Progetto didattico - corso di Logistica

Andrea Maggiordomo
Informatica applicata - Università di Pisa

2012

Progetto n.3

Si consideri il problema di dover distribuire delle merci da un deposito ad un insieme di clienti. Presso il deposito sono disponibili M veicoli che possono fare esattamente un viaggio ciascuno. I veicoli sono tutti uguali e hanno una capacità in peso e in volume. Ciascun cliente richiede una quantità di merce di peso e volume dati. Ogni cliente deve essere visitato esattamente una volta. Dato l'insieme di vertici che rappresentano il deposito ed i clienti, è disponibile una matrice delle distanze per ogni coppia di vertici. Il problema è quello di pianificare i viaggi di consegna minimizzando il costo complessivo.

1 Descrizione del problema

Il problema preso in considerazione è una variazione del *Capacitated vehicle routing problem*, nel quale la merce è caratterizzata da due grandezze distinte: peso e volume.

Sia $U = \{1, \ldots, N\}$ l'insieme dei clienti che devono essere serviti, 0 il deposito; questo problema può essere formulato sul grafo non orientato $G = (V, E)$ con $V = U \cup \{0\}$, $E = \{\{i, j\} : i, j \in V,\ i \neq j\}$. Ad ogni cliente $i \in U$ sono associate le quantità w_i, v_i che definiscono rispettivamente il peso e il volume della merce richiesta. Presso il deposito è disponibile una flotta di M veicoli, tutti di capacità C_w per quanto riguarda il peso e C_v per quanto riguarda il volume. Ad ogni arco del grafo $\{i, j\} \in E$ è inoltre associata la quantità $c_{ij} \in \mathbb{N}$ che ne determina il costo di attraversamento. É stato considerato il caso simmetrico del problema con distanze eculidee: $\forall i, j \in V$, $c_{ij} = c_{ji}$ e $\forall i, j, k \in V$, $c_{ij} + c_{jk} \geq c_{ik}$. Ulteriori vincoli impongono che un cliente sia servito da un solo veicolo, e che lo stesso veicolo non possa effettuare più di un viaggio. L'obiettivo è quello di servire le domande di tutti i clienti rispettando i vincoli operativi menzionati e minimizzando il costo complessivo del percorso.

1.1 Formulazione matematica

La formulazione scelta è un'estensione della classica formulazione a due indici del *CVRP*. Ad ogni arco $\{i, j\} \in E$ è associata la variabile intera x_{ij} che definisce quante volte l'arco $\{i, j\}$ viene percorso. Una soluzione consiste in una copertura del grafo con M cicli, tutti adiacenti al deposito e tali da partizionare U in M sottoinsiemi disgiunti. Le richieste totali dei clienti appartenenti ai singoli cicli (rotte) non devono inoltre eccedere le capacità imposte ai veicoli.

La formulazione proposta è la seguente (dove $S(i)$ indica l'insieme di nodi adiacenti ad i)

$$\min \sum_{\{i,j\}\in E} c_{ij}x_{ij} \tag{1}$$

$$\sum_{j\in S(i)} x_{ij} = 2 \qquad \forall\, i \in U \tag{2}$$

$$\sum_{j\in S(0)} x_{0j} = 2M \tag{3}$$

$$\sum_{\substack{\{i,j\}\in E \\ i\in S, j\notin S}} x_{ij} \geq 2L(S) \qquad \forall\, S \subseteq U,\ |S| \geq 2 \tag{4}$$

$$0 \leq x_{ij} \leq 1 \qquad \forall\, \{i, j\} \in E,\ i, j \neq 0 \tag{5}$$

$$0 \leq x_{0j} \leq 2 \qquad \forall\, \{0, j\} \in E \tag{6}$$

$$x_{ij} \in \mathbb{Z} \qquad \forall\, \{i, j\} \in E \tag{7}$$

I vincoli (2) assicurano che in soluzione ogni cliente abbia grado 2, imponendo l'appartenenza di ciascun nodo in U ad una sola rotta. Il vincolo (3) fissa a $2M$ il grado del deposito, di fatto imponendo l'esistenza di M rotte in soluzione. I vincoli (4), detti vincoli di capacità (*rounded capacity inequalities*), definiscono una valutazione inferiore sul numero di archi incidenti su ogni possibile taglio che separa il deposito dai clienti. In pratica definiscono, per ciascun possibile sottoinsieme S di clienti, una valutazione inferiore sul numero di rotte necessarie per servirne le richieste, garantendo contemporaneamente che in soluzione non esista nessuna componente staccata dal deposito (infatti $\forall\, S \subseteq U,\ L(S) \geq 1$). La quantità $L(S)$ è definita come

$$L(S) = \max\{W(S), Q(S)\}$$

dove

$$W(S) = \left\lceil \frac{\sum_{i\in S} w_i}{C_w} \right\rceil, \quad Q(S) = \left\lceil \frac{\sum_{i\in S} v_i}{C_v} \right\rceil$$

e impone che il numero di rotte scelte per servire S sia almeno pari alla minima quantità di veicoli in grado di servirne le richieste senza eccedere le

limitazioni dei veicoli. In generale ogni arco è percorribile una sola volta, ma il vincolo (6) permette di assegnare agli archi adiacenti al deposito x_{0j} anche il valore 2: il significato di tale assegnamento è una rotta che fa meta verso il solo cliente j (in questo caso infatti j ha automaticamente grado 2 e non può incidere su nessun altro arco in soluzione).

1.2 Istanze

Le istanze del problema sono state generate a partire da quelle disponibili per il *CVRP* presso `http://www.coin-or.org/SYMPHONY/branchandcut/VRP/data/`. I parametri relativi alla quantità di volume richiesta sono C_v che identifica la quantità massima di richiesta in volume servibile da un singolo veicolo, e *tightness* $\in (0, 1)$ che regola il rapporto tra la quantità totale di merce richiesta dai clienti e la capacità dell'intera flotta di veicoli.

Le istanze sono state generate con assegnamenti casuali oppure vincolati ad un intorno della frazione di peso richiesta dallo stesso cliente.

Gli assegnamenti casuali prevedono la distribuzione della quantità di volume richiesta in maniera random attorno alla domanda media.

