UNIVERSITÀ DEGLI STUDI DI MILANO
Facoltà di Scienze Matematiche, Fisiche e Naturali
CORSO DI LAUREA IN INFORMATICA

Algoritmi euristici per il “TSP with rear-loading”

RELATORE
Prof. Giovanni Righini

TESI DI LAUREA DI
Lucio Cassani
Matr. n° 607080

Anno Accademico 2003/2004

Prefazione

La tesi proposta tratta gli algoritmi euristici per risolvere il “TSP with rear-loading”, una variante non ancora studiata del noto problema del commesso viaggiatore, in cui si richiede di calcolare il ciclo hamiltoniano di minimo costo in un dato grafo.

Inizialmente sono illustrati vari algoritmi per la costruzione del tour iniziale. In seguito si passa agli algoritmi di ricerca locale, applicati seguendo le strategie “complex neighbourhood descent” e “variable neighbourhood descent”. Infine vengono trattati algoritmi esatti, sviluppati seguendo il paradigma branch-and-bound, per poter valutare la bontà dei valori trovati con i precedenti.

L’ultima parte della tesi presenta i risultati sperimentali, documentati da tabelle, in cui vengono confrontati i vari algoritmi. I risultati mostrano che i valori ottenuti dalla ricerca locale differiscono da quelli trovati con gli algoritmi esatti per pochi punti percentuali, mantenendo però tempi di esecuzione decisamente inferiori.

Indice

1. Introduzione

1.1 Il “TSP with rear-loading”.. 7
1.2 Obiettivi della tesi.. 7
1.3 Organizzazione dei contenuti………………………………………………………... 8

2. Euristiche di costruzione

2.1 Algoritmo Nearest Neighbour……………………………………………………. 9
2.2 Algoritmo Reversed Nearest Neighbour.. 10
2.3 Algoritmo Randomized Nearest Neighbour.. 10
2.4 Algoritmo Reversed Randomized Nearest Neighbour... 11
2.5 Algoritmo Cheapest Insertion.. 11
2.6 Algoritmo Largest Insertion………………………………………………………. 12
2.7 Algoritmo Farthest Insertion………………………………………………………. 12
2.8 Algoritmo Nearest Insertion………………………………………………………. 13

3. Algoritmi di ricerca locale

3.1 Scambio di coppie…………………………………………………………………. 14
3.2 Scambio di blocchi………………………………………………………………... 15
3.3 Rimozione e reinserimento di una coppia………………………………………… 16
3.4 Rimozione e reinserimento di un blocco…………………………………………….. 17
3.5 Funzionamento dell’euristica……………………………………………………… 18

4. Algoritmi esatti

4.1 Algoritmo con branching monodirezionale e lower bound con Prim……………... 20
4.2 Algoritmo con branching monodirezionale e lower bound con matching bipartito. 21
4.3 Algoritmo con branching bidirezionale e lower bound con Prim………………… 21
4.4 Algoritmo con branching bidirezionale e lower bound con matching bipartito…... 25

5. Risultati sperimentali

5.1 Specifiche tecniche della macchina……………………………………………….. 27
5.2 Formato dei file……………………………………………………………………. 27
5.3 Risultati sperimentali……………………………………………………………… 29

6. Conclusioni

Conclusioni………………………………………………………………………… 33

7. Bibliografia

Bibliografia…………………………………...……………………………………….... 34

Appendice A: codice in linguaggio C

A.1 Algoritmi di costruzione del tour iniziale……………………………………… 35
A.2 Algoritmi di ricerca locale…………………………………………………….... 66
A.3 B&B: branching monodirezionale e lower bound con Prim……………………… 88
A.4 B&B: branching monodirezionale e lower bound con matching bipartito……….. 96
A.5 B&B: branching bidirezionale e lower bound con Prim………………………….. 116
A.6 B&B: branching bidirezionale e lower bound con matching bipartito…………… 128

Capitolo 1

Introduzione

1.1 Il problema del TSP with rear-loading

Il "Traveling Salesman Problem" (TSP) è un noto problema di ottimizzazione NP-hard: dato un grafo G=(V,E) in cui l'insieme dei vertici V={1,…,N} rappresenta le città e una matrice NxN, associata all'insieme degli spigoli $E=\{(i,j): i \neq j; i,j \in V\}$, nella quale l'elemento [i,j] dà la distanza dalla città i alla città j, si richiede di trovare il ciclo hamiltoniano di minimo costo. Per le sue numerose applicazioni il TSP è stato molto studiato e sono state considerate parecchie varianti (grafo asimmetrico, finestre temporali, vincoli di capacità e altro ancora). Si veda a tal proposito Punnen et al. (2002)

Il "TSP with pickup and delivery" (TSPPD) è una variante con vincoli di precedenza e può essere definita nel modo seguente: sia G=(V,E) un grafo. L'insieme V dei vertici è partizionato in tre sottoinsiemi: {1}, P e D, dove il primo è il vertice di partenza (detto anche "città deposito"), P è l'insieme dei clienti *pickup*, cioè che richiedono un'operazione di carico e D è l'insieme dei clienti *delivery*, in cui va effettuato uno scarico. All'insieme E è associata una matrice dei costi che rappresenta la distanza tra le città. Ad ogni cliente pickup è associato un unico delivery e viceversa, quindi la cardinalità di P è la stessa di D. Scopo del problema è trovare il ciclo hamiltoniano di minimo costo tale per cui ogni cliente pickup sia visitato prima del corrispondente delivery. Diversi studi sono stati fatti a riguardo, tra i quali Renaud et al. (2000), Renaud et al. (2002), Gendreau et al. (1998).

Il "TSP with rear-loading" (TSPRL) è una variante del TSPPD in cui si aggiunge un vincolo: le operazioni di carico e scarico seguono un criterio *last in – first out* (LIFO): è sempre possibile visitare un utente pickup, oppure il delivery relativo all'ultimo carico effettuato e non ancora scaricato.

1.2 Obiettivi della tesi

Attualmente in letteratura non esistono studi sul TSPRL. Gli obiettivi della tesi sono

- Studio di algoritmi euristici

Questo obiettivo è stato perseguito con lo studio di euristiche di costruzione del tour iniziale seguendo varie strategie: nearest neighbour e sue varianti (casualizzato, al contrario, casualizzato al contrario), cheapest insertion, largest insertion, nearest insertion, farthest insertion. Ogni algoritmo è stato riformulato seguendo il modello del TSPRL. Ad ogni tour iniziale sono poi applicati gli algoritmi di ricerca locale che consistono in scambi di coppie di utenti, scambi di blocchi, rimozione e reinserimento di coppie, rimozione e reinserimento di blocchi. La ricerca dell'ottimo avviene applicando questi algoritmi a strategie "complex neighbourhood descent" e "variable neighbourhood descent".

- Studio di algoritmi esatti con cui confrontare i risultati trovati con la ricerca locale

Questo obiettivo è stato perseguito studiando alcuni algoritmi branch-and-bound. La politica di esplorazione dell'albero è di tipo best first. Come upper bound è utilizzato il valore ottimo calcolato dall'euristica di ricerca locale. Il lower bound è determinato dal costo dell'albero di copertura minimo, oppure dal costo dei sottocicli indotti dal matching bipartito di minimo costo, calcolato con l'algoritmo ungherese. Con quest'ultimo metodo è possibile considerare il grafo come orientato e quindi si possono proibire alcuni archi, basandosi sulle proprietà del modello e sulla soluzione parziale. Nella fase di branching i nodi figli sono generati partendo dal deposito e scegliendo come prossima tappa tutti gli utenti ammissibili. In alternativa si costruisce la soluzione alternando un utente aggiunto andando in avanti ad uno andando all'indietro, avendo cura di mantenere sempre l'ammissibilità della soluzione. In questo modo è possibile proibire un numero maggiore di archi.

1.3 Organizzazione dei contenuti

Nel capitolo 2 descrivo le euristiche di costruzione del tour iniziale,
Nel capitolo 3 illustro gli algoritmi di ricerca locale, spiegando il funzionamento di ognuno e il modo in cui sono utilizzati per trovare l'ottimo locale.
Il capitolo 4 è dedicato agli algoritmi esatti, in cui mostro come ho seguito il modello branch-and-bound per cercare la soluzione ottima.
Il capitolo 5 contiene i risultati sperimentali, in cui riporto il confronto tra i tempi di esecuzione e i valori trovati dagli algoritmi esatti e da quelli euristici.
Nel capitolo 6 riporto le conclusioni.

Capitolo 2

Euristiche di costruzione

Ho sviluppato alcuni algoritmi per la costruzione di un tour iniziale, su cui poter applicare quelli di ricerca locale. Ognuno di questi algoritmi riceve in input:

- le posizioni delle città nel piano bidimensionale
- gli accoppiamenti tra esse

L'output è un ciclo hamiltoniano che rispetta i vincoli di rear-loading.

2.1 Algoritmo Nearest Neighbour

L'algoritmo inizialmente considera il tour composto dalla sola città deposito. Ad ogni iterazione si aggiunge la città più vicina all'ultima inserita nel cammino tra quelle ammissibili (fig. 2.1).

