UNIVERSITÀ DEGLI STUDI DI MILANO
Facoltà di Scienze Matematiche, Fisiche e Naturali
Dipartimento di Tecnologie dell'Informazione
CORSO DI LAUREA IN INFORMATICA

Algoritmi di Programmazione Matematica per il "TSP with Rear Loading"

RELATORE

Prof. Giovanni RIGHINI

TESI DI LAUREA DI

Federico FICARELLI

Matr. n° 641417

Anno Accademico 2004 / 2005

Sommario

Il lavoro di tesi consiste nello sviluppo e sperimentazione di algoritmi esatti ed approssimati per un problema di instradamento ottimo con vincoli desunti da applicazioni reali nel settore della logistica della distribuzione. L'attenzione verte su una particolare variante ancora poco studiata di Traveling Salesman Problem simmetrico alla quale sono applicati vincoli di "Retro-carica" (o "Rear Loading") i quali impongono alla generica soluzione ammissibile di costruire il percorso Hamiltoniano di costo minimo visitando gli accoppiamenti delle città di pickup e delivery (rispettivamente le origini di carico e le destinazione di scarico) in ordine LIFO.

Il primo approccio considerato è stato quello di costruire un algoritmo esatto basato su tecniche di *Programmazione Dinamica* (P.D.) la quale mi ha permesso di modellare l'esplorazione del grafo come un insieme di stati distribuiti su una matrice triangolare rappresentanti l'avanzamento della computazione e della relativa costruzione del tour. Il tratto significativo della P.D. è la possibilità di eliminare quegli stati riconosciuti come dominati da altri che si rivelano sicuramente più promettenti. Ho assegnato anche un lower bound ad ogni stato per confrontarlo un upper bound ricavato tramite l'esecuzione degli algoritmi euristici sviluppati nella seconda parte del lavoro.

Il passo successivo è stato considerare un approccio basato su algoritmi approssimati, in particolare su diversi metodi di ricerca locale ai quali applicare la *Tabu Search*. Quest'ultima si caratterizza per l'utilizzo di un'area di memoria all'interno della quale conservare, per un periodo di tempo (predefinito o variabile) chiamato *Tabu Tenure*, mosse cronologicamente recenti all'istante attuale della computazione. Fintanto che queste vi risiedono sono considerate vietate, appunto "tabu". Questa meta-euristica fornisce alla politica di ricerca locale alla quale viene applicata la capacità di "liberarsi" da un ottimo locale che la farebbe terminare prematuramente. La probabilità di incorrere in cicli di esplorazione è diminuita proprio dalla memoria Tabu. Grazie a questa importante proprietà sono stato in grado di utilizzare un semplice quanto veloce Nearest Neighbour come euristica costruttiva per la produzione di una soluzione ammissibile iniziale alla quale applicare successivamente tutti gli altri algoritmi di ricerca locale sviluppati: scambio e rilocazione di coppie (sequenza di nodo pickup e delivery ad esso associata), scambio e rilocazione di blocchi (sottosequenze complete che compongono il tour). Per quanto riguarda la gestione del parametro di *Tabu Tenure* ho scelto di utilizzare metodi di *Reactive Tabu Search* che mi hanno permesso di modificarlo dinamicamente. Ho infine utilizzato l'insieme di tutte le tecniche di ricerca locale combinate in una catena di elaborazione gestita tramite politica di *Variable Neighbourhood Descent*.

Keywords: Ricerca Operativa, Programmazione Matematica, Traveling Salesman Problem, Programmazione Dinamica, Rear Loading, Ricerca Locale, Reactive Tabu Search.

Indice

Elenco degli algoritmi

Elenco delle tabelle

Elenco delle figure

Capitolo 1

Introduzione

La categoria di problemi relativi al *Vehicle Routing* (VRP) trova numerose applicazioni reali: il calcolo del percorso ottimale che i veicoli devono seguire per coprire tutti i punti (clienti) di un grafo e soddisfare le relative richieste di servizio. Il metodo di risoluzione deve quindi pianificare i viaggi di un insieme di veicoli localizzati presso un deposito in modo da servire completamente un insieme di clienti minimizzando i costi di trasporto. Nel mio caso, essendo presente un solo veicolo, il *Vehicle Routing* ricade nel sottoinsieme del più specifico Traveling Salesman Problem: oltre alle ovvie interpretazioni in chiave logistica, il problema rappresenta in realtà una sfida per un grande insieme di applicazioni reali che vanno dalla genetica alla pianificazione di catene produttive, dallo *scheduling* di processi in campo informatico alla ricerca biologica (Gutin e Punnen, [1]).

Nella formulazione da me considerata in questo lavoro i punti (nodi) che dovranno essere raggiunti sono suddivisi in due insiemi:

origini (o nodi di carico o di *pickup*), presso i quali il veicolo carica "merce";

destinazioni (o nodi di scarico o di *delivery*), presso i quali il veicolo scarica la "merce" raccolta in precedenza.

Sia $G = (V, E)$ un grafo non orientato ed F l'insieme di tutti i cicli Hamiltoniani in G. Ad ogni arco $e \in E$ viene associato un costo c_e. Il Traveling Salesman Problem (TSP) consiste infatti nel trovare il ciclo Hamiltoniano in G tale per cui la somma di tutti i costi degli archi percorsi è minima. Nel mio caso G è completo; allo stesso modo è possibile considerare G come tale per qualsiasi istanza di TSP utilizzando un costo infinito per rappresentare i mancanti. Sia $V = 1, 2, \ldots, n$ l'insieme dei nodi. La matrice $C = (c_{ij})_{n \times n}$ è la *Matrice dei Costi*

(o *dei Pesi* o *delle Distanze*) dove l'elemento c_{ij} corrisponde al costo dell'arco che unisce il nodo i al nodo j in G.

Dipendentemente dalla natura della matrice (e di conseguena dalla natura di G), le istanze di TSP vengono suddivise in due classi:

Simmetrico (STSP) nei casi in cui G è non orientato (C è simmetrica);

Asimmetrico (ATSP) quando al contrario G è orientato.

Considerando che qualsiasi grafo non orientato può essere rappresentato come un orientato sostituendo ogni arco con due orientati in entrambe le direzioni, la variante simmetrica può essere vista come un caso particolare di ATSP ([1], pag.3).

In una classica formulazione di *pickup and delivery problem* il vincolo di visita impone che, dati O e D, rispettivamente l'insieme delle origini e delle destinazioni, per poter visitare il nodo di scarico $d_i \in D$ è necessario avere già visitato il relativo nodo di carico $o_i \in O$. Da ciò si deduce come i due insiemi debbano essere necessariamente equipotenti e come i nodi di entrambi siano raggruppati a coppie.

1.1 Rear Loading

Nel mio caso ho dovuto considerare un'ulteriore restrizione nell'ordine di visita: il vincolo di *Rear Loading*. Siano $o_l \in O$ l'ultimo nodo di carico visitato ed $d_l \in D$ la destinazione ad esso associata; O' e D' rispettivamente l'insieme delle origini e delle destinazioni già servite, $A \equiv (O \cup D) \setminus (O' \cup D')$ l'insieme dei nodi non ancora raggiunti. All'interno di quest'ultimo individuo un sotto-insieme A' dei nodi prossimi ammissibili applicando il vincolo di *Rear Loading*:

$$A' \subset A \equiv \begin{cases} d_l \\ o_i \notin O' \end{cases}$$

La scelta del nodo successivo è quindi limitata alla destinazione associata all'ultima origine visitata ed alle origini non ancora raggiunte.

1.2 Obiettivi

Durante lo sviluppo di questa tesi mi sono dovuto confrontare con la totale assenza in letteratura di lavori riguardanti il TSP con *Rear Loading*. Non esistono istanze scritte appositamen-

te né tantomeno *benchmarks* o test significativi. Il mio obiettivo è quello di esplorare questa variante ancora così poco studiata e sviluppare differenti approcci per la sua risoluzione nell'ottica di estendere e migliorare il lavoro svolto da Cassani in [2]. Come primo passo ho progettato e studiato un algoritmo esatto basato su tecniche di *Programmazione Dinamica* che mi ha permesso di ottenere i valori ottimi dei percorsi partendo da istanze standard opportunamente adattate. Successivamente ho applicato a differenti politiche di Ricerca Locale la *Tabu Search* allo scopo di renderle più efficaci tramite il supporto di una *meta-euristica* di comprovata flessibilità e potenza.

1.3 Organizzazione del lavoro

Nella stesura di questo elaborato ho seguito la seguente struttura nell'organizzazione dei capitoli:

- Capitolo 1: una breve descrizione del lavoro ed alcuni richiami teorici riguardanti il problema del "TSP with Rear Loading".
- Capitolo 2: la prima parte della trattazione riguarda lo studio e lo sviluppo dell'algoritmo esatto basato su *Programmazione Dinamica*; descrivo inoltre le principali strutture dati come la matrice triangolare sulla quale avviene l'esplorazione ed il vettore hash.
- Capitolo 3: la seconda parte riguarda l'analisi degli algoritmi euristici ai quali ho affiancato la ricerca Tabu, trattata nella prima parte del capitolo insieme alla sua variante "adattativa", la *Reactive Tabu Search*. Espongo infine la catena di elaborazione gestita tramite politica di *Variable Neighbourhood Descent*.
- Capitolo 4: in questa ultima parte riporto i risultati ottenuti durante i numerosi test che ho effettuato per valutare prestazioni e qualità sia dell'algoritmo esatto che degli euristici.
- Capitolo 5: per concludere la trattazione ho riportato alcune considerazioni finali sui risultati ottenuti e le prospettive di ampliamento che si aprono per il futuro.
- Appendice A: in questa prima appendice descrivo in che modo le istanze *TSPLib* vengano modificate per generare i nuovi grafi casuali che mi sono stati indispensabili per effettuare delle prove significative.

Capitolo 2

Programmazione Dinamica

Come primo approccio alla risoluzione del problema ho puntato su di un algoritmo esatto basato su tecniche di *Programmazione Dinamica*.

2.1 Le strutture dati

In questa sezione descrivo le principali strutture dati utilizzate per la modellizzazione del problema.

La prima rappresenta il singolo nodo del grafo e contiene informazioni sufficienti per identificare un punto sul piano euclideo.

Le seguenti sono le più significative e necessarie di una trattazione maggiormente dettagliata.

2.1.1 Grafo

Il grafo viene trattato come matrice di adiacenza nodo-nodo: una matrice di naturali $M[N \times N]$ dove N è il numero di nodi del grafo (la dimensione del problema). In ogni cella $g_{i,j}$ è memorizzata la distanza euclidea tra i nodi i e j che rappresenta il costo di percorrenza dell'arco. Ho deciso di precalcolare l'intera matrice per evitare di ricavare le distanze durante la risoluzione; ho ricondotto al dominio naturale tutti i valori reali risultanti tramite la funzione di libreria ANSI `ceil(3)` dalla quale, sottoponendo il valore *x*, ottengo il minimo intero non minore di *x*.

2.1.2 Stato

Questo record rappresenta lo stato di un *path* ad ogni istante che, partendo dal nodo deposito, viene costruito passo per passo avanzando nell'esplorazione. Ognuno di essi è infatti un cammino differente costruito da una serie di scelte precedenti ricostruibili tramite le strutture in esso conservate e può essere formalizzato con la quadrupla $S = (s, D', u, c)$ dove gli elementi sono nell'ordine:

- uno stack *lifo* delle origini già visitate ma senza che sia stata raggiunta la relativa destinazione;
- un set di valori booleani utilizzati come flag per marcare le destinazioni già visitate;
- l'ultimo nodo raggiunto la cui visita ha generato lo stato stesso;
- il costo del path coperto.

La visita di un nodo provoca una transizione di stato ottenuta con la modifica delle struttura sopra elencate. Per quanto riguarda lo stack di nodi pickup, l'utilizzo della politica LIFO rispecchia il vincolo di *Rear-Loading* dove l'unica destinazione raggiungibile è quella associata all'ultima origine raggiunta (ottenibile con un'operazione di `pop`).

2.1.3 Transizioni di Stato

Dato uno stato, è necessario predisporre una funzione che, visitando un nodo, provveda alla transizione ed all'aggiornamento delle strutture dati per rappresentare l'avanzamento del percorso coperto aggiungendo un nodo. Le funzioni si differenziano a seconda che il nodo raggiunto sia di *pickup* o di *delivery*. Secondo quanto detto in seguito (Paragrafo 2.2.1), non tutte le transizioni devono essere portate a termine.