Gli assegnamenti correlati al peso sono calcolati con il seguente algoritmo: il generatore prende in input un ulteriore parametro *rangeCap* $\in [0, 0.99)$ che regola lo scarto tra la frazione di domanda in peso e quella in volume (relativamente alle capacità del veicolo) richiesta dallo specifico cliente. La quantità di volume v_i richiesta dal cliente i è quindi calcolata come la frazione di volume equivalente alla frazione di peso richiesta, variata con uno scarto casuale nell'intervallo $(-\mathit{rangeCap},\ \mathit{rangeCap})$.

Una volta che le domande in volume sono assegnate ai clienti, i singoli valori vengono aggiustati per rispettare la relazione

$$\frac{\sum_{i \in U} v_i}{M \cdot C_v} = \mathit{tightness}$$

cercando di rimanere dentro la frazione specificata per ogni cliente.

2 Metodi di risoluzione

In questa sezione vengono presentati gli algoritmi utilizzati per risolvere il problema, mentre una descrizione degli aspetti implementativi più interessanti è lasciata alla sezione successiva.

Il primo algoritmo utilizzato è un algoritmo metaeuristico (tabu search) relativamente semplice e basato su alcune delle idee discusse da Gendrau, Hertz e Laporte in [1]. Presenta alcuni aspetti interessanti relativamente all'esplorazione dello spazio delle soluzioni ed al come vengono tollerate soluzioni non ammissibili durante la ricerca locale, ma ha il difetto di dipendere da alcune scelte casuali rivelandosi non troppo stabile nei risultati ottenuti al variare di tali scelte.

Il secondo algoritmo utilizzato è invece un metodo esatto basato sulla risoluzione della formulazione matematica. Il metodo utilizzato è di tipo branch and cut: si parte da un rilassamento combinatorio continuo della formulazione e vengono aggiunti dinamicamente alcuni dei vincoli non considerati nel problema iniziale. Un approccio di questo tipo per risolvere il *CVRP* (del quale il problema preso in esame è parente stretto) è discusso ad esempio in [2] e [3].

2.1 Tabu search

Gli algoritmi di ricerca locale si basano sull'idea di vicinato, considerando soluzioni simili ad una data soluzione iniziale.

Nel caso del problema in esame una soluzione x non è altro che un insieme R_x di rotte (sequenze ordinate di clienti) individuate sul grafo associato al problema. L'insieme delle soluzioni vicine (*neighbourhood* - $N(R_x)$) è costituito da tutte le soluzioni ottenute da R_x rimuovendo un cliente da una particolare rotta $r \in R_x$ e reinserendolo in una delle rotte di R_x (non necessariamente diversa da r) nella posizione più conveniente.

Al fine di evitare che la ricerca si fossilizzi intorno ad una particolare soluzione, l'algoritmo di ricerca tabu impedisce che una soluzione già visitata sia esaminata nuovamente nelle iterazioni immediatamente successive, inserendo la soluzione vietata in una struttura ausiliaria detta *tabu-list* per un numero arbitrario di iterazioni.

Dal momento che la somma delle domande dei clienti presenti in una rotta può eccedere le capacità del veicolo, è necessario "guidare" la ricerca locale verso la regione ammissibile del problema. Questo viene fatto, analogamente a quanto discusso in [1], introducendo una funzione obiettivo ausiliaria c' che penalizzi le soluzioni non ammissibili permettendo comunque all'algoritmo di esaminarle. Le soluzioni durante la ricerca locale vengono valutate con la funzione obiettivo ausiliaria

$$c'x = cx + \sum_{r \in R_x} \alpha w_{over}(r) + \sum_{r \in R_x} \beta v_{over}(r) \quad \alpha, \beta > 0$$

dove $w_{over}(r)$ e $v_{over}(r)$ determinano rispettivamente l'eccesso di domanda in peso e in volume nella rotta r. Ovviamente se x è ammissibile $cx = c'x$, mentre se x non è ammissibile i moltiplicatori α e β determinano una penalità da aggiungere al valore della funzione obiettivo standard. Se durante la ricerca l'algoritmo spende troppo tempo a valutare soluzioni non ammissibili, i moltiplicatori possono essere aggiustati per cercare di ricondurre l'esplorazione verso la regione ammissibile, mentre nel caso contrario possono essere decrementati per indurre la ricerca ad allontanarsi da un minimo locale.

L'algoritmo utilizzato (*TabuRoute*) prevede una fase iniziale nella quale viene identificata una soluzione da utilizzare come punto di partenza nella ricerca, ed una seconda fase che si occupa della ricerca vera e propria.

2.1.1 Scelta della soluzione iniziale

La soluzione iniziale è scelta da un insieme di soluzioni candidate individuate utilizzando delle semplici euristiche. L'ammissibilità in questa fase non è necessaria e viene semplicemente scelta la soluzione che minimizza il valore della funzione obiettivo c' (valutata ponendo $\alpha = \beta = 1$).

Una prima soluzione è identificata utilizzando l'euristica del cliente più vicino per costruire $M - 1$ rotte ammissibili ed un'ultima rotta che includa ogni cliente non servito dalle precedenti.

Successivamente, altre $\frac{\sqrt{N}}{2}$ soluzioni sono costruite a partire da cicli hamiltoniani costruiti sull'insieme dei clienti impiegando ancora l'euristica del nodo più vicino (scegliendo ogni volta un diverso cliente come nodo iniziale), ed eseguendo la procedura *2-Opt* sui cicli individuati. Una soluzione è ottenuta da un ciclo hamiltoniano costruendo le rotte in maniera greedy, partendo dal nodo iniziale. Anche in questo caso l'ultima rotta individuata può rendere la soluzione non ammissibile.

Alla fine di questa fase la soluzione x_u che minimizza c' viene scelta come punto di partenza per la ricerca locale. Se durante questa prima fase si incontra una soluzione ammissibile, questa viene salvata come $\bar{x}$, altrimenti si pone $\bar{x} = \emptyset$.

2.1.2 Ricerca locale

La procedura di ricerca tabù è descritta nell'algoritmo 1.

Gli aspetti fondamentali riguardano la gestione della tabu list, quali soluzioni vicine vengono esaminate ad ogni iterazione e la regolazione dei parametri usati nella funzione obiettivo c'.

Per quanto riguarda la tabu list, dal momento che il passaggio da una soluzione x ad una vicina comporta il semplice reinserimento di un nodo i in una rotta $r_i \in R_x$, è naturale immaginarla come una lista di mosse (i, r_i) vietate. Nello specifico viene impedito che un nodo spostato di recente torni troppo presto nella rotta in cui si trovava precedentemente.