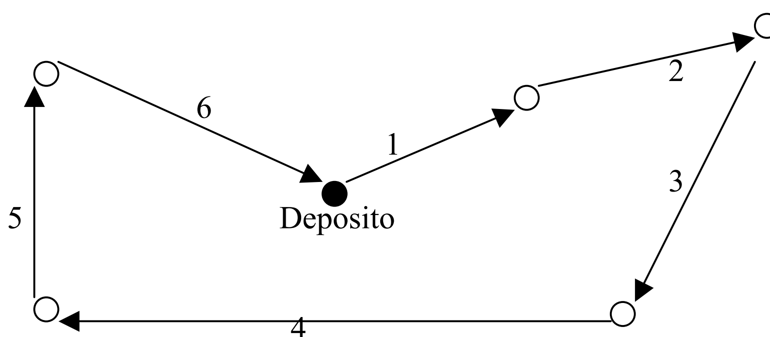


Figura 2.1: algoritmo Nearest Neighbour per il TSP

Per rispettare il vincolo LIFO è necessario controllare il tipo della città da aggiungere: una città pickup può essere sempre aggiunta, mentre una delivery può essere inserita solo se la propria partner è l'ultima pickup visitata della quale non è ancora stata raggiunta la relativa partner (fig. 2.2). L'uso di uno stack permette di gestire con facilità questo vincolo: se si incontra una città di tipo pickup la si inserisce con una procedura *Push*, altrimenti si elimina la città pickup in cima allo stack con una procedura *Pop*.

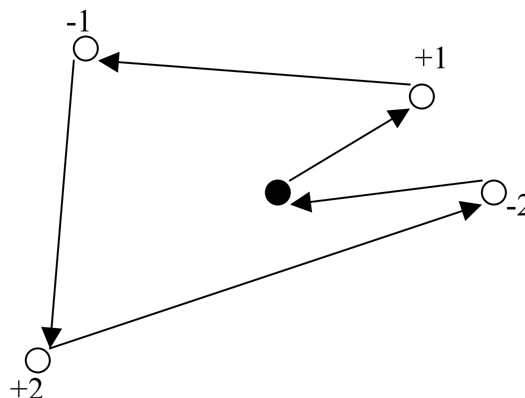


Figura 2.2: algoritmo Nearest Neighbour per il TSPRL

2.2 Algoritmo Reversed Nearest Neighbour

E' una variante dell'algoritmo precedente. Il tour viene costruito muovendosi "a marcia indietro", vale a dire al contrario del Nearest Neighbour. Si parte quindi dal tour costituito dalla sola città deposito e ad ogni iterazione si aggiunge la città più vicina: analogamente al caso precedente, se è di tipo delivery non ci sono vincoli, mentre se è di tipo pickup bisogna controllare che la città partner sia in cima allo stack.

L'esempio in figura 2.3 mostra che il Nearest Neighbour (fig. 2.3a) e il Reversed Nearest Neighbour (fig. 2.3b) non danno necessariamente gli stessi risultati.

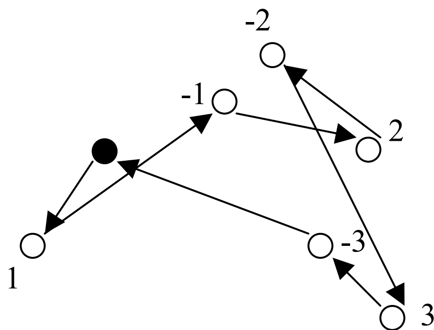


Figura 2.3a: Nearest Neighbour

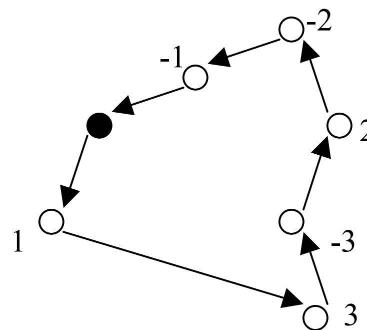


Figura 2.3b: Reversed Nearest Neighbour

2.3 Algoritmo Randomized Nearest Neighbour

L'algoritmo è un'altra variante del Nearest Neighbour. Inizialmente si fissa un parametro λ.
Per ogni città si ordinano le rimanenti per distanza crescente. Ad ogni tappa si determina la prossima città scegliendone casualmente una tra le prime λ: se è già stata usata oppure non è ammissibile si passa alla successiva, continuando finché non si effettua un inserimento.
L'algoritmo è di tipo multistart: eseguendolo più volte si possono ottenere soluzioni iniziali differenti, permettendo all'euristica di ricerca locale di posizionarsi su diversi ottimi locali (fig. 2.4).

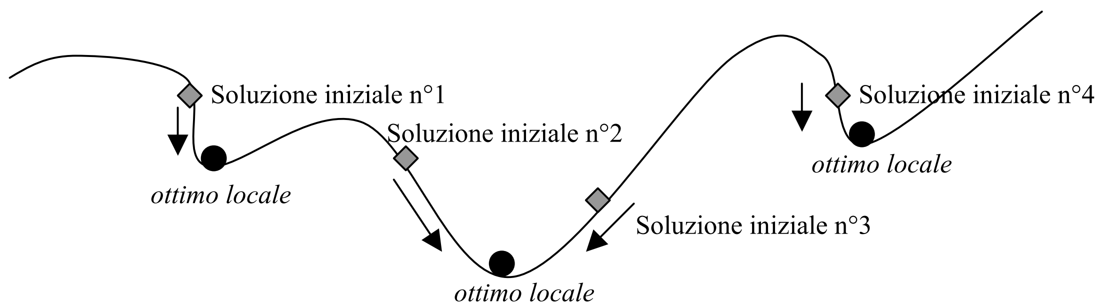


Figura 2.4: ripetendo l'inizializzazione l'algoritmo di ricerca locale può posizionarsi su differenti ottimi locali.

2.4 Algoritmo Reversed Randomized Nearest Neighbour

L'algoritmo è una variante del precedente, in cui il cammino è costruito muovendosi all'indietro.
I valori generati casualmente sono diversi da quelli del Randomized Nearest Neighbour: in questo modo si riduce la possibilità di ottenere lo stesso tour iniziale.

2.5 Algoritmo Cheapest Insertion

L'algoritmo inizia considerando la città deposito come l'unica appartenente al tour. Come primo passo si sceglie la città pickup e la relativa delivery tali per cui il cammino "deposito>pickup>delivery>deposito" è il minimo (fig. 2.5).

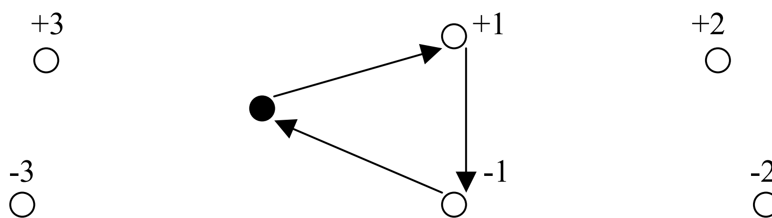


Figura 2.5: costruzione del tour minimo nell'inizializzazione della Cheapest Insertion

Ad ogni iterazione si inserisce una coppia di città alla volta, scegliendo quella che provoca il minimo aumento della lunghezza del tour parziale.
La posizione in cui inserire le città è determinata considerando tutte le soluzioni ammissibili (fig. 2.6).

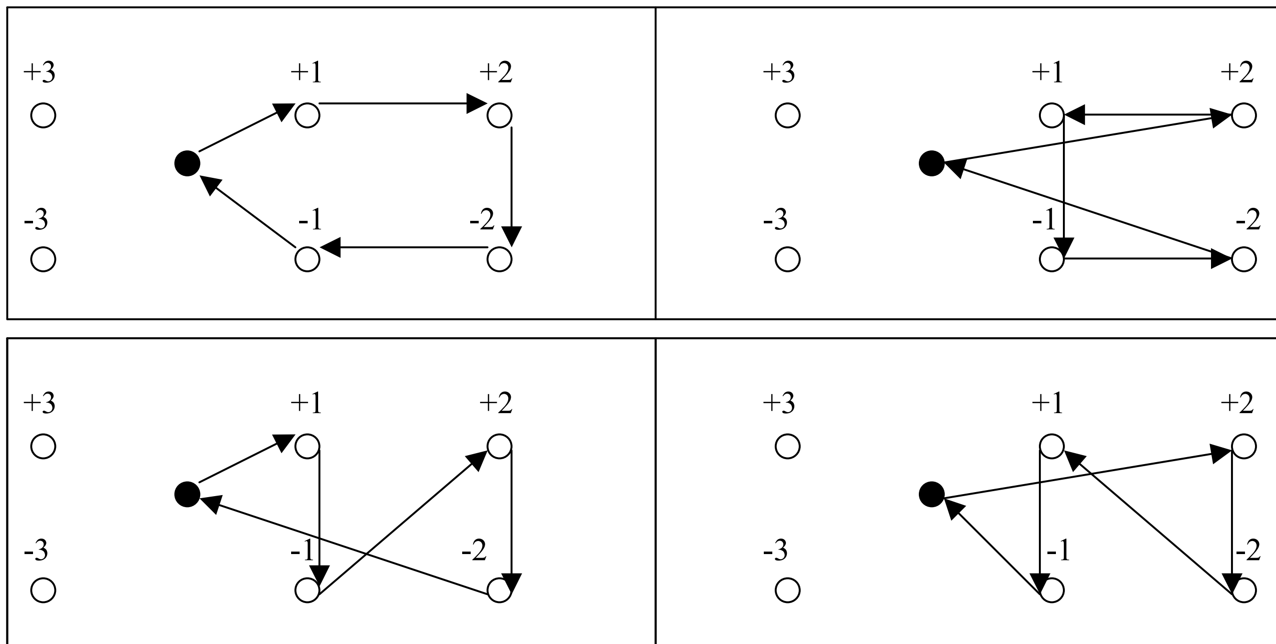


Figura 2.6: inserimento nella Cheapest Insertion. Dato il tour iniziale in figura 2.5, la coppia (2,-2) può essere inserita in quattro modi possibili

2.6 Algoritmo Largest Insertion

L'algoritmo è una variante della Cheapest Insertion. L’inizializzazione è eseguita nello stesso modo. Nella procedura di inserimento per ogni coppia di città si memorizza il migliore risultato ottenibile e si sceglie di inserire il peggiore.

