Scuola Universitaria Professionale
della Svizzera Italiana

Dipartimento
Tecnologie
Innovative

Traveling Salesman Problem
13ª Coppa di Algoritmi
Project "*Swarmlings*"

Studente: Claudio Bonesana

Data: 1 maggio 2013

Introduzione

Il problema del commesso viaggiatore consiste nella ricerca del *ciclo hamiltoniano* più breve all'interno di un grafo pesato completo. In particolare, il grafo pesato rappresenta una mappa dove ad ogni nodo corrisponde una città e ogni città è collegata con una linea retta a tutte le altre città. Dato che le città giacciono tutte sullo stesso piano, il concetto di distanza euclidea è rispettato. Questo tipo di problema appartiene alla classe dei problemi *NP-Completi*. In allegato sono state depositate nella cartella `img` tutte le immagini dei tour scaricate da http://kovarik.felk.cvut.cz/pheromone/data.php?graph=rawdata.

Il compito assegnato consiste nel generare una soluzione ammissibile e quanto più possibile vicina a quella ottimale in 3 minuti di esecuzione del programma. Il programma deve poter prendere in input 10 problemi diversi descritti da altrettanti file:

- *ch130*;
- *d198*;
- *eil76*;
- *fl1577*;
- *kroA100*;
- *lin318*;
- *pcb442*;
- *pr439*;
- *rat783*;
- *u1060*.

Dopo alcune implementazioni inziali, si è scelto di utilizzare l'algoritmo **Ant Colony System** per la risoluzione dei problemi. Nel capitolo a pagina 2 è presente una descrizione completa degli algoritmi implementati.

Il software è interamente scritto in **C++**.

Tutti i dettagli sullo sviluppo e l'esecuzione del software sono disponibili all'interno di questo documento.

Implementazione degli algoritmi

Come da richiesta, per la risoluzione di questo problema si è scelto di implementare un algoritmo ibrido creato unendo tre algoritmi distintinti: un *costruttivo*, un'*ottimizzazione locale*, un *meta euristico*. Il software così costruito è completamente parametrizzato in modo da ottenere il risultato migliore.

Linguaggio scelto: C++

La scelta del linguaggio è caduta fin da subito sul **C++**. L'implementazione ha dapprima seguito i dettami standard della programmazione tuttavia, durante lo sviluppo degli eventi, il codice è terminato con l'abuso di variabili globali, l'abbandono di qualsiasi `Struct` e con l'uso di un unico oggetto: le formiche della classe `Ant`.

Commenti al codice

In questa sezione sono presentati pochi elementi di codice. La maggior parte del codice sorgente è ampiamente documentato. Qui sono riportate unicamente le motivazioni più dettagliate delle varie scelte e implementazioni.

Algoritmo costruttivo: Nearest Neighbour

Come *algoritmo costruttivo* la scelta è caduta sull'algoritmo **Nearest Neighbour**. La prima implementazione di questo Algoritmo ha portato ai risultati mostrati in Tabella 1. In questa prima implementazione la soluzione è stata trovata usando come città iniziale la prima città di ogni lista. Questa è stata una scelta per semplicità e modificata fin da subito per ricercare la soluzione migliore cambiando città iniziale. Infatti, modificando il nodo di partenza la soluzione cambia finale cambia. La soluzione migliore tra tutte queste è mostrata in tabella. Da notare che la media dei risultati (16.35%) di questo algoritmo con i dieci problemi forniti è inferiore alla media attesa secondo le relative tavole (26.27%) analizzate durante il corso.

Le motivazioni principali per la scelta di questo algoritmo sono sostanzialmente due: la semplicità dell'implementazione e la validità della soluzione prodotta. Aggiungendo una città per volta alla soluzione, eliminando la città inserita dalle rimanenti candidate e iterando sul numero di città la soluzione finale è valida per conseguenza.