Il numero di soluzioni vicine esaminate è determinato da due parametri, p e q. Quando un cliente viene valutato per il reinserimento, vengono prese in considerazione solo le rotte a cui appartengono i p clienti più vicini ad esso; q stabilisce invece la cardinalità dell'insieme Z dei clienti candidati ad essere valutati per il reinserimento ad ogni iterazione.

La mossa più conveniente tra quelle esaminate è immediatamente implementata. Se durante l'esecuzione vengono trovate soluzioni migliori di $\bar{x}$ o x_u, queste vengono aggiornate.

Algoritmo 1 Search

Input: t_{max}, p, q, $\bar{x}$, x_u

$\alpha \leftarrow \beta \leftarrow 1.0$;
$x_{iter} \leftarrow x_u$;
while meno di t_{max} iterazioni senza migliorare $\bar{x}$ e x_u **do**
    if x_{iter} non ammissibile (peso) nelle ultime 10 iterazioni **then**
        $\alpha \leftarrow 2\alpha$;
    else if x_{iter} ammissibile (peso) nelle ultime 10 iterazioni **then**
        $\alpha \leftarrow \alpha/2$;
    end if
    if x_{iter} non ammissibile (volume) nelle ultime 10 iterazioni **then**
        $\beta \leftarrow 2\beta$;
    else if x_{iter} ammissibile (volume) nelle ultime 10 iterazioni **then**
        $\beta \leftarrow \beta/2$;
    end if
    $Z \leftarrow q$ elementi di U scelti casualmente;
    while $Z \neq \emptyset$ **do**
        $i \leftarrow next(Z)$;
        $R \leftarrow$ insieme delle rotte dei p clienti più vicini ad i;
        $r_i \leftarrow$ rotta a cui appartiene i;
        $S \leftarrow$ insieme delle soluzioni generate da x_{iter} effettuando le mosse non vietate $(i, r) : r \in R$;
        $x_{next} \leftarrow x \in S : c'x = min\{c'x' : x' \in S\}$;
        if $c'x_{next} < c'x_u$ **then** ▷ Aggiornamento delle soluzioni
            $x_u \leftarrow x_{next}$;
        end if
        if x_{next} ammissibile $\wedge$ $cx_{next} < c\bar{x}$ **then**
            $\bar{x} \leftarrow x_{next}$;
        end if
        $x_{iter} \leftarrow x_{next}$;
        Dichiara tabu per θ iterazioni la mossa (i, r_i); ▷ $\theta \in [5, 10]$
    end while
end while

La funzione obiettivo c' viene ricalibrata quando l'ammissibilità (relativamente al peso o al volume, i casi sono gestiti separatamente) della soluzione non cambia per 10 iterazioni consecutive.

L'algoritmo di ricerca locale viene invocato due volte, una prima volta con $q_1 = \min(N-1, 5)$ ed una seconda volta con $q_2 = N-1$, il parametro p è invece posto sempre a $N-2$: la seconda chiamata si propone di intensificare la ricerca intorno alla miglior soluzione trovata. TabuRoute è descritto dall'algoritmo 2.

Algoritmo 2 TabuRoute

Input: t_{max}
Output: $\bar{x}$ soluzione euristica del problema

1: $I \leftarrow$ insieme di rotte iniziali candidate;
2: $\bar{x} \leftarrow$ miglior soluzione ammissibile (se esiste) di I;
3: $x_u \leftarrow x_i \in I : \forall\, x_j \in I,\ x_j \neq x_i,\ c'x_j \geq c'x_i$;
4: Search($t_{max}|U|$, p, q_1, $\bar{x}$, x_u);
5: Search($M|U|$, p, q_2, $\bar{x}$, x_u);
6: Esegui *2-Opt* su $\bar{x}$;

2.2 Risoluzione della formulazione matematica

La formulazione matematica proposta non è direttamente risolvibile, dal momento che il numero di vincoli di capacità (necessari a renderla completa) cresce esponenzialmente con la quantità di nodi presenti nel grafo. L'approccio utilizzato è stato quindi quello di generare questi vincoli solo quando necessari, nel modo prescritto dagli algoritmi di tipo Branch-and-cut.

Nell'accezione tradizionale gli algoritmi di questo tipo lavorano sulla formulazione completa, risolvendo il rilassamento continuo e individuando disuguaglianze valide (cioè verificate da qualsiasi soluzione ammissibile intera) che escludano la soluzione frazionaria ottenuta - il problema di individuare tali disugiaglianze viene detto *problema di separazione*. Le disuguaglianze trovate vengono quindi aggiunte alla formulazione e viene risolto il "nuovo" problema di programmazione lineare ottenuto. Questa procedura è ripetuta iterativamente fino a quando non viene individuata una soluzione per la quale non è più possibile trovare disuguaglianze valide violate (*algoritmo dei piani di taglio*). Se a questo punto la soluzione individuata non è intera si rende necessario ricorrere al branching ed iniziare ad esplorare l'albero di enumerazione.

Relativamente al problema preso in esame, tuttavia, i vincoli di capacità non sono semplici disuguaglianze valide ma fanno parte a tutti gli effetti della formulazione matematica. Generando dinamicamente questi vincoli può succedere quindi di imbattersi in soluzioni intere non ammissibili per il problema originale. Fortunatamente, in questo caso individuare eventuali vin-

coli di capacità violati è estremamente facile e si riconduce sostanzialmente ad una visita del grafo non orientato ricavato dalla soluzione.

2.2.1 Separazione dei vincoli di capacità

Per risolvere il problema di separazione sono state utilizzate alcune delle procedure euristiche descritte in [2] e in [3]. Questi algoritmi lavorano sul grafo pesato e non orientato $\hat{G} = \{V, \hat{E}\}$ ricavato dalla soluzione parziale $\hat{x}$, con $\hat{E} = \{\{i, j\} \in E : \hat{x}_{ij} > 0\}$ e con peso sugli archi $d_{ij} = \hat{x}_{ij}$. Si ricorre talvolta anche al grafo $\hat{G}_0$ ottenuto da $\hat{G}$ rimuovendo il deposito e tutti gli archi incidenti ad esso.

Dal momento che ogni soluzione deve essere connessa, prima di avviare le euristiche di separazione si effettua una semplice visita del grafo residuo per assicurarsi che tutti i nodi siano visibili dal deposito. In caso contrario, i nodi non raggiungibili inducono un vincolo di capacità violato che viene aggiunto alla formulazione, e si procede immediatamente alla risoluzione del nuovo LP. Questo primo passo garantisce che le euristiche descritte in seguito operino sempre su un grafo connesso.