2.7 Algoritmo Farthest Insertion

L'algoritmo inizia considerando la città deposito come l'unica appartenente al tour. A differenza della Cheapest Insertion questa volta si sceglie la coppia che massimizza il cammino deposito>pickup>delivery>deposito (fig 2.7).

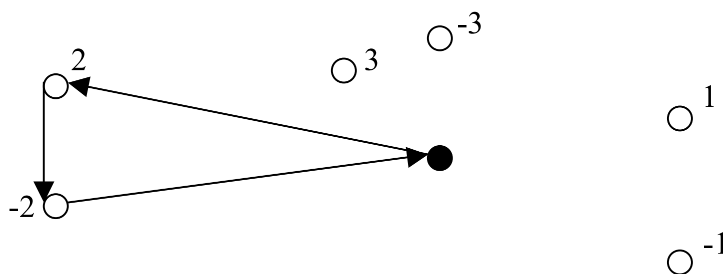


Figura 2.7: inizializzazione nella Farthest Insertion

Ad ogni iterazione, per ciascuna coppia non ancora inserita, si calcola la distanza minima tra ognuna delle città che la compongono e una qualsiasi città appartenente al tour (non necessariamente la stessa per entrambe le componenti della coppia).
Si sceglie la coppia che massimizza la distanza minima. L’inserimento avviene come nella Cheapest Insertion: si inseriscono le città nella posizione ammissibile che dà il minore aumento nella lunghezza del tour (fig.2.8).

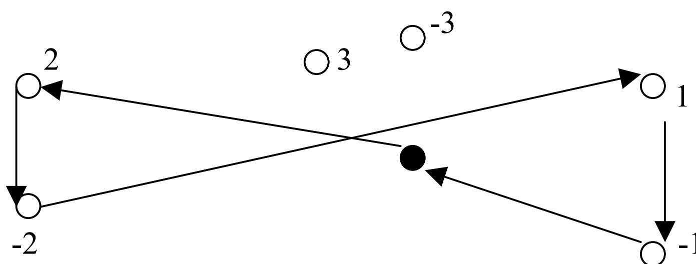


Figura 2.8: inserimento nella Farthest Insertion: 2 e –2 sono le città più lontane da una città appartenente al tour (in questo caso –1) e vengono inserite nella posizione che provoca il minore aumento nella lunghezza del tour

2.8 Algoritmo Nearest Insertion

L’algoritmo è una variante della Farthest Insertion. Il tour iniziale è costruito scegliendo la coppia che minimizza il cammino deposito>pickup>delivery>deposito.
Nella procedura di inserimento si sceglie la coppia in cui la distanza tra le componenti e una città già inserita è la minore.

Capitolo 3

Algoritmi di ricerca locale

Gli algoritmi di ricerca locale ricevono in input:

- le posizioni delle città
- gli accoppiamenti tra di esse
- i tour iniziali generati dalle euristiche di costruzione

In output danno il tour ottenuto posizionandosi su un ottimo locale (Aarts et al., 1999)

3.1 Scambio di coppie

L'algoritmo calcola il costo delle soluzioni ottenute scambiando due coppie di città, per tutte le possibili soluzioni, memorizzando il costo minore. Se il minor costo trovato è migliore del costo iniziale l'algoritmo effettua lo scambio (fig. 3.1). Sono sempre possibili N(N-1) scambi, con N = numero delle coppie.

```
Procedura scambioDiCoppia

   Miglior Valore = Lunghezza del tour

   Per ogni coppia (i,j) {
      Per ogni coppia (k,l) {
         Valuta il costo del tour scambiando i con k e j con l
         Se (nuovo costo < Miglior Valore) {
            Memorizza i*,j*,k*,l*
            Miglior Valore = nuovo costo
         }

      }
   }

   Scambia i* con k* e j* con l*
   Costo del tour = Miglior Valore

End
```

Figura 3.1: procedura di scambio di coppia

Non ci sono problemi per l'ammissibilità, dato che gli scambi coinvolgono entrambi i membri di una coppia (fig. 3.2).

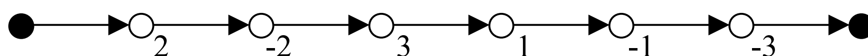
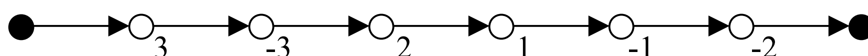


Figura 3.2: l'algoritmo trova che lo scambio tra la coppia 2 e 3 è il migliore



Lo scambio è stato effettuato: la soluzione ottenuta è ancora ammissibile

3.2 Scambio di blocchi

Un blocco è una parte del cammino che inizia con una città pickup e termina con la relativa delivery. L'algoritmo, simile al precedente, prova a scambiare due blocchi non compresi tra loro, provando tutte le possibili combinazioni per ogni blocco. Se la soluzione calcolata è migliore della soluzione iniziale allora effettua lo scambio (fig. 3.3). Nel caso limite, in cui ogni blocco non ne contiene un altro, sono possibili N(N-1)/2 scambi.

```
Procedura scambioDiBlocchi

   Miglior Valore = Lunghezza del tour

   Per ogni blocco (i…j) {
      Per ogni blocco (k…l) {
         Valuta il costo del tour scambiando (i…j) con (k…l)
         Se (nuovo costo < Miglior Valore) {
            Memorizza i*,j*,k*,l*
            Miglior Valore = nuovo costo
         }

      }
   }

   Scambia i*…j* con k*…l*
   Costo del tour = Miglior Valore

End
```

Figura 3.3: procedura di scambio di blocchi

La soluzione trovata è ammissibile perché ogni blocco è una sottosequenza LIFO autonoma: i valori presenti nello stack prima di entrare in un blocco sono gli stessi che si hanno uscendo, perciò spostare un blocco non viola l'ammissibilità (fig. 3.4).

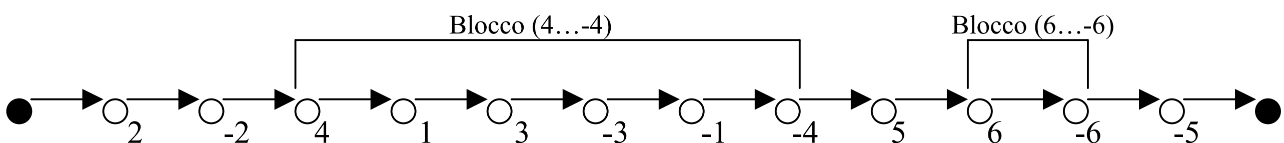
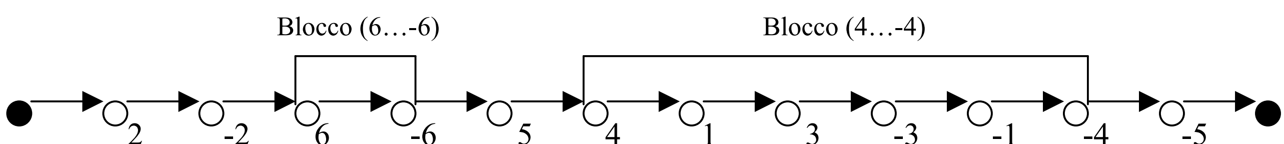


Figura 3.4: l'algoritmo trova che lo scambio più conveniente è quello tra il blocco 4 e 6



Lo scambio è stato effettuato. La soluzione è ancora ammissibile

Va osservato che due o più blocchi affiancati potrebbero essere considerati come un unico blocco, ma per ridurre la complessità l'algoritmo considera solo blocchi singoli.

3.3 Rimozione e reinserimento di una coppia

L'algoritmo cerca la coppia che se rimossa e reinserita nella miglior posizione possibile dà la maggiore riduzione della lunghezza del tour (fig. 3.5). Sono sempre possibili N rimozioni.

```
Procedura rimuoviCoppia

    Miglior Valore = Lunghezza del tour

    Per ogni coppia (i,j) {
        Valuta il costo del tour rimuovendo (i,j) e reinserendola
        nella posizione migliore

        Se (nuovo costo < Miglior Valore) {
            Memorizza i*,j*
            Miglior Valore = nuovo costo
            Memorizza la posizione in cui inserire la coppia
        }
    }

    Rimuovi (i*,j*)

    Reinserisci (i*,j*) nella posizione migliore

End
```

Figura 3.5: procedura di rimozione di una coppia

La figura 3.6 mostra un esempio di rimozione e reinserimento di una coppia.