Tabella 1: Soluzioni con Nearest Neighbour

Problema	Optimum	Risultato	Errore
ch130	6110	7134	16.76%
d198	15780	17705	12.20%
eil76	538	593	10.22%
fl1577	22249	25720	15.60%
kroA100	21282	24659	15.87%
lin318	42029	49139	16.92%
pcb442	50788	58896	15.99%
pr439	107217	127156	18.60%
rat783	8806	10429	18.43%
u1060	224094	275519	22.95%
media			16.35%

Algoritmo per l'ottimizzazione locale: 2-Opt

La scelta dell'ottimizzazione locale è stata più complicata. Inizialmente, sono stati implementati gli algoritmi *2-opt* e *3-opt*. In seguito, sono stati effettuati dei test unendo l'ottimizzazione locale al risultato dell'algoritmo costruttivo. In questo caso è stata presa in considerazione anche la città di partenza: il risultato migliore è dato dall'ottimizzazione migliore applicata ad una soluzione, non necessariamente la migliore migliore prodotta dal *Nearest Neighbour*, definita dalla città di partenza.

In Tabella 2 sono riportate le soluzioni date dall'ottimizzazione *2-opt* e in Tabella 3 sono riportate le soluzione con l'ottimizzazione *3-opt*. La Tabella 3 non riporta alcune soluzioni in quanto i tempi per la ricerca della soluzione ottima sono stati troppo lunghi ed è stato deciso di terminare la ricerca.

Tabella 2: Soluzioni con Nearest Neighbour + 2-opt

Problema	Optimum	Risultato	Errore	Città
ch130	6110	6255	2.37%	43
d198	15780	15950	1.08%	47
eil76	538	545	1.30%	41
fl1577	22249	22685	1.96%	907
kroA100	21282	21360	0.37%	38
lin318	42029	42670	1.53%	86
pcb442	50788	51757	1.93%	99
pr439	107217	109748	2.36%	22
rat783	8806	9135	3.74%	273
u1060	224094	234924	4.83%	940
media			2.15%	

In generale, e in maniera a dir poco sorprendente, l'algoritmo **2-opt** si è dimostrato non solo rapido nell'ottimizzare le soluzioni ma ha anche trovato soluzioni migliori, di

Tabella 3: Soluzioni con Nearest Neighbour + 3-opt

Problema	Optimum	Risultato	Errore	Città
ch130	6110	6239	3.00%	7
d198	15780	15949	1.07%	21
eil76	538	549	2.04%	38
fl1577	22249	-	-	-
kroA100	21282	21767	2.28%	88
lin318	42029	43800	4.21%	49
pcb442	50788	52545	3.48%	337
pr439	107217	111886	4.35%	274
rat783	8806	-	-	-
u1060	224094	-	-	-
media			-	

conseguenza è la prima scelta come algoritmo di ottimizzazione locale. Anche se il *3-opt* non ha fornito risultati eccezionali, l'algoritmo non è stato scartato: è semplicemente rimasto come seconda scelta.

Ottimizzazione dell'ottimizzazione

L'ottimizzazione locale prodotta dal *2-opt* inizialmente effettuava una ricerca continua scorrendo tutti i nodi con due cicli innestati ottenendo una complessità di $O(n^2)$ per la ricerca del miglior scambio locale. Con alcuni aggiustamenti la complessità è scesa a $O(n)$.

Listing 1: Implementazione dell'algoritmo 2-opt

```
void opt2(unsigned int size, unsigned int *tour){
    double best_gain = 0;
    double gain = 0;

    do{
        best_gain = 0;
        unsigned int best_i=0;
        unsigned int best_j=0;

        for (unsigned int i=0; i<size; i++){
            for (unsigned int j=i+2; j<size; j++){
                gain = opt2_compute_gain(size,i,j,tour);
                if (gain < best_gain){
                    best_gain = gain;
                    best_i=i;
                    best_j=j;
                }
            }
        }
        if (best_gain < 0.0){
```

```
                opt2_exchange(size, best_i, best_j, tour);
            }
        }
        while (best_gain < 0.0);
}
```

Purtroppo, l'intera ricerca è all'interno di un ciclo `while` che continua fin quando è possibile ottimizzare. Questo da una parte garantisce un'ottima soluzione finale, ma dall'altra rende l'ottimizzazione di un grande numero di città, come ad esempio per il caso del *rat783* o del *fl1577*, molto lenta. Questo è purtroppo l'anello debole dell'intero software in quanto è l'operazione che richiede più tempo ma garantisce anche dei risultati migliori.

