*Metodi e Modelli per l'Ottimizzazione Combinatioria*

**Catalin Copil [m. 1109420]** **Padova, 15/07/2017**

# Risoluzione del problema 'Perforatore'

### Introduzione e presentazione del progetto svolto

La prima parte dell'esercitazione è stata risolta utilizzando il solver CPLEX. Per la seconda parte sono stati realizzati 3 Solver diversi, che implementano euristiche di Ricerca Locale. I risultati ottenuti sono stati messi a confronto, anche se l'utilizzo del solver CPLEX è stato fattibile solo per istanze fino a 100 nodi a causa dei lunghi tempi di attesa.

Per rendere più comprensibile e facilitare lo sviluppo, il progetto è composto da varie classi e file, ecco una spiegazione del ruolo delle classi

| Classe | Funzionalità |
|---|---|
| Punto | Rappresenta un nodo dell'istanza |
| Istanza | Istanza di un problema, contiene una lista di oggetti Punto |
| Soluzione | Soluzione di un'istanza, contiene un puntatore ad un oggetto Istanza, contiene la lista ordinata di oggetti Punto che rappresenta l'ordine da seguire. Calcola la Funzione Obiettivo. |
| Solver | Classe base per tutte le classi Solver. Contiene un metodo risolvi() che avvia il solver; contiene altri metodi che permettono il recupero della Funzione Obiettivo, del tempo impiegato e di un oggetto Soluzione. |
| CPLEX_Solver | Solver che utilizza CPLEX |
| NearSolver | Solver che utilizza l'euristica Nearest Neighbor |
| TwoOptSolver | Solver che utilizza Ricerca Locale con mosse 2-opt |
| TwoOptMove | Classe che rappresenta una mossa 2-opt per una soluzione |
| TabuSearchSolver | Solver che implementa la Tabu Search con mosse 2-opt |

Il file `perforatore.cpp` contiene il metodo main che avvia i vari solver in base ai parametri ricevuti all'avvio.

La maggior parte delle istanze dei problemi sono state generate in modo casuale, ma è stata utilizzata anche un'istanza ricavata da un problema reale di scheda perforata (fonte Google: https://developers.google.com/optimization/routing/tsp/tsp )

**Classe Solver (file `Solver.h` e `Solver.cpp`)**

La classe Solver è alla base della gerarchia dei solver implementati. Il suo obiettivo è quello di dare un'interfaccia standard per chi utilizza le classi derivate. Il costruttore di Solver riceve in input un puntatore ad un oggetto Istanza, grazie a questo puntatore (`ist`) il solver può recuperare tutti i dati necessari al problema da risolvere. La classe Solver ha anche un puntatore (`sol`) ad un oggetto Soluzione. Inizialmente `sol` punta ad una soluzione di default (i punti sono nell'ordine in cui sono memorizzati nell'istanza) ma nelle classi base questo puntatore viene assegnato alla soluzione ottenuta dopo aver eseguito il solver tramite l'opportuno metodo `risolvi()`.
Per il recupero della soluzione c'è il metodo `getSoluzione()`, che ritorna il puntatore `sol`.

## CPLEX Solver (file `CPLEXSolver.h` e `CPLEXSolver.cpp`)

Analogamente alle esercitazioni del laboratorio, per utilizzare la libreria CPLEX è stato utilizzato il file `cpxmacro.h` e gli oggetti di tipo `CEnv` e `Prob` per le chiamate alla libreria.
Al momento della costruzione dell'oggetto `CPLEX_Solver` viene passato il puntatore dell'istanza che il solver dovrà risolvere; l'ambiente CPLEX viene configurato tramite il metodo `setupLP()` che viene invocato dal costruttore della classe CPLEX_Solver.

**Il metodo `setupLP()`**

Il modello realizzato per la libreria CPLEX corrisponde con quello della consegna, anche per i nomi delle variabili e dei costi. N è il numero dei nodi che l'istanza contiene.
La matrice delle distanze è stata definita come un vector a due dimensioni, che quindi può essere trattato come matrice. L'utilizzo degli oggetti vector è stata una scelta fatta per migliorare le prestazioni del software al crescere della dimensione delle istanze. Dato che per CPLEX le variabili corrispondono a delle colonne numerate, è stato utile realizzare una struttura matriciale che permetta di localizzare le variabili $x_{i,j}$ e $y_{i,j}$ tra le colonne del problema CPLEX. Questa è l'utilità delle matrici indiciX e indiciY, restituire la posizione della rispettiva variabile (x o y) di indici i e j.