Indicando con S un insieme di clienti e con $\delta(S)$ la somma dei pesi degli archi incidenti sul taglio di $\hat{G}$ $(S, V \setminus S)$, osserviamo che S viola un vincolo di capacità se $\delta(S) < 2L(S)$. Le euristiche di separazione descritte in seguito cercano insiemi di clienti di questo tipo.

ConnectedComponents La procedura *ConnectedComponents* descritta in [3] inizia considerando l'insieme C delle componenti connesse massimali di $\hat{G}_0$. Se una componente C_i viola un particolare vincolo di capacità questo viene aggiunto all'insieme di disuguaglianze trovate, altrimenti l'algoritmo procede iterativamente rimuovendo da C_i un nodo v tale da lasciare invariata la valutazione inferiore sul numero di veicoli necessari a servirne le richieste fino a quando trova un vincolo di capacità violato, oppure non esiste più nessun nodo con queste caratteristiche. Il procedimento è ripetuto per ogni elemento di C.

Come già menzionato le euristiche di separazione sono invocate sempre su soluzioni connesse. Nel caso di soluzione intera, le componenti connesse massimali di $\hat{G}_0$ sono esattamente gli insiemi di clienti serviti dalle rotte individuate in soluzione. In questo caso l'euristica *ConnectedComponents* si può considerare un separatore "esatto" e si ha la garanzia che ogni soluzione intera non ammissibile sia tagliata dalla visita (se non connessa) oppure da questa euristica (se una qualche rotta viola i vincoli di capacità). Ogni soluzione intera non separata è quindi sicuramente ammissibile per il problema originale.

Shrinking La procedura *Shrinking*, anch'essa descritta in [3], considera invece gli archi attivi della soluzione frazionaria, cercando coppie di nodi

Algoritmo 3 ConnectedComponents

Input: $\hat{G}_0$ ricavato da $\hat{x}$
Output: D insieme di disuguaglianze violate da $\hat{x}$

Seleziona l'insieme C delle componenti connesse massimali di $\hat{G}_0$;
for all $C_i \in C$ **do**
    if $\delta(C_i) < 2L(C_i)$ **then**
        Aggiungi a D la disuguaglianza violata indotta da C_i;
    else
        Determina, se esiste, il nodo $v \in C_i$ che minimizza $\delta(C_i \setminus \{v\})$ e tale che $L(C_i) = L(C_i \setminus \{v\})$;
        if v non esiste **then**
            Passa alla prossima componente;
        else
            $C_i \leftarrow C_i \setminus \{v\}$;
            goto 3;
        end if
    end if
end for

Algoritmo 4 Shrinking

Input: $\hat{G}$ ricavato da $\hat{x}$
Output: D insieme di disuguaglianze violate da $\hat{x}$

while *true* **do**
    if $\exists\, \{i,j\} \in \hat{E}$ tale che $S = \{i,j\}$, $\delta(S) < L(S)$ **then**
        Aggiungi a D la disuguaglianza violata da S;
    end if
    Seleziona l'arco $\{i,j\}$ non adiacente al deposito che massimizza w_{ij};
    if $\nexists\, \{i,j\} \vee w_{ij} < 1$ **then**
        Break;
    else
        Fondi i nodi i e j in un unico supernodo;
        Aggiorna $\hat{G}$ e il peso degli archi w;
    end if
end while

$\{i, j\}$ di U (deposito escluso) che violino qualche vincolo di capacità in $\hat{G}$. Una volta esaminate tutte le coppie, viene individuato l'arco $\{h, k\}$ non adiacente al deposito di peso d_{hk} massimo in $\hat{E}$: se $d_{hk} \geq 1$, h e k vengono fusi in un unico supernodo u con

$$w_u = w_h + w_k, v_u = v_h + v_k,\ d_{ui} = d_{hi} + d_{ki} \quad \forall i \in S(h) \cup S(k).$$

L'algoritmo procede iterativamente cercando coppie di (super)nodi che violino qualche vincolo di capacità e collassando un arco come specificato in precedenza fino a quando tutti gli archi sono adiacenti al deposito (nel qual caso non è possibile fondere i due estremi senza includere il deposito stesso) oppure hanno peso minore di 1.

GreedyRoundCap La procedura *GreedyRoundCap* è analoga ad una delle euristiche greedy descritte in [2]; inizia considerando l'insieme S composto da un singolo cliente e cerca di espanderlo aggiungendo iterativamente un nodo v che minimizzi il valore di $\delta(S \cup \{v\})$, controllando ad ogni incremento se S viola un vincolo di capacità. Il procedimento è ripetuto usando ciascun cliente come elemento di partenza in S;

Algoritmo 5 GreedyRoundCap

Input: $\hat{G}$ ricavato da $\hat{x}$
Output: D insieme di disuguaglianze violate da $\hat{x}$

```
1: for all i ∈ U do
2:     S ← {i};
3:     repeat
4:         Aggiungi a S il nodo v che minimizza δ(S ∪ {v});
5:         if S viola un vincolo di capacità then
6:             Aggiungi la disuguaglianza violata da S a D;
7:         end if
8:     until Non è più possibile estendere S
9: end for
```

3 Implementazione

In questa sezione vengono discusse alcune delle scelte implementative effettuate, il codice sorgente è scritto interamente in C++. La realizzazione dell'algoritmo metaeuristico è abbastanza lineare ed è trattata brevemente; più attenzione è dedicata invece alla descrizione di come è stato implementato il branch and cut.

Per risolvere i modelli matematici sono state utilizzate le librerie messe a disposizione dal progetto COIN-OR (`http://www.coin-or.org/`). Nello specifico sono state impiegate la libreria Clp per risolvere i problemi di

programmazione lineare, la libreria Cgl per interfacciare le euristiche di separazione implementate con i solutori e la libreria Cbc che offre un solutore specifico di tipo branch and cut.

Del branch and cut sono state realizzate due implementazioni: una si appoggia ai solutori solamente per risolvere i problemi di programmazione lineare, controllando direttamente la strategia di visita dell'albero di enumerazione, la strategia di branching e la generazione delle righe della formulazione. Una seconda implementazione affida invece il processo di ricerca interamente a Cbc, limitandosi a fornire il generatore di tagli specializzato.