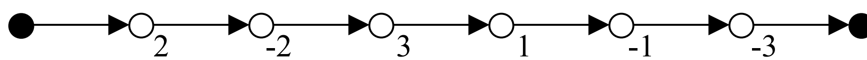
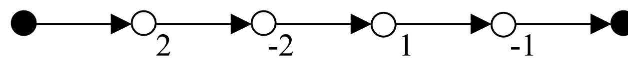
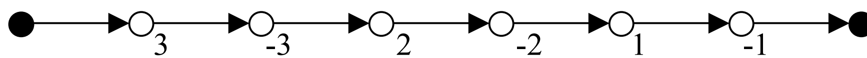


Figura 3.6: l'algoritmo determina che la coppia da rimuovere è (3,-3)



La coppia è rimossa



La coppia è reinserita in posizione diversa

3.4 Rimozione e reinserimento di un blocco

Simile al precedente, l'algoritmo cerca il blocco che se rimosso e reinserito nella miglior posizione possibile dà la maggiore riduzione della lunghezza del tour (fig. 3.7). In questo caso sono possibili N rimozioni nel caso migliore, vale a dire quello in cui ogni blocco non ne contiene altri.

```
Procedura rimuoviBlocchi

    Miglior Valore = Lunghezza del tour

    Per ogni blocco (i…j) {
        Valuta il costo del tour rimuovendo (i…j) e reinserendolo nella
        posizione migliore

        Se (nuovo costo < Miglior Valore) {
            Memorizza i*,j*
            Miglior Valore = nuovo costo
            Memorizza la posizione in cui inserire il blocco
        }
     }

    Rimuovi (i*…j*)

    Reinserisci (i*…j*) nella posizione migliore


End
```

Figura 3.7: procedura di rimozione di un blocco

Dato che ogni blocco è una sottosequenza LIFO e che entrando e uscendo da esso non ci sono variazioni al carico complessivo è possibile effettuare l'inserimento in ogni punto del cammino mantenendo l'ammissibilità (fig. 3.8).

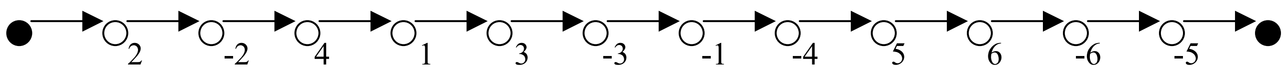
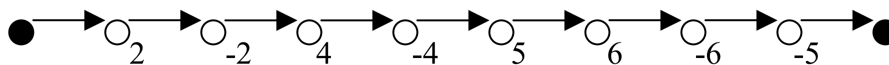
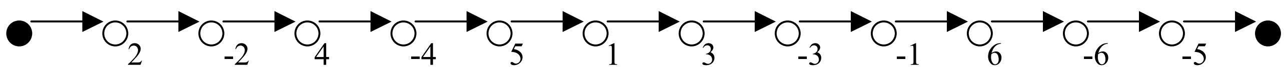


Figura 3.8: l'algoritmo trova che il blocco da rimuovere è (1…-1)



Il blocco (1…-1) è rimosso



Il blocco è reinserito dopo la città 5

3.5 Funzionamento dell'euristica

Per ognuno dei tour iniziali generati con le euristiche di costruzione, l'algoritmo applica le procedure descritte sopra. L'ottimo viene cercato con strategia "variable neighbourhood descent" e "complex neighbouhood descent".
Nella prima le euristiche vengono ordinate secondo un criterio stabilito e si ottimizza. Quando non si può più migliorare si passa alla seconda. Se si migliora si ricomincia dalla prima, altrimenti si passa alla successiva e così via (fig. 3.9).

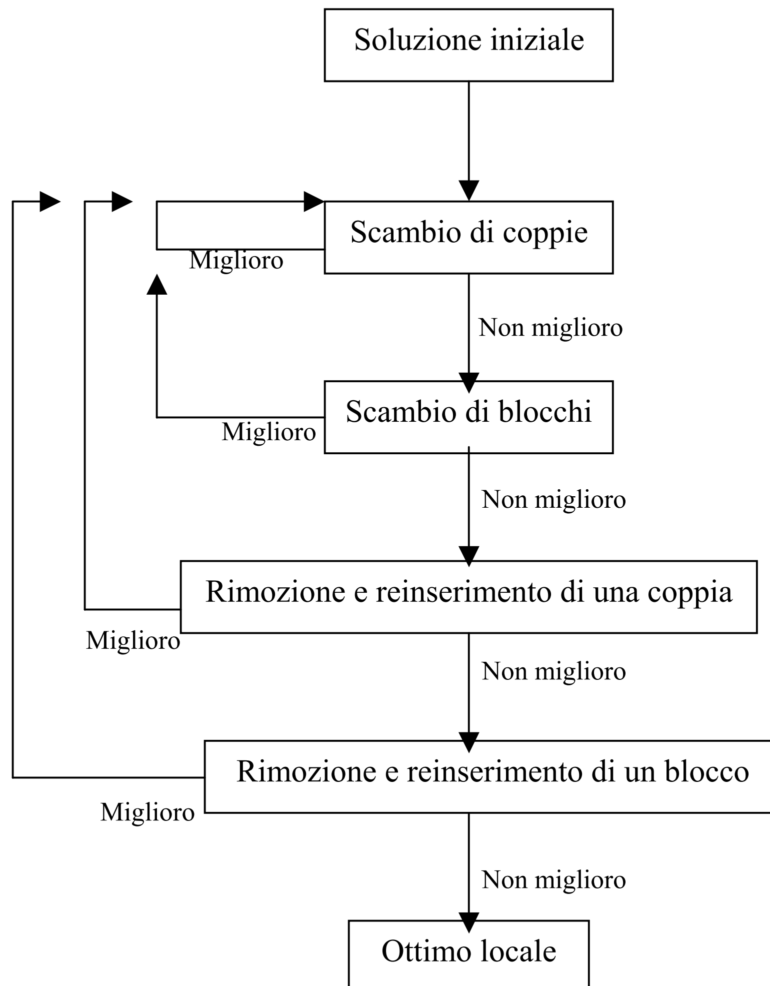


Figura 3.9: funzionamento dell'algoritmo Variable Neighbourhood Descent

Con la seconda strategia l'ottimo viene cercato esplorando tutti gli intorni ad ogni iterazione, scegliendo ogni volta la soluzione migliore (fig. 3.10).

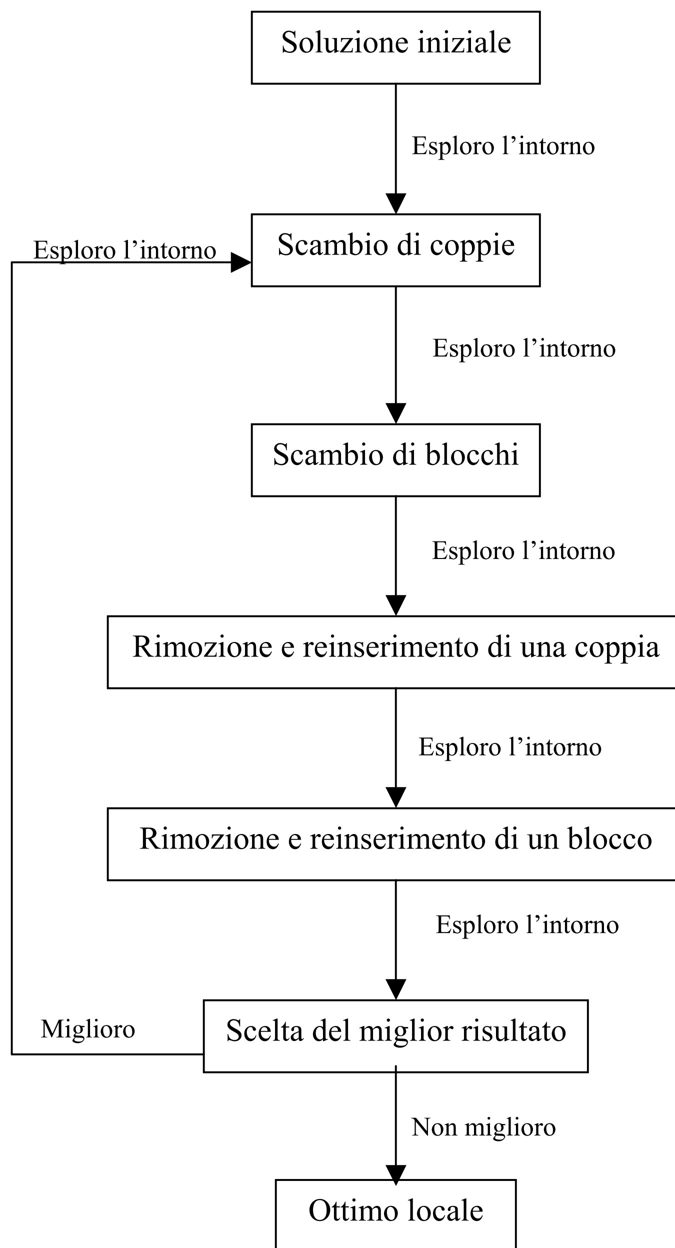


Figura 3.10: strategia "complex neighbourhood descent". L'algoritmo prova ogni euristica, scegliendo la soluzione migliore. Se migliora prosegue, altrimenti si arresta su un ottimo locale.

