TABUROUTE: euristica locale per la risoluzione di VRP

Stefano Mercogliano, Daniele Ottaviano, Francesco Vitale

07/06/2020

Contents

La presente documentazione espone dapprima una breve panoramica sul problema di *Capacitated Vehicle Routing*, al fine di introdurre una delle possibili euristiche adottabili per la risoluzione del problema, data la sua intrattabilità con gli strumenti di programmazione matematica lineare classica. Successivamente viene presentata concretamente un'euristica implementata in linguaggio C++, `TABUROUTE`, in stile prettamente procedurale.

1 Capacitated Vehicle Routing

Il problema ricade nella classe dei problemi decisionali combinatori, notoriamente difficili da risolvere a causa dell'ingente numero di possibili soluzioni ammissibili ritrovabili per il problema.
Una sua possibile descrizione informale è la seguente:

Il problema del Capacitated Vehicle Routing riguarda, data una flotta di veicoli e un insieme di punti di domanda e un deposito, l'instradamento dei veicoli su un certo numero finito di rotte, che si originano tutte dal deposito, al fine di minimizzare una certa funzione di costo, quale ad esempio la distanza totale percorsa calcolata per ogni rotta, rispettando al contempo un insieme di vincoli quali il soddisfacimento della domanda di ogni punto di domanda.

Il motivo per cui questo problema risulta essere particolarmente ostico, come già detto, sta nel possibile numero di soluzioni ammissibili. Definiti l'insieme di tutte le possibili soluzioni $S = \{S_1, S_2, \ldots, S_k\}$, dove $S_i = \{C_1, C_2, \ldots, C_m\}$, viene da sè che il numero di soluzioni risulta essere particolarmente alto, oltre al fatto che il problema di Capacitated Vehicle Routing può essere visto come due sottoproblemi tra loro correlati, che sono il *routing* dei veicoli (assimilabile a un problema TSP), e il *clustering* dei nodi (assimilabile, appunto, a un problema di clustering); i sottoproblemi stessi sono notoriamente difficili da risolvere, e a maggior ragione lo è il problema principale.

1.1 Dati, vincoli e funzione obbiettivo

Esistono tante varianti del problema di Vehicle Routing, per cui fornire un'elencazione esaustiva di tutti i dati, vincoli e possibili funzioni obbiettivo non sarebbe possibile. Tuttavia, a partire dalla formulazione informale precedentemente sviluppata, possiamo sicuramente definire:

- Numero di veicoli m;
- Capacità Q dei veicoli;
- Insieme dei punti vendita $V = \{v_0, v_1, \ldots, v_{n-1}\}$, dove il punto v_0 è definito deposito;

- Domanda d_i associata all'i-esimo punto vendita;
- Un insieme di rotte $C = \{C_1, C_2, \ldots, C_m\}$, dove $C_j = \{v_k \in V - \{v_0\} : v_k \notin C_i, \forall i \neq j\} \cup \{v_0\}$, ossia ogni rotta è costituita da un sottoinsieme dell'insieme di punti vendita V, meno il deposito, dove ogni punto vendita presente nel cluster non è al contempo presente in alcun altro cluster;
- Costo c_{ij} per passare da un punto vendita i a un punto vendita j in una rotta;
- Lunghezza L della rotta, definita come la somma dei costi c_{ij}.

Per ciò che riguarda i vincoli, ci concentreremo sul vincolo che predica la non eccedenza della capacità Q associata al generico veicolo della flotta per la rotta programmata per tale veicolo, e la non eccedenza della lunghezza massima L del percorso svolto dal generico veicolo della flotta, considerando anche, in questo caso, un'informazione aggiuntiva che riguarda il tempo di servizio $\delta(i)$ dell'i-esimo nodo della rotta. Dunque avremo, per ogni rotta C, i seguenti vincoli:

$$\sum_{(i,j)\in C} c_{ij} + \sum_{i \in C} \delta(i) \leq L$$

$$\sum_{i \in C} d_i \leq Q$$

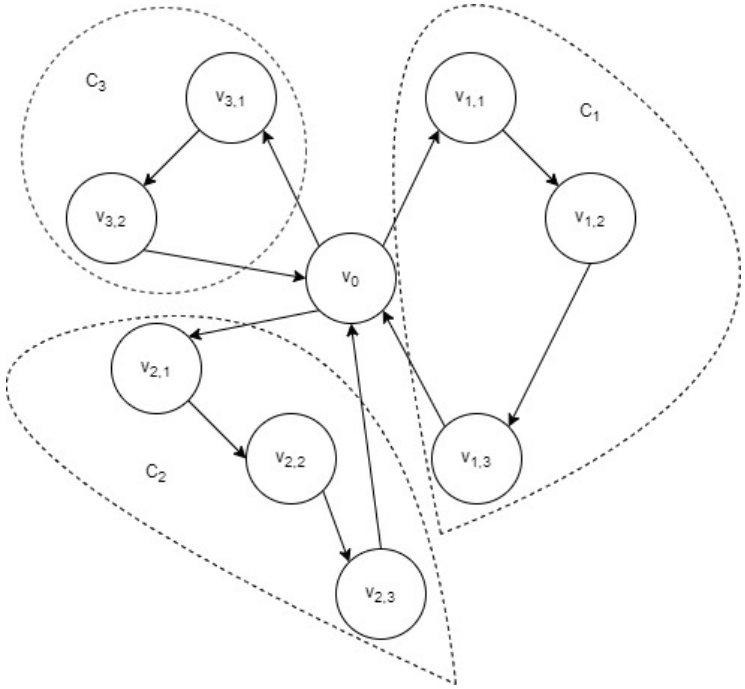
Come funzione obiettivo considereremo la minimizzazione di:

$$f = \sum_{r} \sum_{(i,j)\in C_r} c_{ij}$$

Ossia la minimizzazione della distanza totale percorsa per ogni rotta presente.

1.2 Rappresentazione

Il problema, come si è intuito, può essere naturalmente modellabile tramite l'ausilio di un grafo. Per quanto riguarda l'intero problema, possiamo considerare un grafo $G(V, A)$, dove V è l'insieme dei nodi, ossia dei punti vendita, e A è l'insieme degli archi, ossia dei collegamenti tra i punti vendita. Questa modellazione risulta essere particolarmente agevole soprattutto a livello implementativo: sfruttando strutture dati ad-hoc (matrice di adiacenza, liste di adiacenza) l'implementazione rende piuttosto naturale tutte le operazioni che saranno necessarie per le operazioni dell'algoritmo che svilupperemo.
I singoli cluster possono essere descritti come dei percorsi chiusi (cicli hamiltoniani) che collegano il nodo deposito con tutti gli altri nodi della rotta; da qui si evince come il problema sia scomponibile in un problema TSP e in un problema di clustering: tramite clustering vengono addensati nodi 'simili' in una singola rotta, ossia nodi più vicini in termini di costi di collegamento, e tramite TSP vengono sviluppati dei circuiti hamiltoniani per collegare in un percorso chiuso questi nodi. Graficamente possiamo apprezzare la potenza espressiva dei grafi nella descrizione del problema, tramite la seguente visualizzazione del problema:



Il modello lineare può essere più o meno agevolmente sviluppato tramite gli strumenti di programmazione matematica; tuttavia è opportuno dire che i tempi di esecuzione sono irragionevoli per istanze anche relativamente contenute in dimensione.