Visita di un'origine

Siano S lo stato corrente ed o l'origine visitata. Lo stato prossimo S' viene generato aggiornando le strutture dati secondo le regole:

$$S' = \begin{cases} s' = push(s, o) \\ D' = D \\ c' = c + d(u, o) \\ u' = o \end{cases}$$

dove la funzione $d(u, o)$ ritorna la distanza euclidea fra i nodi u ed o recuperandone il valore precalcolato dalla matrice di adiacenza M. Per realizzare lo stack utilizzo un vettore allocato dinamicamente: partendo da una dimensione iniziale, non appena risulta satura l'area di memoria viene riallocata (`realloc(3)`) aggiungendo un blocco di grandezza costante impostata tramite macrodefinizione. Con questa implementazione mi è stato possibile sfruttare la funzione di libreria standard per la copia di blocchi di memoria (`memcpy(3)`) sicuramente molto efficiente. Ne giova anche la fase di deallocazione effettuabile con un'unica chiamata a `free(1)`.

Visita di una destinazione

Siano S ed S' come nel paragrafo precedente; sia inoltre d la destinazione raggiunta associata all'origine o. La transizione può essere descritta come:

$$S' = \begin{cases} s' = pop(s) \\ D' = D \cup d \\ c' = c + d(u, d) \\ u' = d \end{cases}$$

2.1.4 Matrice di esplorazione

La struttura dati usata è una matrice triangolare E sulla quale durante la risoluzione vengono memorizzati tutti gli stati.

A causa dei vincoli e del fatto che ad ogni passo è possibile visitare un solo nodo per volta (c'è un solo "veicolo"), il movimento che l'algoritmo effettua assume una conformazione *Manhattan*. Infatti, raggiungendo un nodo di carico, si effettua uno spostamento incrementale

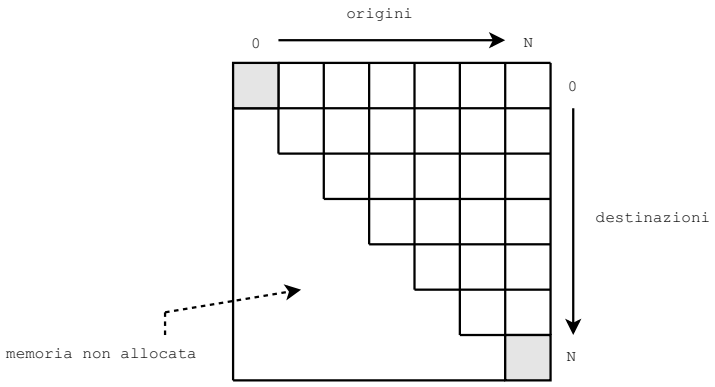

Figura 2.1: matrice di esplorazione E con gli assi di spostamento lungo i quali si muove l'algoritmo. Le due celle opache rappresentano la posizione iniziale $E[0,0]$ e finale $E[N,N]$ di tutti i percorsi.

lungo l'asse delle origini; viceversa lungo l'asse delle destinazioni nel caso di visita di un nodo di scarico.

In ogni cella, le coordinate rappresentano lo stato di esplorazione del grafo. Sia $e = E[i,j]$ la cella di coordinate i, j: essa conterrà tutti gli stati generati in seguito alla visita di i origini e di j destinazioni. Una qualsiasi coppia di coordinate naturali è valida se identifica una cella all'interno dello spazio riservato per E; questo avviene se risulta soddisfatta la condizione:

$$(i \geq j) \ \wedge \ (i \leq N) \ \wedge \ (j \leq N)$$

La risoluzione parte dalla posizione $E[0,0]$ ed ha come obiettivo il raggiungimento della $E[N,N]$.

2.1.5 Cella

Ogni posizione $[i,j]$ della matrice di esplorazione è un record contenente tutte le informazioni relative agli stati generati in seguito alla visita di i origini (colonne) e j destinazioni (righe). I puntatori ad essi sono memorizzati in N tabelle di hashing dove ognuna è utilizzabile accedendo al vettore descritto al punto [2] dell'elenco seguente: ad ogni indice posso raggiungere tutti gli stati generati dal livello precedente in seguito alla visita del nodo u. La singola cella è rappresentata in Figura 2.2 e le strutture dati che ho utilizzato sono:

[1] un record costituente la singola cella della matrice d'esplorazione; contiene [2] ed i dati necessari al calcolo delle statistiche ed alla gestione delle strutture;

[2] un vettore di puntatori a record contenenti ognuno una tabella di hashing presso la quale sono mantenuti tutti gli stati non dominati aventi lo stesso nodo u come ultimo visitato;

[3] la funzione di hash non è una vera e propria struttura dati ma, dato il suo ruolo nel funzionamento della cella, è necessario menzionarla. Fornendo in input lo stack di origini "aperte" s, viene restituita da $h(s)$ la testa della sotto-lista di stati all'interno della quale completare il test di dominanza (trattato in seguito nel Paragrafo 2.2.1). La funzione di ricerca vera e propria è descritta nel Paragrafo 2.1.6;

[4] la tabella di hashing vera e propria è rappresentata da un vettore di puntatori a liste. La sua lunghezza (d'ora in poi chiamata parametro $\mathfrak{K}$) rimane costante durante l'esecuzione ed è impostata staticamente a tempo di compilazione. Per ottenere i valori ottimali ho condotto alcuni test i cui risultati sono riportati nel Paragrafo 4.1 a pagina 36. Per una trattazione completa consultare il paragrafo seguente.

[5] una semplice lista a puntatori doppiamente collegata mantenuta ordinata per costo crescente. Ad ogni invocazione della funzione di hash viene ritornata la testa della stessa contenente il riferimento allo stato più promettente. Non ho dovuto creare una struttura dati ad hoc per mantenere efficientemente ordinati gli stati in quanto, per il controllo della dominanza, essa deve comunque essere scandita completamente: durante il processo rilevo la posizione corretta (in coda all'ultimo stato con costo minore) nella quale inserire lo stato da creare. Ottengo così l'ordinamento in modo "gratuito" sfruttando il tempo utilizzato per un'altra operazione.

Gli stati vengono quindi raggruppati in base all'ultimo nodo che hanno raggiunto: ciò è risultato molto comodo nella verifica della terza condizione di dominanza (Paragrafo 2.2.1) in quanto mi permette di considerare per ogni nodo visitabile solo gli stati che la soddisfano escludendo gli altri senza effettuarne il controllo.

2.1.6 Vettore e Funzione di Hashing

La tabella di hashing non è altro che un vettore di puntatori a liste ordinate di stati. Grazie a questo espediente mi è stato possibile evitare di scandire la lista di tutti gli stati ma solo

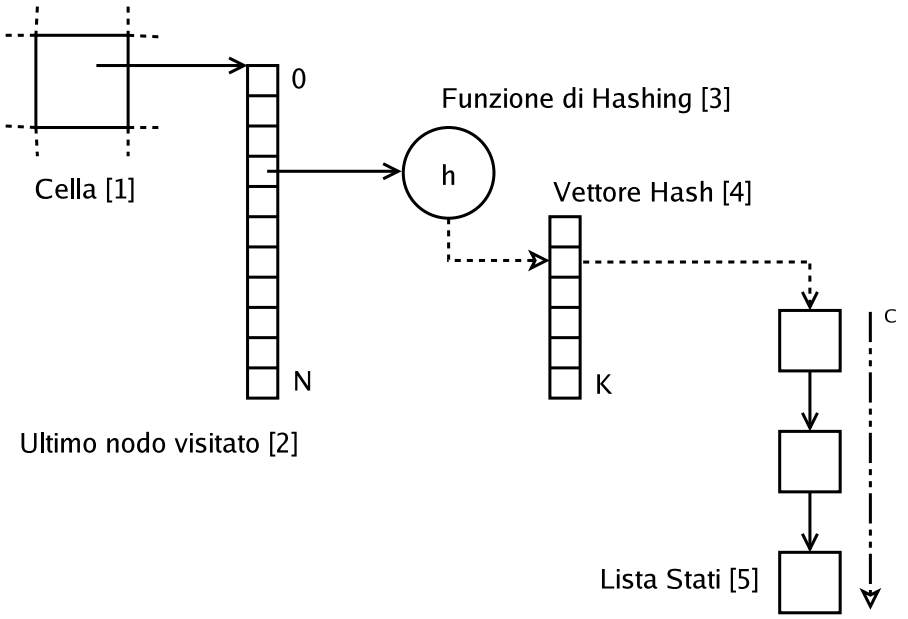

Figura 2.2: strutture dati utilizzate per una cella della matrice d'esplorazione. Descritte in dettaglio nel Paragrafo 2.1.5

un suo sotto-insieme. La funzione $\mathfrak{h}(s)$ processa lo stack tramite un semplice metodo di *Hash con Divisione* per ricavare l'indice presso il quale trovare la lista di stati corretta:

$$\mathfrak{h}(s) \ : \ \Big(\sum_{i=1}^{|s|} s_i \Big) \ mod \ \mathfrak{K}$$

dove $\mathfrak{K}$ è la dimensione del vettore di hashing scelta arbitrariamente e costante, impostata tramite macrodefinizione ed indipendente dalla cella. Proprio questo parametro è stato soggetto ad un necessario *tuning* che mi ha imposto di trovare il valore ottimale per il funzionamento del solutore; i risultati di questa fase sono riportati in Tabella 4.1 a pagina 38. Purtroppo la funzione utilizzata non mi permette di raggruppare solo gli stack identici ma anche quelli *simili*, ciò non mi consente di evitare del tutto il test di uguaglianza fra stack della stessa lista.

2.2 L'algoritmo

In questo paragrafo descrivo l'algoritmo che ho realizzato per effettuare l'esplorazione della matrice degli stati E. Inizio con l'analisi delle regole di dominanza e per il calcolo dei *bounds*, necessarie alla *Programmazione Dinamica*, per poi passare all'algoritmo vero e proprio ed alla relativa stima della complessità computazionale.

2.2.1 Dominanza

L'utilizzo di un metodo di *Programmazione Dinamica* richiede dei criteri che permettano di decretare la dominanza tra una coppia di stati. Durante il procedimento, solo coloro che non vengono dominati da nessun altro devono essere mantenuti per dare origine a quelli del livello successivo; in caso contrario possono essere scartati evitando la generazione di nuovi path che non potrebbero dare luogo a percorsi ottimi come affermato dal *Principio di Ottimalità.* Ho definito la relazione di dominanza come:

$$S' \succ S \equiv (D' \supseteq D) \wedge (s' \equiv s) \wedge (u' = u) \wedge (c' \leq c)$$

Lo stato S' domina S se (con $S' \neq S$):

1. l'insieme delle destinazioni visitate D è sottoinsieme di D';
2. lo stack s' è uguale ad s;
3. entrambi gli stati sono stati generati dalla visita allo stesso nodo;
4. il costo del path $c' \in S'$ è minore o uguale a $c \in S$.

Solo se tutte le condizioni sono verificate si ha dominanza, in caso contrario gli stati non sono comparabili e devono essere mantenuti ed espansi entrambi. E' inoltre valida la relazione di transitività per tutte le condizioni. Il controllo di uguaglianza degli stack viene effettuato tramite una singola chiamata alla funzione di libreria ANSI `memcmp(3)` che permette di verificarne la validità su porzioni di memoria di dimensione arbitraria in modo efficiente.

2.2.2 Bounding

Allo scopo di limitare il più possibile l'esplosione combinatoria del numero di stati generati ho imposto quanti vincoli ho desunto dalla formulazione del problema. Il mio obiettivo è

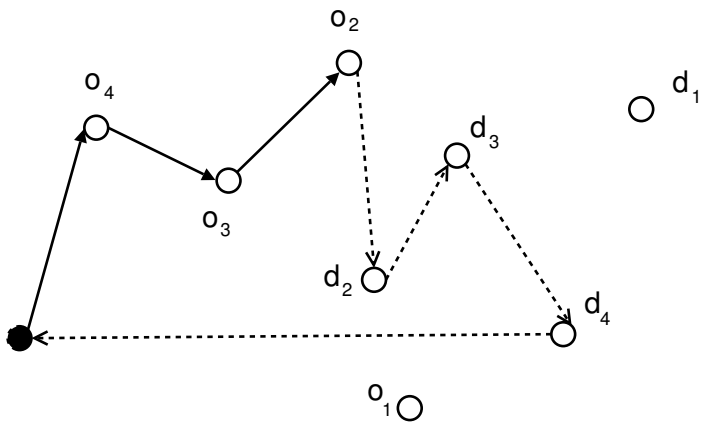

Figura 2.3: calcolo del primo *Lower Bound*. Il percorso continuo rappresenta il cammino già completato mentre la parte discontinua descrive le distanze che entrano a far parte del valore del limite. Come si nota dalla figura il valore è dato dal percorso necessario per chiudere lo stack associato ad uno stato.

stato quello di calcolare e migliorare il valore del *lower bound* associato ad ogni stato aggiungendo tutte le stime utili a prevedere con il maggior anticipo possibile il costo minimo dei percorsi generati.

