZIONE

Scopo di questo progetto era la realizzazione di una classe wrapper che permettesse l'interoperabilità fra le librerie CoinMP.dll (versione 1.4) e cplex101.dll, entrambe realizzate in linguaggio C++ unmanaged, con una qualsiasi applicazione scritta in C#.
Oltre a ciò, il progetto, qui presentato, fornisce un set di operazioni unificate che permettono all'utente di utilizzare il solver da lui ritenuto più consono, in maniera del tutto trasparente.
La libreria CoinMP.dll fa parte del progetto open source Computational Infrastructure for Operations Research (COIN-OR) e fornisce la maggior parte delle funzionalità dei progetti COIN-OR LP, COIN-OR Branch-and-Cut e Cut Generation Library.
Per maggiori informazioni riguardo COIN-OR si consiglia di visitare il sito ufficiale all'indirizzo http://www.coin-or.org.
La libreria cplex101.dll, invece, esporta i metodi principali utilizzati dal famoso solver proprietario della IBM, per la soluzione di problemi di programmazione lineare, intera, mista e quadratica. Anche in questo caso si rimanda al sito ufficiale (http://www.ilog.com/products/cplex) per ottenere informazioni più dettagliate.
Ritornando all'elaborato qui svolto, l'obiettivo è stato raggiunto minimizzando l'utilizzo del Marshalling per la gestione dei tipi di dato, e quindi preferendo a questo approccio la creazione di blocchi unsafe. Questo ha permesso, innanzitutto, di gestire in maniera esplicita i puntatori, oltreché di creare dei blocchi fixed, consentendo di evitare problemi di gestione della memoria, dovuti all'incompatibilità fra le politiche del garbage collector del framework .NET e i metodi di allocazione tradizionali.
La classe wrapper così realizzata consente, dunque, la totale interoperabilità con le librerie originarie, permettendo di richiamare i metodi principali esportati da entrambe.
Si passerà dunque a descrivere, nel prossimo capitolo, come includere ed utilizzare il wrapper nella propria applicazione C#.

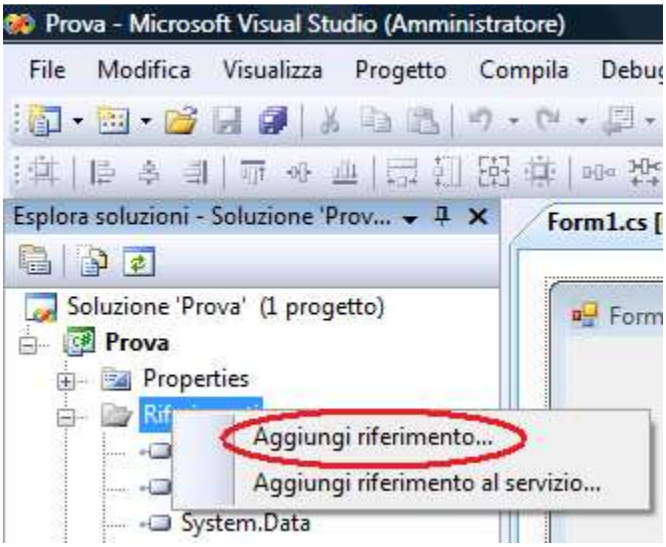
2. UTILIZZO DELLA CLASSE WRAPPER

In questo capitolo verrà descritto in primo luogo come creare un nuovo progetto che includa la libreria CoinCplexWrapper.dll, mentre in seguito si mostrerà un esempio di codice, atto a chiarificare in che modo inizializzare e risolvere un problema di ottimizzazione con la classe Wrapper.

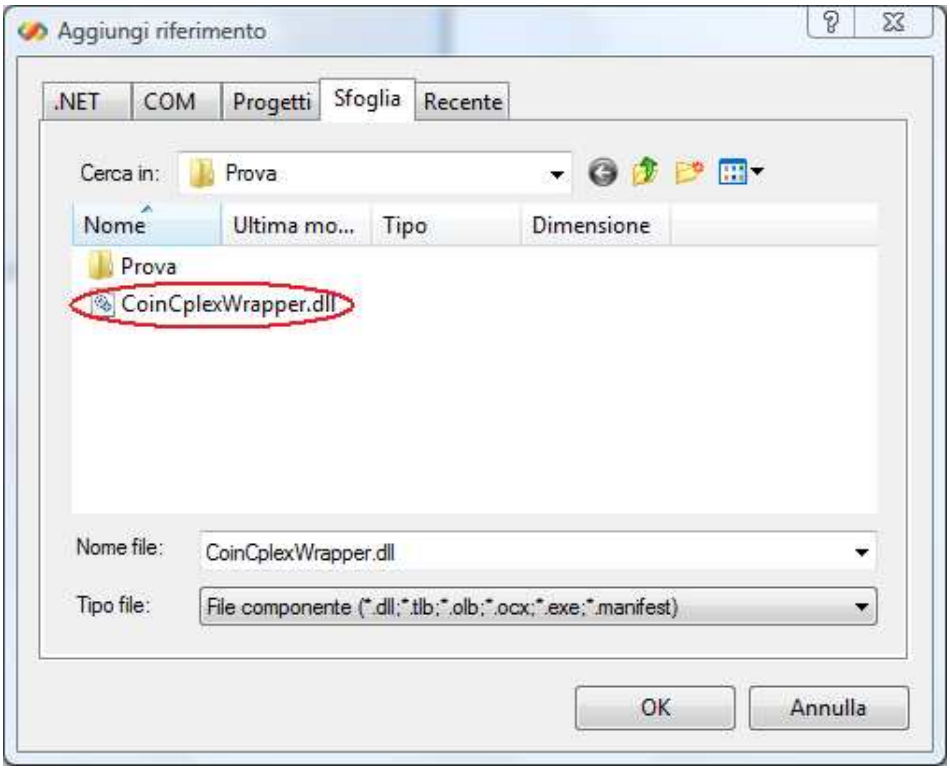
2.a. Inclusione del wrapper nella propria applicazione