Esempio: indiciX[4][5] = posizione della variabile $x_{4,5}$

L'aggiunta delle variabili al modello viene fatta separatamente per le variabili x e per le y. Grazie a due cicli for innestati vengono fatti variare i valori dei due indici per le variabili, inserendole in ordine nel modello. Attraverso ulteriori cicli for vengono aggiunti al modello anche i vincoli.

Il metodo `risolvi()` avvia il solver e ne misura il tempo impiegato.

Per motivi di tempo non è stato implementato il metodo `getSoluzione()` che avrebbe dovuto creare una lista di punti in base ai valori dati dal modello alle variabili.

## Nearest Solver (file `NearSolver.h` e `NearSolver.cpp`)

Il Nearest Solver esegue una ricerca locale esplorando un vicinato che consiste in soluzioni costruite con l'euristica Nearest Neighbor ma con nodo di partenza diverso.

Il metodo risolvi() prende in input l'indice del nodo iniziale ed un tempo limite per l'esplorazione del vicinato. Per poter avviare la ricerca nel vicinato il nodo iniziale viene impostato a -1; così facendo l'algoritmo cicla tutti i possibili nodi iniziali. Durante l'esplorazione del vicinato

l’algoritmo tiene in memoria la soluzione migliore, sostituita ogni volta che viene trovata una soluzione con minor valore per la funzione obiettivo.

## TwoOpt Solver (file `TwoOptSolver.h` e `TwoOptSolver.cpp`)

Il Solver TwoOpt costruisce una soluzione iniziale utilizzando l’euristica Nearest Neighbor e successivamente esplora il vicinato delle possibili mosse 2-opt che portano un miglioramento all’attuale soluzione.

La strategia di esplorazione del vicinato è un compromesso tra *first Improvement* e *best Improvement* perché vengono considerate le prime **`maxMoves`** mosse che porterebbero un miglioramento e tra quelle viene scelta la mossa che porta il maggior guadagno. La variabile **`maxMoves`** è impostata a 5 per le istanze fino a 300 nodi, a 7 per le istanze fino a 2000 nodi e a 10 per le istanze più grandi di 2000 nodi.

Per il TwoOpt Solver ci sono 2 criteri di arresto: il tempo di esecuzione ed il numero massimo di mosse 2-opt. Tali parametri hanno valori di default (tempo esecuzione = 10 sec, maxOpt = 300) oppure possono essere specificati dall’utente all’avvio del programma perforatore (vedere il paragrafo “Il programma perforatore”).

## TabuSearch Solver (file `TabuSearchSolver.h` e `TabuSearchSolver.cpp`)

Il Solver TabuSearch è molto simile al Solver TwoOpt perché costruisce allo stesso modo la soluzione iniziale (con euristica Nearest Neighbor) ed esplora il vicinato delle possibili mosse 2-opt. La differenza sostanziale è il fatto che è stata implementata una Tabu Search. Quindi si ha una Tabu List dove vengono memorizzate le ultime mosse. Tali mosse saranno tabu per un determinato numero di iterazioni (dipende dalla lunghezza della Tabu List).

La strategia di esplorazione è la best Improvement perché per passare alla soluzione successiva viene selezionata la migliore mossa (che non sia presente nella Tabu List) del vicinato 2-opt.

L’implementazione della Tabu List è stata fatta in modo simile all’esempio visto in laboratorio quindi per ogni punto dell’Istanza viene tenuta traccia dell’ultima iterazione nella quale il nodo è stato utilizzato per una operazione di 2-opt. Per capire se una determinata mossa è tabu o meno viene controllato se i nodi che dovranno effettuare lo scambio sono stati utilizzati per una mossa nelle k iterazioni precedenti (tale k è definito come la lunghezza della Tabu List).