Il primo *lower bound* è dato dal costo di chiusura del *path*. Ad ogni stato ho associato uno *Stack Bound*; il calcolo considera il costo del percorso da coprire per raggiungere tutte le destinazioni associate alle origini nell'ordine in cui queste ultime sono estratte dallo stack. Come si vede dalla Figura 2.3, il costo del *path* aggiuntivo è quello che permette di chiudere tutte le coppie *pickup-delivery* aperte.

Nel tentativo di migliorare ulteriormente il limite fin qui calcolato, ho introdotto un *Lookup Bound* da aggiungere allo *Stack Bound*. Per ricavarlo applico la seguente metodologia ad ogni stato generato in via definitiva (che abbia superato in controllo di dominanza):

1. il percorso iniziale è quello fin qui considerato per il calcolo dello *Stack Bound*. É importante sottolineare che sto calcolando una stima del costo minimo che assumeranno i *tour* generati dallo stato corrente; in tutto il procedimento non mi curo del mantenimento dell'ammissibilità.

2. Per ogni nodo non ancora raggiunto ricavo il minimo costo di inserimento all'interno di uno dei segmenti del *tour* prodotto dal passo precedente; in Figura 2.4 si vede come

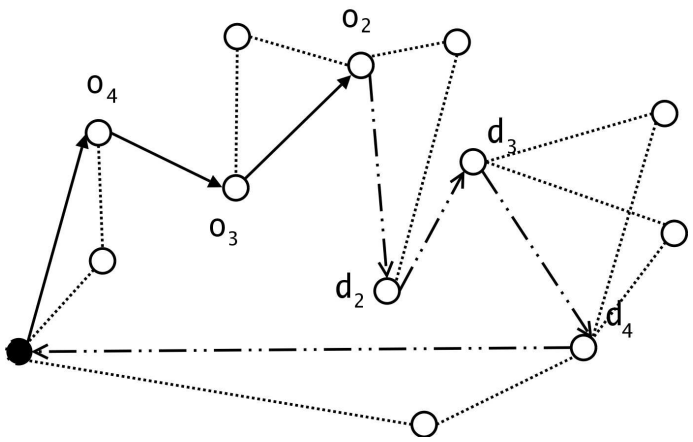

Figura 2.4: prima fase, assegnazione dei nodi non visitati agli archi. Ognuno dei valori è calcolato come inserimento di minimo costo all'interno del percorso prodotto per il calcolo del primo bound.

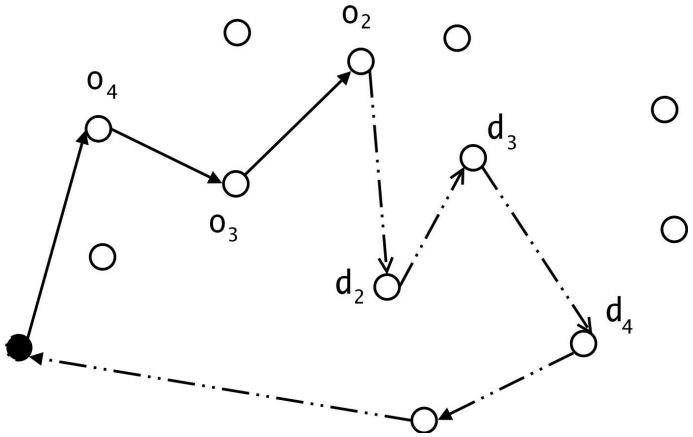

Figura 2.5: seconda fase, rilevamento dell'inserimento peggiore. Il costo degli archi discontinui rappresenta il limite vero e proprio che andrà aggiunto allo *Stack Bound*.

ognuno di essi venga assegnato all'arco in luogo del quale potrebbe essere inserito in modo ottimo. Questo valore rappresenta la stima minima del valore di costo che dovrà essere sicuramente pagato per ottenere una soluzione completa ed ammissibile.

3. Rilevo il più dispendioso fra questi inserimenti: il suo valore rappresenta il *Lookup Bound* vero e proprio che vado a sommare allo *Stack Bound* (Figura 2.5).

Per essere generato con successo il *lower bound* complessivo dello stato deve essere minore dell'*upper bound* globale a tutto l'algoritmo calcolato tramite ricerca locale. Il numero di stati eliminati grazie al *bounding* (Tabella 4.4) rivela come questo approccio sia stato estremamente utile.

2.2.3 Esplorazione

L'esplorazione, come descritto nel paragrafo 2.1.4, avviene effettuando una serie di spostamenti sulla matrice triangolare E. In realtà essa è stata organizzata in livelli di esplorazione (Figura 2.6) dove in ogni strato sono incluse tutte le celle direttamente raggiungibili dal precedente.

Siano $L_i \subset E$ il livello attuale di avanzamento dell'algoritmo, $e_1, e_2, \ldots, e_l \in L_i$ le celle del livello attuale ed $r(e)$ la funzione che, data una posizione, ritorna l'insieme delle raggiungibili. Posso definire il prossimo livello di esplorazione come:

$$L_{i+1} = r(e_1) \cup r(e_2) \cup \ldots \cup r(e_l)$$

La politica adottata è quella dell'avanzamento *breadth-first*: dato un livello, ne vengono esplorati tutti gli stati generando le celle che costituiranno il successivo.

L'algoritmo è descritto sotto forma di pseudo-codice in Algoritmo 1; la condizione di terminazione è che nel livello corrente si trovi la cella $E[N, N]$ nella quale i path hanno visitato tutte le coppie di nodi.

Come già esposto nel Paragrafo 2.2.1, uno stato viene preso in considerazione solo quando non ne esiste nessun altro che lo domini tra quelli appartenenti alla *stessa cella*. Il controllo viene effettuato prima che abbia inizio la transizione vera e propria in modo da evitare la creazione di strutture dati che andrebbero immediatamente deallocate dopo poche istruzioni. Non è necessario che venga generato per intero il livello attuale per poi essere "ripulito" dagli

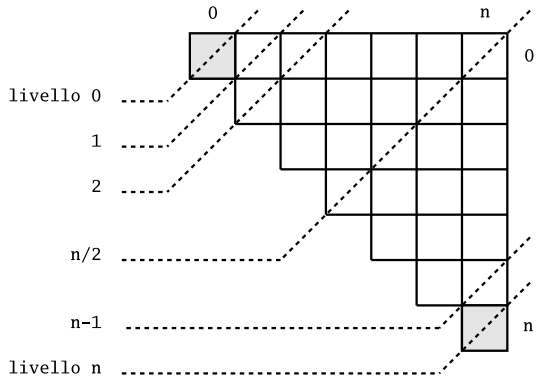

Figura 2.6: stratificazione in livelli delle celle della matrice di esplorazione.

stati dominati: oltre ad essere un procedimento facilmente evitabile sarebbe un enorme spreco di risorse. Quello che avviene è un controllo incrementale, effettuato durante l'esecuzione del ciclo di generazione. Invocando la funzione di transizione creo un'istanza temporanea di stato prossimo: se e solo se anche una sola delle quattro condizioni di dominanza non viene soddisfatta per ogni confronto, lo stato dovrà essere generato, in caso contrario si passerà oltre evitando che il dominato venga espanso nel seguito della risoluzione (non viene inserito nella lista). Per ogni posizione del vettore della cella attuale è memorizzata una tabella hash che, dato l'insieme di flag di nodi di scarico da verificare, permette il raggiungimento in tempo costante del sotto-insieme per il quale il confronto è significativo: solo tra quelli raggruppati all'interno della stessa lista (output della funzione h(s)) è possibile infatti riscontrare uno o più esiti positivi della prima condizione (Paragrafo 2.2.1).

La relazione di dominanza viene verificata tra il nuovo stato e quelli presenti nella lista; il test avviene in quest'ordine fino a quando il candidato all'inserimento non domina a sua volta uno stato già presente. Quest'ultimo viene eliminato, il nuovo entra definitivamente a far parte della cella ma l'algoritmo non interrompe la scansione: da questo momento l'ordine del test è invertito per rimuovere quei *path* che vengono dominati dall'ultimo arrivato. Questo è reso possibile dalla transitività della relazione di dominanza.

Algoritmo 1 Costruzione del path tramite *Programmazione Dinamica*. La funzione $t(S, v)$ effettua il processo di transizione (Paragrafo 2.1.3); l'insieme U_e è il vettore hash della cella e attuale; le funzioni $\texttt{lookupb}(S')$ e $\texttt{stackb}(S')$ calcolano rispettivamente il *Lookup Bound* e lo *Stack Bound* associati allo stato S'; $upperb$ è l'*upperbound* ottenuto tramite gli algoritmi euristici del Capitolo 3.

$e \ = \ E[0,0]$

BEGIN

$\forall e \in E$ $O(N^2)$

| $\forall u \in U_e$ $O(N)$

| | $S \ = \ (s, D, u, c)$

| | $\forall v \in N$ $O(N)$

| | | $S' \ = \ t(S \in U, v)$

| | | $L' \ = \ \mathfrak{h}(s \in S')$

| | | $\forall S'' \in L'$ $O(m)$

| | | | $\texttt{se}\ S' \not\prec S''$

| | | | $\texttt{se}\ (\texttt{lookupb}(S') + \texttt{stackb}(S')) \ < \ upperb$

| | | | $L' \ \equiv \ L' \cup S'$

| | | | $\texttt{se}\ S'' \prec S'$

| | | | $L' \ \equiv \ L' \setminus S''$

END $O(mN^4)$

$e \ = \ E[N,N]$

2.2.4 Complessità

Posso stimare la complessità computazionale dell'algoritmo che ho usato per il solutore (Algoritmo 1) con:

1. $\forall e \in E$, scansione di tutte le celle della matrice triangolare: $O(N^2)$;
2. $\forall U \in e$, scansione del vettore di tabelle di hashing, una posizione per ogni nodo del grafo: $O(N)$;
3. recupero della lista-sottoinsieme corretta tramite hash: $O(1)$. Nonostante il calcolo della chiave di hashing viene effettuato tramite una sommatoria, non scandisco completamente lo stack ogni volta che questo viene alterato; la chiave dello stack ereditata dallo stato padre è alterata diminuendone o incrementandone il valore a seconda che un nodo venga estratto o inserito. Il tempo necessario all'aggiornamento è quindi costante.
4. $\forall S'' \in L'$, scansione della sotto-lista prossima di stati recuperata dal vettore: $O(m)$ (con $m \ll m'$, dove il secondo termine rappresenta la lungezza della lista degli stati completa, senza utilizzo di vettori di hashing). Questo valore dipende interamente dalla scelta del parametro $\mathfrak{K}$ come mostrato dai risultati di Tabella 4.1 a pagina 38.
5. $\forall v \in N$, generazione dello stato prossimo per ogni nodo visitabile; non tutti saranno sempre ammissibili ma nel caso pessimo ho: $O(N)$;

per una complessità d'insieme stimabile in mN^4.

Capitolo 3

Algoritmi Euristici

Per quel che riguarda un approccio di tipo euristico al problema ho scelto di utilizzare diverse politiche di esplorazione dell'intorno alle quali applicare la *Tabu Search*, trattata in dettaglio nel Paragrafo 3.1.

Tutti gli algoritmi descritti in seguito necessitano come dato in ingresso un percorso completo ed ammissibile; la qualità dello stesso non ne influenza il funzionamento anche se la scarsa ricerca effettuata dall'euristica costruttiva potrebbe posizionare il punto di partenza in un ottimo locale dal costo molto alto.

Essendo la posizione del nodo deposito prefissata per ogni istanza, ho utilizzato un semplice *Nearest Neighbour* dovendo rinunciare alla variabile di casualità introdotta dal *Randomized Nearest Neighbour*. Questo inconveniente è risultato superfluo grazie all'utlizzo della ricerca *Tabu* la quale mi ha permesso di sfruttare la sua capacità di "sbloccarsi" dagli ottimi locali per diversificare il più possibile la soluzione ricavata molto velocemente tramite l'euristica costruttiva.

3.1 Tabu Search

La *Tabu Search* è una *meta-euristica* che fornisce ad un algoritmo di ricerca locale gli strumenti per evadere un ottimo locale che lo farebbe terminare prematuramente. Essa è stata utilizzata, ad esempio, per la risoluzione del classico Traveling Salesman Problem simmetrico (Malek [3]) e, come sottolineato da Tsubakitani ed Evans [4], la dimensione della memoria Tabu influenza pesantemente le prestazioni generali della ricerca. La sua natura di

meta-euristica dalla grande flessibilità la rende adatta a problemi molto vincolati come nel mio caso.