Sono state pensate due varianti per ottimizzare ulteriormente questo algoritmo. La prima consiste nell'utilizzare degli *array di vicinanza* in cui l'algoritmo cerca le città da scambiare: ad ogni città viene assegnato arbitrariamente un numero variabile di città entro una certa distanza. Sarebbe interessante provare ad implementare questa idea sfruttando anche dei `kd-tree`.
La seconda, invece, consiste nel limitare il numero massimo di iterazioni ad ogni step di ricerca. Questo garantisce un numero maggiore di iterazioni globali ma una soluzione inizialmente peggiore.

Controllo della soluzione

Il controllo della soluzione è stato fatto in maniera molto semplice. Dato che, ad eccezione dell'ottimizzazione locale, non ci sono algoritmi che spezzano i tour creati ma aggiungono una città dopo l'altra alla soluzione, il controllo si basa sulla somma degli indici delle città e del controllo di questo valore con la somma attesa di una progressione aritmetica. Nel caso in cui una città si ripetesse oppure che mancasse una città allora tale somma non sarebbe corretta. Se le città non si ripetono, allora il tour non ha anelli e se le città non mancano allora sono tutte visitate.

```
int sum=0;
for (unsigned int i=0; i<size; i++){
        sum += tour[i] +1;
}
/* somma di una progressione aritmetica */
int expected = ((size * (size+1))/2);
```

Algoritmi meta euristici

Per gli algoritmi meta euristici, un criterio di ricerca ritenuto inizialmente fondamentale, in seguito abbandonato per forze di causa maggiore, è la ricerca di una soluzione debolmente dipendente dal *random*. Purtroppo, il caso ha troppo potere in queste scelte. Di conseguenza, la scelta dell'algoritmo meta euristico è risultata la scelta più complessa. Ponendo come obiettivo il trovare la soluzione migliore imponendo però ora il tempo di *3 minuti* massimi.

Simulated Annealing

La prima scelta di implementazione è caduta sul **Simulated Annealing**. Fin da subito questo algoritmo si è però dimostrato inferiore alle aspettative. La prima versione del *Simulated Annealing* consisteva nell'implementazione classica senza alcuna ottimizzazione locale: l'algoritmo calcola la soluzione successiva mediante la scelta di punti casuali e lo sfruttamento della funzione di scambio del 2-opt.

Nella Tabella 4 sono riportati i risultati di questa prima implementazione. Nella colonna *Algoritmo migliore* viene segnato se la soluzione è quella iniziale (*2opt - Nearest Neighbour* con *2opt*) oppure se è stata prodotta dal nuovo algorimo (*SA*). Da notare che non è stato registrato il seed e che la città iniziale è data da una scelta casuale.

Un problema comune a tutti gli algoritmi per la risoluzione di del TSP che inizia ad affiorare fin da subito, è la bassa efficienza con problemi composti da molte città. Sebbene l'algoritmo riesca a trovare l'ottimo per alcuni problemi, fatica con quelli più grandi.

Tabella 4: Soluzioni con Simulated Annealing

Problema	Optimum	Risultato	Errore	Algoritmo migliore
ch130	6110	6110	0.00%	SA
d198	15780	15817	0.23%	SA
eil76	538	538	0.00%	SA
fl1577	22249	23050	3.60%	2opt
kroA100	21282	21828	0.00%	SA
lin318	42029	42718	1.64%	SA
pcb442	50788	51558	1.54%	SA
pr439	107217	109025	1.69%	SA
rat783	8806	9248	5.02%	2opt
u1060	224094	239514	6.88%	2opt
media			2.06%	

Iterated Local Search

Per tentare di migliorare il *Simulated Annealing* si è scelto di trasformarlo in un **Iterated Local Search** sfruttando gli algoritmi di ottimizzazione locale con l'aggiunta di varie funzioni di mutazione per la ricerca di nuove soluzioni. Non solo viene utilizzata la coppia classica **double bridge - 3opt** ma anche ricerche parallele e ottimizzazioni locali.

Purtroppo, in Tabella 5 sono riportati i pessimi risultati del migliore di questi metodi (*ILS* - double bridge + 3opt) questi metodi posti a confronto con un Simulated Annealing (*SA*) con ottimizzazione locale e mutazione con scambio di punti casuali medianti 3-opt.