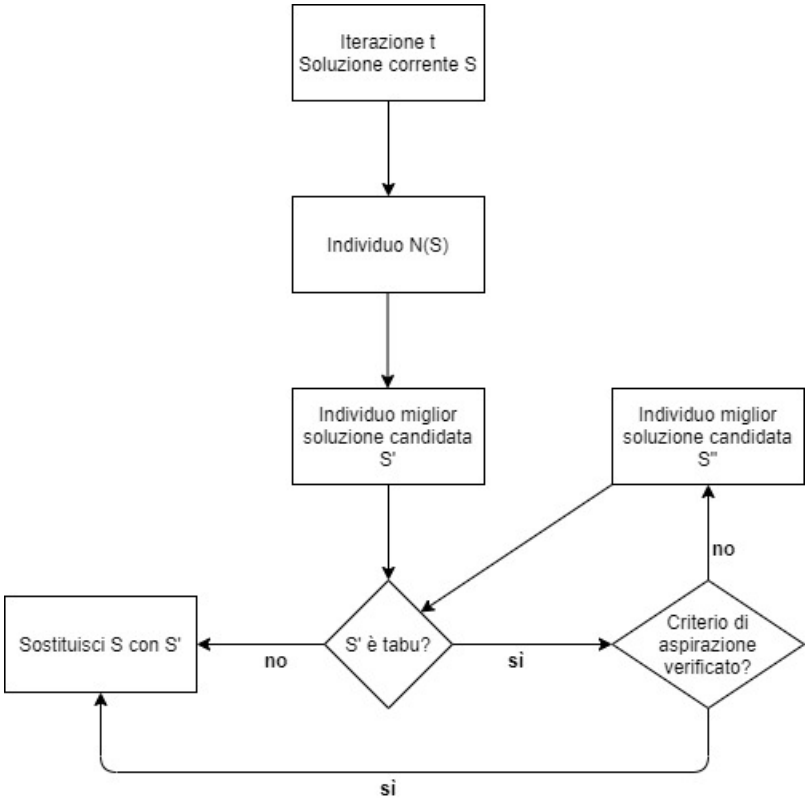
1.3 Euristiche

Com'è noto, le tipologie di euristiche sfruttabili per risolvere problemi computazionalmente intrattabili con metodi esatti sono molte; ci sono le varie classi di algoritmi genetici, tabu-search, algoritmi greedy, ecc. In generale ciò che accomuna tutte le euristiche è l'ausilio di alcuni strumenti per cercare di avvicinarsi alla soluzione ottima; ogni tipologia presenta, oltretutto, un macrocodice che descrive ad alto livello gli step elaborativi che compie.
Tra tutte le euristiche, ci concentriamo sulla tabu-search, in quanto è l'approccio che adotteremo per risolvere il problema di Capacitated Vehicle Routing. L'outline di un generico algoritmo tabu-search è il seguente:

1. Inizializza le strutture dati con la soluzione trovata tramite euristiche costruttive;
2. Costruisci un intorno della soluzione S;
3. Scegli, tra tutte le soluzioni dell'intorno, quella migliore;
4. Verifica il criterio di arresto.

Questo schema dev'essere ovviamente parametrizzato secondo la metodologia scelta per la costruzione dell'intorno (l'applicazione di un'opportuna *mossa*), per il cosiddetto 'criterio di aspirazione' (per bypassare, per una certa iterazione, una mossa classificata precedentemente come tabu) e per il criterio di arresto.

Lo schema generale per la generica iterazione k-esima è il seguente:



Alla base dell'euristica tabu-search c'è il principio di ogni euristica locale, ossia accettare di avere delle soluzioni candidate inizialmente peggiorative, per poi trovare successivamente delle soluzioni che possono arrivare a migliorare la soluzione di partenza: la lista tabu serve proprio a questo, ossia per scartare le soluzioni precedentemente prese (come la soluzione iniziale), peggiorare, e poi eventualmente migliorare il valore di funzione obbiettivo.

Oltre all'applicazione dello schema tabu-search, consideriamo anche l'ausilio di una tecnica RFCS per la costruzione di un insieme di rotte iniziali per la flotta di veicoli. L'euristica più banale riguarda la costruzione di un ciclo hamiltoniano per tutti i nodi dell'insieme V (a meno del nodo v_0 deposito); costruito questo ciclo hamiltoniano, si considera anche il nodo deposito, e si procede a costruire le m rotte nel rispetto dei vincoli di lunghezza e capacità massima.

Consideriamo anche le tecniche che vengono adoperate per migliorare i

risultati dell'euristica. Le due tecniche principali sono:

- Intensificazione;
- Diversificazione.

Per quanto concerne l'intensificazione, la tecnica prevede di trovare degli intorni più ampi di una soluzione. Per ciò che concerne la diversificazione, invece, abbiamo il cosiddetto *approccio multi-start*, per cui si applica l'euristica migliorativa per soluzioni iniziali diverse.

2 TABUROUTE

Definiamo TABUROUTE l'euristica, basata sui principi della tipologia di euristiche tabu-search, per la risoluzione del problema precedentemente esposto di vehicle routing. L'obiettivo, dunque, riguarda la minimizzazione della seguente funzione obiettivo:

$$F = \sum_{r} \sum_{(i,j) \in C_r} c_{ij}$$

L'approccio seguito riguarda, tuttavia, l'aggiunta di alcuni elementi peculiari, tra cui l'introduzione dei concetti di *soluzione ammissibile*, *soluzione inammissibile*, *penalizzazione delle soluzioni*.
L'idea è quella di scaturire, a partire da una soluzione S (la soluzione iniziale alla prima iterazione, modificata dalle successive 'soluzioni candidate' nelle iterazioni a venire), un insieme di soluzioni $S' = \{S'_1, S'_2, \cdots, S'_q\}$, che definiremo insieme delle *soluzioni temporanee*. A partire da queste soluzioni temporanee, si determina quali tra queste sono tabu: per quelle tabu, si applica il precedentemente citato *criterio di aspirazione*, per bypassare il loro fatto di essere tabu. Una volta discriminate tutte le soluzioni temporanee tabu si determina quale tra queste è la migliore, la cosiddetta *soluzione candidata*, che verrà comparata con la soluzione attualmente mantenuta (successivamente verrà scandito meglio cosa si intende per 'soluzione mantenuta'), ed eventualmente la sostituirà.

L'algoritmo tiene conto, nel corso dell'esecuzione di una tabu-search, della *migliore soluzione finora ritrovata*; ogni qualvolta la soluzione mantenuta

viene sostituita dalla soluzione candidata alla k-esima iterazione, essa viene anche comparata con la migliore soluzione fino a quel momento ritrovata. Alla fine dell'esecuzione di una tabu-search (TABUROUTE ne farà diverse durante il corso della sua esecuzione) verrà restituita la soluzione migliore tra tutte quelle trovate nel corso dell'esecuzione della procedura.