La memoria Tabu può contenere, a seconda della natura del problema, un insieme di mosse o attributi di soluzioni considerate proibite, non ripetibili fintanto che vi risiedono. La finestra temporale di permanenza è comunemente chiamata *Tabu Tenure*.

Grazie alla *Tabu Search* il generico algoritmo di ricerca locale al quale viene applicata può essere modificato per accettare mosse peggioranti facendo in modo che possa sfuggire ad un ottimo locale che, nonostante sia associato ad una pessima soluzione, lo farebbe comunque terminare. La memoria Tabu aiuta la ricerca ad evitare la ripetizione di mosse già effettuate recentemente (o a considerare la stessa soluzione), il che comporterebbe un inutile spreco di tempo di calcolo. É importante sottolineare che non si fuga del tutto l'eventualità di incorrere in cicli nell'esplorazione, nonostante un accorto dimensionamento del *Tabu Tenure* possa diminuirne l'incidenza.

La possibilità di variare la finestra temporale rappresentata dal parametro *T* (*Tabu Tenure*) permette di forzare l'algoritmo a comportarsi in modo differente:

- con *T* piccolo si aumenta il livello di *intensificazione* della ricerca; l'utilizzo di un valore troppo ridotto porta però al pericolo di ritornare ad un ottimo locale già visitato ed impedisce l'esplorazione in profondità dello spazio delle soluzioni;

- con *T* grande aumenta la *diversificazione* consentendo il raggiungimento di un possibile ottimo locale (o globale) molto lontano dalla soluzione di partenza. Di contro, l'utilizzo di un *T* troppo ampio porta allo spreco di tempo di calcolo per la ricerca in zone inutili dello spazio impedendo di intensificare soluzioni promettenti già raggiunte.

Durante i test effettuati ho fatto riferimento alle considerazioni di Glover [5] e Tsubakitani [4] per la scelta del parametro di *Tabu Tenure*. Ho optato per mantenere nella memoria Tabu gli archi eliminati in modo che per un certo numero di passi essi non possano essere reinseriti. Questo mi ha permesso di evitare di ricadere su ottimi locali già raggiunti e profondamente esaminati, di ripetere mosse effettuate con troppa frequenza e di tutelarmi, senza comunque evitarli completamente, contro possibili cicli di esplorazione. Nell'implementazione ad ogni arco ho associato una marca temporale che segnala il passo di ricerca più recente nel quale è stato rimosso dal tour. L'inserimento di un nuovo arco avviene con successo se e solo se la seguente asserzione

$$(t = init) \vee (t + T < step)$$

Algoritmo 2 : **schema di *Tabu Search***. La descrizione si riferisce alla struttura utilizzata per tutti gli algoritmi di ricerca locale. Con I mi riferisco all'intorno di ricerca; $step$ è la marca temporale del passo corrente mentre $limit$ è il limite di terminazione; $Tabu[move]$ è il valore associato alla mossa, identificata con $move$, contenuto nella memoria; G e G_{best} sono rispettivamente il *tour* attuale e quello globalmente migliore. La funzione `valuta()` esegue la valutazione del costo della mossa mentre la funzione `esegui()` modifica definitivamente il percorso.

```
BEGIN
finchè step ≤ limite
  |  ∀move ∈ I
  |     |  t = Tabu[move]
  |     |  se (t = init) ∨ ((t + T) < step)
  |     |        c = valuta(move, G)
  |     |        se c < c_cur
  |     |              c_cur = c
  |     |        G = esegui(move, G)
  |     |        Tabu[move] = step
  |     |  altrimenti
  |     |        reactiveTuning(Tabu)
  |  se c_cur < c_best
  |        G_best = G
END
```

risulta soddisfatta. Con t indico la marca temporale associata all'arco, con $init$ il valore relativo ad un arco mai rimosso, con T il valore di *Tabu Tenure* mentre con $step$ la marca temporale del passo corrente.

Per quanto riguarda il criterio di terminazione ho optato per un numero massimo di step di ricerca. Tutti gli algoritmi di ricerca locale che ho utilizzato impiegano un tempo comparabile per eseguire una mossa: sarebbe stato inutile considerare il numero di cicli di CPU come suggerito da Tsubakitani [4]. Lo schema generale di *Tabu Search* che ho utilizzato è rappresentato in Algoritmo 2.

3.1.1 Reactive Tabu Search

Mantenere fisso il parametro di *Tabu Tenure* è la soluzione più semplice ma esistono diverse politiche che permettono di rendere "reattiva" la ricerca in base alla situazione nella quale viene a trovarsi. Questa tecnica è detta di *Reactive Tabu Search*, originariamente ideata da Battiti e Tecchiolli e formalizzata in [6].

Tra le possibilità sono stato costretto, sia a causa della natura della mia variante di Traveling Salesman Problem, sia in seguito alla scelta di cosa mantenere nella memoria, a scartare l'*Aspiration Level* (conservo singole mosse e non soluzioni, il suo utilizzo avrebbe richiesto una dispendiosa fase di ricostruzione del *path* per ogni mossa tabu) e l'*Acceptable Infeasibility* (l'infrazione dei vincoli avrebbe reso inutile le proprietà di mantenimento dell'ammissibilità degli algoritmi che ho utilizzato). Le politiche implementate sono:

- nessuna reazione, la ricerca viene bruscamente interrotta.
- *Tabu Reset*: la memoria Tabu viene completamente svuotata rendendo così ammissibile qualsiasi mossa successiva.
- *Adaptive Tenure*: la dimensione della finestra temporale viene leggermente corretta ogni volta che non è possibile trovare mosse ammissibili.

Nonostante abbia implementato e testato tutte le tecniche, solo l'ultima si è rivelata veramente utile al funzionamento della ricerca: la prima è insufficiente e preclude alla ricerca il raggiungimento di importanti soluzioni, la seconda si è rivelata troppo drastica eliminando l'informazione raccolta fino al momento dell'applicazione.

3.2 Algoritmi di Ricerca Locale

Gli Algoritmi di Ricerca locale descritti in seguito ricevono in input un percorso completo ed ammissibile e, tramite operazioni specifiche per ogni politica, esplorano l'intorno dell'ottimo corrente (che viene aggiornato ad ogni passo) al fine di ricadere in uno migliore. Un algoritmo classico di questo tipo non accetta mosse peggioranti con il grave difetto di non potersi liberare dall'ottimo locale corrente a meno che questo sia adiacente ad un altro migliore. Come si intuisce, questo costituisce un grande handicap soprattutto quando vengono

utilizzate euristiche costruttive che non producono soluzioni soddisfacenti (è il caso del veloce *Nearest Neighbour* usato). Come già esposto, questo difetto diventa insignificante grazie all'utilizzo della ricerca *Tabu*.

In tutti gli algoritmi descritti in seguito viene calcolato, per ogni possibile mossa di esplorazione dell'intorno, il costo del nuovo *tour* nel caso in cui essa venga effettivamente eseguita. Per non dover provvedere all'operazione di precalcolo, ogni volta che un'azione viene considerata, sfrutto le risorse allocate per la memoria Tabu: essa contiene infatti una certa quantità di informazione per ogni possibile mossa; quando una di queste viene valutata per la prima volta, "salvo" il risultato della valutazione di costo in modo che, per tutte le future valutazioni, non sia più necessario ripetere il calcolo.

3.2.1 Scambio di Coppie

La ricerca locale effettuata tramite lo scambio di coppie (O,D) consiste nell'esplorare ogni combinazione di coppie *pickup-delivery* memorizzando ad ogni ottimo locale incontrato il migliore: quest'ultimo sarà il *tour* prodotto dall'algoritmo. In Figura 3.1 è rappresentata la suddivisione in coppie di un percorso e l'effetto di uno scambio effettuato su di esso.

In Algoritmo 3 è riportato lo pseudo-codice. Come si nota dalla descrizione, il criterio di terminazione è dato da un limite di passi oltre il quale l'algoritmo deve interrompere la ricerca. Per ogni combinazione di coppie viene simulato lo scambio valutandone il costo; se e solo se esso è minore dell'ottimo locale migliore trovato fino ad ora, viene memorizzato ed eseguito al termine del passo corrente. Per essere portato a termine, lo scambio deve prevedere mosse che non siano contenute nella memoria tabu, altrimenti viene marcato come inammissibile e scartato; un arco che sta per essere inserito non deve essere stato rimosso da meno di T passi prima. Inoltre, se lo scambio porta ad un ottimo locale che si rivela essere il migliore raggiunto fino ad ora, viene memorizzato come ottimo locale "complessivo": quello rimasto in questa variabile al termine della ricerca viene applicato al percorso iniziale e ritornato come output dell'algoritmo di Scambio di Coppie; inoltre, data l'ammissibilità, qualsiasi scambio effettuato ne garantisce il mantenimento.

Come descritto nel Paragrafo 3.1 la *Tabu Search* non verifica che l'ottimo locale del passo corrente sia migliore del precedente: esso viene comunque eseguito anche se peggiorante.

Senza considerare i vincoli *Tabu*, sono sempre possibili $O(N^2)$ scambi.

Algoritmo 3 : **funzione Scambio di Coppie**. L'algoritmo valuta tutti gli scambi possibili tra le coppie *pickup-delivery* verificando per ognuno se è il migliore del passo corrente e se comporta solo mosse non presenti nella memoria *Tabu*. Il valore finale di $best_{global}$ è il costo del *tour* migliore incontrato durante l'esplorazione dell'intorno.

$best_{global} = costo_{init}$
$best = costo_{init}$
$best_{cur} = \infty$
$best_{swap} = \emptyset$
`finchè` $(step \leq max_{steps})$
  $\forall Coppia_{i,j} \mid i, j \in N$
    $\forall Coppia_{h,k} \mid h, k \in N,\ h, k \neq i, j$
      $costo =$ `valutaScambio`$(Coppia_{i,j},\ Coppia_{h,k})$
      `se` $(costo < best_{cur}) \wedge (costo \neq TABU)$
        $best_{cur} = costo$
        $best_{swap} = \{Coppia_{i,j},\ Coppia_{h,k}\}$
  `se` $(best_{swap} \neq \emptyset)$
    `eseguiScambio`$(best_{swap})$
    $best_{global} =$ `min`$(best_{cur},\ best_{global})$
    $best = best_{cur}$
    $best_{swap} = \emptyset$
    $best_{cur} = \infty$
  `altrimenti`
    `Reactive Tabu Search Tuning`

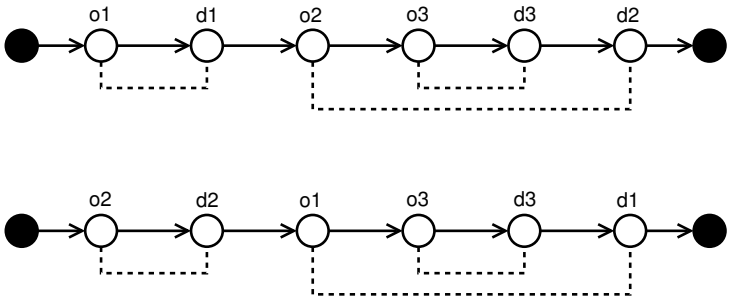

Figura 3.1: suddivisione in coppie *pickup-delivery* di un generico percorso ammissibile ed effetto che ha su di esso lo scambio tra le coppie 1 e 2.

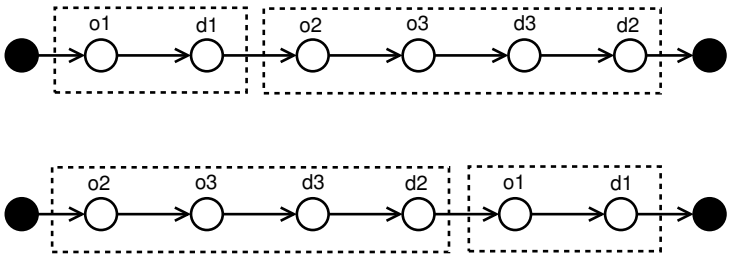

Figura 3.2: suddivisione in blocchi *lifo* di un generico percorso ammissibile ed effetto che ha su di esso lo scambio tra i blocchi 1 e 2. Considero solo le sequenze massimali: la (o3, d3) non è visibile all'algoritmo.

3.2.2 Scambio di Blocchi

La Ricerca Locale effettuata tramite Scambio di Blocchi prevede come operazione preliminare l'esplorazione del *tour* iniziale per la sua suddivisione nelle sottosequenze LIFO più ampie possibili come rappresentato in Figura 3.2. É importante sottolineare che, considerando le sequenze massimali, non si trattano le sotto-sequenze in esse contenute.