Come valore di default la lunghezza della Tabu List è 7, ma può essere modificata tramite i parametri del programma perforatore (vedere il paragrafo “Il programma perforatore”).

## Classe TwoOptMove

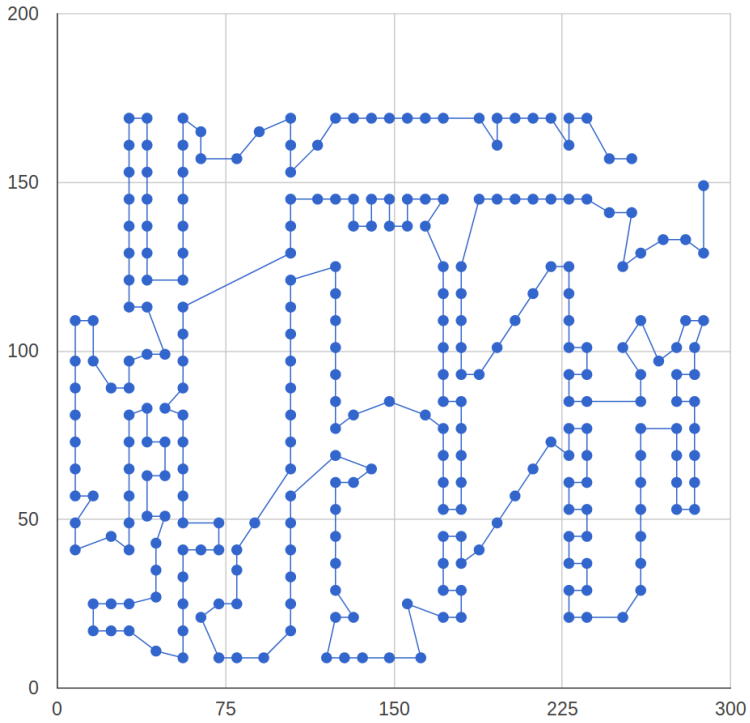
Per rappresentare una mossa 2-opt viene utilizzato un oggetto della classe TwoOptMove. L’oggetto contiene 2 indici, un puntatore all’istanza e uno ad un oggetto Soluzione. Grazie a questi puntatori è possibile valutare in modo rapido se la mossa porterà un miglioramento alla soluzione passata.

## Classe Soluzione

Per un utilizzo efficiente della memoria le soluzioni sono codificate come una sequenza di indici. Gli indici fanno riferimento alla posizione dei punti nell'oggetto Istanza. Così facendo è stato possibile gestire anche istanze di 25000 nodi.

## Un'istanza reale

Oltre all'obiettivo principale, ho voluto utilizzare un'istanza reale e metterla a confronto con delle istanze generate casualmente. L'istanza reale (chiamata IstanzaG280.txt) utilizzata contiene 280 nodi, e rappresenta una vera scheda elettronica che necessita di 280 fori. (fonte: https://developers.google.com/optimization/routing/tsp/tsp )
L'istanza è stata trovata comprensiva della soluzione ottima, quindi non è stato necessario eseguire CPLEX per l'intera istanza. Per motivi pratici, l'istanza è stata spezzata, ottenendo due derivati di 57 e 90 nodi (`IstanzaG57.txt` e `IstanzaG90.txt`)




Le rimanenti istanze sono state realizzate, generando coppie di interi casuali tra 0 e 1000.