Capitolo 4

Algoritmi esatti

Per verificare l'efficacia delle euristiche ho sviluppato alcuni algoritmi esatti, basandomi sulla tecnica branch-and-bound (Ibaraki, 1988).

4.1 Algoritmo con branching monodirezionale e lower bound con Prim

Il primo algoritmo che ho realizzato è il più semplice e probabilmente il più ovvio.
Per la fase di branching l'algoritmo sceglie la prossima tappa in tutti i modi ammissibili possibili: data una soluzione parziale, la prossima tappa può essere una qualsiasi città pickup oppure la delivery relativa alla pickup in cima allo stack (fig. 4.1).

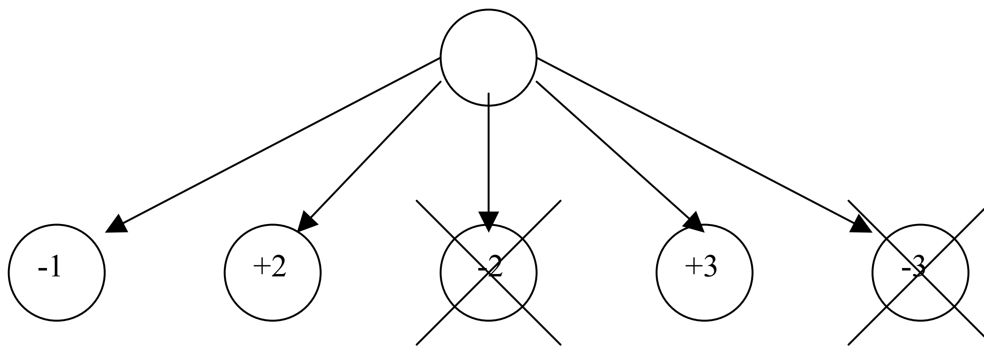


Figura 4.1: se il cammino parziale è "deposito->+1" posso aggiungere tutte le pickup e solo –1 tra le delivery

La politica di esplorazione dell'albero è di tipo best first: il valore del lower bound indica quanto un nodo è "promettente" e l'esplorazione avviene dando la precedenza ai nodi più promettenti, a qualunque livello siano.
L'upper bound iniziale è il valore calcolato dagli algoritmi di ricerca locale, aggiornato ogni volta che si trova una soluzione con un valore migliore.
Il lower bound è determinato dal costo dell'1-albero minimo, calcolato con l'algoritmo di Prim (1957). In questa fase è necessario considerare la soluzione parziale come un unico macronodo.

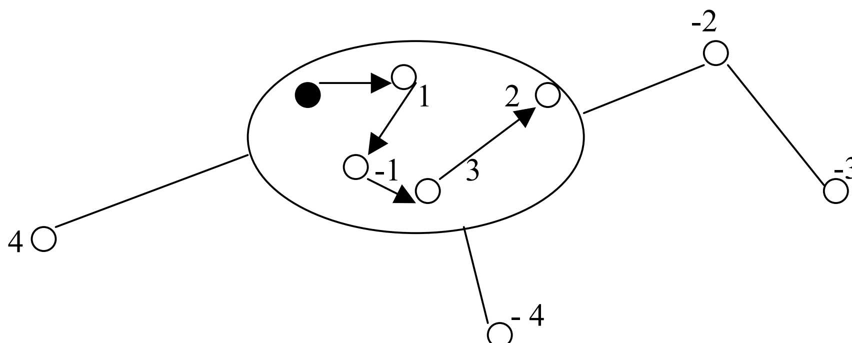


Figura 4.2: con la procedura shrink si comprime il cammino parziale in un unico nodo per il calcolo dell'albero di copertura minimo. Il costo dal nodo compresso J ad un altro nodo n è pari al costo da 2 a n. Il costo da un nodo n al nodo J è pari al costo da n al deposito.

Il costo dell'1-albero viene calcolato con l'algoritmo di Prim considerando la soluzione parziale composta dal cammino *i…j*. Per il calcolo dell'albero di copertura minimo si utilizza il costo degli spigoli incidenti a *j* (cioè l'ultima città inserita nel macronodo) a cui va aggiunto il più piccolo spigolo incidente a *i* (la città deposito).

4.2 Algoritmo con branching monodirezionale e lower bound con matching bipartito a costo minimo

Dopo avere sviluppato il primo algoritmo ho cercato di migliorare il calcolo del lower bound, in modo da poter stimare meglio il valore della soluzione e limitare l'esplosione combinatoria del numero dei nodi.
Ho provato quindi a utilizzare l'algoritmo ungherese (Kuhn, 1955) per il calcolo del matching bipartito a costo minimo (fig. 4.4a), che genera dei sottocicli a costo minimo (fig. 4.4b). Dato che il tour del TSP è un unico ciclo, il suo costo totale non può essere minore del costo di tali sottocicli.

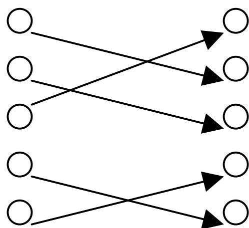


Figura 4.4a: calcolo del matching bipartito a costo minimo

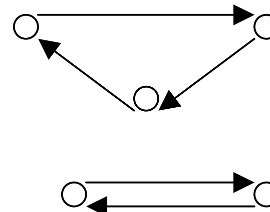


Figura 4.4b: determinazione dei sottocicli generati dal matching

Ho quindi modificato l'algoritmo precedente, cambiando la procedura di calcolo del lower bound con quella appena descritta. Grazie al matching è possibile considerare il grafo come orientato e quindi, basandosi sulla soluzione parziale, è possibile proibire alcuni archi assegnando loro costo infinito in modo da poter raffinare il calcolo del lower bound. La spiegazione è rimandata al paragrafo 4.4, nel quale esamino la versione con branching bidirezionale.

4.3 Algoritmo con branching bidirezionale e lower bound con Prim

In questo algoritmo ho modificato la fase di branching: la soluzione viene costruita alternando città aggiunte normalmente ad altre aggiunte "in retromarcia". In figura 4.5 è riportato un esempio di costruzione della soluzione muovendosi in due direzioni

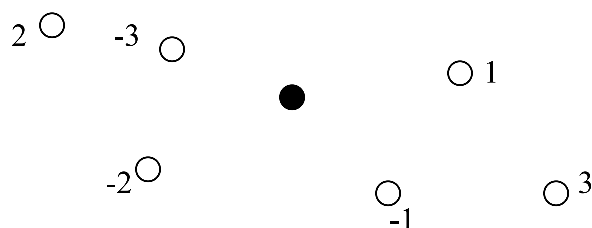
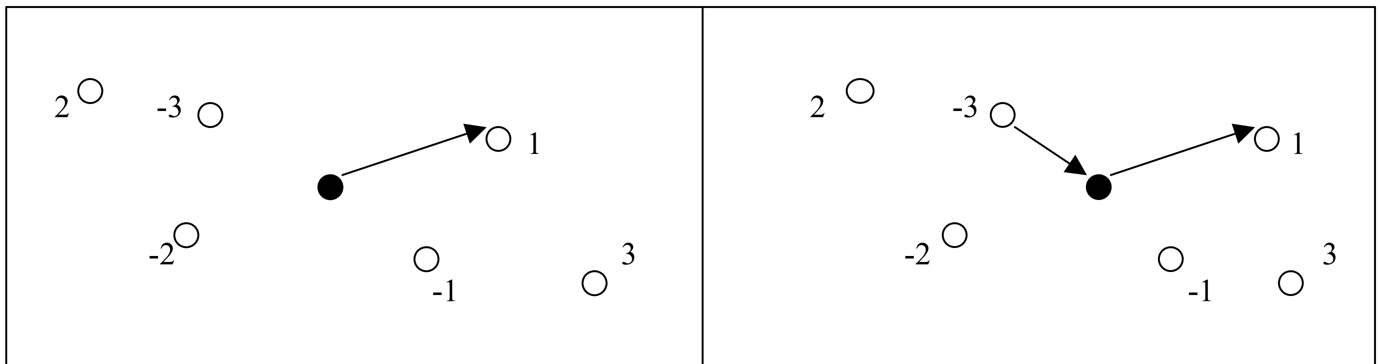


Figura 4.5: esempio di problema



L'algoritmo costruisce la soluzione alternando archi aggiunti in avanti ad altri aggiunti all'indietro

L'idea è di poter sfruttare meglio il calcolo del lower bound dell'algoritmo precedente, proibendo più archi possibile. Lo svantaggio è che diventa più complicato garantire l'ammissibilità delle soluzioni: negli algoritmi precedenti è molto semplice, dato che basta proibire alle delivery "sbagliate" di essere inserite. In questo caso bisogna aggiungere il controllo fatto andando al contrario, aggiungendo uno stack che memorizzi le città che vengono man mano incontrate muovendosi all'indietro e che funzioni all'opposto dell'altro: si possono quindi inserire tutte le delivery, mentre per le pickup è necessario che la delivery sullo stack sia la sua partner (fig 4.6). Questo però non è sufficiente, dato che possono comunque esserci soluzioni non ammissibili.