2.1 Nomenclatura

TABUROUTE parametrizza una generica euristica di ricerca locale basata su tabu-search facendo uso di componenti specifici. Consideriamo, dunque, la nomenclatura necessaria che accompagnerà l'intera trattazione:

- `TABUROUTE`: il nome assegnato all'algoritmo che si basa sulla chiamata ricorrente di un algoritmo di tabu_search per la ricerca della soluzione a un problema di capacitated vehicle routing;
- `SEARCH`: il nome assegnato all'algoritmo che implementa una tabu_search;
- $G(V, A)$: il grafo che descrive la configurazione di nodi e archi del problema, dove V è l'insieme dei punti vendita comprensivo di deposito, e A è l'insieme degli archi che collegano tutti i vari nodi;
- m: numero di veicoli disponibili;
- $R = \{R_1, R_2, \ldots, R_m\}$: insieme di rotte, dove $R_i = \{v_{i1}, v_{i2}, \ldots, v_{ik}\}$;
- Q, L, δ_i: dati del problema, essi esprimono la *capacità massima di un veicolo*, la *lunghezza massima di un percorso*, il *tempo di servizio* dell'i-esimo nodo;
- F_1 e F_2: un concetto importante portato avanti da TABUROUTE è quello della presenza di due funzioni obbiettivo, e di conseguenza di soluzioni ammissibili e inammissibili. Quando una soluzione è inammissibile non viene a priori scartata, ma ad essa viene associata una 'penalità'; questa penalità consente concettualmente di peggiorare una soluzione, e quindi implementare il concetto alla base di un'euristica locale: il peggioramento iniziale della soluzione attuale nella speranza che si arrivi a determinare una soluzione migliore col passare delle iterazioni.

Detto ciò, le funzioni obbiettivo hanno le seguenti espressioni:

$$F_1 = \sum_r \sum_{(i,j)\in C_r} c_{ij}$$

$$F_2(S) = F_1(S) + \alpha \sum_r \max\left[0, \left(\sum_{v_i \in R_r} q_i\right) - Q\right] + \beta \sum_r \max\left[0, \left(\sum_{v_i, v_j \in R_r} c_{ij} + \sum_{v_i \in R_r} \delta_i\right) - L\right]$$

- $S^*, \tilde{S}^*, F_1^*, F_2^*$: come già accennato, l'algoritmo presenta i concetti di *migliore soluzione ammissibile fino a quel momento calcolata* S^*, a cui è di conseguenza associato il miglior valore di funzione obbiettivo F_1^*, e di *migliore soluzione fino a quel momento calcolata*, ossia $\tilde{S}^*$, che non è in generale ammissibile, ma possiede il minor valore di F_2 calcolato per ogni soluzione ritrovata dalla tabu-search fino a un dato punto del flusso d'esecuzione;

- α, β: i parametri di penalizzazione, assegnati nel caso la capacità o la lunghezza di uno dei cluster della soluzione considerata vengano violati;

- Ψ: nel corso dell'esecuzione della tabu_search, sarà necessario campionare un numero di vertici dall'insieme W passato in ingresso, in maniera casuale. L'insieme di nodi campionati viene qui nominato Ψ. La cardinaltià di questo insieme è indicata con q;

- W: sottoinsieme dell'insieme di nodi V gestito dalla tabu_search;

- λ: numero di soluzioni di partenza considerate nell'approccio multistart di TABUROUTE;

- *Soluzione temporanea*: soluzione derivata dall'applicazione di una singola mossa a un singolo nodo di Ψ;

- *Mossa*: componente fondamentale di tutti gli algoritmi basati su tabu_search, nel nostro caso conterrà informazioni chiave per poter costruire le *Soluzioni temporanee*;

- *Miglior soluzione temporanea (o Soluzione candidata)*: soluzione derivata dal campionamento della migliore soluzione temporanea tra tutte quelle calcolate per il singolo nodo di Ψ

- *Soluzione mantenuta*: è il nome assegnato alla soluzione mantenuta a una generica iterazione da parte di `SEARCH`; essa verrà comparata con la miglior soluzione **temporanea** di una iterazione (soluzione candidata): se in quella data iterazione la miglior soluzione candidata è migliore della soluzione mantenuta, allora avverrà la sostituzione di quest'ultima con quell'altra;

- *Lista tabu*: componente fondamentale di tutti gli algoritmi basati su tabu_search, essa terrà conto di tutte le mosse tabu finora riscontrate;

- TTL: tempo di permanenza di una mossa tabu all'interno della lista tabu;

- $\theta_{min}, \theta_{max}$: parametri in ingresso alla tabu_search, servono a stabilire il TTL di una mossa;

- p_1: dimensione del 'vicinato' di un nodo; esso darà origine a un sottoinsieme dell'insieme dei nodi V;

- *Numero di spostamenti*: nella valutazione di una soluzione candidata, un vertice viene spostato da una rotta a un'altra. È importante tenere conto del numero di volte che un vertice viene spostato da una rotta a un'altra al fine di determinare la soluzione candidata;

- f_v: parametro correlato al numero di spostamenti di un vertice;

- $g, h, nmax$: parametri d'ingresso alla tabu_search; variando nmax è possibile concentrarsi sull'intensificazione di una soluzione;

- F_{temp}: valore di funzione obbiettivo temporaneo gestito dalla `SEARCH`.

2.2 Gestione delle penalità

È doveroso aprire una parentesi sulla *gestione delle penalità* che segue l'algoritmo. Gestendo le penalità, l'algoritmo riesce a modulare le mosse in maniera tale che:

- Le mosse che peggiorano eccessivamente il valore di $F_2(S)$, dove S è la soluzione candidata calcolata nella precedente iterazione di `SEARCH`, è meno probabile vadano a formare una soluzione temporanea che possa sostituire la precedente soluzione candidata. Ciò viene assicurato grazie a una gestione peculiare di F_{temp} nel corso della valutazione delle mosse candidate. Questo valore di F_{temp} viene gestito secondo la seguente logica nel caso in cui venga eseguita una mossa:

$$\begin{cases} F_{temp}(S') = F_2(S') & \text{se } F_2(S') < F_2(S) \\ F_{temp}(S') = F_2(S') + \Delta_{max}\sqrt{mg}f_v & \text{altrimenti} \end{cases}$$

  dove S' è la soluzione temporanea derivata dalla mossa svolta.
  Il motivo per cui questa gestione penalizza le mosse più svantaggiose, sta nel fatto che F_{temp} verrà usato per determinare la prossima soluzione candidata.