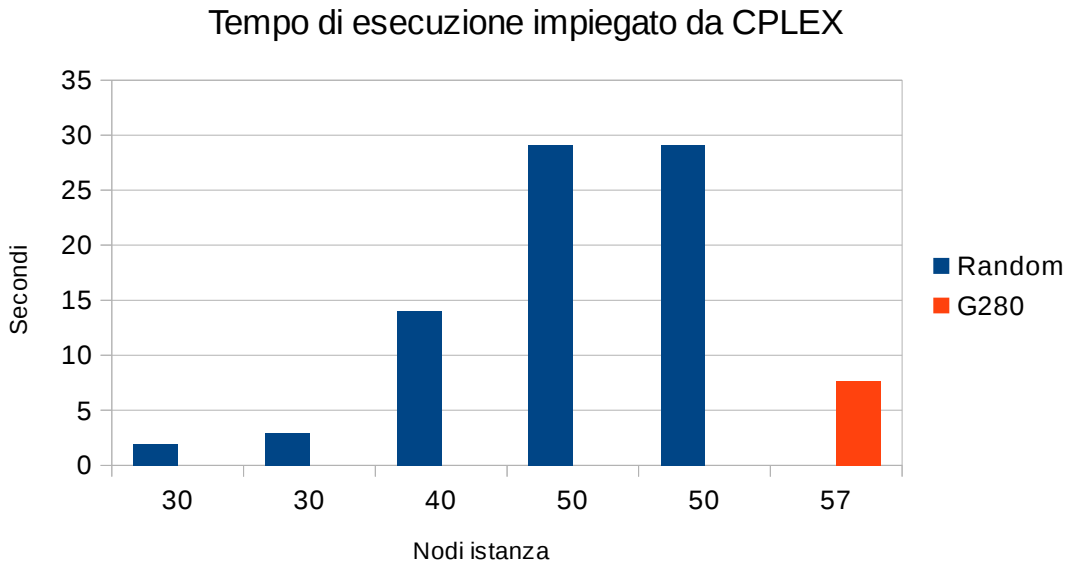
Di seguito una tabella riassuntiva con i risultati inerenti alle istanze del problema reale:

| Istanze Google | FO CPLEX | FO Nearest | FO TwoOpt | FO TabuSearch |
|---|---|---|---|---|
| istanzaG57 | 563,286 | 635,802 | 601,966 | 597,227 |
| istanzaG90 | 949,879 | 1016,23 | 983,544 | 983,544 |
| istanzaG280 | 2701,59 | 3118,8 | 2837,59 | 2860,61 |

Per l’istanza con 280 nodi, CPLEX non è stato avviato, ma la soluzione esatta è stata calcolata utilizzando i nodi ordinati dell’istanza. La tabella seguente riporta l’errore relativo per le 3 istanze reali e per altre istanze casuali secondo la formula: Err = (FO CPLEX – FO Solver)/FO CPLEX.

| Istanze | Err Nearest | Err TwoOpt | Err TabuSearch |
|---|---|---|---|
| istanzaG57 | 0,1287 | 0,0687 | 0,0603 |
| Random 57 | 0,1357 | 0,1548 | 0,1535 |
| istanzaG90 | 0,0699 | 0,0354 | 0,0354 |
| Random 90 | 0,1350 | 0,0386 | 0,0662 |
| istanzaG280 | 0,1544 | 0,0503 | 0,0589 |

Dalla tabella contenente gli errori relativi si può notare come per le istanze reali siano inferiori agli errori di una istanza casuale (soprattutto nel caso dei 57 nodi), a parità di numero nodi. Possiamo quindi dedurre che i Solver ottengono risultati migliori per delle istanze reali.




Il grafico illustra il variare del tempo di esecuzione tra l’istanza reale e quelle casuali. Nel caso di CPLEX, l’istanza reale risulta più “facile”, mentre per gli altri solver la differenza è minima.

## Test Istanze casuali

La maggior parte delle prove sono state fatte su istanze casuali. L’utilizzo del solver CPLEX è stato fattibile solo per piccole istanze (fino a 100 nodi) perché i tempi di esecuzione crescevano esponenzialmente.

La seguente tabella illustra alcuni risultati ottenuti per piccole istanze:

| ISTANZA | NODI | FO CPLEX | FO Nearest | FO TwoOpt | FO TabuSearch |
|---|---|---|---|---|---|
| Ist_rand10.txt | 10 | 2.823,76 | 2.823,8 | 2.823,8 | 2.823,8 |
| Ist_rand20.txt | 20 | 4134,4 | 4.175,0 | 4.305,14 | 4.305,14 |
| Ist_rand30.txt | 30 | 4.579,01 | 4.826,6 | 4.839,51 | 4.839,51 |
| Ist_rand50.txt | 50 | 5.928,68 | 6.457,86 | 6.200,7 | 6.200,7 |
| Ist_rand70.txt | 70 | 6.038,05 | 6.662,8 | 7.101,68 | 7.192,63 |
| Ist_rand80.txt | 80 | 6.637,82 | 7.549,49 | 7.252,47 | 7.167,6 |
| Ist_rand100.txt | 100 | 7.501,46 | 8.871,06 | 8.229,43 | 8.178,8 |

Osservando i risultati ottenuti dai Solver con euristiche (evidenziati in verde le soluzioni più vicine alla soluzione ottima), si può notare che per le istanze di piccole dimensioni il Nearest Solver sia decisamente più efficiente degli altri due. Si può notare come per le istanze più piccole non c'è differenza tra TwoOptSolver e TabuSearchSolver.