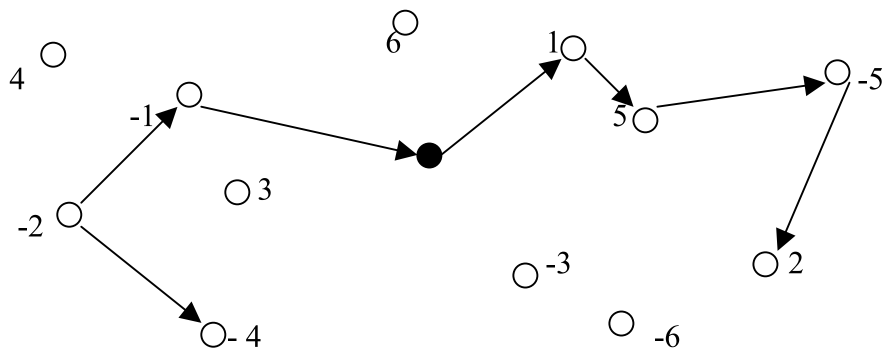


Figura 4.6: in questo esempio di problema lo stack in avanti è composto da 1 e 2.
Lo stack all'indietro è composto da –1, -2 e –4

Studiando il problema si nota una proprietà delle soluzioni ammissibili: ogni sottosequenza, cioè ogni cammino che va da una pickup alla relativa delivery, è a sua volta una sequenza LIFO. Questo implica che, per avere una soluzione ammissibile, le città pickup che si trovano sullo stack della soluzione costruita "in avanti" devono essere le partner delle città delivery che sono sullo stack della soluzione costruita "in retromarcia". Inoltre devono apparire nello stesso ordine sugli stack (fig. 4.7).

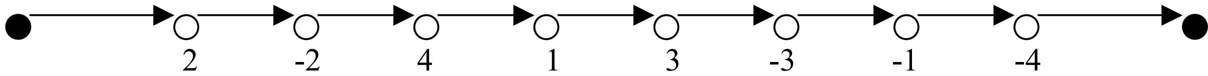


Figura 4.7: una possibile soluzione ammissibile per il problema con quattro coppie di città

Stack delle città da assegnare:	
Sottosequenza 2 *In avanti: vuoto* *All'indietro: vuoto*	*Sottosequenza 4* *In avanti: vuoto* *All'indietro: vuoto*
Sottosequenza 1 *In avanti: 4* *All'indietro: -4*	*Sottosequenza 3* *In avanti: 4,1* *All'indietro: -4, -1*

Quando costruisco la soluzione "in avanti" e inserisco una città pickup devo controllare se la relativa delivery è già inserita (nella costruzione all'indietro devo controllare, ogni volta che inserisco una delivery, se la relativa pickup è già inserita): se è così allora ho individuato una sottosequenza e devo fare la verifica degli stack (fig. 4.8).

Figura 4.8: inserendo la città 3 trovo una sottosequenza e controllo gli stack: in avanti ho 1, all'indietro ho –1. La soluzione parziale è ammissibile e posso inserire 3.

Se gli stack sono diversi allora la soluzione parziale non è ammissibile e la scarto (fig. 4.9).

Figura 4.9: in questo caso la città 3 non può essere inserita: sullo stack in avanti ho 2, all'indietro ho –5. Tutte le soluzioni generate da questa non sono ammissibili, quindi non la considero

Il controllo degli stack è effettuato seguendo lo schema in figura 4.10. L'esempio si riferisce all'inserimento in avanti (per il caso all'indietro il ragionamento è simmetrico).

```
Procedura controllaStack

1° caso: inserimento di una città pickup +i
Se -i appartiene allo stack backward
        Se –i è in fondo allo stack backward e lo stack forward è vuoto {
                Inserisci +i
                Togli –i dal fondo dello stack backward
        }

Se –i non appartiene allo stack backward {
        Inserisci +i
        Aggiungi +i allo stack forward
}

2° caso: inserimento di una città delivery –i
Se +i è in cima allo stack forward {
        Inserisci –i
        Togli +i dallo stack forward
}
```

Figura 4.10: inserimento di una città in avanti nel caso con branching bidirezionale

In figura 4.11 illustro il caso di inserimento di una città pickup in cui la delivery è già inserita

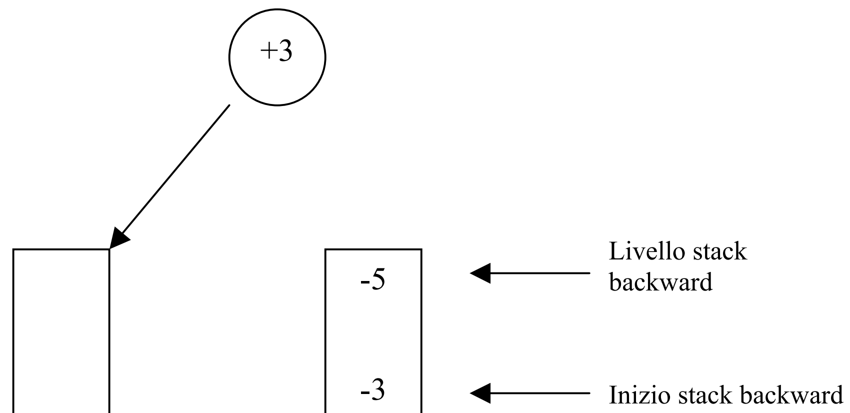


Figura 4.11a: l'algoritmo valuta se è possibile inserire +3. Lo stack forward è vuoto e il primo elemento dello stack backward è –3, quindi l'inserimento è possibile

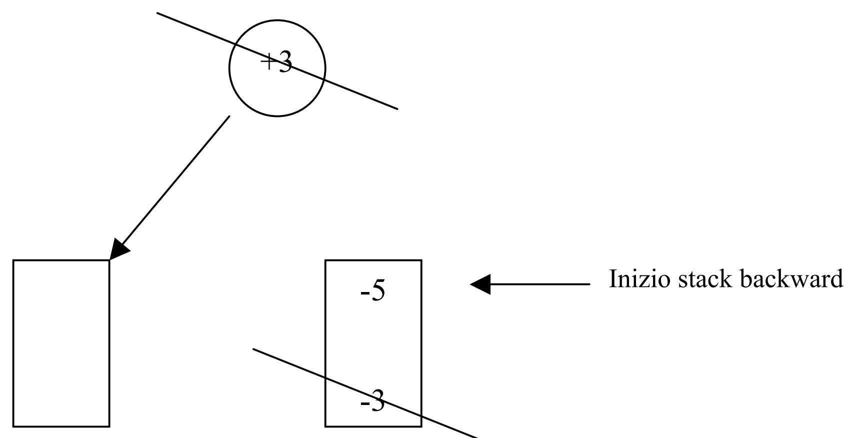


Figura 4.11b: l'algoritmo ha inserito +3. In fondo allo stack backward ora c'è –5.

4.4 Algoritmo con branching bidirezionale e lower bound con matching bipartito a costo minimo

In questo algoritmo ho utilizzato la versione bidirezionale del branching e l'algoritmo ungherese per il calcolo del lower bound. Grazie al matching bipartito è possibile considerare il grafo come orientato e quindi si possono proibire gli archi che non possono esistere in una soluzione ammissibile.
In figura 4.10 è riportato un esempio di proibizione.