- I valori di α e β partecipano alla determinazione della F_2 delle successive soluzioni temporanee nel corso dell'algoritmo. Seppure inizializzati a 1, essi verranno gestiti per fare in modo che se le precedenti h soluzioni migliori temporanee finora calcolate hanno ecceduto sempre la capacità massima del veicolo, oppure la lunghezza massima del percorso in cui sono immessi, esse vedranno un aumento. Questo aumento porterà di conseguenza a scartare più spesso le 'migliori soluzioni temporanee', ossia eviterà che la soluzione mantenuta venga sostituita a queste ultime, e di conseguenza consentirà, di fatto, la convergenza dell'algoritmo.
  La gestione dei valori di α e β è come segue:
  a valle del trascorso di h iterazioni, i due parametri devono essere aggiornati. I casi sono essenzialmente quattro:

$$\begin{cases} \alpha = \frac{\alpha}{2}, \beta = \frac{\beta}{2} & \text{se non eccedute capacità e lunghezza} \\ \alpha = \frac{\alpha}{2}, \beta = 2\beta & \text{se ecceduta lunghezza ma non capacità} \\ \alpha = 2\alpha, \beta = \frac{\beta}{2} & \text{se ecceduta capacità ma non lunghezza} \\ \alpha = 2\alpha, \beta = 2\beta & \text{se eccedute capacità e lunghezza} \end{cases}$$

- Come già detto, quando viene identificata la migliore soluzione temporanea, non è detto che questa andrà a sostituire la soluzione mantenuta. Perchè ciò avvenga, è necessario che alcune condizioni siano rispettate. In particolare, nominata come $\hat{S}$ la soluzione candidata e S la soluzione mantenuta:

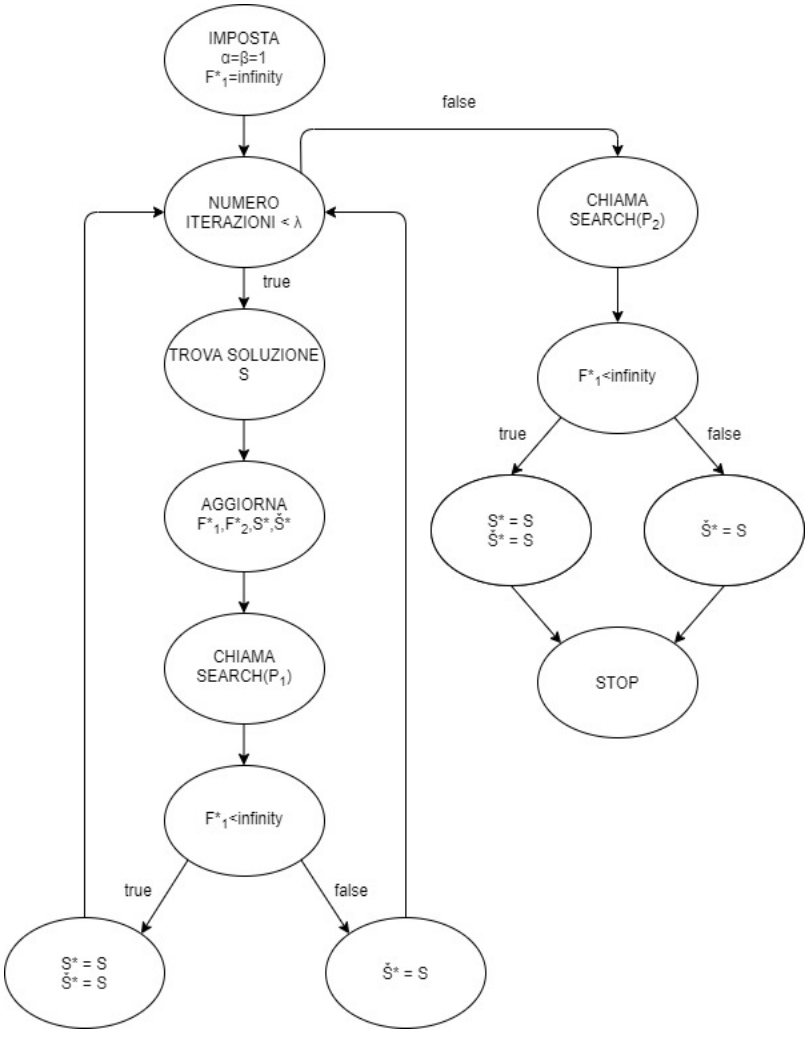
  $$S \neq \hat{S} \quad \text{Se } F_2(\hat{S}) > F_2(S) \text{ e } S \text{ è ammissibile}$$

  Tuttavia possiamo riconoscere che l'andamento peggiorativo ci sarà, e non si incorrerà in cicli; basterà infatti che una delle migliori soluzioni temporanee, fattibile o meno, abbia un valore di F_2 inferiore a quella della soluzione di partenza, perchè la soluzione iniziale sia sostituita. Oltretutto, abbiamo che la mossa appena svolta sarà tabu, e pertanto non potranno accadere dei cicli.

2.3 Struttura algoritmica

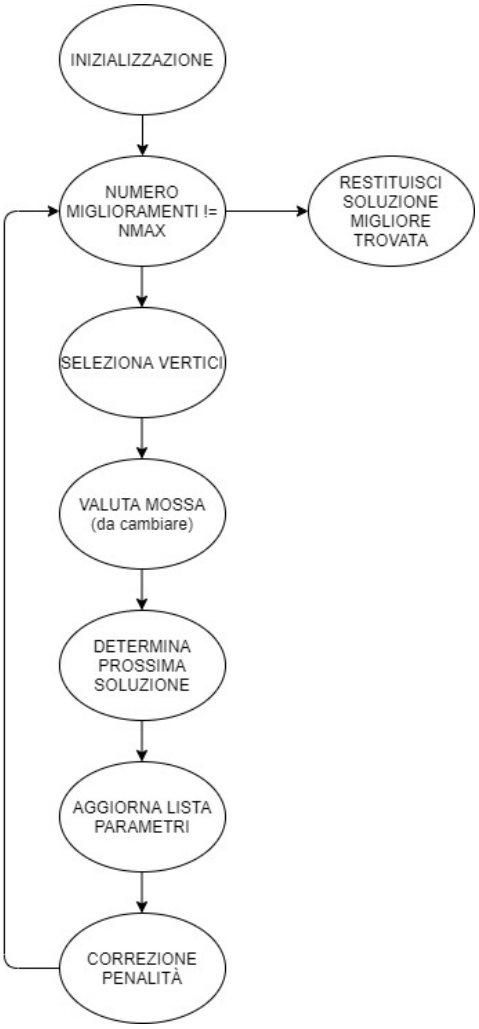
La struttura algoritmica è pensata per un'implementazione prettamente procedurale, e quindi organizzata in subroutines

TABUROUTE Il primo grafo che presentiamo è quello relativo alla procedura TABUROUTE:



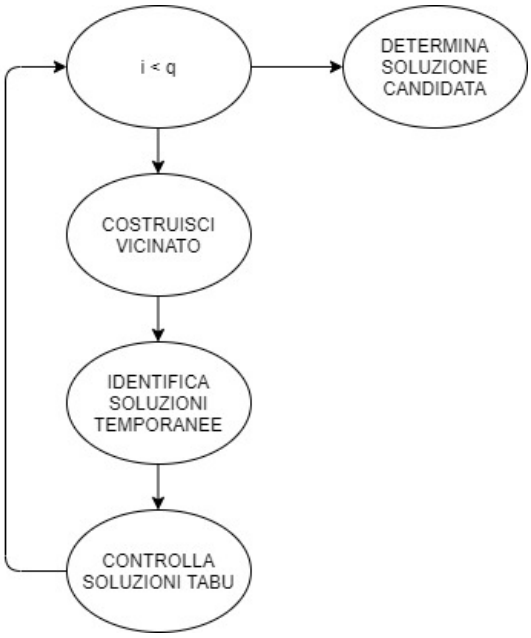
CFG di `TABUROUTE`

SEARCH Dato che la funzione `SEARCH` viene chiamata più volte durante il corso dell'algoritmo, viene qua presentata anche la sua struttura algoritmica:

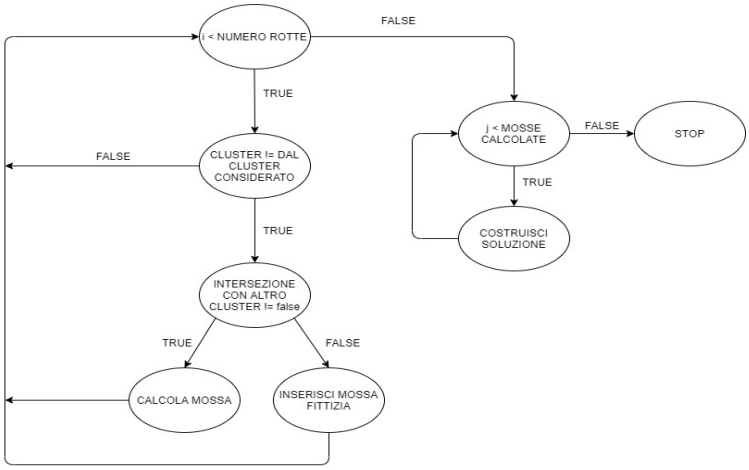


CFG di `SEARCH`

Alcune delle procedure richiamate dalla SEARCH sono meritevoli di approfondimento poichè notevolmente più complesse. La più complessa tra tutte risulta nella fattispecie essere VALUTA MOSSA, di cui presentiamo il grafo, insieme a una sua sottoprocedura, ossia IDENTIFICA SOLUZIONI TEMPORANEE:



CFG di `VALUTA_MOSSA`



CFG di `IDENTIFICA_SOLUZIONI_TEMPORANEE`

Come si vede, `VALUTA_MOSSA` consente, alla fine, di determinare la soluzione candidata, poichè per ogni nodo dell'insieme Ψ, vengono determinate un insieme di *soluzioni temporanee* (per cui va anche controllato il *criterio di aspirazione*), a cui segue una discriminazione per la soluzione temporanea migliore (ossia la cosiddetta soluzione candidata).
La funzione `IDENTIFICA_SOLUZIONI_TEMPORANEE` risulta essere piuttosto complicata poichè bisogna tenere traccia di diverse cose, tra cui:

- Il nodo correntemente scelto e la rispettiva rotta di appartenenza;
- Il cluster (rotta) attualmente considerato per effettuare la mossa (bisogna oltremodo evitare di considerare il cluster in cui è attualmente presente il nodo);

- La gestione della consistenza tra la rotta di partenza del nodo, e la rotta destinazione della mossa: ciò porta ad avere una gestione degli indici piuttosto complicata.
- Una volta costruite le mosse, derivare per queste ultime le soluzioni temporanee.

2.3.1 Strutture dati utilizzate

Data la complessità dell'algoritmo e la rappresentazione adoperata per il problema, è necessario esporre di seguito le strutture dati adoperate, che sono tutte contenute nel file di intestazione `HGL.h`. Prima di esporle, tuttavia, poniamo l'attenzione al fatto che per rappresentare l'intero grafo, abbiamo sfruttato una matrice di adiacenza, ossia una matrice allocata dinamicamente che, per ogni posizione, manterrà il costo per arrivare da un nodo a un altro (dunque una classica rappresentazione per i grafi), e oltretutto abbiamo sfruttato il tipo `vector` della libreria di C++ per tenere traccia degli indici dei nodi (che sia l'insieme V dei nodi, che l'insieme W, che l'insieme Ψ, che l'insieme dei vicini). Una piccola eccezione va citata per la gestione dei percorsi interni alle rotte, che farà uso del tipo `list`. Detto ciò, cominciamo:

- `Nodo`

```
struct Nodo {
   float x;
   float y;

   int indice;
   int domanda;
   int tempo;
   int spostamenti;
};
```

I primi due campi sono le coordinate x e y del nodo all'interno del grafo, sono necessarie per un eventuale rendering grafico.
Il campo `indice` è servito per numerare il nodo, ma per questioni pratiche, non per ordinamento; difatti si è tenuto traccia del nodo deposito assegnandogli indice nullo.
I campi `domanda` e `tempo` servono a tenere traccia dei due dati insiti

del nodo stesso, ossia, appunto, la domanda, e il tempo di servizio.
Il campo `spostamenti` tiene traccia degli spostamenti del nodo.

- `Rotta`

```
struct Rotta {
   list<Nodo> percorso
   float lunghezza;
   float tempo_servizio;
   float domanda_totale;
   int indice;
};
```

La rotta ha una lista linkata di *nodi*; la scelta della lista linkata è motivata dalla facilità con cui i nodi possono essere inseriti arbitrariamente all'interno delle posizioni della lista, data la peculiare gestione di quest'ultima. Specifichiamo che l'ordine con cui i nodi sono immessi è determinato dalla lista linkata, ma internamente i nodi avranno degli indici che consentiranno la loro identificazione univoca all'interno del vettore V dei nodi di partenza.
I campi `lunghezza`, `tempo_servizio`, `domanda_totale` sono serviti a modellare la lunghezza totale della rotta, il tempo di servizio cumulato per ogni nodo del percorso, e la domanda totale dei nodi nel percorso.
Il campo `indice` ci consente, nel contesto di una soluzione, di poterci riferire agevolmente a una delle rotte di quest'ultima nell'ambito delle numerose operazioni che necessitano di tracciare univocamente la rotta.

- `Mossa`

```
struct Mossa {
   int nodo;
   int rotta_origine;
   int rotta_destinazione;
   int posizione_in_origine;
   int posizione_in_destinazione;
   float costo;
};
```

La mossa è una delle strutture dati più importanti. Come si vede, essa mantiene un indice chiamato `nodo`; esso sarà utile per riferirsi

all'indice del nodo all'interno di uno dei `vector` che verranno sfruttati nel corso di `SEARCH`. Oltre a ciò, abbiamo quattro interessanti campi, ossia `rotta_origine`, `rotta_destinazione`, `posizione_in_origine`, `posizione_in_destinazione`. Come detto, le rotte sono numerate, nel contesto di una soluzione; i primi due campi elencati servono a riferirsi alla rotta origine del nodo, alla rotta destinazione scelta per la mossa. Oltretutto, bisogna tenere conto della posizione nella rotta origine, e poi della posizione nella rotta destinazione.

- `Soluzione`

```
struct Soluzione {
   float F1;
   float F2;
   float F_temp;
   float penalita_capacita;
   float penalita_tempo;
   bool ammissibile;
   vector<Rotta*> cluster;
   Mossa m;
};
```

La soluzione è ovviamente il punto focale dell'algoritmo. Oltre alle varie funzioni obbiettivo già menzionate (`F1`, `F2`, `F_temp`), e a un paio di parametri di debug (`penalita_capacita`, `penalita_tempo`), abbiamo un parametro booleano `ammissibile` che rende più facile stabilire se una soluzione sia ammissibile o meno, e poi un `vector` di rotte: questa voce è importante, e i puntatori sono gisutificati dal fatto che ogni soluzione manterrà il proprio vettore di rotte, con i propri percorsi. La gestione di questo vettore dunque è di fondamentale importanza, poichè dev'essere gestito per ogni soluzione presente nel sistema.
In ultimo, abbiamo la mossa `m`, che serve a stabilire qual è la mossa che ha originato tale soluzione: questo campo è particolarmente utile quando bisogna stabilire se una soluzione temporanea dev'essere scartata poichè originata da una mossa tabu, o meno.