Ant Colony System - Swarmlings

Rinomato per le sue qualità di risoluzione di questo problema e ritenuto più rapido dell'*Algoritmo Genetico*, la scelta dell'**Ant Colony System** è risultata obbligata.

Tabella 5: Soluzioni con Iterated Local Search e Simulated Annealing

Problema	Optimum	Risultato	Errore	Seed	Algoritmo migliore
ch130	6110	6121	0.18%	59628155	SA
d198	15780	15867	0.55%	915477690	SA
eil76	538	541	0.56%	731648788	SA
fl1577	22249	150094	574.61%	628689340	ILS
kroA100	21282	21320	0.18%	963618778	SA
lin318	42029	42535	1.21%	504338432	SA
pcb442	50788	52623	3.63%	207729145	SA
pr439	107217	108757	1.44%	695178218	SA
rat783	8806	24299	175.94%	240631284	ILS
u1060	224094	292491	30.52%	883602352	SA
media			78.88%		

In Tabella 6 sono mostrati i primi risultati dell'implementazione semplice dell'ACS. Questa implementazione utilizza un'ottimizzazione locale di ogni soluzione e usa i seguenti parametri interni:

$$\begin{aligned} F &= 4 \\ \beta &= 2.0 \\ \alpha &= 1.0 \\ \rho &= 1.0 \\ q_0 &= 0.95 \end{aligned} \tag{1}$$

L'algoritmo si affida alla costruzione iniziale di una soluzione mediante l'algoritmo Nearest Neighbour con ottimizzazione locale 2-opt. In seguito vengono preparate F formiche posizionandole ognuna su una città a caso appartenente all'insieme delle città. Ogni formica costruisce la sua soluzione che alla fine viene ottimizzata grazie al 2-opt. Le soluzioni sono quindi paragonate e la migliore contribuisce a deporre il feromone.

Sfortunatamente, nonostante le numerose soluzioni ottime trovate, la percentuale complessiva di errore è ancora troppo elevata. L'idea di sostituire l'ottimizzazione 2-opt con un 3-opt non ha portato frutti in quanto quest'ultimo si è rivelato molto lento e ha limitato fortemente il numero di iterazioni, e quindi soluzioni prodotte dalle formiche. Meno iterazioni significa una soluzione meno precisa e quindi un errore maggiore. Il punto di forza quindi è iterare maggiormente.

Partendo dall'implementazione base si è poi proceduto ad ottimizzare il codice semplificando operazioni (abuso di *memcpy*, sostituzione di vector con *array*, eliminazione di array di struct in favore di più *array di primitive*). Grazie all'utilizzo di un *profiler* si è potuto guadagnare ulteriori, preziosi iterazioni.

Miglioramento dell'ACS

Per migliorare ulteriormente l'*ACS* si è proceduto all'analisi dei punti di forza e debolezze con i vari set di città a disposizione. Fin da subito appare abbastanza ovvio come i tour più

Tabella 6: Ant Colony System - versione base con 2-opt

Problema	Optimum	Risultato	Errore	Seed	Città iniziale
ch130	6110	6110	0.00%	632414000	43
d198	15780	15780	0.00%	979031000	47
eil76	538	538	0.00%	652939000	41
fl1577	22249	22691	1.99%	195918947	907
kroA100	21282	21282	0.00%	572359000	38
lin318	42029	42029	0.00%	347628000	86
pcb442	50788	51140	0.71%	753776000	99
pr439	107217	107578	0.34%	104757000	22
rat783	8806	8921	1.31%	1365371332	273
u1060	224094	230837	3.01%	15458000	940
media			0.74%		

lunghi siano i più difficili da elaborare, mentre tour sotto le 200 città vengono mangiati in poche iterazioni. Occorre quindi analizzare perché con dei tour lunghi l'algoritmo fatichi.

Un problema comune all'*ACS*, che appare analizzando l'evoluzione delle soluzioni create dalle formiche e della matrice del feromone, è dovuto alla gestione del feromone. Nei casi più piccoli, dove l'evaporazione era maggiore a causa dell'elevato numero di iterazioni, il feromone aveva la cattiva abitudine di scendere sino a provocare un errore di *underfow*. Tale errore si propagava in seguito ad altri valori della matrice feromonica rendendo instabile il tutto. La soluzione è stata quella di limitare il valore minimo delle celle della matrice con una soglia minima. Tale soglia è data dall'equazione

$$soglia = t_0/threshold \tag{2}$$

con $treshold = 10.0$. In altre parole se il feromone scende sotto un decimo di t_0, il valore viene reimpostato a t_0.