Dopo aver aperto il proprio progetto con Visual Studio, per prima cosa è necessario aggiungere la libreria CoinCplexWrapper.dll fra i riferimenti. Per fare questo occorre:

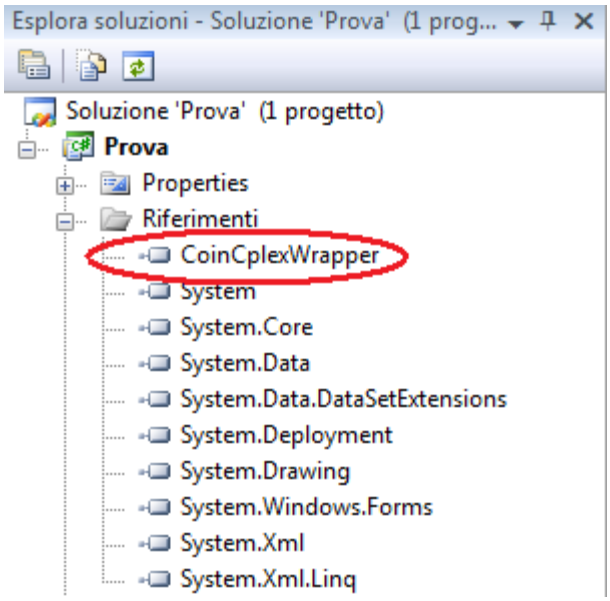
1) Cliccare con il pulsante destro del mouse su **Riferimenti** nella finestra **Esplora soluzioni** e poi scegliere nel menu **Aggiungi riferimento…**



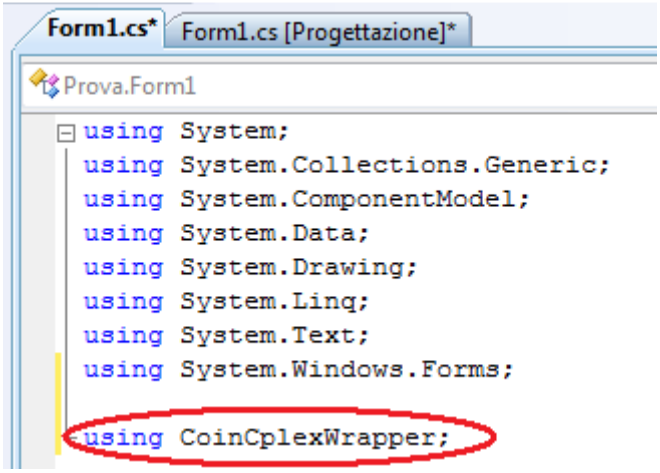
2) Nella finestra che si aprirà di seguito spostarsi nel tab **Sfoglia**, cercare il file **CoinCplexWrapper.dll** (anche se non è strettamente necessario, si consiglia di mettere il file nella stessa cartella della soluzione), e cliccare su **OK**.



A questo punto fra i riferimenti dovrebbe essere presente **CoinCplexWrapper**.



3) L'ultimo passo che resta da fare consiste nell'aggiungere nel file in cui s'intende utilizzare il wrapper il namespace di quest'ultimo, scrivendo nella posizione opportuna `using CoinCplexWrapper;` (vedi figura seguente).



A questo punto, sarà possibile inizializzare nel progetto un oggetto della classe Wrapper, ad esempio ne l seguente modo:

```
Wrapper w = new Wrapper();
```

2.b. Esempio di utilizzo

Ciò che s'intende chiarire in questo paragrafo, con un esempio, è come inizializzare e risolvere un problema di ottimizzazione – utilizzando uno qualsiasi dei due solver supportati - attraverso la classe Wrapper.