Per capire quanto le soluzioni ottenute dai vari Solver con euristiche si distanziano dalla soluzione ottima sono stati effettuati test su istanze di varie dimensioni. Per ogni test è stato calcolato l'errore relativo secondo la formula seguente: Err = (FO CPLEX – FO Solver)/FO CPLEX.

Per ogni dimensione sono stati eseguiti 5 test su istanze diverse e nella tabella seguente sono riportate le medie degli errori relativi.

| NODI | err Nearest | err TwoOpt | err TabuSearch |
|---|---|---|---|
| 10 | 0,040 | 0,023 | 0,023 |
| 20 | 0,066 | 0,071 | 0,059 |
| 30 | 0,079 | 0,097 | 0,119 |
| 50 | 0,072 | 0,099 | 0,096 |
| 70 | 0,113 | 0,117 | 0,118 |
| 100 | 0,151 | 0,113 | 0,113 |

Per le istanze di 10 nodi, in 2 test (su 5) tutti i 3 Solver riportavano la soluzione ottima.

Si può notare come l'errore relativo del Solver Nearest sia in generale più contenuto rispetto agli altri due. Questo risultato però cambia per le istanze di dimensione più grande (ad esempio con 100 o più nodi)

Per mettere a confronto i 3 solver che fanno uso di euristiche, è stato utile utilizzare istanze di dimensioni maggiori. Nella tabella seguente sono riportati i risultati ottenuti per istanze più grandi, ponendo come tempo massimo di esecuzione 10 secondi:

| NODI | FO Nearest | FO TwoOpt | FO TabuSearch | t Nearest | t TwoOpt | t TabuSearch |
|---|---|---|---|---|---|---|
| 300 | 16.463,1 | 14.897,2 | 14.675,8 | 0,109886 | 0,089302 | 0,127510 |
| 500 | 19.316,0 | 18.139,9 | 17.722,7 | 0,56955 | 0,395101 | 0,448805 |
| 700 | 23.122,0 | 21.418,0 | 21.161,9 | 1,35994 | 1,28185 | 1,32865 |
| 800 | 23.890,6 | 21.769,7 | 21.426,4 | 2,03831 | 1,389450 | 1,82944 |
| 1000 | 28.168,0 | 25.640,3 | 25.274,1 | 3,66611 | 2,79440 | 3,57384 |
| 2000 | 38.806,7 | 37.452,8 | 36.415 | 10 | 7,34028 | 10 |
| 3000 | 48.885,0 | 47.672,0 | 46.948,4 | 10 | 10 | 10 |
| 5000 | 62.512,3 | 61.848,0 | 61.594,2 | 10 | 10 | 10 |
| 8000 | 79.139,2 | 78.505,9 | 78.796,3 | 10 | 10 | 10 |
| 10000 | 88.338,10 | 88.096,50 | 88.118,10 | 10 | 10 | 10 |

Come è possibile notare dai risultati evidenziati, i migliori risultati sono ottenuti dal Solver che implementa la Tabu Search, anche se per le istanze più voluminose TwoOpt ottiene risultati migliori. Infatti, TwoOpt converge più rapidamente perché non esplora tutto il vicinato prima di attuare una mossa, allo stesso tempo questa tecnica può portare a scegliere delle mosse poco