Continuando analizzando l'evoluzione delle soluzioni, si è potuto notare come, a lungo andare, le formiche tendano a *fossilizzarsi* su una striscia di feromone non per forza di cose ottimale. Una volta generatasi questa forte striscia di feromone, alle formiche è praticamente impedito uscirne in quanto il peso del feromone incide troppo sulla scelta della città ottimale.

Per ovviare a questo problema si sono inseriti due contatori:

$$\begin{matrix} steady\_best\_counter \\ steady\_local\_counter \end{matrix} \tag{3}$$

Il funzionamento di questi due contatori è simile, cambia solamente il livello in cui operano, ed entrambi agiscono sul deposito aggiuntivo di feromone. Il primo contatore, `steady_best_counter` lavora a livello globale, ovvero permette oppure no alla formica con la soluzione migliore di quella attuale uscita da ogni iterazione, di depositare del feromone aggiuntivo. Ogni iterazione genera una soluzione migliore. Quando questa soluzione è peggiore o uguale alla soluzione globale migliore, il contatore viene aumentato. Quando questo contatore supera il valore di guardia, `max_best_counter`, il deposito di feromone

viene bloccato in modo che solamente l'evaporazione agisca sulla modifica del feromone. Questo porta il feromone a diminuire *sbloccando* il funzionamento delle formiche che possono tornare a trovare nuove strade migliori.

Nella versione di questo *Ant Colony System*, tuttavia, è stata aggiunta una seconda fonte di feromone. Ad ogni iterazione la migliore formica della soluzione locale, indipendentemente dal fatto che questa sia migliore oppure no della soluzione globale, ha la possibilità di depositare del feromone. Da un lato questo fatto premia le formiche più esploratrici che hanno trovato il percorso più corto tra tutte le soluzioni locali, dall'altro incrementa il problema della fossilizzazione della soluzione. Questo meccanismo si attiva solamente quando la soluzione locale non genera alcuna nuova soluzione globale, per evitare un doppio deposito.

Per ovvie ragioni, anche la soluzione locale deve essere messa sotto controllo. Se per troppe volte le formiche non trovano soluzioni migliori, allora il deposito aggiuntivo viene bloccato.

In questo modo, con la sola evaporazione, il feromone scende a livelli accettabili e le formiche tornano a trovare nuove vie attraverso zone inesplorate o meno trafficate. I parametri di base scelti sono i seguenti:

$$\begin{aligned} max\_best\_counter &= 30 \\ max\_local\_counter &= 100 \end{aligned} \tag{4}$$

Infine, per incentivare le formiche ad esplorare ulteriormente nuove strade, si è scelto anche di manipolare il parametro q_0. Come noto, questo parametro determina il meccanismo di *Explotation/Exploration* delle formiche. Per forzare una maggiore esplorazione basta decrementarlo. Il segreto sta nel manipolare questa scelta: se le formiche rimangono per troppe iterazioni su una soluzione senza trovarne di migliori, allora viene forzata la parte di *Exploration* decrementando il valore di q_0 per un valore prestabilito $q0\_step$. Per evitare che le formiche termino con l'effettuare delle ricerche quasi completamente casuali, il decremento ha un limite prestabilito oltre il quale l'operazione si ferma. Quando una nuova soluzione viene trovata, il valore di q_0 viene riportato al valore iniziale. Di base i valori sono i seguenti:

$$\begin{aligned} q_0 &= 0.95 \\ q_0\_min &= 0.4 \\ q_0\_step &= 0.005 \end{aligned} \tag{5}$$

Esecuzione programma

Piattaforma

Le macchine utilzzate per l'esecuzione del programma, come definito dalle regole della coppa, sono i computer dell'aula 227. L'aula 227 comprende 30 macchine uguali a quella qui proposta. Nella Tabella 7 sono riportati i dettagli della macchina DTI-227-01. Le altre macchine sono cloni di questa sia per caratteristiche che per sistema operativo.