A questo scopo, si consideri il problema caratterizzato dalla seguente formulazione matematica, in forma canonica:

$$\text{obj} = Max \sum_{i=1}^{8} x_i$$

$$\begin{aligned} 3x_1 + x_2 \quad - 2x_4 - x_5 \quad - x_8 &\leq 14 \\ 2x_2 + 1.1x_3 &\leq 80 \\ x_3 \quad + x_6 &\leq 50 \\ 2.8x_4 \quad - 1.2x_7 &\leq 50 \\ 5.6x_1 \quad + x_5 \quad + 1.9x_8 &\leq 50 \end{aligned}$$

$$0 \leq x_i \leq 1000000 \qquad \forall i = 1,..,8$$

Per poter risolvere il dato problema è necessario innanzitutto inizializzare il numero di colonne e di righe del tableau, che rappresenteremo con due variabili intere ***colCount*** e ***rowCount*** e che dunque, in questo caso, assumeranno rispettivamente i valori 8 e 5. Fatto questo, sarà necessario inizializzare una variabile, che chiameremo ***nonZeroCount***, nella quale memorizzeremo il numero di elementi diversi da 0 (in questo caso 14).
Ora è possibile inizializzare una stringa (***objectName***) nella quale scrivere il nome della funzione obiettivo ("obj"), e una variabile intera (***objectSense***) che specifica se il problema da risolvere è di massimo (-1, come in questo caso) o di minimo (1).
Dopo aver impostato questi valori, è possibile cominciare ad inserire i dati del problema nelle opportune strutture; serviranno dunque:

- ***objectCoeffs:*** vettore di double contenente i coefficienti della funzione obiettivo (in questo caso tutti a 1);
- ***lowerBounds:*** eventuali lower bound per le variabili (in questo caso tutti a 0);
- ***upperBounds:*** eventuali upper bound per le variabili (in questo caso tutti a 1000000), se il valore che le variabili possono assumere non è limitato superiormente è comunque necessario inizializzare questo vettore con il valore massimo rappresentabile dal tipo di dato utilizzato (nell'esempio il *double*);
- ***rowType***: una stringa che ha tanti caratteri quante sono le righe, questo perché ogni suo carattere esprime se il vincolo cui fa riferimento è di tipo "minore o uguale" ('L'),

"uguale" ('E'), oppure "maggiore o uguale" ('G') (in questo caso abbiamo cinque vincoli tutti di tipo minore o uguale e quindi *rowType* sarà una stringa composta da cinque 'L');

- ***rhsValues***: un vettore che contiene i valori *right hand side* (nell'esempio 14, 80, 50, 50, 50);
- ***matrixBegin, matrixCount, matrixIndex, matrixValues***: questi quattro vettori sono il frutto della rappresentazione della matrice dei coefficienti compressa per colonne;
- ***colNames, rowNames:*** due vettori di stringhe che contengono rispettivamente i nomi delle colonne e delle righe del tableau.

Il codice per inizializzare le variabili finora descritte sarà dunque il seguente:

```
String problemName = "CoinTest";
int colCount = 8;
int rowCount = 5;
int nonZeroCount = 14;

String objectName = "obj";
int objectSense = -1;
double[] objectCoeffs = new double[8] { 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0};
double[] lowerBounds = new double[8] { 0, 0, 0, 0, 0, 0, 0, 0 };
double[] upperBounds = new double[8] { 1000000.0, 1000000.0, 1000000.0,
                                       1000000.0, 1000000.0, 1000000.0,
                                       1000000.0, 1000000.0 };

String rowType = "LLLLL";
double[] rhsValues = new double[5] { 14.0, 80.0, 50.0, 50.0, 50.0 };

int[] matrixBegin = new int[8 + 1] { 0, 2, 4, 6, 8, 10, 11, 12, 14 };
int[] matrixCount = new int[8] { 2, 2, 2, 2, 2, 1, 1, 2 };
int[] matrixIndex = new int[14] { 0, 4, 0, 1, 1, 2, 0, 3, 0, 4, 2, 3, 0, 4 };
double[] matrixValues = new double[14] { 3.0, 5.6, 1.0, 2.0, 1.1, 1.0, -2.0,
                                         2.8, -1.0, 1.0, 1.0, -1.2, -1.0, 1.9};

String[] colNames = new String[8] { "c1", "c2", "c3", "c4", "c5", "c6", "c7",
                                    "c8" };
String[] rowNames = new String[5] { "r1", "r2", "r3", "r4", "r5" };
```

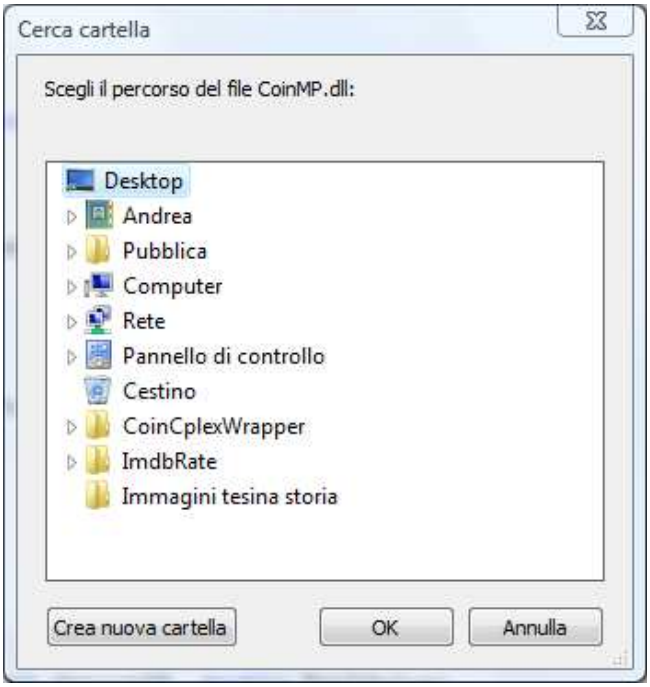
Dopo aver completato la dichiarazione delle variabili del problema, dovremo scegliere quale solver, fra i due supportati dal wrapper, utilizzare e poi inizializzarlo. Ad esempio, se si vuole inizializzare il solver CoinMP, si dovrà aggiungere al codice la seguente riga:

```
w.initSolver(w.getCoinMPSolver());
```