Come si nota dallo pseudo-codice riportato in Algoritmo 4 la ricerca opera come la precedente considerando però interi blocchi invece che singole coppie. Essendo ogni elemento una sequenza LIFO completa, lo spostamento di uno qualsiasi di questi mantiene l'ammissibilità del percorso fornito in ingresso.

Senza considerare i vincoli *Tabu* sono sempre possibili $O(M^2)$ scambi con M numero di blocchi rilevati.

Algoritmo 4 : **funzione Scambio di Blocchi**. L'algoritmo valuta tutti gli scambi possibili tra i blocchi *lifo* rilevati nel percorso verificando per ognuno se è il migliore del passo corrente e se comporta solo mosse non presenti nella memoria *Tabu*. Il valore finale di $best_{global}$ è il costo del *tour* migliore incontrato durante l'esplorazione dell'intorno.

$best_{global} = costo_{init}$
$best = costo_{init}$
$best_{cur} = \infty$
$best_{swap} = \emptyset$
`finchè` $(step \leq max_{steps})$
  $\forall Blocco_i \mid i \in N$
    $\forall Blocco_k \mid k \in N, k \neq i$
      $costo =$ `valutaScambio`$(Blocco_i, Blocco_k)$
      `se` $(costo < best_{cur}) \wedge (costo \neq TABU)$
        $best_{cur} = costo$
        $best_{swap} = \{Blocco_i, Blocco_k\}$
  `se` $(best_{swap} \neq \emptyset)$
    `eseguiScambio`$(best_{swap})$
    $best_{global} =$ `min`$(best_{cur}, best_{global})$
    $best = best_{cur}$
    $best_{swap} = \emptyset$
    $best_{cur} = \infty$
  `altrimenti`
    `Reactive Tabu Search Tuning`

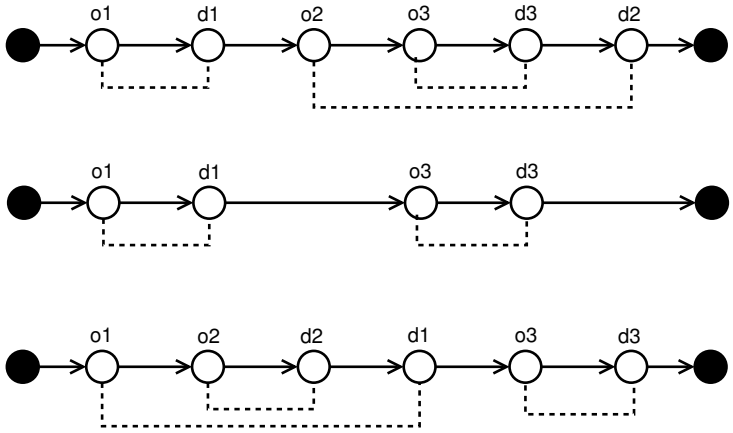

Figura 3.3: suddivisione in coppie *pickup-delivery* di un generico percorso ammissibile ed effetto che ha su di esso la rimozione e successivo reinserimento (rilocazione) della coppia 2 in coda al nodo *pickup* della coppia 1.

3.2.3 Rilocazione di Coppie

L'algoritmo che prevede la Rilocazione di Coppie lavora sugli accoppiamenti di nodi *pickup-delivery* così come per il procedimento descritto nel Paragrafo 3.2.1. A differenza di quest'ultimo, la coppia considerata in ogni passo è una sola che viene rimossa, ricostruita rendendo adiacenti i nodi estratti e, successivamente, reinserita in una nuova posizione del percorso. Gli effetti sono mostrati in Figura 3.3.

Come si nota dalla descrizione riportata in Algoritmo 5 il ciclo più esterno considera tutte le possibili coppie mentre quello immediatamente più interno tutti i nodi, singoli e non accoppiati, del percorso. Il controllo che il nodo corrente k sia differente dai costituenti della coppia i, j non viene effettuato: il risultato della rilocazione sarà semplicemente rendere adiacenti i due *pickup-delivery*.

Anche in questo caso dato che l'algoritmo considera per la rilocazione singole coppie *pickup-delivery* che non sono altro che sequenze LIFO minime, l'ammissibilità garantita dal *tour* iniziale viene mantenuta.

Algoritmo 5 : **funzione di Rilocazione di Coppie**. Il ciclo esterno passa in rassegna tutte le possibili associazioni *pickup-delivery* mentre il più interno considera tutti i singoli nodi del percorso, compresi i costituenti della coppia corrente. Sia il costo derivante dalla valutazione di rimozione che quello di inserimento in nuova posizione, sono soggetti alla verifica di ammissibilità: entrambe le operazioni devono prevedere mosse non *Tabu* per non essere scartate.

$best_{global} = costo_{init}$
$best = costo_{init}$
$best_{cur} = \infty$
$best_{move} = \emptyset$
`finchè` $(step \leq max_{steps})$
  $\forall Coppia_{i,j} \mid i, j \in N$
    $\forall k \mid k \in N$
      $costo =$ `valutaRimozione`$(Coppia_{i,j})$
      $costo = costo +$ `valutaInserimento`$(Coppia_{i,j},\ k)$
      `se` $(costo < best_{cur}) \wedge (costo \neq TABU)$
        $best_{cur} = costo$
        $best_{move} = \{Coppia_{i,j},\ k\}$
  `se` $(best_{move} \neq \emptyset)$
    `eseguiRilocazione`$(best_{move})$
    $best_{global} =$ `min`$(best_{cur},\ best_{global})$
    $best = best_{cur}$
    $best_{move} = \emptyset$
    $best_{cur} = \infty$
  `altrimenti`
    `Reactive Tabu Search Tuning`

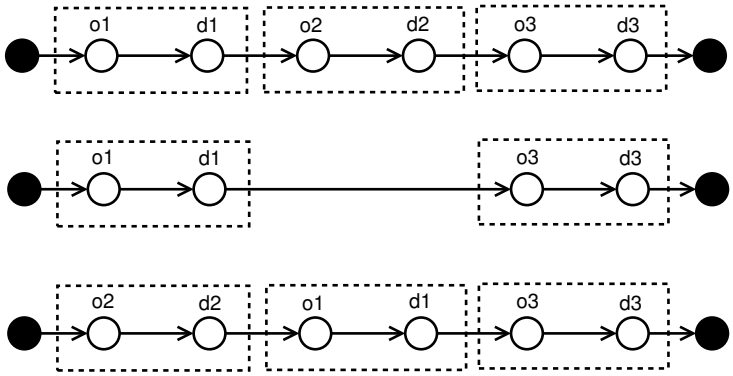

Figura 3.4: suddivisione in blocchi *lifo* di un generico percorso ammissibile ed effetto che ha su di esso la rimozione. Successivo reinserimento (rilocazione) del blocco 2 in testa al blocco 1.

3.2.4 Rilocazione di Blocchi

L'algoritmo di Rilocazione di Blocchi lavora considerando le sottosequenze LIFO complete più ampie possibili rilevate nel percorso in ingresso, come avviene per lo Scambio di Blocchi descritto nel Paragrafo 3.2.2. A differenza della Rilocazione di Coppie, non è necessario il passo di ricostruzione in quanto il blocco rimosso è già completo: i nodi interni non vengono considerati mentre gli unici nodi visibili sono quelli esterni, gli estremi della sottosequenza che si comporta come una sorta di *black-box*. Gli effetti dell'operazione di rilocazione sono rappresentati in Figura 3.4.

Come si vede dalla descrizione riportata in Algoritmo 6, sia il ciclo esterno che quello interno considerano a turno tutti i blocchi rilevati: a differenza del precedente, la base della rilocazione è la sequenza *lifo*; questa scelta si è resa necessaria in quanto, se avessi utilizzato i singoli nodi, ad ogni spostamento sarei stato costretto a ripetere la procedura di suddivisione in blocchi (complessità $O(N)$).

Anche in questo caso l'ammissibilità del *tour* viene garantita in seguito a qualsiasi rilocazione.

Algoritmo 6 : **funzione di Rilocazione di Blocchi**. Entrambi i cicli di scansione operano sulle sotto-sequenze *lifo* rilevate. Per essere ammissibile, sia la rimozione che il reinserimento devono prevedere mosse non contenute nella memoria *Tabu*.

$best_{global} = costo_{init}$
$best = costo_{init}$
$best_{cur} = \infty$
$best_{move} = \emptyset$
`finchè` $(step \leq max_{steps})$
  $\forall Blocco_i \mid i \in N$
    $\forall Blocco_k \mid k \in N$
      $costo =$ `valutaRimozione`$(Blocco_i)$
      $costo = costo +$ `valutaInserimento`$(Blocco_i,\ Blocco_k)$
      `se` $(costo < best_{cur}) \wedge (costo \neq TABU)$
        $best_{cur} = costo$
        $best_{move} = \{Blocco_i,\ Blocco_k\}$
  `se` $(best_{move} \neq \emptyset)$
    `eseguiRilocazione`$(best_{move})$
    $best_{global} =$ `min`$(best_{cur},\ best_{global})$
    $best = best_{cur}$
    $best_{move} = \emptyset$
    $best_{cur} = \infty$
  `altrimenti`
    `Reactive Tabu Search Tuning`

3.3 Variable Neighbourhood Descent

Per la ricerca dell'ottimo, utilizzo tutti gli algoritmi di ricerca locale già visti. Il *tour* iniziale generato tramite *Nearest Neighbour* viene fornito in ingresso ad una catena di elaborazione gestita dalla politica di *Variable Neighbourhood Descent*: come si vede dalla Figura 3.5, le euristiche sono ordinate staticamente secondo un criterio desunto dai risultati di numerosi test; ogni fase prevede un processo completo di ricerca al termine del quale si decide a quale algoritmo passare il *tour* intermedio. Nel caso si ottenga un miglioramento si riparte dall'inizio della catena, in caso contrario si tenta di migliorare avanzando alla fase successiva. Questo ordinamento mi permette di sfruttare al massimo gli algoritmi che esplorano l'intorno con una "granularità" maggiore, come quelli che effettuano scambi, per passare in seguito, solo quando non è più possibile migliorare, a quelli che continuano la ricerca con più precisione considerando le tutte le possibili rilocazioni.

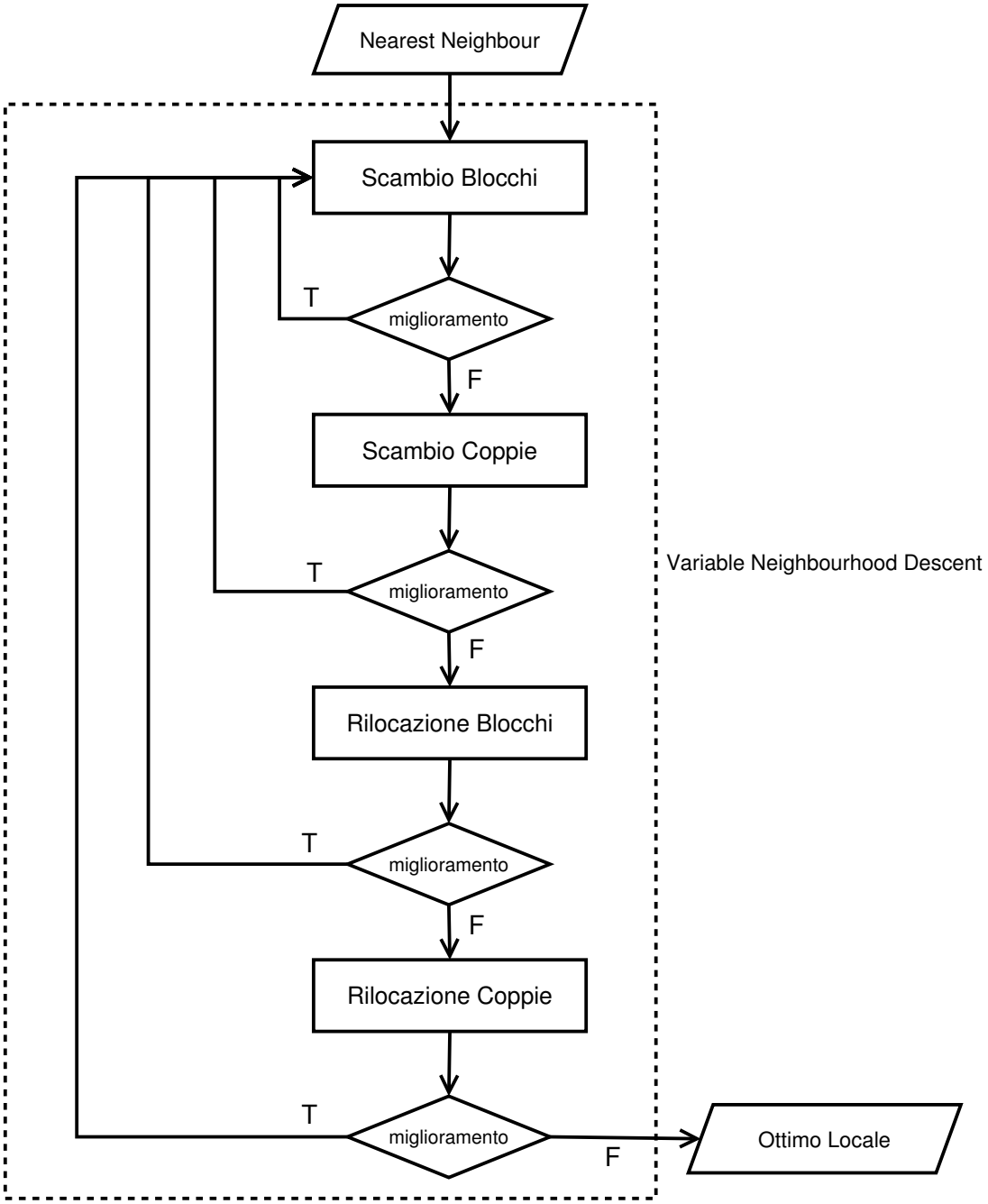

Figura 3.5: *Flow-chart* relativo al funzionamento della catena di elaborazione nella quale utilizzo tutti gli algoritmi euristici visti. La politica di gestione è la *Variable Neighbourhood Descent*.