3.1 TabuRoute

Come anticipato l'implementazione dell'algoritmo TabuRoute non presenta aspetti particolarmente interessanti.

Per quanto riguarda la tabu-list, visto che le mosse vietate possono essere denotate da coppie (i, r) viene naturale rappresentarla usando una matrice intera T di N righe e M colonne che "etichetta" le mosse con il numero di iterazioni rimanenti per le quali queste sono ancora da considerare vietate. Stabilire se la mossa (i, r) è vietata è corrisponde a controllare se l'elemento T_{ij} sia positivo.

Per il resto sono state utilizzate quasi esclusivamente strutture dati e algoritmi messi a disposizione dalla Standard Template Library. Una soluzione (insieme di rotte) è rappresentata da M vettori di interi (singole rotte, l'ordine degli elementi determina la sequenza con la quale i clienti sono visitati a partire dal deposito). La selezione dei p clienti più vicini ad un dato cliente i è effettuata ordinando l'insieme U con la funzione `std::sort` utilizzando un operatore di confronto specializzato per il quale $j < k \iff c_{ij} < c_{ik}$, e selezionando i primi p elementi dell'insieme così ordinato.

3.2 Branch and cut

I separatori sono implementati estendendo la classe `CglCutGenerator` fornita dalla libreria Cgl per renderli compatibili con i solutori Clp e Cbc. La generazione delle disuguaglianze è delegata all'implementazione del metodo `generateCuts(OsiSolverInterface&, OsiCuts&, CglTreeInfo)` . Il generatore ricava la soluzione corrente dall'oggetto `OsiSolverInterface` e inserisce i tagli individuati nella struttura `OsiCuts`.

L'applicazione effettiva dei tagli individuati alla formulazione è responsabilità di chi invoca questo metodo.

3.2.1 Implementazione esplicita dell'algoritmo

L'implementazione esplicita dell'algoritmo branch and cut (algoritmo 6) utilizza un oggetto `OsiClpSolverInterface` per risolvere i problemi di programmazione lineare via via generati a partire dal modello iniziale $\mathcal{P}$. I

sottoproblemi (nodi dell'albero di enumerazione) sono rappresentati come una lista di vincoli associati ad ogni variabile; ogni vincolo è identificato da una coppia $\langle$*upper bound*, *lower bound*$\rangle$.

Algoritmo 6 Branch and cut

Input: $(G, c, w, v, C_w, C_v), \mathcal{P}, \epsilon$
Output: $\bar{x}$ soluzione ottima del problema

1: $\bar{x} \leftarrow Euristica(G, c, w, v, C_w, C_v)$;
2: $z_{ub} \leftarrow c\bar{x}$;
3: $Q \leftarrow \{\mathrm{X}\}$;
4: **repeat**
5: $\quad \mathrm{X}_i \leftarrow next(Q)$;
6: $\quad$ **repeat**
7: $\quad\quad \hat{x} \leftarrow LP(\mathcal{P}, \mathrm{X}_i)$;
8: $\quad\quad$ **if** $\hat{x} = \emptyset \vee c\hat{x} \geq z_{ub}$ **then**
9: $\quad\quad\quad$ Chiudi il nodo X_i;
10: $\quad\quad$ **else if** $\hat{x}$ è ammissibile per il problema originale **then**
11: $\quad\quad\quad \bar{x} \leftarrow \hat{x},\ z_{ub} \leftarrow c\hat{x}$;
12: $\quad\quad\quad$ Chiudi il nodo X_i
13: $\quad\quad$ **else**
14: $\quad\quad\quad \mathcal{L} \leftarrow Cut(\hat{x}, \mathcal{P})$;
15: $\quad\quad\quad \mathcal{P} \leftarrow \mathcal{P} \cup \mathcal{L}$;
16: $\quad\quad$ **end if**
17: $\quad$ **until** X_i chiuso $\vee\ \mathcal{L} = \emptyset$
18: $\quad$ **if** X_i non è chiuso **then**
19: $\quad\quad Q \leftarrow Q \cup Branch(\mathrm{X}_i, \hat{x})$;
20: $\quad$ **end if**
21: $\quad$ **if** Tutti i nodi al livello corrente sono visitati **then**
22: $\quad\quad$ Aggiorna z_{lb} con il più piccolo valore $c\hat{x}$ rilevato;
23: $\quad$ **end if**
24: **until** $Q = \emptyset \vee \frac{z_{ub} - z_{lb}}{z_{lb}} \leq \epsilon$

Il nodo radice dell'albero di enumerazione, nel quale i vincoli sulle variabili sono quelli imposti dal problema iniziale, è indicato con X. Quando l'algoritmo dei piani di taglio termina generando una soluzione frazionaria $\hat{x}$, è necessario ricorrere al branching. Supponiamo che la variabile candidata al branching (scelta nel modo descritto in seguito) sia i, il problema X viene decomposto nei due sottoproblemi $\mathrm{X}_{i1} = \mathrm{X}_i \cup \{x_i \leq \lfloor \hat{x}_i \rfloor\}$ e $\mathrm{X}_{i2} = \mathrm{X}_i \cup \{x_i \geq \lceil \hat{x}_i \rceil\}$. È intuitivo verificare come questa regola (applicata ad ogni nodo per il quale sia necessario ricorrere al branching) partizioni lo spazio delle soluzioni indotto da X in due sottoinsiemi disgiunti. Questo comporta che l'albero di enumerazione esamini al caso peggiore ogni possibile soluzione del problema originale, garantendo quindi la correttezza

dell'algoritmo.

La visita dell'albero è eseguita in ampiezza. I nodi sono memorizzati in due code, quella dei nodi al livello corrente e quella dei nodi al livello successivo. Quando un livello è esaminato per intero la corrispondente coda si svuota, e il lower bound globale z_{lb} è aggiornato con il più piccolo lower bound individuato esaminando i nodi al livello appena completato.

L'identificazione delle disuguaglianze violate è affidata al generatore descritto in precedenza: queste, come accennato, sono memorizzate in un oggetto `OsiCuts` e sono poi applicate al modello contenuto nel solutore con il metodo `OsiClpSolverInterface::applyCuts( OsiCuts & )`.

Sottoproblemi per i quali il valore della funzione obiettivo ecceda l'upper bound trovato o non esistano soluzioni vengono potati. Allo stesso modo vengono chiusi sottoproblemi per i quali si riesca ad individuare una soluzione intera ammissibile.