Se invece, si ha intenzione di utilizzare Cplex, la riga da aggiungere sarà la seguente:

```
w.initSolver(w.getCPLEXSolver());
```

Ovviamente è possibile, nel caso di necessità, utilizzare entrambi i risolutori all'interno della stessa applicazione, semplicemente aggiungendo entrambe le righe di codice riportate precedentemente.
È importante non dimenticarsi l'inizializzazione dei solver, poiché è durante questa fase che il wrapper esegue l'invocazione delle dll. La libreria viene cercata in un percorso predefinito, che consiste in quello della stessa applicazione per CoinMP, e nella directory d'installazione di default per Cplex. Nel caso in cui, le librerie non si trovino nella posizione predefinita, viene visualizzata una finestra di dialogo (vedi immagine sotto) che permette di selezionare la cartella in cui esse si trovano.



La fase, successiva all'inizializzazione dei risolutori, consiste nel caricamento dei dati all'interno della struttura, atta a rappresentare il problema. Infatti, contrariamente alla versione del wrapper per coin, in questo progetto è stato generato il tipo di dato WrapProblem che contiene il puntatore al problema e il valore booleano del solver con cui esso è stato creato. In questo modo, non è necessario che l'utente specifichi per ogni metodo invocato quale solver utilizzare, poiché, ove possibile, ciò viene gestito in automatico ed in maniera completamente trasparente all'utilizzatore.
Per instanziare un oggetto di tipo WrapProblem è sufficiente scrivere una riga di codice, basata sul seguente modello:

```

```

Dove *problemName* è la variabile con il nome che si vuole dare al problema, mentre *solver* è il valore booleano che identifica il risolutore di cui si vuole far uso.

Come nell'esempio riguardante l'inizializzazione, sarà sufficiente passare al posto di *solver* il valore ritornato dal metodo *getCoinMPSolver()*, se si ha intenzione di utilizzare la libreria CoinMP, oppure *getCPLEXSolver()* se, invece, si preferisce usare Cplex.
Dopo aver creato il problema, i dati precedentemente dichiarati possono essere caricati al suo interno richiamando il metodo *loadProblem()* secondo il modello seguente:

```
w.loadProblem(prob, colCount, rowCount, nzCount, rangeCount, objsense, objconst,
              objCoeffs, lowerBounds, upperBounds, rowType, rhsValues,
              rangeValues, matrixBegin, matrixCount, matrixIndex,matrixValues,
              colNames, rowNames, objName);
```

Il primo argomento accettato in input da *loadProblem()* è proprio la struttura che identifica il problema, seguita da tutti i dati che abbiamo visto come dichiarare all'inizio del paragrafo.
Se si ritiene che uno dei parametri richiesti in input dal metodo *loadProblem()* non sia necessario, per rappresentare il problema che si sta tentando di caricare, è sufficiente passare:

- **""**, se il parametro è una stringa;
- **0**, se il parametro è un int;
- **0.0**, se il parametro è un double;
- **null**, se il parametro è un array.

Infine, si fa notare che il metodo *loadProblem()* non richiede più in input il solver, infatti, come già accennato in precedenza, questo è memorizzato all'interno della struttura del problema e dunque, d'ora in avanti, questa informazione sarà gestita in modo del tutto trasparente all'utente.
Una volta caricati i dati, è possibile risolvere il problema richiamando il metodo *optimizeLpProblem()* (vedi figura sotto).

```
w.optimizeLpProblem(prob);
```

Al termine, se tutto si è svolto correttamente, sarà possibile ottenere il costo finale richiamando *getObjectValue()*, e i valori della soluzione richiamando il metodo *getSolutionValues()*:

```
double obj = w.getObjectValue(prob);
double[] solvars = new double[colCount]; //Vettore deputato a contenere i valori
                                          della soluzione
w.getSolutionValues(prob, solvars, null, null, null);
```

Dopo aver effettuato le due chiamate rappresentate sopra, infatti, si avranno in *obj* il costo finale ed in *solvars* i valori della soluzione.
Quando un problema, allocato e risolto, non è più d'interesse, è possibile liberare la memoria da esso occupata richiamando il metodo *unloadProblem()*, secondo l'esempio rappresentato di seguito:

```
w.unloadProblem(prob);
```

Infine, prima di terminare la propria applicazione, sarebbe buona norma chiudere tutti i solver che si sono inizializzati. Per farlo, è sufficiente richiamare il metodo *closeSolver()*, passandogli come argomento il valore booleano corrispondente al solver che s'intende chiudere. Dunque, analogamente a quanto visto nel caso dell'inizializzazione, le chiamate saranno le seguenti:

```
w.closeSolver(w.getCoinMPSolver()); //Per chiudere CoinMP
w.closeSolver(w.getCPLEXSolver()); //Per chiudere Cplex
```

3. INTEROPERABILITÀ

In questo capitolo, verranno presentate e discusse le scelte riguardanti l'interoperabilità. In particolare, dal momento che, come già accennato nell'introduzione, si è preferito al marshalling implicito una gestione esplicita della conversione dei tipi, si focalizzerà l'argomentazione proprio sul modo con cui tutto ciò è stato realizzato.