- `Tabu`

```
struct Tabu {
   Mossa m;
   int TTL;
};
```

  La struttura dati che servirà a tenere traccia di una certa mossa `m` e del suo `TTL` all'interno della lista tabu.

- `Tabulist`

```
struct Tabulist {
   vector<Tabu> mossa_tabu;
};
```

  La struttura dati che wrappa un vettore di `Tabu` e che rappresenta la nostra lista tabu.

2.3.2 SEARCH

La SEARCH, come già detto, implementa una logica tabu_search. È una funzione piuttosto complessa, dunque schematizziamo inizialmente i suoi parametri di ingresso:

$$S \quad W \quad q \quad p_1 \quad p_2 \quad \theta_{min} \quad \theta_{max} \quad g \quad h \quad n_{max}$$

e la sua unica uscita:

$$S$$

I parametri d'ingresso li abbiamo già definiti in precedenza, l'uscita S, data la sua ambiguità nel caso in cui dicessimo unicamente che è la miglior soluzione, cerchiamo di specificarla meglio. In ingresso vediamo che c'è una soluzione iniziale S; questa soluzione iniziale viene gestita durante il corso dell'algoritmo, e quindi assumerà il senso della 'soluzione mantenuta', ossia quella che verrà comparata con le varie soluzioni candidate che vengono scelte a ogni iterazione. L'algoritmo gestisce anche, ricordiamo, S^*. Alla fine dell'esecuzione dell'algoritmo, la soluzione mantenuta si predisporrà ad essere il parametro d'uscita, venendo pertanto sostituita da S^*.

Detto ciò, consideriamo il corpo della procedura, che va sotto al nome di tabu_search:

```
void tabu_search(Soluzione& S, vector<Nodo>& W, int q, int p1, int p2,
int theta_min, int theta_max, int g, int h, int n_max) {

//STRUTTURA DELL'ALGORITMO DI TABU SEARCH

//Inizializzazione delle variabili locali
int t = 1;
Tabulist listaTabu;
vector<Nodo> PSI;

// Contatori per gestire i coefficienti di penalita
int L_ammissibile=0;
int L_inammissibile=0;
int C_ammissibile=0;
int C_inammissibile=0;
int n_miglioramenti=0;
```

```
Soluzione Soluzione_candidata;
vector<Soluzione> Soluzioni_temporanee;
Soluzione Soluzione_migliore_fattibile;
Soluzione Soluzione_migliore;
float delta_max = 0.0;
float f_v = 0.0;
float F2_candidata_precedente = 0;

//INIZIALIZZAZIONE DELLE SOLUZIONI
inizializza_soluzione(Soluzione_candidata,S.cluster.size());
inizializza_soluzione(Soluzione_migliore_fattibile, S.cluster.size());
inizializza_soluzione(Soluzione_migliore, S.cluster.size());

while (n_miglioramenti != n_max && t < STOP) {

   t == 1 ? F2_candidata_precedente = 0 : F2_candidata_precedente =
   Soluzione_candidata.F2;

   seleziona_vertici(PSI, q, W, W.size());

   assegna_soluzione(Soluzione_candidata, valutazione_mossa(PSI, q, p1,
   Soluzioni_temporanee, listaTabu, S, Soluzione_migliore_fattibile,
   Soluzione_migliore, delta_max, f_v));

   for (int i = 0; i < Soluzioni_temporanee.size(); i++)
      distruggi_soluzione(Soluzioni_temporanee[i]);
   Soluzioni_temporanee.clear();

   determina_prossima_soluzione(Soluzione_candidata, S, L_ammissibile,
   L_inammissibile, C_ammissibile, C_inammissibile);

   aggiorna_lista_parametri(Soluzione_candidata, listaTabu,
   Soluzione_migliore_fattibile, Soluzione_migliore, delta_max, f_v, t,
   n_miglioramenti, theta_min, theta_max, F2_candidata_precedente);

   correzione_penalita(L_ammissibile, L_inammissibile, C_ammissibile,
   C_inammissibile, h);
}

//ASSEGNA SOLUZIONE
```

```
print_soluzione(Soluzione_migliore);
assegna_soluzione(S,Soluzione_migliore_fattibile);

//DISTRUZIONE DELLE SOLUZIONI

distruggi_soluzione(Soluzione_migliore_fattibile);
distruggi_soluzione(Soluzione_migliore);
distruggi_soluzione(Soluzione_candidata);
}
```

L'algoritmo è già opportunamente commentato, tuttavia è giusto puntualizzare che la cosiddetta 'soluzione mantenuta' è la soluzione S che viene mandata in ingresso e manipolata durante tutto il corso dell'algoritmo. Insieme ad essa, abbiamo `Soluzione_candidata` che è per l'appunto la soluzione candidata derivata dal ponderamento di tutte le `soluzioni_temporanee` nel contesto di una iterazione. Infine, le soluzioni $\tilde{S}^*$ e S^* sono rispettivamente `Soluzione_migliore` e `Soluzione_migliore_fattibile`. Per il resto è opportuno dire che `F2_candidata_precedente` servirà per la gestione di Δ_{max}.

- La prima cosa interessante da notare è `inizializza_soluzione()`, che accetta in ingresso un argomento di tipo Soluzione, e uno che specifica la dimensione delle rotte; ciò è dovuto al fatto che le rotte, essendo allocate dinamicamente, vanno opportunamente inizializzate.
- Dal costrutto `while` in poi abbiamo essenzialmente l'algoritmo descritto precedentemente tramite CFG.
- Al di là di gestioni e controlli utili a mantenere coerenza con i parametri gestiti internamente alla sottofunzioni, abbiamo `seleziona_vertici` che consente di costruire Ψ a partire dalla dimensione q e dall'insieme W.
- Successivamente vediamo il primo sfruttamento di una funzione giustificata dalla presenza di rotte allocate dinamicamente: `assegna_soluzione`; questa funzione è necessaria poichè per le due soluzioni coinvolte, se si facesse una semplice assegnazione, il vettore di rotte interno sarebbe soggetto a una *shallow copy*, ossia avremmo due vettori di puntatori che puntano allo stesso riferimento (puntatori diversi, indirizzi di memoria puntati uguali); è dunque necessario sfruttare una copia ad-hoc, wrappata da `assegna_soluzione`.

- Abbiamo dunque che la `Soluzione_candidata` per la corrente iterazione viene sostituita all'uscita di `valutazione_mossa`: `valutazione_mossa` incapsula tutte le operazioni necessarie a determinare, dato l'insieme di nodi Ψ in ingresso, la nuova soluzione candidata per la corrente iterazione (illustreremo successivamente questa sottofunzione in dettaglio).