3.2.2 Cbc

Come già discusso, la strategia di risoluzione del problema usa un approccio di tipo branch and cut per generare dinamicamente la formulazione.

Il solutore messo a disposizione da Cbc nella classe `CbcModel` implementa un algoritmo branch and cut che, nella configurazione standard, opera sul modello IP specificato sotto l'ipotesi che questo sia completo, ed offre la possibilità di invocare algoritmi di separazione generici.

Oltre ad implementare un separatore specializzato per il problema, è necessario quindi configurare il solutore affinchè sia compatibile con l'approccio considerato. In particolare, visto che alcune soluzioni intere potrebbero essere tagliate, bisogna impedire al solutore di considerare qualsiasi soluzione come ammissibile prima che questa sia esaminata dai separatori. Occorre inoltre continuare a tagliare le soluzioni frazionarie fino a quando non è più possibile individuare ulteriori vincoli violati.

Il primo obiettivo è raggiunto passando al solutore un oggetto `OsiBabSolver` che contiene informazioni supplementari sulla strategia di risoluzione da impiegare.

```
CbcModel model;
OsiBabSolver solverCharacteristics;
solverCharacteristics.setSolverType( 4 );
model.solver()->setAuxiliaryInfo( &solverCharacteristics );
```

Qui `setSolverType( 4 )` informa il solutore che una soluzione intera potrebbe ancora essere tagliata.

Per imporre che il separatore venga chiamato fino a quando questo non fallisce nel trovare tagli è sufficiente l'istruzione

```
model.cutGenerator( i )->setMustCallAgain( true );
```

che applica tale condizione all'i-esimo separatore aggiunto al modello.

Oltre a queste istruzioni occorre anche vietare l'ammissibilità delle soluzioni trovate durante l'esecuzione dello strong branching, dal momento che queste non sono mai processate attraverso i separatori. Nel codice questa funzionalità è implementata attraverso la classe `CbcFeasibilityNoStrong`, passando al modello un oggetto di questo tipo con il metodo `setFeasibility`.

La strategia di visita utilizzata è sostanzialmente una depth-first fino alla prima soluzione ammissibile trovata, dopodiché questa varia dinamicamente tra depth-first e breadth-first a seconda di come evolve l'albero di enumerazione.

4 Risultati computazionali

La tabella 1 riporta i parametri utilizzati per generare le istanze di prova.

Istanza	Sorgente	C_q	Tightness	RangeCap
n16-k8.data	P-n16-k8	50	0.94	0.25
n31-k5.data	B-n31-k5	80	0.87	0.15
n37-k5.data	A-n37-k5	100	0.91	0.30
n44-k6.data	A-n44-k6	200	0.91	0.40
n46-k7.data	A-n46-k7	100	0.88	0.30
n51-k7.data	B-n51-k7	1000	0.92	0.35
n57-k9.data	B-n57-k9	250	0.83	0.15
n64-k9.data	B-n64-k9	400	0.85	0.35
n22-k4.data	E-n22-k4	1000	0.95	n/a (random)
n30-k3.data	E-n30-k3	1000	0.88	n/a (random)
n34-k5.data	A-n34-k5	100	0.95	n/a (random)
n38-k6.data	B-n38-k6	200	0.90	n/a (random)

Tabella 1: Istanze generate

I costi associati agli archi sono calcolati come suggerito dalla libreria di istanze TSPLIB, arrotondando i valori frazionari con la funzione `nint`.

Tutti i risultati sono stati ricavati eseguendo i test su Linux, la versione di Cbc usata è la 2.7.

I risultati dell'algoritmo TabuRoute sono riportati nella tabella 2. Il gap è espresso rispetto al valore ottimo del problema ricavato in fase di test. Per quanto riguarda i parametri algoritmici, incrementare oltre una certa soglia il valore di t_{max} non ha comportato miglioramenti rilevanti. La scelta di q e p è stata guidata dalla volontà di permettere alla ricerca di poter esaminare diverse soluzioni (ogni cliente viene reinserito in ciascuna delle rotte individuate) senza rallentare eccessivamente le singole iterazioni - valori di q troppo elevati nella prima invocazione di *Search* hanno palesato

Istanza	t_{max}	z_{ub}	Tempo	Gap
n16-k8	250	484	0	0
n22-k4	250	412	2	0
n30-k3	250	583	0	6%
n31-k5	250	686	1	0.73%
n34-k5	250	830	1	2.47%
n37-k5	250	747	1	5.96%
n38-k6	250	862	1	4.10%
n44-k6	250	996	107	5.28%
n46-k7	250	981	3	4.58%
n51-k7	250	1189	3	10.91%
n57-k9	250	1630	8	2%
n64-k9	250	1052	65	22.18%

Tabella 2: Risultati TabuRoute (seed 12345)

questo effetto collaterale. È giusto menzionare che in alcuni casi si sono verificati cicli che hanno di fatto stallato la ricerca.

Relativamente ai tempi di esecuzione, non emerge alcun legame rilevante tra questi e la qualità delle soluzioni fornite, evidenziando un comportamento erratico dell'algoritmo. Un esame (abbastanza superficiale, ma sufficiente a chiarire questo aspetto) della dipendenza dalla sequenza pseudo-casuale considerata, mostrato nella tabella 3, sembra suffragare questa ipotesi: i valori ricavati possono essere estremamente variabili.

Istanza	z_{ub} migliore	z_{ub} peggiore	Variazione
n37-k5	722	735	1.8%
n38-k6	855	889	3.97%
n44-k6	1028	1144	11.28%
n46-k7	971	982	1.13%
n51-k7	1092	1572	43.95%
n64-l9	905	1080	19.34%

Tabella 3: Variazione nei risultati di TabuRoute (5 run con $t_{max} = 100$)

La tabella 4 riporta invece i risultati ottenuti con il branch and cut, usando sia l'implementazione di Cbc che quella proposta nel progetto; la prima valutazione superiore (indicata nella colonna *cut-off*) è ottenuta invocando *TabuRoute* con $t_{max} = 50$, inizializzato con 5 semi diversi.