miglioranti. Anche se non è riportato nella tabella è bene sapere che nei 10 secondi di esecuzione per l'istanza da 8000 nodi TwoOpt ha effettuato 102 mosse, mentre TabuSearch solo 6. Per l'istanza da 10000 nodi TwoOpt ha effettuato 106 mosse, mentre TabuSearch solo 4. Questi numeri fanno capire quanto sia diversa l'esplorazione del vicinato per i due Solver e anche come le prestazioni sono influenzate dal tempo limite imposto. Infatti, con le stesse istanze ed un tempo limite di 90 secondi il Solver TabuSearch 'vince' sull'istanza più grande. Nella tabella seguente i risultati ottenuti aumentando il tempo limite:

| NODI | FO Nearest | FO TwoOpt | FO TabuSearch |
|---|---|---|---|
| 3000 | 48.692,1 | 43.226,4 | 42.472,90 |
| 5000 | 62.265,3 | 59.519,0 | 57.655,20 |
| 8000 | 78.816,8 | 76.454,50 | 76.492,8 |
| 10000 | 87.710,10 | 86.728,70 | 86.045,60 |

E' chiaro quindi che per istanze più grandi il limite di esecuzione a 10 secondi è troppo basso ma, come si può notare dalla successiva tabella, anche il limite di 30 secondi potrebbe essere troppo basso affinché gli algoritmi che applicano mosse 2-opt possano convergere verso soluzioni accettabili. (Per i test di durata più ampia è stato incrementato il numero massimo di iterazioni per consentire ai solver di utilizzare appieno il tempo a loro disposizione)

| limite tempo | NODI | FO Nearest | FO TwoOpt | FO TabuSearch |
|---|---|---|---|---|
| 15 sec | 12000 | 96995,1 | 97960,7 | 98080,9 |
| 15 sec | 15000 | 107754 | 108133 | 108082 |
| 15 sec | 20000 | 124576 | 126233 | 126409 |
| 30 sec | 15000 | 107596 | 107883 | 107735 |
| 30 sec | 20000 | 124576 | 125900 | 126119 |
| 30 sec | 25000 | 139287 | 139295 | 139638 |

Uno dei parametri che possono influenzare il funzionamento della Tabu Search è la lunghezza della Tabu List; questo valore è stato fissato a 7 inizialmente e sono stati eseguiti dei test per capire se per alcune istanze sarebbe stato benefico incrementare tale valore. Nella tabella i risultati utilizzando l'istanza da 1000 nodi (numero massimo iterazioni = 1000).

| limite tempo | TabuList length | FO TabuSearch | 2-opt | FO TwoOpt |
|---|---|---|---|---|
| 5 sec | 7 | 37696,6 | 54 | 38275,6 |
| | 10 | 37662,1 | 55 | |
| | 15 | 37662,1 | 55 | |
| | 20 | 37696,6 | 54 | |
| 10 sec | 7 | 36344,6 | 109 | 36526,2 |
| | 10 | 36344,6 | 109 | |
| | 15 | 36379,3 | 107 | |
| | 20 | 36344,6 | 109 | |
| | 30 | 36368,1 | 108 | |
| 15 sec | 7 | 35642 | 165 | 35778,2 |
| | 10 | 35658,1 | 163 | |
| | 15 | 35717,7 | 158 | |
| | 20 | 35649,9 | 164 | |
| | 30 | 35816,5 | 148 | |
| 30 sec | 7 | 35027,4 | 315 | 35378,8 |
| | 15 | 35027,4 | 315 | |
| | 20 | 35027,4 | 315 | |
| | 30 | 35027,4 | 315 | |

Di seguito i risultati ottenuti con un'istanza da 6000 nodi:

| limite tempo | TabuList length | FO TabuSearch | 2-opt | FO TwoOpt |
|---|---|---|---|---|
| 15 sec | 7 | 68677,8 | 17 | 68660,9 |
| | 10 | 68608,2 | 18 | |
| | 15 | 68608,2 | 18 | |
| 20 sec | 7 | 68278,8 | 24 | 68528 |
| | 10 | 68278,8 | 24 | |
| | 15 | 68271,7 | 24 | |
| | 30 | 68336,2 | 23 | |
| 30 sec | 7 | 67686,8 | 36 | 68377,8 |
| | 10 | 67686,8 | 36 | |
| | 15 | 67693 | 36 | |
| | 20 | 67693 | 36 | |

Si evince che la lunghezza ideale della Tabu List dipende da vari fattori, anche dalla dimensione dell'istanza e dal tempo di esecuzione a disposizione.