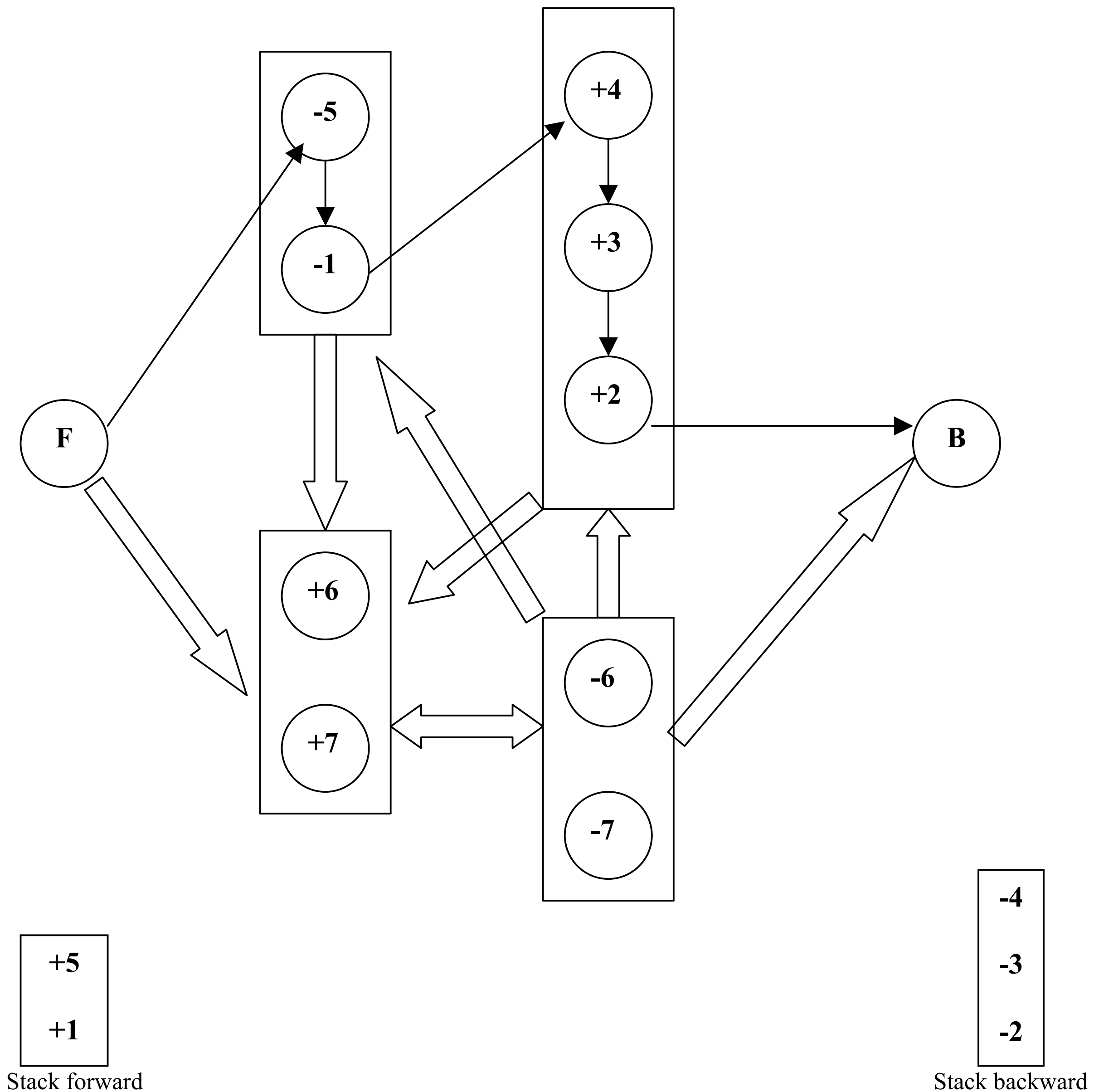


Figura 4.11: esempio di proibizione

Nell'esempio proposto sono riassunte tutte le regole:

- dall'ultima città inserita in avanti (F) si può andare a qualsiasi pickup o alla partner della pickup in cima allo stack forward

- da una delle città gemellate con quelle nello stack forward si può passare alla partner della precedente città nello stack o a una qualunque pickup non gemellata
- se la città +a precede +b allora –a non può precedere –b
- da una delivery si può passare alle pickup gemellate alle città nello stack backward o alle delivery partner delle città nello stack forward
- da una pickup gemellata ad una delivery nello stack backward si può passare alla gemella della delivery precedente nello stack oppure ad una qualsiasi pickup non gemellata
- l'ultima città inserita al contrario (B) può essere raggiunta da qualsiasi delivery non gemellata oppure dalla pickup gemellata alla delivery in cima allo stack backward
- non è ammesso l'arco che collega una delivery alla propria pickup.

Capitolo 5

Risultati sperimentali

5.1 Specifiche tecniche della macchina usata

Gli algoritmi sono stati codificati in C standard ISO 99 e compilati con GCC 3.2. I programmi sono stati lanciati su una macchina con processore Intel Pentium3 500Mhz con 256 MB di RAM e sistema operativo Windows 2000 Professional.

5.2 Formato dei file

I programmi ricevono in ingresso un file di testo con estensione "tsp", appartenente alla nota libreria TSPLIB, che riporta le coordinate sul piano delle città. Modificando il parametro "DIMENSION" del file è possibile scegliere con quante città lavorare (fig. 5.1).

```
NAME : fnl4461
COMMENT : Die 5 neuen Laender Deutschlands (Ex-DDR) (Bachem/Wottawa)
TYPE : TSP
DIMENSION : 15
EDGE_WEIGHT_TYPE : EUC_2D
NODE_COORD_SECTION
1   5639   6909
2   5652   6142
3   5654   6101
4   5659   6910
5   5659   6920
6   5661   6182
7   5663   6830
8   5669   6213
9   5670   6425
10   5671   6870
11   5673   6132
12   5682   6840
13   5690   6891
14   5694   6801
15   5698   6265
        [...]
```

Figura 5.1: esempio di contenuto di un file TSPLIB

Gli accoppiamenti tra le città sono scritti in un file con estensione "mtc", creato da un programma che ho scritto. Le coppie sono create utilizzando il generatore di numeri casuali fornito nella libreria standard del C. Il seme del generatore è determinato dal valore dell'orologio di sistema.
Il file contiene, per ogni città, il tipo nella seconda colonna (zero se di tipo delivery, uno se pickup) e la città partner nella terza colonna (fig. 5.2). Per la città deposito il tipo e la città partner sono valori fittizi.

0	3	0
1	0	13
2	0	14
3	0	11
4	1	9
5	1	10
6	0	8
7	0	12
8	1	6
9	0	4
10	0	5
11	1	3
12	1	7
13	1	1
14	1	2

Figura 5.2: esempio di accoppiamento tra le città. La prima colonna indica la città, la seconda il tipo e la terza la partner

Il programma di costruzione dei tour iniziali riceve in input la mappa e le coppie. L'output consiste in un file per ogni euristica costruttiva, che ha estensione "str" e contiene il tour iniziale e la sua lunghezza (fig. 5.3).

0
11
4
12
13
5
14
2
10
1
7
8
6
9
3
0

2029

Figura 5.3: esempio di tour iniziale calcolato con il Reversed Nearest Neighbour

Il programma per la ricerca locale prende in ingresso il file tsp, il file mtc e i tour generati dalle euristiche costruttive. Una volta terminato dà in output i risultati sui file di testo "complexNeighbourhoodDescent.txt" e "variableNeighbourhoodDescent.txt".
I programmi di branch-and-bound necessitano in input della mappa e degli accoppiamenti. L'output è scritto in un file di testo.

5.3 Risultati sperimentali

Nelle tabelle in figura 5.4 e 5.5 ho riportato i seguenti valori:

- **Nome mappa** indica la mappa utilizzata.
- **Ottimo globale** è il valore ottimo calcolato con il più veloce algoritmo esatto.
- **Tempo del B&B** esprime il tempo impiegato dall'algoritmo branch-and-bound per la ricerca della soluzione ottima.
- **Valore dell'euristica di costruzione** mostra il miglior valore calcolato dalle euristiche di costruzione.
- **Gap %** indica il divario in percentuale tra la lunghezza del miglior tour iniziale e quella del tour ottimo.
- **Tempo dell'euristica di costruzione** esprime il tempo impiegato per la costruzione di tutti i tour iniziali.
- **Valore della ricerca locale** mostra il valore ottimo calcolato con gli algoritmi di ricerca locale.
- **Gap %** è il divario in percentuale tra l'ottimo locale e l'ottimo globale.
- **Tempo della ricerca locale** indica il tempo impiegato dagli algoritmi di ricerca locale.

Ho utilizzato le mappe della libreria TSPLIB in cui le coordinate delle città sono espresse tramite una coppia di numeri interi. I tempi di esecuzione sono espressi nel formato "minuti.secondi".

Nella tabella 5.6 ho riportato i test effettuati sulla mappa "fnl4461" con un numero di città da 19 a 23. Ho scelto questa mappa perché è quella che negli altri test ha dato i tempi di esecuzione minori per la ricerca della soluzione esatta. Ho dovuto arrestare le prove al caso con 23 città, dato che i tempi di esecuzione cominciano a diventare notevoli.

I tempi di esecuzione degli algoritmi di costruzione del tour e di quelli di ricerca locale sono inferiori al secondo.

Nome mappa	Ottimo globale	Tempo del B&B	Valore dell'euristica di costruzione	Gap %	Valore della ricerca locale	Gap %
nrw1379	2.663	00.03	2.976	11,7	2.663	0,0
nrw1379	2.545	00.04	2.936	15,4	2.545	0,0
brd14051	4.416	01.05	4.972	12,6	4.531	2,6
brd14051	4.243	00.10	4.389	3,4	4.243	0,0
d15112	71.634	00.00	79.451	10,9	71.634	0,0
d15112	77.132	00.04	94.098	22,0	77.132	0,0
d18512	4.055	09.33	4.120	1,6	4.078	0,6
d18512	4.451	03.09	4.589	3,1	4.451	0,0
fnl4461	1.805	00.00	2.202	22,0	1.896	5,0
fnl4461	1.945	00.00	2.060	5,9	1.950	0,2
pr1002	11.777	00.03	12.982	10,2	11.777	0,0
pr1002	12.626	00.00	15.224	20,5	12.626	0,0

Figura 5.4: test effettuati con 15 città sulle mappe TSPLIB

Nome mappa	Ottimo globale	Tempo del B&B	Valore dell'euristica di costruzione	Gap %	Valore della ricerca locale	Gap %
nrw1379	2.604	01.04	2.785	6,9	2.723	4,5
nrw1379	2.789	00.17	3.078	10,0	2.798	0,3
brd14051	4.636	01.56	4.770	2,9	4.694	1,2
brd14051	4.304	00.42	4.578	6,4	4.332	0,6
d15112	72.237	19.14	85.236	18,0	73.609	1,9
d15112	71.356	00.52	89.297	25,1	72.954	2,2
d18512	4.761	06.22	5.030	5,6	4.811	1,0
d18512	4.241	01.39	4.799	13,1	4.363	2,9
fnl4461	2.031	00.00	2.112	3,9	2.031	0,0
fnl4461	2.199	00.22	2.345	6,6	2.229	1,4
pr1002	12.850	01.57	14.551	13,2	12.920	0,5
pr1002	13.830	01.26	16.137	16,6	14.186	2,6

Figura 5.5: test effettuati con 17 città sulle mappe TSPLIB

Numero di città	Ottimo globale	Tempo del B&B	Valore dell'euristica di costruzione	Gap %	Valore della ricerca locale	Gap %
19	2.000	00.05	2.217	10,8	2.000	0,0
19	2.148	05.22	2.332	8,5	2.196	2,2
21	2.018	08.36	2.351	16,5	2.018	0,0
21	2.521	41.14	2.621	4,0	2.521	0,0
23	2.014	47.57	2.490	23,6	2.099	4,2
23	2.058	160.59	2.210	7,4	2.128	3,4

Figura 5.6: test effettuati sulla mappa fnl4461

In figura 5.7 ho riportato il confronto tra i tempi di esecuzione degli algoritmi branch-and-bound.
In tutte le mappe ho considerato 13 città ad eccezione delle mappe fnl4461 e pr1002, nella quale ne ho considerate rispettivamente 17 e 15.
L'algoritmo numero 1 si riferisce al branch-and-bound con branching monodirezionale e calcolo del lower bound con Prim.