3.a. P/Invoke

Per poter richiamare i metodi dalle librerie originarie – implementate in linguaggio C++ unmanaged – tramite il wrapper – scritto in C# – è stato sfruttato un utile strumento messo a disposizione dal framework .NET: la ***Platform invoke*** (abbrev.: P/Invoke).
Questa piattaforma sfrutta, per l'importazione dei metodi, l'attributo ***DllImport*** seguito dal nome della dll e dal metodo che s'intende adottare:

```
[DllImport("CoinMP.dll")]
        static extern int CoinGetVersionStrBuf(IntPtr VersionStr, int buflen);
```

Come si può notare, l'intestazione del metodo è preceduto dalle due parole chiave *static* ed *extern*, entrambe richieste dalla clausola DllImport; in particolare, la seconda esplicita il fatto che il codice, che implementa il metodo, si trova in un ambiente esterno all'applicazione.
I tipi di dato dei parametri in input devono essere, inoltre, riformattati in base alle politiche adottate; per questo motivo, vedremo un po' più nel dettaglio, nei prossimi paragrafi, quali scelte sono state fatte, in tal senso, in questo progetto.
Prima rimane, però, da ricordare un ultimo concetto fondamentale riguardo l'utilizzo dello strumento P/Invoke, e, cioè, quello di rendere più pulito il codice, facendo un uso ampio ed intelligente dell'incapsulamento. Questo è possibile, ad esempio, rendendo *private* la dichiarazione del metodo importato, e costruendo parallelamente un metodo *public* che si preoccupi di richiamarlo in maniera trasparente all'utente.

3.b. La conversione dei tipi

Tratteremo in questo paragrafo le scelte che sono state fatte, durante la realizzazione del progetto, riguardo la conversione dei tipi, passando dal codice gestito a quello non gestito e viceversa.

3.b.1. I tipi *int* e *double*

I tipi *int* e *double* sono stati indubbiamente i tipi di dato più facili da gestire, in quanto non hanno necessitato di conversione.

3.b.2. I puntatori

I primi interrogativi si sono posti con la comparsa dei puntatori; in effetti una delle principali differenze, fra codice gestito e non, consiste proprio nella gestione della memoria. In particolare, ci si è resi conto che le politiche utilizzate dal garbage collector del framework entravano in conflitto con la logica di allocazione dei linguaggi unmanaged.
In sintesi, accadeva spesso inizialmente, che il garbage collector entrasse in funzione durante l'esecuzione del codice, spostando fisicamente i dati da una cella di memoria all'altra, facendo perdere i riferimenti al codice non gestito.
Per ovviare a questo fastidioso fenomeno, che inevitabilmente avrebbe condotto all'arresto dell'applicazione, è stata sfruttata un'altra comoda funzionalità messa a disposizione dal linguaggio C#, ovvero la clausola *fixed*. Tramite questo costrutto, è possibile dichiarare all'interno di un blocco *unsafe* – che consiste in una sezione di codice non gestito – un puntatore "fisso", il quale, quindi, non verrà toccato dal garbage collector.
Infine, per poter passare come parametro, al metodo che l'avesse richiesto, un puntatore così costruito, si è resa necessaria la sua generalizzazione al tipo di dato *IntPtr*.
È facile intuire, quindi, come una gestione così artificiosa dei tipi di dato abbia richiesto più che mai l'utilizzo dell'incapsulamento, al fine di evitare all'utente di doversi cimentare, come un tempo, con le vecchie politiche di allocazione – l'abbandono delle quali costituisce forse uno dei più grandi vantaggi dell'utilizzo di un linguaggio come il C#.
Al fine di rendere ancora più chiaro il processo di conversione dei puntatori, si fornirà ora un esempio. Si consideri, a tal proposito, l' intestazione originale del metodo *CoinGetSolutionValues()*, riportata di seguito:

```
int CoinGetSolutionValues(HPROB hProb, double* Activity, double* ReducedCost,
                          double* SlackValues, double* ShadowPrice);
```

dove HPROB rappresenta il puntatore alla struttura del problema, definita dai creatori della libreria CoinMP.