Tabella 7: Macchina usata per i test

Sistema Operativo	Ubuntu 10.04.4 LTS Lucid Lynx
Kernel	2.6.32-45-generic-pae i686
Processore	Intel(R) Core(TM)2 Quad CPU Q9400 @2.66GHz
RAM	4GB

Esecuzione del framework

Per poter eseguire correttamente gli script del *framework*, occorre ricostruire la seguente struttura ad albero. I file risultanti dall'esecuzione dei vari script si troveranno nella cartella `output` e nelle sottocartelle.

```
/
├── data/
│   └── *.tsp
├── launchme
├── launchme.py
├── makefile
├── output/
│   ├── tour/
│   └── text/
├── testme
└── TSP
```

Compilazione ed esecuzione

Il software è stato scritto con il linguaggio `C++` ed è stato fornito un makefile.

```
$ make
```

Per l'esecuzione esistono due possibilità: lanciare lo script `testme` presente nel framework, che eseguirà i dieci problemi uno dopo l'altro con i parametri ottimali trovati;

```
$ ./testme
```

oppure eseguire singolarmente i vari problemi. La sintassi minima accettata dal software è la seguente

```
$ ./TSP -d file <parametri>
```

In aggiunta il software accetta i parametri mostrati in Tabella 8:

Output del programma

Il software produce dell'output a schermo che viene scritto prima e dopo l'esecuzione dei vari algoritmi. L'output normale è il seguente:

```
Computazione in corso... fatto!

Tour:                    rat783
Algoritmo utilizzato:    Ant Colony System
Città iniziale:          273
Distanza minima trovata: 8844
SEED utilizzato:         10351898
Tempo trascorso:         180 secondi
Soluzione valida
```

Tabella 8: Parametri

Parametro	Descrizione
-d	Lista delle città su cui l'algoritmo andrà a lavorare.
-c	Città di partenza utilizzato dall'algoritmo Nearest Neighbour con 2-opt.
-s	Seed utilizzato dagli algoritmi con parti meta euristiche.
-D	Abilita la modalità debug con la stampa a schermo di dati e informazioni aggiuntive.
-h	Mostra questo help.
-o	Imposta l'output del file .tour.
-F	Numero di formiche da utilizzare (intero).
-A	Parametro α (dobule).
-R	Parametro ρ (dobule).
-Q	Parametro q_0 (double).
-q	Parametro q_o_min (double).
-S	Parametro q_o_step (double).
-B	Parametro $Max\_best\_counter$ (intero).
-L	Parametro $Max\_local\_counter$ (intero).

Framework per run automatizzate

Per eseguire più *run* possibili è stato creato un piccolo framework di esecuzione utilizzando il linguaggio di scripting per shell linux **bash** e uno script di raccolta risultato in **python**. Per la verifica finale dei risultati è stato creato un ulteriore script, sempre in *bash*, per l'esecuzione dei programmi in un ambiente sicuro e controllato. In aggiunta a questi script è stato creato anche uno script python per il lancio analitico di un problema con parametri differenti e uno script in *bash* per il controllo remoto della disponibilità delle macchine presenti nell'aula 227.
Tutti gli script sono presenti nella cartella `framework` dell'allegato.

launchme

Questo è lo script in **bash** principale per la raccolta dei dati. Lo script è nato per gestire più algoritmi per la risoluzione differenti. Nella versione finale, questa caratteristica viene utilizzata unicamente per lanciare parallelamente più problemi. Lo script viene configurato per ogni macchina con i seguenti parametri:

- `ALGORITHMS`: originariamente il tipo di algoritmi da lanciare, ora definisce semplicemente quanti processi paralleli eseguire. Uno zero in questo array corrisponde ad un processo.
- `CITIES`: array delle città di partenza da assegnare ad ogni algoritmo.
- `FILES`: nome del problema.
- `CYCLES`: numero di iterazioni da compiere, solitamente tante da riempire un ora di elaborazione.
- `HOURS`: se il parametro `CYCLES` è impostato correttamente, rappresenta il numero di ore continue di esecuzione.