L'algoritmo numero 3 è il branch-and-bound con branching bidirezionale. Il lower bound anche in questo caso è calcolato con l'1-albero minimo. Gli altri due algoritmi hanno tempi di esecuzione decisamente più alti.

Nome mappa	Ottimo globale	Migliore algoritmo	Tempo di esecuzione
brd14051	4.156	1	<1s
d15112	70.139	1 e 3	<1s
d18512	4.284	1	<1s
fnl4461	1.821	3	<1s
nrw1379	2.406	3	<1s
pr1002	12.572	3	<1s

Figura 5.7: confronto tra gli algoritmi branch-and-bound

In figura 5.8 ho illustrato il confronto tra i valori calcolati dagli algoritmi di costruzione del tour iniziale. In tutte le mappe ho considerato 15 città.

Nome mappa	Ottimo globale	Valore N.N.	Valore Rev. N.N	Valore Ran. N.N	Valore Rev. Ran. NN	Valore C.I.	Valore L.I.	Valore F.I.	Valore N.I.
brd14051	4.143	4.499	6.295	6.804	6.601	4.848	5.117	6.191	4.848
brd14051	3.977	6.606	6.756	7.265	6.699	4.355	4.849	5.358	4.468
d15112	69.042	85.585	99.667	107.634	120.947	79.434	99.991	113.795	91.257
d15112	69.053	89.588	98.780	87.062	94.770	83.234	99.172	111.029	83.234
d18512	4.043	4.431	4.785	6.952	5.377	4.447	4.635	6.031	4.447
d18512	4.237	6.615	6.636	7.130	4.927	4.306	5.922	6.100	4.306
fnl4461	1.744	3.273	1.911	3.180	2.274	2.217	2.890	2.707	2.328
fnl4461	1.791	2.936	4.356	3.583	4.405	1.889	1.950	2.021	1.889
nrw1379	2.753	5.576	4.642	5.575	6.100	3.629	2.996	3.262	3.480
nrw1379	2.458	4.714	2.933	4.952	4.854	3.199	2.607	2.487	2.936
pr1002	11.838	17.002	12.826	18.460	15.847	17.171	18.032	18.915	17.171
pr1002	13.068	18.290	22.207	19.680	24.810	17.926	16.376	15.828	16.616

Figura 5.8 : confronto tra gli algoritmi di costruzione del tour iniziale con 15 città

Per evidenziare le differenze tra i valori trovati dagli algoritmi di costruzione del tour iniziale ho considerato il caso con 100 coppie di città (oltre al deposito). In questo caso ho considerato il valore ottimo calcolato dagli algoritmi di ricerca locale: l'alto numero di città rende improponibile l'uso degli algoritmi esatti. Il risultato è nella tabella in figura 5.9. Accanto al valore ottimo la lettera "C" o "V" indica se è stato calcolato con strategia "complex neighbourhood descent" o "variable neighbourhood descent". Nella stessa riga, in grassetto, è evidenziato il tour iniziale dal quale è stato calcolato l'ottimo locale.

Nome Mappa	Valore euristica	Valore N.N.	Valore Rev. N.N	Valore Ran. N.N	Valore Rev. Ran. NN	Valore C.I.	Valore L.I.	Valore F.I.	Valore N.I.
brd14051	23.576 V	82.123	78.241	**90.257**	75.601	25.805	29.167	28.270	28.118
brd14051	23.037 C	79.014	74.653	79.505	73.420	**25.000**	28.622	25.984	25.207
d15112	533.468 V	1.139.632	908.972	1.169.119	1.034.050	**575.104**	630.187	663.130	595.111
d15112	560.513 V	1.071.741	1.060.614	1.061.713	**1.142.069**	613.975	662.172	691.898	625.834
d18512	22.090 C	76.482	83.813	77.521	78.069	25.504	29.239	28.204	**25.980**
d18512	22.420 C	79.813	66.369	81.314	71.418	26.255	28.324	25.005	**24.966**
fnl4461	20.850 V	49.270	47.578	**52.267**	49.123	24.851	26.157	26.147	24.839
fnl4461	22.769 C	50.594	47.944	52.465	**50.420**	26.096	28.795	27.215	26.139
nrw1379	23.938 C	57.270	54.864	58.307	54.892	**26.201**	29.439	27.767	27.244
nrw1379	23.165 C	54.566	58.415	59.348	60.711	26.469	28.173	28.351	**26.321**
pr1002	185.926 C	441.180	428.228	644.207	449.097	221.209	223.127	**236.857**	199.820
pr1002	177.128 C	323.422	485.106	374.811	450.086	210.698	**214.848**	212.818	213.059

Figura 5.9: confronto tra gli algoritmi di costruzione del tour iniziale con 100 coppie di città

Capitolo 6

Conclusioni

Dai test risulta che gli algoritmi di costruzione del tour iniziale basati sul Nearest Neighbour in generale non hanno un buon comportamento dato che spesso producono valori molto alti. Gli altri algoritmi elaborano tour iniziali decisamente migliori.

Gli algoritmi di ricerca locale riescono ad avvicinarsi all'ottimo globale discostandosi di pochi punti percentuale. In particolare nel caso con 15 città spesso si raggiunge la soluzione ottima, mentre aumentando il numero delle coppie tende ad aumentare anche il numero delle soluzioni non ottime. L'errore si mantiene comunque molto basso.

Va osservato che i tempi di esecuzione sono nettamente a favore degli algoritmi di ricerca locale: nel caso con 100 coppie, sulla macchina utilizzata, hanno richiesto circa 2 minuti l'euristica di tipo complex neighbourhood descent e 30 secondi la variable neighbourhood descent. Gli algoritmi esatti in questi casi sono assolutamente improponibili.

La strategia di tipo complex neighbourhood descent si è dimostrata la migliore come qualità delle soluzioni, anche se la variable neighbourhood descent ha fornito il valore ottimo nel 30% dei casi.

Riguardo gli algoritmi esatti, c'è da dire che non si è verificata una netta predominanza. Il titolo di migliore è un testa a testa tra gli algoritmi in cui il lower bound è calcolato con l'algoritmo di Prim. Gli altri, in cui si utilizza l'algoritmo ungherese, non hanno dato i risultati sperati. Probabilmente ha inciso la maggiore complessità (cubica, contro la complessità quadratica dell'algoritmo di Prim).

Per quanto riguarda possibili futuri sviluppi, va osservato che il TSPRL è un problema non ancora studiato, quindi si possono considerare sia molte varianti (con capacità, su grafo asimmetrico, con finestre temporali e molte ancora) che nuovi algoritmi euristici o esatti.

Capitolo 7

Bibliografia

Traveling Salesman Problem

- Gregory Gutin, Abraham P. Punnen "Traveling Salesman Problem and Its Variations (Combinatorial Optimization, V. 12)", Kluwer (2002)

Traveling Salesman Problem with Pickup and Delivery

- Jacques Renaud, Fayez F. Boctor, Jamal Ouenniche "A heuristic for the pickup and delivery traveling salesman problem" Computers & Operations Research 27 (2000) 905-916
- Jacques Renaud, Fayez F. Boctor, Gilbert Laporte "Perturbation heuristics for the pickup and delivery traveling salesman problem" Computers & Operations Research 29 (2002) 1129-1141
- Michel Gendreau, Gilbert La porte, Daniele Vigo "Heuristics for the traveling salesman problem with pickup and delivery" Computers & Operations Research 26 (1999) 699-714

Branch-and-Bound

- T. Ibaraki "Enumerative Approaches to Combinatorial Optimization. Parts I, II. Annals of Operations Research, Vol. 10, 11. J.C. BALTZER AG (1988)"

Algoritmo di Prim

- Prim, R. C. (1957). Shortest connection networks and some generalizations. Bell Syst. Tech. J. 36, 1389-1401.

Ricerca locale

- E. H. L. Aarts, J. K. Lenstra "Local Search in Combinatorial Optimization", Kluwer (1999)

Algoritmo ungherese

- H.W. Kuhn "The Hungarian Method for the assignement problem", Naval Research Logistics Quarterly, 2, 83-87, (1955).