Un aspetto interessante è la comparazione tra le prestazione dell'implementazione esplicita e quella di Cbc. Su istanze di dimensioni contenute l'implementazione esplicta si comporta in maniera accettabile, risultando più lenta ma riuscendo comunque a completare la visita dell'albero di enumerazione – eccetto il caso di n34-k5. Su istanze di dimensioni maggiori

Istanza	CbcModel B&C			VrpSolver B&C			cut-off
	z_{ub}	Tempo	Gap	z_{ub}	Tempo	Gap	
n16-k8	484	0	0	484	0	0	485
n22-k4	412	6	0	412	12	0	502
n30-k3	550	12	0	550	65	0	584
n31-k5	681	1	0	681	25	0	687
n34-k5	810	19	0	815	3600	2.26%	842
n37-k5	705	6	0	705	5	0	721
n38-k6	828	115	0	828	3600	0.85%	856
n44-k6	946	2206	0	-	-	-	1029
n46-k7	938	843	0	940	3600	1.18%	972
n51-k7	1072	392	0	1072	3600	0.37%	1093
n57-k9	1598	10027	0	-	-	-	1616
n64-k9	861	38	0	861	593	0	929

Tabella 4: Risultati branch and cut

la chiusura del gap si fa via via più problematica con l'eccezione dell'istanza n64-k9: in quest'ultimo caso la rapidità con la quale si trovano "buone" soluzioni ammissibili permette di contenere le dimensioni dell'albero di enumerazione e di convergere in tempi relativamente brevi alla soluzione ottima.

4.1 Configurazione del solutore general purpose

Il solutore è stato provato in diverse configurazioni per valutare l'impatto che alcuni dei diversi aspetti che caratterizzano un algoritmo branch and cut possono avere sulle prestazioni. Nello specifico le possibilità esaminate sono state l'impiego o meno dei separatori generici implementati nella libreria Cgl (tra i quali solo `CglProbing` e `CglGomory` si sono rivelati efficaci), ed il passare al solutore una valutazione superiore valida sulla funzione obiettivo.

Nelle tabelle seguenti l'etichetta `CO` si riferisce all'imposizione della limitazione superiore (`1`) o meno (`0`) sulla funzione obiettivo, mentre `S` riguarda la combinazione di separatori usata (`0` le sole euristiche, `1` ricorre ai tagli di Gomory se le euristiche falliscono, `2` inizialmente invoca anche CglProbing per cercare di fissare qualche variabile).

La tabella 5 riporta i risultati ottenuti impiegando i separatori generici. In maniera inaspettata, l'uso di CglProbing si rivela controproducente in quasi tutti i casi ed indipendentemente dall'imposizione del cut off sulla funzione obiettivo. Dai raffronti la configurazione migliore sembra essere `[CO=1 S=1]`: la versione senza cut off si comporta mediamente meglio ma evidenzia enormi difficoltà nella risoluzione dell'istanza n57-k9, mostrando come in questo caso la disponibilità di un buon upper bound aiuti significativamente la risoluzione dell'istanza.

	CO=1 S=1			CO=1 S=2			CO=0 S=1			CO=0 S=2		
Istanza	#Nodi	Tempo	Gap	#Nodi	Tempo	Gap	#Nodi	Tempo	Gap	#Nodi	Tempo	Gap
n16-k8	10	0.25	-	10	0.24	-	32	0.65	-	32	0.67	-
n22-k4	1317	9	-	1170	7	-	882	5	-	995	9	-
n30-k3	2019	20	-	1358	12	-	2964	36	-	1393	12	-
n31-k5	100	1	-	363	3	-	284	3	-	394	5	-
n34-k5	1116	23	-	1249	32	-	744	14	-	852	18	-
n37-k5	268	6	-	249	7	-	219	6	-	227	7	-
n38-k6	2051	56	-	3526	126	-	1740	50	-	3154	109	-
n44-k6	21554	3600	1.19%	18548	3600	1.56%	9936	1040	-	18282	2704	-
n46-k7	1757	95	-	7515	629	-	1378	71	-	1419	91	-
n51-k7	19859	2577	-	4000	231	-	6696	603	-	2684	133	-
n57-k9	17097	3744	-	-	-	-	39000	29932	4.89%	-	-	-
n64-k9	448	40	-	446	40	-	446	40	-	446	44	-

Tabella 5: Confronto tra varie combinazioni di separatori

	CO=1 S=0		CO=0 S=0		CO=1 S=1		
Istanza	#Nodi	Tempo	#Nodi	Tempo	#Nodi	Tempo	Gap
n16-k8	12	0.17	47	0.74	10	0.25	-
n22-k4	1124	6	1148	7	1317	9	-
n30-k3	1562	12	1684	15	2019	20	-
n31-k5	94	1	290	4	100	1	-
n34-k5	1297	19	916	13	1116	23	-
n37-k5	275	6	233	6	268	6	-
n38-k6	4212	114	3323	91	2051	56	-
n44-k6	18894	2209	24089	3350	21554	3600	1.19%
n46-k7	13019	848	1701	83	1757	95	-
n51-k7	6630	394	9872	637	19859	2576	-
n57-k9	30176	10027	24267	5924	17097	3744	-
n64-k9	450	38	450	39	448	40	-

Tabella 6: Confronto tra diverse configurazioni del solutore Cbc

Il confronto con le configurazioni che fanno uso delle sole euristiche è riportato nella tabella 6. Per quanto riguarda queste ultime la situazione è se possibile ancora meno definita, e curiosamente la valutazione superiore che con i separatori generici risultava determinante per risolvere l'istanza n57-k9 in questo caso comporta un decadimento delle prestazioni significativo.

Risulta estremamente difficile trarre conclusioni definitive su quale sia la configurazione migliore tra quelle provate. L'utilizzo dell'upper bound sembra determinante in un solo caso, mentre negli altri il contributo è molto meno evidente e diverse volte addirittura dannoso. Questa scelta a livello teorico pare ragionevole, ed è stata operata con l'intento di contenere le dimensioni dell'albero di enumerazione nel caso in cui l'algoritmo non riesca a trovare "velocemente" soluzioni ammissibili, ma con le istanze provate ciò non si è verificato in maniera consistente. Similmente, anche l'impatto dei separatori generici è di difficile lettura e fortemente dipendente dalle singole istanze, come evidenziato sempre dalla tabella 6. Interessante come non emerga alcun nesso definito tra l'impiego dei separatori generici e una diminuzione del numero di nodi esaminati dall'algoritmo che anzi talvolta risulta maggiore. Questo fatto può essere in parte giustificato con l'ipotesi che le soluzioni intere a cui si converge con l'ausilio dei separatori generici si rivelino spesso non ammissibili e vengano pertanto tagliate, oppure che il contributo dei separatori alla progressione del lower bound non sia significativo.