- Successivamente vediamo la distruzione delle `Soluzioni_temporanee`, in quanto dobbiamo deallocare questo vettore (verrà riutilizzato nella successiva iterazione; per non creare memory leaks bisogna dapprima deallocare le rotte).

- A questo punto abbiamo la funzione `determina_prossima_soluzione`. Questa soluzione si preoccupa di comparare la soluzione candidata con la soluzione attualmente mantenuta (per intendersi, verificare se l'assegnazione $S = \hat{S}$ venga effettuata oppure no); insieme a ciò, la gestione delle variabili locali per la gestione delle penalità.

- Abbiamo poi `aggiorna_lista_parametri`. La soluzione si occupa di aggiornare tutti i parametri, meno che i coefficienti di penalità, coinvolti nell'esecuzione dell'algoritmo; poniamo particolare interesse alla gestione già più volte citata dell'aggiornamento di `Soluzione_migliore_fattibile` e `Soluzione_migliore`. In questo caso l'aggiornamento è gestito rispetto alla `Soluzione_candidata` piuttosto che alla soluzione mantenuta, tuttavia ciò non incide sulla correttezza dell'algoritmo.

- Alla fine del ciclo, abbiamo `correzione_penalita`, che si occupa di aggiornare i termini di penalità.

seleziona_vertici Per ciò che concerne questa funzione, essa è piuttosto semplice: seleziona casualmente dei nodi dall'insieme W e li inserisce allinterno di Ψ.

```
void seleziona_vertici(vector<Nodo>& PSI, int q, vector<Nodo> W,
int dim) {
   PSI.clear();

   for (int i = 0; i < q; i++) {
      int j = (int)(rand() % (dim-i));
      PSI.push_back(W[j]);
      W.erase(W.begin() + j);
}
```

valutazione_mossa Al di là della lunghezza del codice, questa è decisamente la funzione più articolata dell'algoritmo, e necessita quindi di un'attenta analisi. Consideriamo innanzitutto il suo codice per poi procedere a commentarlo:

```
Soluzione valutazione_mossa(const vector<Nodo>& PSI, int q,int p1,
vector<Soluzione>& Soluzioni_temporanee, Tabulist& listaTabu,
Soluzione & S, Soluzione& Soluzione_migliore_fattibile,
Soluzione& Soluzione_migliore, const float& delta_max, const int& f_v)
{
   vector<int> N;

   for (int i = 0; i < q; i++) {
      costruzione_vicinato(PSI[i], N, p1);
      identifica_soluzioni_temporanee(N, PSI[i].indice, listaTabu,S,
      Soluzioni_temporanee, i);
      N.clear();
   }
   controlla_soluzioni_tabu(Soluzioni_temporanee, listaTabu, S,
   Soluzione_migliore_fattibile, Soluzione_migliore, delta_max, f_v);
   if (Soluzioni_temporanee.size() > 0)
      return determina_soluzione_candidata(Soluzioni_temporanee);
   else return S;
}
```

Innanzitutto, così come già detto, viene considerato un vettore di vicini N; per ogni nodo estratto dall'insieme Ψ, viene popolato il vettore di vicini N attraverso la funzione `costruzione_vicinato`. Questa funzione, il cui codice è:

```
void costruzione_vicinato(const Nodo& v, vector<int>& N, const int& p1)
{
   vector<float> adjVector;
   for (int k = 0; k < dim; k++) {
      adjVector.push_back(adjMatrix[v.indice][k]);
   }
   for (int j = 0; j < p1; j++) {
      int min = minimo_indice(adjVector, v.indice);
      N.push_back(min);
      adjVector[min] = FLOAT_INF;
   }
}
```

fa uso della matrice di adiacenza; infatti, viene campionata la i-esima riga corrispondente all'indice del nodo in esame, e popola il vettore dei vicini N campionando i primi p_1 nodi più vicini in termini di distanza.

La successiva funzione, ossia `identifica_soluzioni_temporanee` fa uso del vettore dei vicini. Il codice è:

```
void identifica_soluzioni_temporanee(const vector<int>& N,
const int& indice_nodo, const Tabulist listaTabu, Soluzione & S,
vector<Soluzione>& Soluzioni_temporanee, const int & indice_q) {

   int i = 0;
   int indice_intersezione;
   float costo_min = FLOAT_INF;
   vector<Mossa> temp;
   while (i < S.cluster.size()) {
      if (posizione_nodo(*S.cluster[i], V[indice_nodo]) == -1) {
         if (intersezione(N, S.cluster[i], indice_intersezione) ||
         S.cluster[i]->percorso.size() == 1) {
            temp.push_back(
            calcola_mossa(*S.cluster[rotta_del_nodo(S, V[indice_nodo])],
            *S.cluster[i], indice_nodo, indice_intersezione));
         }
         else {
            Mossa NoMossa;
            NoMossa.costo = FLOAT_INF;
            temp.push_back(NoMossa);
         }
      }
      i++;
   }
   for (int i = 0; i < temp.size(); i++)
      if (temp[i].costo - FLOAT_INF < 0) {
         costruisci_soluzione(Soluzioni_temporanee, temp[i], S);
      }
}
```

Questa funzione consente di ottenere il vettore di soluzioni temporanee. Innanzitutto abbiamo che il ciclo viene ripetuto per ogni cluster della soluzione mantenuta `S`. A questo punto bisogna verificare se il nodo estratto da Ψ risulta essere nel cluster correntemente analizzato, e ciò lo facciamo tramite la funzione `posizione_nodo`. Successivamente, se il cluster non ha il nodo in esame, se ne calcola la *intersezione*, tramite la funzione `intersezione`, con il vettore dei vicini. Se è presente almeno un vicino nel cluster con cui

è effettuata l'intersezione, viene salvata la posizione del vicino (può anche succedere che il cluster non abbia nodi, e in tal caso si procede lo stesso a calcolare la mossa per lo scambio).
A questo punto viene richiamata una funzione importante, ossia `calcola_mossa`, di cui presentiamo anche il codice:

```
Mossa calcola_mossa(Rotta Rr, Rotta Rs, const int& indice_origine,
const int& indice_destinazione) {

   Mossa mossa_temporanea;

   mossa_temporanea.posizione_in_origine =
   posizione_nodo(Rr, V[indice_origine]);

   mossa_temporanea.nodo = V[indice_origine].indice;
   mossa_temporanea.rotta_origine = Rr.indice;
   mossa_temporanea.rotta_destinazione = Rs.indice;


   if (Rs.percorso.size() > 1)
      mossa_temporanea.posizione_in_destinazione = indice_destinazione;
   else
      mossa_temporanea.posizione_in_destinazione = 1;

   return mossa_temporanea;
}
```

La funzione prende in ingresso due rotte, ossia la rotta origine e la rotta destinazione, e gli altri due ingressi vanno interpretati in maniera attenta. `indice_origine` sta a identificare l'indice riferito al vettore V dei nodi, ma non alla posizione del nodo nella rotta origine, che va successivamente calcolato tramite la funzione `posizione_nodo`, invece `indice_destinazione` sta a indicare proprio la posizione in cui viene spostato il nodo nella rotta destinazione. A parte ciò, il resto dei campi della mossa viene gestito in maniera chiara, con l'accortenza, ovviamente, di verificare se il cluster destinazione sia vuoto o meno.
Un'altra cosa da specificare bene, è che il primo parametro di `calcola_mossa` viene ritrovato grazie alla funzione `rotta_del_nodo`, che a partire dalla soluzione mantenuta S e dall'indice del nodo, riesce a ricavare la rotta di partenza.