Capitolo 4

Risultati Sperimentali

Per valutare le prestazioni degli algoritmi realizzati ho condotto numerosi test su istanze standard; in questo capitolo riporto i risultati ottenuti e le relative valutazioni.

Tutte le prove, così come lo sviluppo, sono state portate a termine utilizzando la seguente piattaforma:

- Intel Pentium-M 710, 1,4 GHz
- 512 MB di memoria fisica
- S.o. GNU/Linux 2.6.14.1
- Librerie di sistema GLibC 2.3.5
- Compilatore GCC 4.0.2

Per i dettagli sui flag di compilazione (`std=c99 O3 march=pentium-m fomit-frame-pointer`) fare riferimento al manuale del compilatore[1]. Gli altri tool che ho utilizzato durante lo sviluppo del codice utilizzato nei test sono: GDB 6.3.0, DDD 3.10 e Valgrind 2.0.4.

Le istanze del problema considerate durante i test provengono dalla libreria *TSPLib*, una collezione che ne raccoglie un gran numero, tutte studiate specificamente per l'utilizzo con algoritmi per la risoluzione del Traveling Salesman Problem[2]. Un esempio del formato dei file è riportato in Figura 4.1. Alcune di queste presentano purtroppo una distribuzione dei punti

[1] Using the GNU Compiler Collection, Free Software Foundation, reperibile presso `http://gcc.gnu.org/onlinedocs/gcc-4.0.2/gcc/`

[2] Di fatto, nonostante non sia una libreria standardizzata, è accettata come tale dalla comunità di Ricerca Operativa.

```
NAME : fnl4461
COMMENT : Die 5 neuen Laender Deutschlands (Ex-DDR) (Bachem/Wottawa)
TYPE : TSP
DIMENSION : 4461
EDGE_WEIGHT_TYPE : EUC_2D
NODE_COORD_SECTION
1    5639    6909
2    5652    6142
3    5654    6101
4    5659    6910
...
```

Figura 4.1: Estratto dal file istanza `fnl4461` dalla *TSPLib*. Le informazioni relative ai punti contenute nella sezione `NODE_COORD_SECTION` sono espresse nel formato `NODEID XCOORD YCOORD` .

nello spazio euclideo che non è sufficientemente varia: solo applicando l'euristica costruttiva (*Nearest Neighbour*) riuscivo ad ottenere buone soluzioni prossime all'ottimo globale; il *tour* migliore era spesso ottenuto visitando i nodi nell'ordine in cui comparivano nel file. Per casualizzare la topografia delle città ho scritto un apposito tool; come le istanze vengano modificate è descritto in dettaglio nell'Appendice A.

In questo paragrafo ho riportato all'interno di tabelle i risultati delle prove effettuate. Spesso alcune celle solo vuote; ad esempio, si vedano i valori temporali della colonna "Tempo RL" nella Tabella 4.4. Avendo per semplicità rilevato i tempi di calcolo con la funzione ANSI `clock(3)`, tramite la macrodefinizione `CLOCKS_PER_SEC` mi è stato possibile ottenere una granularità a livello del millisecondo. Tempi inferiori risultano quindi nulli ed ho preferito la cella vuota allo zero (che in caso di intervalli temporali è poco significativo)[3].

4.1 Considerazioni sulla Programmazione Dinamica

In Tabella 4.1 ho riportato i risultati dei test eseguiti sull'algoritmo di *Programmazione Dinamica*. Le colonne riguardano:

[3]Per ulteriori dettagli sulle funzioni e macrodefinizioni di libreria fare riferimento a *The GNU C Library Reference Manual*, Free Software Foundation, reperibile presso `http://www.gnu.org/software/libc/manual/`.

- N: cardinalità del grafo considerato nella risoluzione;
- L: numero di livelli nei quali è stata stratificata la matrice di esplorazione;
- C: celle visitate, è la dimensione della matrice E;
- GEN: numero totale di stati generati;
- DOM: numero di stati scartati per dominanza;
- BND: numero di stati eliminati tramite *bounding*;
- DEL/GEN%: rapporto percentuale fra il numero di stati eliminati sia per *bounding* che per dominanza ed il numero di generati totali;
- T: tempo in secondi impiegato per la risoluzione;
- $\mathfrak{K}$: dimensione del vettore di hashing;
- $\overline{m}$: lunghezza media delle liste degli stati (elemento [5] in Figura 2.2).

Il parametro $\mathfrak{K}$ determina la lunghezza del vettore di hash e di conseguenza la frequenza delle collisioni: per ricavare il valore ottimale relativo ad ogni dimensione d'istanza, ho dovuto effettuare una fase di *tuning* durante la quale ho rilevato sia i tempi di computazione che i valori medi della saturazione delle liste di stati. Come si vede dai dati, non è possibile aumentare arbitrariamente $\mathfrak{K}$ in quanto l'implementazione impone un'*upper bound* dipendente dalla cardinalità del grafo: nonostante $\overline{m}$ diminuisca, la dimensione dei *chunk* (blocchi di allocazione) da gestire diventa tale da peggiorare il tempo richiesto per il completamento. E' interessante notare inoltre come il rapporto fra stati dominati e generati rimanga sostanzialmente costante rivelandosi quindi dipendente solo dalla natura dei dati.

Come si nota, il grande limite dell'algoritmo di *Programmazione Dinamica* sta nell'esplosione combinatoria del numero di stati generati. Nonostante abbia previsto l'eliminazione degli stessi tramite bounding e dominanza, questo rimane il fronte sul quale si dovrebbero prevedere dei miglioramenti.

Come si nota da Figura 4.2, l'andamento del numero di stati generati è simmetrico (con leggeri scostamenti in pochi casi) rispetto al livello $\frac{L}{2}$ con L il numero di strati della matrice di esplorazione.

D022-04g - Gaskell, 1967; Eilon, Watson-Gandy and Christofides, 1971									
N	L	C	GEN	DOM	BND	DEL/GEN%	T (s)	$\mathfrak{K}$	$\overline{m}$
10	12	18	1265	144	651	62,85	–	1	5,04
							–	**10**	**2,05**
							0,010	100	2,05
12	14	30	6882	787	3780	66,36	0,010	1	12,50
							0,010	**50**	**3,66**
							0,020	100	3,66
14	16	38	40464	4783	29401	84,48	0,090	10	7,72
							0,090	**100**	**6,42**
							0,120	300	6,42
16	18	40	95856	9170	63257	75,56	0,330	1000	8,46
							0,320	**2000**	**7,40**
							0,510	3000	7,40
18	20	56	826128	64848	569845	76,83	2,100	1000	25,80
							2,000	**3000**	**8,30**
							2,100	5000	8,30
20	22	68	1343910	189376	698265	66,05	20,450	5000	28,40
							19,987	**8000**	**9,40**
							20,340	10000	9,37

Tabella 4.1: risultati dei test effettuati sul solutore. L'istanza utilizzata è la `D022-04g` dove per ogni dimensione sono stati considerati i primi **N** punti. La colonna $\mathfrak{K}$ indica il parametro del modulo di hash: tra le righe facenti riferimento alla stessa dimensione d'istanza i valori in grassetto sono i migliori rilevati. Per ogni caso ho riportato solamente i tre casi più significativi; $\overline{m}$ è la quantità di stati memorizzati in una singola sotto-lista riportata come media calcolata su tutti i passi di risoluzione, una sorta di indicatore della saturazione globale delle liste-stati.

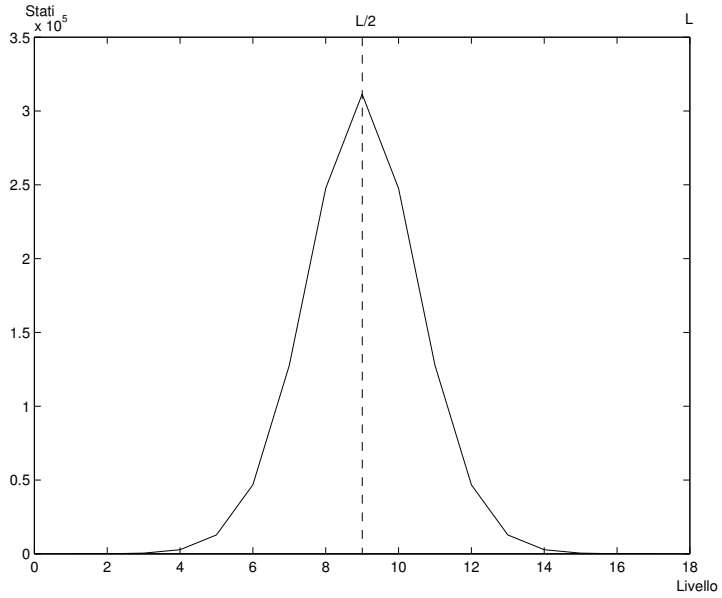

Figura 4.2: andamento del numero di stati visitati all'avanzare dell'esplorazione. Il picco si trova presso lo strato $\frac{L}{2}$ collocato lungo la diagonale della matrice. I valori sono stati rilevati sull'istanza `D022-04g` limitata a 16 nodi.

4.2 Considerazioni sulla Ricerca Locale

Per la quasi totalità degli algoritmi euristici la prima fase della valutazione da affrontare è quella relativa alla taratura dei parametri di funzionamento che, a seconda della natura della ricerca, possono influenzare enormemente sia il tempo richiesto per la computazione, sia la qualità delle soluzioni ricavate.

Nel mio caso i valori chiave da considerare sono:

Tabu Tenure l'ampiezza della finestra temporale all'interno della quale le mosse di ricerca sono considerate "tabu" e quindi non effettuabili. Nel Paragrafo 3.1 ho già illustrato come questo valore possa forzare l'algoritmo verso l'intensificazione o la diversificazione della soluzione.

Limit il limite dei passi di ricerca oltre il quale l'algoritmo è costretto a fermarsi; indispensabile per sopperire alla mancanza di condizione di terminazione della *Tabu Search*. Il

fnl4461 (200 nodi)			
Valore	**Tabu Tenure**	Passi VND	Tempo VND (s)
137701	1	5	11.970
135556	2	9	20.400
128724	**10**	7	15.450
131410	50	12	24.200
133816	100	10	19.400
133816	150	10	18.600
133816	200	10	17.920
133816	250	10	17.500

Tabella 4.2: andamento del valore della soluzione trovata tramite Ricerca Locale al variare del parametro di *Tabu Tenure*. Tutte le prove sono state portate a termine con un fattore di limite per la ricerca su blocchi pari a 2 e con un fattore di limite per la ricerca su coppie pari a $2N$ (400 in questo caso), i migliori riportati in Tabella 4.3. L'istanza è la `fnl4461` limitata a soli 200 nodi.

valore interessa gli algoritmi che adottano politiche per coppie *pickup-delivery* (Paragrafi 3.2.1 e 3.2.3).

Factor l'equivalente di **Limit** per le politiche di esplorazione dell'intorno che fanno uso di blocchi LIFO come quelle dei Paragrafi 3.2.2 e 3.2.4. Per questi metodi infatti non è possibile conoscere a priori in quante sottosequenze il *tour* verrà suddiviso; siccome mi è stato utile esprimere **Limit** in forma di fattore moltiplicativo per la dimensione dell'istanza, così **Factor** rappresenta il fattore relativo al numero di blocchi rilevati in un percorso a tempo di esecuzione.