## Commenti personali

Dalle prove fatte si evince che le caratteristiche di un'istanza possono influire molto sulle prestazioni di un algoritmo. L'utilizzo dell'istanza reale ha dimostrato che avere delle regolarità tra i punti dell'istanza può aiutare molto il solver. In particolar modo il Solver CPLEX ha risolto in maniera decisamente più rapida l'istanza reale da 57 nodi, rispetto ad altre casuali della stessa dimensione. Tra i Solver che utilizzano euristiche reputo migliori i Solver che utilizzano le mosse 2-opt, soprattutto il Solver TwoOpt che riesce a convergere rapidamente in un ottimo locale. L'implementazione della Tabu Search potrebbe avere miglioramenti soprattutto per quanto riguarda la strategia di esplorazione del vicinato perché la *best Improvement* attuale su istanze voluminose

porta via molto tempo prima di eseguire una mossa. Probabilmente sarebbe profiquo implementare un compromesso esplorando solo una parte del vicinato. Penso che il Solver Tabu Search debba essere adattato in base alla dimensione delle istanze che deve risolvere, scegliendo anche una dimensione adeguata della Tabu List. Per istanze da 100-2000 nodi la versione attuale è una buona implementazione, ma per altre istanze necessita miglioramenti.

## Il programma `perforatore`

Il programma perforatore richiede in input un altro parametro che può essere il filename di un'istanza da risolvere oppure un intero che indica la dimensione dell'istanza da generare.
Di default nel programma sono attivati tutti i 4 solver, anche se il Solver CPLEX non viene avviato in caso di istanza con più di 100 nodi. Tali impostazioni si possono cambiare modificando le variabili MAX_CPLEX_PROB e MAX_CPLEX_EXEC.

Il programma accetta in input anche altri parametri che sono opzionali, quindi la sintassi completa sarebbe:

```
./perforatore nomeFileIstanza.txt tempoInSecondi tabuLength maxIter
```

```
./perforatore numeroNodiIstanza tempoInSecondi tabuLength maxIter
```

con:

- `numeroNodiIstanza: intero>0 che indica il numero di nodi dell'istanza casuale da generare`
- `tempoInSecondi: double che indica il tempo limite massimo per l'esecuzione dei Solver con euristiche (di default = 10)`
- `tabuLength: lunghezza della TabuList (di default = 7)`
- `maxIter: numero massimo di Iterazioni per i Solver TwoOpt e TabuSearch (di default = 300)`

**Nota bene**: il programma di default salva le istanze (se il flag scriviIstanza è true) e le soluzioni in una sottocartella **util**. Se essa è inesistente non viene fatto il salvataggio (tale scelta può essere cambiata modificando il main).

Per disattivare alcuni solver oppure il salvataggio su file delle soluzioni bisogna impostare a false le variabili corrispondenti:

```
// FLAG DI ATTIVAZIONE / DISATTIVAZIONE DEI SOLVER E DELLE ALTRE FUNZIONALITÀ
bool attivoCPLEX = true;            // solver CPLEX attivato
bool attivoNS = true;               // NearSolver attivato
bool attivoTWO = true;              // TwoOptSolver attivato
bool attivoTS = true;               // TabuSearchSolver attivato
bool scriviIstanza = true;          // scrittura istanza attivata
bool SolNS = true;                  // scrittura soluzione NearSolver attivata
bool SolMIO = true;                 // scrittura soluzione MioSolver attivata
bool SolTWO = true;           // scrittura soluzione TwoOptSolver attivata
```

**Dettagli tecnici**

Il progetto è stato sviluppato in ambiente Linux, Ubuntu LTS 16.04.
Versione g++: 5.4.0
Specifiche PC: portatile DELL, 4 GB RAM, Intel i5-2520M, 4 core, 2.5 Ghz

Il progetto viene fornito con il `Makefile`, al quale bisogna correggere il percorso CPLEX in base al proprio ambiente.

Il comando `make` compila tutti i file generando il programma `perforatore`.
Il comando `make clean` elimina tutti i file oggetto.