4.2 Prove supplementari

In questa sezione sono riportate le prove effettuate per valutare l'impatto che i parametri relativi al volume hanno sulla difficoltà delle istanze. Le prove sono state effettuate utilizzando il solutore Cbc in configurazione `[CO=1`

Tightness = 0.90			
Istanza sorgente	rc = 0.20	rc = 0.35	rc = 0.50
A-n32-k5.vrp	9.71	11.73	7.55
A-n33-k5.vrp	9.09	9.42	4.76
A-n33-k6.vrp	7.16	9.40	104.97
A-n34-k5.vrp	9.44	6.54	8.01
A-n37-k5.vrp	10.10	12.71	56.93
A-n39-k5.vrp	36.91	20.41	42.93
A-n39-k6.vrp	19.52	7.21	11.45
A-n44-k6.vrp	428.23	1172.78	20356.20
A-n45-k6.vrp	69.83	2134.80	262.80
B-n31-k5.vrp	5.54	55.18	3.26
B-n35-k5.vrp	3.58	135.14	54.86
B-n38-k6.vrp	5.57	9.49	5.17
B-n41-k6.vrp	13.57	11.08	230.61
B-n45-k5.vrp	15.07	12.16	15.26
B-n50-k7.vrp	21.72	20.96	22.25
E-n30-k3.vrp	6.16	6.62	6.14
E-n33-k4.vrp	1.90	3.59	1.61
P-n40-k5.vrp	7.91	6.51	7.64
P-n45-k5.vrp	18.09	191.18	16.02

Tabella 7: Variazione RangeCap

`S=0]`, ovvero calcolando una limitazione superiore sulla funzione obiettivo e ricorrendo alle sole euristiche specifiche per risolvere il problema di separazione. I tempi riportati includono il calcolo dell'upper bound iniziale.

La tabella 7 evidenzia le prove effettuate variando i rapporti tra le frazioni di peso e di volume richieste dai clienti (ovvero modificando il parametro *rangeCap*), mantenendo la quantità di volume totale richiesta (regolata dal parametro *tightness*) pari al 90% della quantità massima servibile. Le prove riportate nella tabella 8 invece si riferiscono ad istanze nelle quali viene variata la domanda totale dei clienti mantenendo costante il parametro *rangeCap*. In questo secondo caso è stato posto un tempo limite di 3600 secondi, per le istanze non risolte entro questo limite è riportato il gap relativo.

In generale variare le proporzioni tra le richieste in un intervallo più ampio non si è rivelato un modo particolarmente affidabile di "complicare" le istanze. Più interessanti sono invece gli esperimenti effettuati variando la quantità totale di domanda richiesta: in questo caso infatti la tendenza è più delineata. Nelle prove effettuate con Tightness = 0.75 molte istanze sono risolte con le stesse rotte della soluzione ottima del CVRP standard, cioè per tale valore le quantità di volume richieste sono dominate in larga parte da quelle di peso. Con valori più alti invece, le richieste relative

RangeCap = 0.50			
Istanza sorgente	t = 0.75	t = 0.85	t = 0.95
A-n32-k5.vrp	2.44	3.02	8.73
A-n34-k5.vrp	6.50	10.50	9.31
A-n36-k5.vrp	11.67	431.83	19.32
A-n37-k5.vrp	5.98	13.46	51.91
A-n38-k5.vrp	11.38	17.96	9.90
A-n39-k5.vrp	38.37	35.60	54.35
A-n39-k6.vrp	24.69	38.71	121.55
A-n44-k6.vrp	299.06	289.01	8.39%
A-n46-k7.vrp	24.58	136.34	1350.17
B-n31-k5.vrp	10.38	17.04	4.69
B-n34-k5.vrp	4.31	3.32	14.47
B-n35-k5.vrp	3.04	3.32	3.86
B-n38-k6.vrp	10.69	7.87	11.66
B-n39-k5.vrp	5.66	3.67	6.46
B-n41-k6.vrp	10.67	24.04	8.21%
B-n43-k6.vrp	23.01	30.36	716.57
B-n44-k7.vrp	29.49	285.97	32.29
B-n45-k5.vrp	14.40	13.22	51.41
B-n50-k7.vrp	14.53	15.74	24.41
E-n23-k3.vrp	0.42	0.42	0.40
E-n30-k3.vrp	6.21	2.79	7.88
E-n33-k4.vrp	2.08	1.59	18.66
P-n45-k5.vrp	19.01	19.87	44.85
P-n50-k7.vrp	514.83	885.135	325.27

Tabella 8: Variazione Tightness

al volume entrano in conflitto con quelle relative al peso, invalidando un maggior numero di soluzioni e prolungando la procedura di ricerca.

5 Osservazioni

Il contributo delle richieste di volume all'aumento della complessità del problema è evidente. Più la componente volume è ingombrante (sia nel rapporto tra domanda totale e domanda servibile, sia – in misura minore – nella non correlazione tra i tipi di richieste) più le istanze diventano di difficile risoluzione. Del resto un risultato di questo tipo potrebbe non essere troppo sorprendente: le richieste di volume non fanno altro che porre un'ulteriore condizione sull'ammissibilità delle rotte. Come evidenziato in [3], nel *CVRP* i vincoli di capacità impongono un legame tra gli instradamenti e la componente di *packing* del problema; nel caso qui esaminato i vincoli

di capacità riguardano due dimensioni "parallele", e riducono ulteriormente l'insieme delle rotte ammissibili (quelle compatibili con entrambe le richieste). La componente di *routing* si interseca quindi non con una ma con due diverse strutture di packing, aggiungendo un ulteriore grado di difficoltà alla risoluzione del problema.

Riferimenti bibliografici

[1] M. Gendreau, A. Hertz, G. Laporte: "A Tabu Search Heuristic for the Vehicle Routing Problem", Management Science, Vol. 40, pp. 1276-1290, (1994)

[2] P. Augerat, J. M. Belenguer, E. Benavent, A. Corberan, D. Naddef: "Separating capacity constraints in the CVRP using tabu Search", European Journal of Operations Research 106, pp. 546–557, (1998)

[3] T. K. Ralphs, L. Kopman, W. R. Pulleyblank, L. E. Trotter Jr.:"On the Capacitated Vehicle Routing Problem", Mathematical Programming, Vol. 94, pp. 343-359, (2003)