Lo script procede quindi a lanciare con un distacco di 2 secondi un processo dopo l'altrofino al numero di elementi di `ALGORITHM`. In seguito, entra in un'attesa di 190 secondi. L'output a schermo viene salvato in un file indicante l'ora dell'inizio del'iterazione tramite un semplice redirect.
Nella variante finale, ad ogni ciclo lo script si preoccupa di creare un *tar gz* dei file in output e di eliminare gli originali per creare spazio. Questa operazione si è resa necessaria a causa dell'elevato numero di run effettuate complessivamente (ringrazio i sistemisti per aver aumentato spazio e numero di file).

launchme.py

Script di raccolta dati scritto in python. Analizza tutti i file presenti nella cartella `output/text/` e scrive nella cartella `output/` un file con i migliori run con relativi seed utilizzati.
Si notino due cose. I file analizzati devono essere tutti corretti e nella forma di output della versione finale dell'algoritmo. Eventuali file errati (spesso di dimensione minore) andranno eliminati *prima* della raccolta finale. La presenza di file interrompe la raccolta dati.

testme

Script in *bash* di test finale. Esegue i 10 problemi con i parametri migliori ottenuti.

digger.py

Script in *Python* di analisi dei parametri. Sviluppato per l'analisi dei problemi con grandi numeri di città. Mai sfruttato a pieno.

ctrlssh

Script in *bash* maledetto e richiesto da molti. Fornita una lista di IP, ad esempio quelli delle macchine dell'aula 227, indica il sistema operativo attualmente in esecuzione sulle macchine bersaglio ed esegue un rapporto completo dei processi in esecuzione sulle macchine Linux. Si è rivelato molto utile per la ricerca di macchine disponibili. Una versione arcaica di questi script è stata gentilmente offerta ad *Alessio Scannapieco*. La versione finale è stata registrata nel **crontab** di Linux1 in modo da inviare un rapporto sullo stato dell'aula ininterrottamente ad un indirizzo email personale ogni 20 minuti.
L'idea finale consisteva nel riuscire ad eseguire il lancio automatico dei programmi. Fortunatamente per gli utilizzatori dell'aula 227 questo optional non è stato incluso.

Risultati

In Tabella 9 sono stati riportati i risultati migliori completi di parametri e seed utilizzati. Nella cartella *result* sono riportati i relativi file *.tour* e la tabella originale.

Tabella 9: Soluzioni migliori per ogni problema

Problema	Optimum	Risultato	Errore	Seed	Città
ch130	6110	6110	0.00%	42	43
d198	15780	15780	0.00%	136598329	47
eil76	538	538	0.00%	893451273	41
fl1577	22249	22391	0.64%	258376960	907
kroA100	21282	21282	0.00%	983231872	38
lin318	42029	42029	0.00%	322745052	86
pcb442	50788	50791	0.03%	209239427	99
pr439	107217	107217	0.00%	311124730	22
rat783	8806	8844	0.43%	10351898	273
u1060	224094	225547	0.65%	221818014	940
media			0.17%		

Conclusioni

Da un punto di vista algoritmico, l'*Ant Colony System* è un algoritmo che mi ha affascinato ancora prima della sua presentazione durante il corso. Appena ho avuto l'occasione mi sono messo ad implementarlo rapidamente e si è subito rivelato un algoritmo pensato anche nella sua implementazione. A livello di codice non è stato difficile implementarlo, la parte piû complessa è stata l'analisi dei risultati e del comportamento del feromone. Fortunatamente alcune idee si sono rivelate giuste e mi hanno permesso di scendere con i risultati.

Per migliorare ulteriormente le performance si potrebbe provare a convertire completamente in **C puro** il software eliminando gli oggetti `Ant` ora utilizzati per semplicità.

Swarmlings

Il progetto è stato amichevolmente chiamato *Swarmlings* in quanto il comportamento di queste formiche ricorda più il comportamento degli *Zergling*.
Gli Zergling sono delle creature presenti nell'universo di *StarCraft* e sono fisicamente molto debole ma disponibili in grande quantità dato il ritmo di produzione doppio per ogni uovo. Si tratta di corridori xenomorfi a quattro zampe dotati di robuste zanne e artigli, potenzialmente anche ali, con l'abitudine di muoversi in sciame contro la preda.

Un po' di colore non guasta mai.