Innanzitutto, è stato necessario importare il metodo, utilizzando la P/Invoke (vedi sotto):

```
[DllImport("CoinMP.dll")]
        static extern int CoinGetSolutionValues(IntPtr hProb, IntPtr Activity,
                                                IntPtr ReducedCost, IntPtr SlackValues,
                                                IntPtr ShadowPrice);
```

Come si può notare, tutti i puntatori, di qualsiasi tipo fossero, sono stati convertiti nel puntatore generico *IntPtr*.
Dopodiché, è stato dichiarato il metodo *W_CoinGetSolutionValues()* con l'intestazione riportata qui sotto:

```
int W_CoinGetSolutionValues(IntPtr hProb, double[] Activity,
                            double[] ReducedCost, double[] SlackValues,
                            double[] ShadowPrice)
```

dove l'unico puntatore generico rimasto è quello al problema, mentre tutti gli altri parametri sono stati gestiti col tipo di dato C# che meglio rappresenta il corrispondente tipo unmanaged; e dunque, in questo caso, una serie di array di double.
La conversione degli array di double in IntPtr avviene all'interno di questo metodo, ed è stata ottenuta col seguente blocco di codice – a titolo di esempio, si mostra la conversione solo del vettore *Activity*, che è rappresentativa anche per tutti gli altri:

```
IntPtr a = new IntPtr();
unsafe
{
      if (Activity != null)
      {
            fixed (double* ap = &Activity[0])
            {
                  a = (IntPtr)ap;
            }
      }
}
```

Il primo passo è stato dunque quello di dichiarare un nuovo *IntPtr,* da utilizzare come riferimento all'array di double che intendiamo passare in input al metodo importato. In seguito, è stato dichiarato il blocco *unsafe*, all'interno del quale, se il vettore non è nullo, verrà eseguito il codice richiesto per la sua conversione. Questo, come già detto in precedenza, consiste in un blocco *fixed.* Dovendo convertire un vettore di double managed, in un vettore unmanaged, è stato necessario ottenere il puntatore al primo elemento e fare in modo che i dati non venissero spostati dal garbage collector. Una volta quindi ottenuto il riferimento al primo elemento del vettore, ed avendolo "bloccato" con la clausola *fixed*, è stato sufficiente generalizzarlo ad *IntPtr* attraverso un cast esplicito.

A questo punto, l'*IntPtr 'a'* è pronto per essere passato al metodo importato *CoinGetSolutionValues()*, come secondo parametro di input.

3.b.3 Le stringhe e i vettori di stringhe

Un caso complicato è stato quello di conversione delle stringhe, e, ancora di più, quello di conversione dei vettori di stringhe – richiesti, ad esempio, dal metodo di caricamento dei dati, per la rappresentazione dei nomi delle variabili.
In C# esiste la classe *String* che permette la rappresentazione di stringhe, ed è dunque subito apparso utile e soprattutto comodo che l'utente, utilizzando il wrapper, potesse sfruttare le potenzialità di questo nuovo tipo di dato. In C++ unmanaged, invece, com'è noto, le stringhe sono rappresentate attraverso dei vettori di *char*. Il primo problema da risolvere, quindi, è consistito nella conversione di una stringa da formato *string*, in un vettore di caratteri. L'unica cosa a cui fare attenzione, in questo frangente, è il fatto che il corrispondente del tipo di dato *char* unmanaged non è il *char* managed, bensì il *byte*. Superato questo piccolo ostacolo, ed ottenuto il vettore di *byte* correttamente costruito – funzionalità che nel progetto è stata implementata nel metodo privato del wrapper *ConvertStringToByteArray()* – è possibile convertirlo in un puntatore generico seguendo lo stesso ragionamento già visto per i vettori di double. A volte, però, si è resa necessaria anche la trasformazione inversa, ovvero quella da puntatore a stringa; in particolare, per quei metodi che ritornavano un vettore di caratteri. Questa operazione è effettuata dalla funzione privata *ConvertIntPtrToString()*, la quale, preso in input il puntatore generico da convertire, non fa altro che un cast esplicito ad un puntatore a *byte* e, infine, la trasformazione di quest'ultimo in *string*, scorrendolo come un vettore qualsiasi.
Il caso di conversione più difficile, come già detto inizialmente, è stato quello dei vettori di stringhe. Questo passo ha richiesto innanzitutto la conversione dell'array di partenza in una matrice di *byte* – operazione svolta dal metodo *ConvertStringArrayToByte2DArray()*. Fatto ciò, la conversione in *IntPtr* è stata realizzata con il codice riportato sotto.

```
IntPtr cn = new IntPtr();
unsafe
{
      if (ColNames != null)
      {
            fixed (byte* c = &Cols[0, 0])
            {
                  byte** vet = (byte**)Marshal.AllocHGlobal(sizeof(byte*) *
                              ColNames.Length);

                  for (i = 0; i < ColNames.Length; i++)
                  {
                        vet[i] = &c[i * (colmax + 1)];
                  }

                  cn = (IntPtr)vet;
             }
       }
}
```

Per una migliore comprensione del codice di cui sopra, si consideri che: *ColNames* è il vettore di *string* da convertire; *Cols* è la matrice di *byte* ottenuta dalla trasformazione di *ColNames*; ed infine *colmax* è la lunghezza delle righe di *Cols*.
Come visto per gli array monodimensionali, anche in questo caso viene fissato il riferimento al primo elemento della matrice. Ciò che cambia qui è la necessità di allocare un vettore di puntatori, di dimensione pari al numero di stringhe contenute nell'array di partenza, in cui andare ad inserire il riferimento iniziale di ogni riga della matrice *Cols*. Una volta terminata questa operazione, non resta altro da fare, se non il cast esplicito da *byte*** a *IntPtr*.