In Tabella 4.2 ho riportato i risultati delle prove eseguite allo scopo di ricavare il valore ottimale di **Tabu Tenure**. Come si nota chiaramente dai valori, il parametro influenza enormemente sia la qualità del ciclo trovato sia il tempo utilizzato per completare il processo di *Variable Neighbourhood Descent* ed il numero di passi da esso effettuati. Analizzandoli, ho desunto dai dati che le considerazioni di Tsubakitani ed Evans [4] sono poco utili per il mio caso in quanto, come si osserva anche dalla Tabella 4.5, i cicli Hamiltoniani relativi al Traveling Salesman Problem senza vincoli di retro-carica sono molto differenti. Ho quindi stimato che il valore ottimale di Tabu Tenure si trova nell'intervallo $\frac{N}{10} \div \frac{N}{20}$ con N dimensione dell'istanza.

fnl4461 (200 nodi)				
Limit	**Factor**	Valore	Passi VND	Tempo VND (s)
10	1	137812	20	2.190
20	1	136394	13	1.630
50	1	134465	7	2.030
100	1	130865	13	7.360
200	1	129869	11	12.230
200	2	129869	11	12.200
300	2	129839	10	16.590
400	**2**	**128724**	**7**	**15.550**
500	3	128724	7	19.370
600	4	128724	7	23.270

Tabella 4.3: andamento del valore della soluzione trovata tramite Ricerca Locale al variare del limite dei passi di ricerca e del fattore limite per la ricerca basata su blocchi. Il valore del parametro di *Tabu Tenure* è 10, il migliore ricavato dal test riportato in Tabella 4.2. L'istanza è la `fnl4461` limitata a soli 200 nodi.

In Tabella 4.3 sono riportati i dati ottenuti a seguito di test atti a ricavare il valore ottimale di **Limit** e **Factor**. Anche in questo caso i parametri in questione influenzano pesantemente il funzionamento delle euristiche: su valori troppo ridotti la ricerca non ha sufficiente libertà per posizionarsi su di un ottimo locale accettabile; di contro, per valori troppo elevati l'ottimo (presumibilmente globale o con *gap* di ottimalità molto ridotto) viene raggiunto ma la ricerca non viene interrotta utilizzando inutilmente tempo di calcolo. Dai dati ho desunto come il valore di Limit ottimale sia attorno a $2N$ mentre, essendo il numero di blocchi *lifo* dipendente solo dalla disposizione dei nodi del grafo e non dalla dimensione, il valore utile di Factor è compreso tra 2 e 3. É interessante notare come aumentare Limit non significhi protrarre il ciclo di *Variable Neighbourhood Descent* per un maggior numero di iterazioni: nel caso del valore ottimale vengono eseguiti 7 cicli, la soluzione viene migliorata per altrettante volte. Osservando la prima riga si vede come nonostante i miglioramenti successivi siano 20, essi sono di entità estremamente inferiore dato il pessimo valore del *tour* finale.

Per avere una valutazione della qualità delle soluzioni ricavate dalla Ricerca Locale ho condotto alcuni test per quantificarne i *gap* di ottimalità. I valori dei parametri sono quelli ottimali appena illustrati; i risultati di questa fase sono riportati in Tabella 4.4. Dalla colonna dei tempi necessari al completamento della *Programmazione Dinamica* si evince la profonda

Nodi	Opt. TSP	V PD	S%	T PD (s)	V RL	T RL (s)	Stati Bnd	**GAP%**
`fnl4461`								
10	10562	30202	185,9	0,08	30875	–	20	2,23
12	11323	31011	173,8	0,14	31011	–	4262	0,00
14	11423	31063	171,9	0,88	31208	–	35514	0,47
16	11625	32641	180,7	2,50	32672	–	337615	0,09
18	12534	32810	161,7	19,22	32810	0,01	2323426	0,00
20	12335	33654	172,8	302,12	33676	0,01	5786102	0,07
22	14465	41302	185,5	1143,40	41311	0,01	9088472	0,02
`linhp318`								
10	9160	13011	42,0	0,08	13063	–	655	0,40
12	10022	14948	49,2	0,17	15671	–	3938	4,84
12	8427	14930	77,2	0,33	14930	–	23400	0,00
16	12080	17989	48,9	0,56	18221	0,010	63204	1,29
18	11935	21436	79,6	1,83	21763	0,010	569845	1,53
20	13895	23064	66,0	11,37	23251	0,010	2780820	0,81
22	14741	21779	47,7	27,87	21779	0,010	6707700	0,00
`kroE100`								
10	10243	12312	20,2	0,090	12546	–	552	1,90
12	9377	13958	48,9	0,170	14091	–	2855	0,95
14	9332	14236	52,6	0,290	14555	–	18940	2,24
16	10054	15175	50,9	0,490	15175	0,010	67154	0,00
18	11500	20339	76,9	0,850	20380	0,010	239267	0,20
20	11157	16030	43,7	1,880	16030	0,010	813096	0,00
22	11996	17584	46,6	31,090	18392	0,010	10626562	4,60
`att48`								
10	5275	26048	393,8	0,100	26223	–	597	0,67
12	5693	25372	345,7	0,170	26199	–	2843	3,26
14	5933	31919	438,0	0,290	32241	–	17052	1,01
16	6518	34314	426,4	0,570	34657	0,010	92179	1,00
18	6043	34289	467,4	1,230	34290	0,010	410171	0,00
20	6686	33290	397,9	3,010	33330	0,010	1123945	0,12
22	6922	34100	392,6	49,330	34268	0,010	15098387	0,49
`bier127`								
10	42954	71497	66,5	0,100	71497	–	749	0,00
12	22734	45763	101,3	0,180	45763	–	4522	0,00
14	35771	68063	90,3	0,370	68063	–	29254	0,00
16	30121	66183	119,7	1,540	66685	0,010	246837	0,76
18	38699	65016	68,0	2,090	65016	0,010	578915	0,00
20	48683	68911	41,6	720,010	75219	0,010	1109520	9,15
22	49884	69871	40,1	749,560	75477	0,010	15600734	8,02
		$\overline{S}\% =$	161,97					

Tabella 4.4: Risultati dei test effettuati sul diverse istanze *TSPLib*.

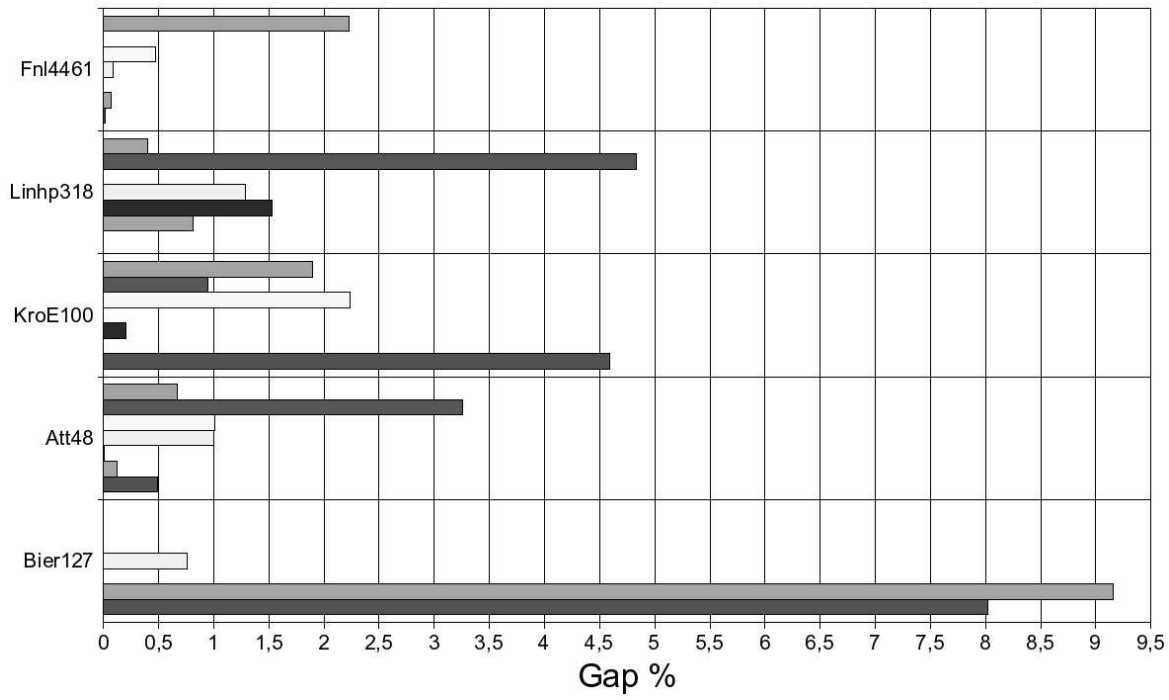

Figura 4.3: istogramma dell'andamento del *gap* di ottimalità percentuale sulle istanze dei test di Tabella 4.4 riguardanti la ricerca locale.

differenza tra quest'ultima e gli algoritmi euristici. Sotto la voce "GAP%" ho riportato i valori del *gap* per ognuna delle istanze: il risultato peggiore presenta una differenza del 9,15% mentre in tutti gli altri casi viene raggiunta l'ottimalità o una soluzione prossima ad essa; ciò testimonia la qualità dei *tour* ricavati dalla Ricerca Locale supportata dalla *Tabu Search*. A conferma di questa analisi ho riportato i dati relativi al numero di stati eliminati grazie al bounding sulla generazione, descritto nel Paragrafo 2.2.2 (colonna "S BND"). La colonna "Opt TSP" contiene i costi delle soluzioni relative alle istanze risolte da *Concorde*, un solutore per Traveling Salesman Problem simmetrico senza vincoli di retro-carica che si appoggia sulle librerie *Cplex* (versione 6.5). I valori di scostamento percentuale dall'ottimo della versione calcolata dal mio algoritmo (S%) sono molto significativi ed evidenziano chiaramente l'importante effetto dei vincoli aggiuntivi sulla soluzione; nonostante le differenze dovute alla natura dei dati delle istanze, il mantenimento dell'ammissibilità in caso di *Rear Loading* è sempre estremamente dispendioso: la media di S% è infatti pari a 161,97.

In Figura 4.3 è rappresentato un istogramma dell'andamento del *gap* di ottimalità dei

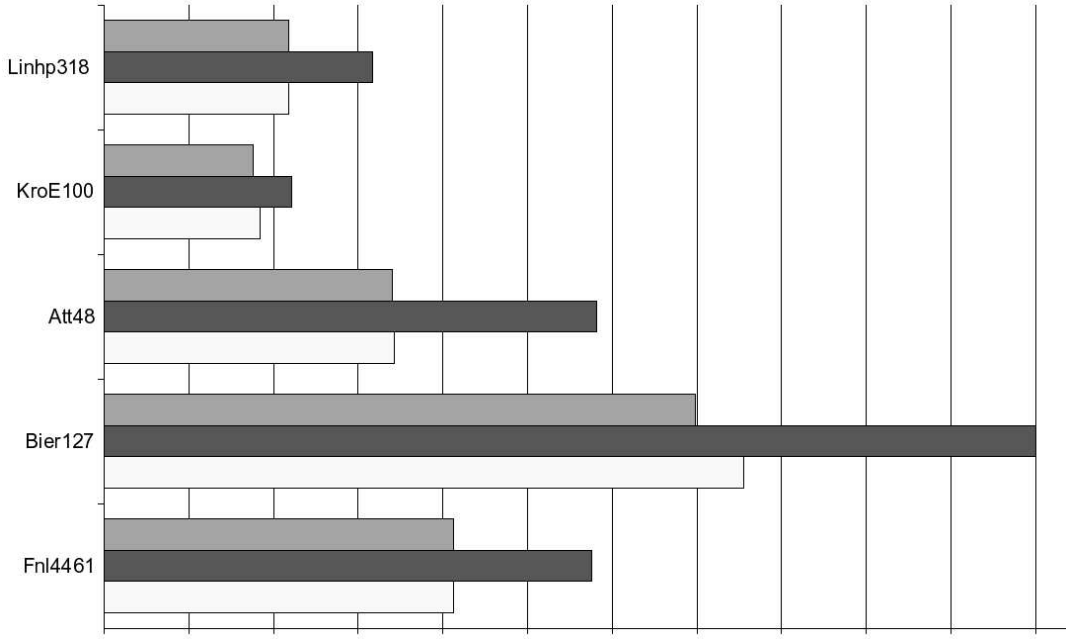

Figura 4.4: istogramma che rappresenta i valori delle soluzioni ricavate per le istanze riportate, tutte limitate a 22 nodi e casualizzate. I dati raccolti sono quelli riportati in Tabella 4.4. Per ognuno dei casi, la prima colonna dall'alto rappresenta il valore del ciclo ottimo, la seconda quello prodotto dall'euristica costruttiva mentre l'ultima il costo del ciclo ricavato dalla ricerca locale.

test effettuati mentre il grafico di Figura 4.4 riguarda il confronto fra i valori delle soluzioni prodotte con i metodi utilizzati per ricavare i dati di Tabella 4.4.