APPENDICE A

Descrizione dei metodi esportati dalla classe wrapper

- `public bool` **`getCoinMPSolver`**`()`
  ritorna il valore booleano corrispondente al solver Coin;

- `public bool` **`getCPLEXSolver`**`()`
  ritorna il valore booleano corrispondente al solver Cplex;

- `public int` **`initSolver`**`(bool solver)`
  inizializza il solver passato in input;

- `public int` **`closeSolver`**`(bool solver)`
  chiude il solver passato in input;

- `public string` **`getSolverName`**`(bool solver)`
  ritorna la stringa contenente il nome del solver passato in input;

- `public string` **`getVersion`**`(bool solver)`
  ritorna la stringa contenente la versione del solver passato in input;

- `public WrapProblem` **`createProblem`**`(string problemName, bool solver)`
  crea e ritorna un problema con nome uguale a *problemName* utilizzando il solver passato in input;

- `public void` **`loadProblem`**
```
(WrapProblem problem, int colCount,
     int rowCount, int nzCount, int rangeCount, int objSense,
     double objConst, double[] objCoeffs, double[] lowerBounds,
     double[] upperBounds, string rowType, double[] rhsValues,
     double[] rangeValues, int[] matrixBegin, int[] matrixCount,
     int[] matrixIndex, double[] matrixValues, string []colNames,
     string []rowNames, string objName)
```
  inserisce nell'oggetto della classe WrapProblem i dati del problema passati in input;

- `public void` **`loadInteger`**`(WrapProblem problem, string colType)`
  carica nella struttura del problema il tipo dei vincoli interi passati in input con la stringa *colType*;

- `public void` **`loadPriority`**
```
(WrapProblem problem, int priorCount, int[]
     priorIndex, int[] priorValues, int[] branchDirection)
```
  carica, per i problemi che lo richiedono, i dati relativi alla priorità;

- `public void` **`loadSos`**`(WrapProblem problem, int sosCount, int sosNzCount, int[] coinSosType, int[] sosPrior, int[] sosBegin, int[] sosIndex, double[] sosRef, string cplexSosType, string[] sosNames)`
  carica, per i problemi che lo richiedono, i dati Sos;

- `public void` **`loadQuadratic`**`(WrapProblem problem, int[] quadBegin, int[] quadCount, int[] quadIndex, double[] quadValues)`
  carica, per i problemi che lo richiedono, i dati quadratici;

- `public void` **`unloadProblem`**`(WrapProblem problem)`
  libera la memoria occupata dalla struttura del problema;

- `public string` **`getProblemName`**`(WrapProblem problem)`
  ritorna la stringa corrispondente al nome del problema passato in input;

- `public int` **`getColCount`**`(WrapProblem problem)`
  ritorna il numero delle variabili colonna;

- `public int` **`getRowCount`**`(WrapProblem problem)`
  ritorna il numero delle variabili riga;

- `public string` **`getColName`**`(WrapProblem problem, int colIndex)`
  ritorna il nome della colonna con indice *colIndex*;

- `public string` **`getRowName`**`(WrapProblem problem, int rowIndex)`
  ritorna il nome della riga con indice *rowIndex*;

- `public void` **`optimizeLpProblem`**`(WrapProblem problem)`
  risolve il problema se è di programmazione lineare;

- `public void` **`optimizeMipProblem`**`(WrapProblem problem)`
  risolve il problema se è di programmazione intera o mista intera;

- `public int` **`getSolutionStatus`**`(WrapProblem problem)`
  ritorna il valore intero che descrive lo stato della soluzione;

- `public string` **`getSolutionText`**`(WrapProblem problem, int solutionStatus)`
  ritorna la stringa che descrive lo stato della soluzione corrispondente al codice *solutionStatus*;

- `public double` **`getObjectValue`**`(WrapProblem problem)`

ritorna il costo ottimo;

- `public double` **`getMipBestBound`**`(WrapProblem problem)`
  ritorna il miglior costo trovato fino a quel momento;

- `public int` **`getIterCount`**`(WrapProblem problem)`
  ritorna il contatore dell'iterazione corrente;

- `public int` **`getMipNodeCount`**`(WrapProblem problem)`
  ritorna il numero di nodi generati**;**

- `public void` **`getSolutionValues`**`(WrapProblem problem,`
  `double[] activity, double[] reducedCost, double[] slackValues,`
  `double[] shadowPrice)`
  inserisce i valori della soluzione negli array passati in input;

- `public int` **`getSolverStatus`**`(WrapProblem problem)`
  restituisce il codice relativo allo stato del solver**;**

- `public void` **`getSolutionBasis`**`(WrapProblem problem, int[] colStatus,`
  `double[] rowStatus)`
  …;

- `public void` **`readFile`**`(WrapProblem problem)`
  carica un problema da file (completamente funzionante solo per Cplex);

- `public void` **`writeFile`**`(WrapProblem problem)`
  scrive i dati del problema in un file;

- `public string` **`getParamName`**`(WrapProblem problem, int paramID)`
  ritorna il nome del parametro con identificativo *paramID*;

- `public void` **`getIntParamMinMax`**`(WrapProblem problem, int paramID, out`
  `int minValue, out int maxValue)`
  inserisce nelle variabili passate in input come riferimento il valore minimo e quello massimo (interi) che può assumere il parametro con identificativo *paramID*;

- `public void` **`getRealParamMinMax`**`(WrapProblem problem, int paramID, out`
  `double minValue, out double maxValue)`
  inserisce nelle variabili passate in input come riferimento il valore minimo e quello massimo (real) che può assumere il parametro con identificativo *paramID*;

- `public int` **`getIntParam`**`(WrapProblem problem, int paramID)`

restituisce il valore attuale del parametro con identificativo *paramID*, ma solo se è di tipo intero;

- `public void` **`setIntParam`**`(WrapProblem problem, int paramID, int intValue)`
  cambia il valore del parametro con identificativo *paramID*, settandolo uguale a *intValue*;

- `public double` **`getRealParam`**`(WrapProblem problem, int paramID)`
  restituisce il valore attuale del parametro con identificativo *paramID*, ma solo se è di tipo double;

- `public void` **`setRealParam`**`(WrapProblem problem, int paramID, double realValue)`
  cambia il valore del parametro con identificativo *paramID*, settandolo uguale a *realValue*;

- `public string` **`getStringParam`**`(WrapProblem problem, int paramID)`
  restituisce il valore attuale del parametro con identificativo *paramID*, ma solo se è di tipo string (non ancora implementata in CoinMP);

- `public void` **`setStringParam`**`(WrapProblem problem, int paramID, string strValue)`
  cambia il valore del parametro con identificativo *paramID*, settandolo uguale a *strValue*;

- `public void` **`addrows`**`(WrapProblem problem, int ccnt, int rcnt, int nzcnt, double[] rhs, string sense, int[] rmatbeg, int[] rmatind, double[] rmatval, string[] colname, string[] rowname)`
  permette di aggiungere nuove righe ad un problema (solo per Cplex);

- `public void` **`addcols`**`(WrapProblem problem, int ccnt, int nzcnt, double[] obj, int[] cmatbeg, int[] cmatind, double[] cmatval, double[] lb, double[] ub, string[] colname)`
  permette di aggiungere nuove colonne ad un problema (solo per Cplex);

- `public void` **`delrows`**`(WrapProblem problem, int begin, int end)`
  permette di eliminare le righe, comprese nell'intervallo definito da *begin* e *end*, da un problema (solo per Cplex);

- `public void` **`delcols`**`(WrapProblem problem, int begin, int end)`
  permette di eliminare le colonne, comprese nell'intervallo definito da *begin* e *end*, da un problema (solo per Cplex);

APPENDICE B

Risultati ottenuti nella risoluzione di alcuni problemi

COINMP:

Problema CoinTest:

Optimal solution: 1428729,28571429
Variables:
c2: 40
c4: 428589,285714286
c5: 50
c6: 50
c7: 1000000

Problema Bakery:

Optimal solution: 506,666666666667
Variables:
Sun: 8000
Moon: 3000

Problema Afiro:

Optimal solution: -464,753142857143
Variables:
x01: 80
x02: 25,5
x03: 54,5
x04: 84,8
x06: 18,2142857142857
x14: 18,2142857142857

x16: 19,3071428571428
x22: 500
x23: 475,92
x24: 24,08
x26: 215
x36: 339,942857142857
x37: 383,942857142857

Problema P0033:

Optimal solution: 3089
Variables:
c157: 1
c163: 1
c164: 1
c166: 1
c170: 1
c174: 1
c175: 1
c178: 1
c180: 1
c181: 1
c182: 1
c183: 1
c184: 1
c185: 1
c186: 1

Problema Exmip1:

Optimal solution: 3,23684210526316
Variables:
col01: 2,5
col02: 1,05
col04: 1
col05: 0,5
col06: 4
col08: 0,263157894736842

CPLEX:

Problema air02:

Optimal solution: 7810
Variables:
CL000004: 1
CL000136: 1
CL000200: 1
CL000421: 1
CL000874: 1
CL001538: 1
CL002901: 1
CL003845: 1

Problema CoinTest:

Optimal solution: 1428729,28571429
Variables:
c2: 40
c4: 428589,285714286
c5: 50
c6: 50
c7: 1000000

Problema Bakery:

Optimal solution: 640
Variables:
Sun: 8000
Moon: 3000

Problema Afiro:

Optimal solution: -464,753142857143
Variables:
x01: 80
x02: 25,5
x03: 54,5
x04: 84,8

x06: 18,2142857142857
x14: 18,2142857142857
x16: 19,3071428571429
x22: 500
x23: 475,92
x24: 24,08
x26: 215
x36: 339,942857142857
x37: 383,942857142857

Problema P0033:

Optimal solution: 3089
Variables:
c157: 1
c163: 0,999999999999999
c164: 1
c166: 0,999999999999999
c170: 1
c174: 1
c177: 1
c178: 1
c179: 1
c180: 1
c182: 1
c183: 0,999999999999999
c184: 0,999999999999999
c185: 1
c186: 1

Problema Exmip1:

Optimal solution: 3,23684210526315
Variables:
col01: 2,5
col02: 1,05
col04: 1
col05: 0,5
col06: 4
col08: 0,263157894736845