In Tabella 4.5 ho raccolto i dati ricavati dai test condotti su numerose istanze *TSPLib*. Le colonne riportate sono:

- Istanza: nome *TSPLib* dell'istanza considerata;
- Opt TSP: valore del ciclo ottimo relativo al TSP senza vincoli di retro-carica;
- V NN: valore del *tour* iniziale prodotto dall'euristica costruttiva;
- V RL: valore del ciclo prodotto dagli algoritmi di ricerca locale combinati nella catena di elaborazione VND;

- M NN%: valore percentuale del miglioramento tra il percorso iniziale prodotto dall'eurstica costruttiva ed il finale, output del VND;
- DIFF TSP%: differenza percentuale tra il ciclo finale della ricerca locale e l'ottimo globale del TSP senza retro-carica;
- NORM%: stima del possibile *gap* di ottimalità ottenuto con il VND. i valori sono calcolati sottraendo ai risultati della colonna "DIFF TSP%" il valore medio di scostamento $\overline{S}\%$ riportato in Tabella 4.4;
- T VND: tempo in secondi impiegato dal *Variable Neighbourhood Descent*;
- Step VND: passi effettuati dal VND;
- Factor, Limit e Tenure: parametri della *Tabu Search*.

Nonostante non avessi a disposizione i costi dei cicli ottimi per la mia versione vincolata, ho comunque rilevato l'entità dello scostamento dagli ottimi del TSP "semplice" per poi produrre, utilizzando lo scostamento percentuale medio $\overline{S}\%$ ottenuto dai dati della Tabella 4.4, una sorta di "lasca" approssimazione del *gap* ottenuto dalla Ricerca Locale (colonna NORM%). Queste prove mi hanno comunque fornito dati interessanti come il miglioramento percentuale (M NN%) ottenuto dal *Variable Neighbourhood Descent* sulla soluzione iniziale costruita dal *Nearest Neighbour* o i tempi di computazione richiesti per il completamento. Questi ultimi rivelano la grande snellezza e velocità degli algoritmi euristici che ho sviluppato.

Istanza	Opt TSP	V NN	V RL	M NN%	DIFF TSP%	NORM%	T VND (s)	Step VND	Factor	Limit	Tenure
a280	2579	5152	5152	0,00	99,77	–	0,26	1	3	560	28
att48	10628	111897	15673	85,99	47,47	–	0,43	6	2	96	5
att532	27686	671253	259659	61,32	837,87	675,87	615,04	15	6	1064	54
bier127	118282	438231	173640	60,38	46,80	–	9,01	17	2	252	13
eil51	426	1322	747	43,49	75,35	–	0,36	8	2	100	5
eil76	538	1724	1154	33,06	114,50	–	0,62	5	2	152	8
eil101	629	2096	1302	37,88	107,00	–	8,23	15	2	200	10
kroA100	21282	107780	56810	47,29	166,94	4,94	2,72	10	2	200	10
kroB100	22141	103576	51993	49,80	134,83	–	2,53	7	2	200	10
kroC100	20749	95742	55222	42,32	166,14	4,14	2,67	10	2	200	10
kroD100	21294	97784	57171	41,53	168,48	6,48	2,69	10	2	200	10
kroE100	22068	111866	57779	48,35	161,82	–	1,64	6	2	200	10
kroA150	26524	156069	78974	49,40	197,75	35,75	8,85	10	2	300	15
kroB150	26130	159711	73766	53,81	182,30	20,30	9,73	11	2	300	15
kroA200	29368	208318	96012	53,91	226,93	64,93	25,58	12	2	400	20
kroB200	29437	177244	93320	47,35	217,02	55,02	30,11	14	2	400	20
lin105	14379	36342	25798	29,01	79,41	–	1,49	5	2	208	11
lin318	42029	133522	103520	22,47	146,31	–	85,57	10	4	636	32
pr76	108159	232488	149914	35,52	38,61	–	1,36	11	2	152	8
pr107	44303	102934	78362	23,87	76,88	–	2,16	5	2	212	11
pr124	59030	182586	163360	10,53	176,74	14,74	1,52	3	2	248	13
pr136	96772	262163	182015	30,57	88,09	–	2,01	3	2	272	14
pr144	58537	139699	127899	8,45	118,49	–	2,29	3	2	288	15
pr152	73682	201665	155363	22,96	110,86	–	3,66	4	2	304	16
pr226	80369	201099	168892	16,02	110,15	–	9,02	3	3	452	23
pr264	49135	150878	102499	32,06	108,61	–	24,24	5	3	528	27
pr299	48191	169179	125034	26,09	159,46	–	28,92	4	3	596	30
pr439	107217	437460	306327	29,98	185,71	23,71	160,61	7	5	876	44
rat99	1211	2130	1786	16,15	47,48	–	0,97	4	2	196	10
rat195	2323	3828	3683	3,79	58,54	–	5,84	3	2	388	20
st70	675	2193	786	64,16	16,44	–	0,80	8	2	140	7
ts225	126643	262545	261567	0,37	106,54	–	6,19	2	3	448	23

Tabella 4.5: risultati ottenuti dalla Ricerca Locale su diverse istanze TSPLib. I valori della colonna NORM sono stati calcolati diminuendo della quantità $\overline{S}\%$ (dalla Tabella 4.4) i valori della colonna DIFF TSP.

Capitolo 5

Conclusioni e sviluppi futuri

Nello sviluppo del lavoro di tesi ho dovuto affrontare la grande mancanza di materiale in letteratura: il "TSP with Rear Loading" rimane ancora una variante molto poco studiata nonostante le possibili applicazioni reali già menzionate nel Capitolo 1.

L'algoritmo esatto basato sulla *Programmazione Dinamica* si è rivelato poco performante su grafi di cardinalità superiore ai 20-22 nodi. Con istanze di dimensione inferiore, comunque adeguate all'ambito dei problemi logistici di instradamento, il tempo richiesto per il completamento rimane al di sotto dei 2-3 secondi dimostrando una buon agilità.

Inoltre, con l'introduzione della combinazione tra *Stack Bound* e *Lookup Bound*, sono stato in grado di eliminare dal 60% all'80% di nuovi stati alleggerendo notevolmente il processo di esplorazione della matrice.

L'handicap principale rimane l'esplosione combinatoria del numero di stati generati durante la risoluzione. In futuro sarebbe possibile un ulteriore miglioramento del lower bound associato ad ogni *path* prevedendo l'eventualità di considerare parallelamente inserimenti multipli sul percorso fittizio; sarebbe utile anche lo studio di un radicale cambio di politica di esplorazione della matrice da *breadth first* a *depth first*. Questo consentirebbe di utilizzare un ulteriore *bound* basato su di una soluzione completa ed ammissibile, ottenuta in breve tempo e senza l'ausilio di algoritmi euristici "esterni".

Gli algoritmi di ricerca locale si sono rivelati un'ottima scelta dimostrando doti di velocità e snellezza producendo in breve tempo soluzioni di grande qualità: sulle stesse istanze utilizzate per valutare l'algoritmo esatto, il gap rimane al di sotto del mezzo punto percentuale nel 54% dei casi mentre l'ottimalità è raggiunta per il 20%; il tempo impiegato supera il millisecondo nella sola metà delle prove.

Come si nota dalla Tabella 4.5 la ricerca Tabu arriva ad ottenere una soluzione anche per grafi di oltre 500 nodi.

L'utilizzo di una *meta-euristica* come la *Tabu Search* è stato fondamentale permettendomi di non sprecare tempo di calcolo per far fronte alle scarse soluzioni iniziali prodotte dall'euristica costruttiva. Durante lo svolgimento del lavoro, essa ha dimostrato squisite caratteristiche di flessibilità e potenza, contenendo nel contempo il numero dei parametri indispensabili da dimensionare a soli due: il limite di terminazione della ricerca ed il valore di *Tabu Tenure*. Questo mi ha permesso di ottenere risultati significativi senza arenarmi in una estenuante fase di *tuning*.

Per il futuro vedo molto utile un ulteriore studio riguardante le politiche di *Reactive Tabu Search* per l'utilizzo di metodi raffinati e maggiormente adattativi. Sarebbe inoltre interessante valutare il comportamento di politiche di esplorazione dell'intorno come di euristiche costruttive differenti e quanto queste ultime influenzino direttamente le soluzioni prodotte dalla ricerca Tabu.

Per quanto riguarda il Traveling Salesman Problem con vincoli di *Rear Loading*, lo studio di altre varianti, come quella con capacità, ad esempio, rimane tuttora un territorio inesplorato.

Bibliografia

[1] Gregory Gutin and Abraham P. Punnen, *The Traveling Salesman Problem and Its Variations*, Kluwer Academic Publishers, 2002.

[2] L. Cassani, *Algoritmi Euristici per il "TSP with Rear Loading"*, Tesi di Laurea, Dip. Tecnologie dell'Informazione, Università degli Studi di Milano, 2004.

[3] Malek, M., *Search Methods for Traveling Salesman Problems*, Departement of Electrical Engineering, The University of Texas, Austin, 1988.

[4] Shigeru Tsubakitani and James R. Evans, *Optimizing Tabu List Size for the Traveling Salesman Problem*, Computers Ops Res. Vol. 25, No. 2, pp. 91-97, 1998.

[5] Glover, F., *Tabu Search - Part I*, ORSA Journal of Computing, vol. 1, N. 3, pag. 190-206 1989.

[6] Battiti R., Tecchiolli G., *The Reactive Tabu Search*, ORSA Journal of Computing, vol. 6, N. 2, pag. 126-140, 1994.

Appendice A

Generazione di istanze casuali

Per ovviare al problema della scarsa casualità della distribuzione spaziale dei nodi di *pickup* e *delivery* (come già sottolineato nel Paragrafo 4), ho scritto un piccolo tool che mi ha permesso di modificare le istanze in formato *TSPLib* (Figura 4.1). Il brano di codice che esegue la randomizzazione vera e propria è riportato in Algoritmo 8; come si nota i dati dei nodi non vengono modificati, per produrre una nuova istanza modificata copio nel file destinazione delle righe casuali prese dall'originale. Il prodotto è un grafo formato da un sottoinsieme casuale dei punti della mappa *TSPLib*.

In Algoritmo 7 ho riportato lo pseudo-codice relativo al ciclo di generazione. La funzione $\texttt{r}(0 \div N_{in})$ ritorna un naturale casuale compreso nell'intervallo $0 \div N_{in}$ dove il secondo estremo è la cardinalità del grafo originale I; O è il grafo d'uscita mentre N_{out} la cardinalità che quest'ultimo deve presentare al termine della generazione.

Volendo fare un parallelo tra entrambe le rappresentazioni, I è il set di punti descritti nel file di ingresso mentre O il grafo risultante nel file di output; N_{out} è la dimensione desiderata del grafo d'uscita, ovvero il numero di nodi, scelti casualmente, che vengono copiati da input ad output.

Algoritmo 7 Algoritmo di generazione di istanze casuali.

BEGIN

$\forall n \in N_{out}$

| finchè $\neg i \in O$

| | $i = \mathtt{r}(0 \div N_{in})$

| $O \equiv O \cup I_i$

END

Algoritmo 8 Brano significativo relativo alla generazione delle istanze. Le funzioni e macrodefinizioni utilizzate sono tutte conformi allo standard ANSI.

```
120
121 #define rew() fseek(fInputFile, pos2, SEEK_SET)
122
123   /* Copia di righe casuali nel file di output */
124   rew();
125   fscanf(fInputFile, %*s\n);
126   pos2 = ftell(fInputFile);
127
128   srand((unsigned)time(NULL));
129
130   if(RAND_MAX < dimension)
131     printf(Attenzione: la dimensione dell'istanza eccede RAND_MAX!\n);
132
133
134
135   for(count=1, flags[0] = 1; count<=outdim; count++)
136   {
137     cur = rand() % (dimension+1);
138
139     while(flags[(cur = cur % (dimension+1))]) cur++;
140     flags[cur] = 1;
141
142     fscanf(fInputFile, %ld , &id);
143
144     while(id != cur) fscanf(fInputFile, %*d %*d\n %ld , &id);
145
146     fscanf(fInputFile, %d %d\n, &x, &y);
147
148     fprintf(fOutputFile, %ld %d %d\n, count, x, y);
149
150     rew();
151   }
152
153 #undef rew
154
```

Scritto con jEdit 4.2, compilato con pdfeTEX 3.14 e stampato con GNU Ghostscript 7.07.

29 gennaio 2006