A questo punto viene richiamata `costruisci_soluzione`, che ci consentirà di calcolare, a partire dalle mosse, il vettore `Soluzioni_temporanee`:

```
void costruisci_soluzione(vector<Soluzione>& Soluzioni_temporanee,
const Mossa & mossa, const Soluzione & S ) {
   Soluzione S_temp ;
   std::list<Nodo>::iterator it_erase, it_insert;


   inizializza_soluzione(S_temp, S.cluster.size());
   assegna_soluzione(S_temp, S);
   S_temp.m = mossa;


   it_erase = S_temp.cluster[mossa.rotta_origine]->percorso.begin();
   it_insert = S_temp.cluster[mossa.rotta_destinazione]->
   percorso.begin();

   advance(it_erase, mossa.posizione_in_origine);
   advance(it_insert, mossa.posizione_in_destinazione);

   S_temp.cluster[mossa.rotta_origine]->percorso.erase(it_erase);

   if (S_temp.cluster[mossa.rotta_destinazione]->percorso.size() > 1)
      S_temp.cluster[mossa.rotta_destinazione]->percorso.insert(
      it_insert, V[mossa.nodo]);
   else {
      S_temp.cluster[mossa.rotta_destinazione]->
      percorso.push_back(V[mossa.nodo]);
      S_temp.m.posizione_in_destinazione = posizione_nodo(
      *S_temp.cluster[mossa.rotta_destinazione], V[mossa.nodo]);
   }

   calcola_lunghezza_percorso(*S_temp.cluster[mossa.rotta_origine]);
   calcola_domanda_percorso(*S_temp.cluster[mossa.rotta_origine]);
   calcola_tempo_percorso(*S_temp.cluster[mossa.rotta_origine]);

   calcola_lunghezza_percorso(
   *S_temp.cluster[mossa.rotta_destinazione]);
   calcola_domanda_percorso(*S_temp.cluster[mossa.rotta_destinazione]);
```

```
   calcola_tempo_percorso(*S_temp.cluster[mossa.rotta_destinazione]);

   calcola_F(S_temp);

   Soluzioni_temporanee.push_back(S_temp);
}
```

La funzione, al di là della forzata gestione tramite iteratori, risulta essere piuttosto semplice; costruisce la soluzione temporanea riferendosi ai parametri mantenuti dalla mossa. In particolare farà riferimento alle posizioni origine e destinazione di rotte e all'interno delle rotte, e al nodo in V da salvare. Effettua anche il calcolo di tutti i parametri della soluzione temporanea.

Ritornando a `valutazione_mossa`, a questo punto disponiamo di tutte le `Soluzioni temporanee`, e ciò che bisogna fare è discrimanare quali mantenere e quali no, applicando il *criterio di aspirazione*, attraverso la funzione `controlla_soluzioni_tabu`.

determina_prossima_soluzione La sottofunzione preposta al calcolo della prossima soluzione:

```
void determina_prossima_soluzione(Soluzione& Soluzione_candidata,
Soluzione& S, int& L_ammissibile, int& L_inammissibile,
int& C_ammissibile, int& C_inammissibile) {
   if (Soluzione_candidata.penalita_capacita > 0) {
      C_inammissibile++;
      C_ammissibile = 0;
   }
   else {
      C_ammissibile++;
      C_inammissibile = 0;
   }

   if (Soluzione_candidata.penalita_tempo > 0) {
      L_inammissibile++;
      L_ammissibile = 0;
   }
   else {
      L_ammissibile++;
```

```
      L_inammissibile = 0;
   }

   if (!(Soluzione_candidata.F2 > S.F2 && S.ammissibile)) {
      assegna_soluzione(S, Soluzione_candidata);
   }
   else;
}
```

La correzione delle penalità viene fatta grazie alla gestione di queste variabili `C_ammissibile` e simili. Alla fine, abbiamo anche la condizione per cui la soluzione candidata viene a sostituirsi alla soluzione mantenuta.

aggiorna_lista_parametri In questa sottofunzione come detto avvengono tutti gli aggiornamenti necessari per la corretta esecuzione dell'algoritmo.

```
void aggiorna_lista_parametri(const Soluzione& Soluzione_candidata,
Tabulist& lista_tabu, Soluzione& Soluzione_migliore_fattibile,
Soluzione& Soluzione_migliore, float& delta_max, float& f_v,
int& t, int& n_miglioramenti, const int& theta_min,
const int& theta_max, const float & F2_prec)
{
   /* Iteratore per scorrere il vettore di nodi V */
   std::vector<Nodo>::iterator it;

   /* Inserimento della mossa nella lista tabu */
   srand(t);
   int TTL_temp = rand() % (theta_max - theta_min) + theta_min;
   Tabu tabuTemp;
   tabuTemp.m = Soluzione_candidata.m;
   tabuTemp.TTL = TTL_temp + t;

   lista_tabu.mossa_tabu.push_back(tabuTemp);

   /* Aggiornamento S*, S~*, F*, F~* */

   bool soluzione_migliore_aggiornata = 0;
   if (Soluzione_candidata.ammissibile == 1)
      if (Soluzione_candidata.F1 < Soluzione_migliore_fattibile.F1)
```

```
      {
         assegna_soluzione(Soluzione_migliore_fattibile,
         Soluzione_candidata);
         soluzione_migliore_aggiornata = 1;
      }
   else if (Soluzione_candidata.ammissibile == 0)
      if (Soluzione_candidata.F2 < Soluzione_migliore.F2)
      {
         assegna_soluzione(Soluzione_migliore,Soluzione_candidata);
         soluzione_migliore_aggiornata = 1;
      }

   /* Aggiornamento di delta_max = |F2(Soluzione_candidata) -
   F2(Soluzione_candidata_precedente)| */

   delta_max = abs(Soluzione_candidata.F2 - F2_prec);

   /* Aggiornamento di f_v=numero_spostamenti_nodo/t */
   Nodo Nodo_temp = V[Soluzione_candidata.m.nodo];

   f_v = float(Nodo_temp.spostamenti) / float(t);

   /* Aggiornamento di n_miglioramenti per verificare
   la condizione di stop */
   if (soluzione_migliore_aggiornata == 0)
      n_miglioramenti++;
   else n_miglioramenti = 0;
   /* Aggiornamento dell'indice del costrutto iterativo */
   t = t + 1;

}
```

In questa funzione avviene l'aggiornamento dei parametri che verranno sfruttati nel corso dell'algoritmo, ma la cosa più importante, come detto, è l'aggiornamento della miglior soluzione fattibile S^* e della miglior soluzione $\tilde{S}^*$ a fronte della soluzione candidata calcolata nella corrente iterazione.

correzione_penalita L'ultima funzione, che è molto semplice data la gestione peculiare delle variabili locali, è la seguente:

```
void correzione_penalita(int& L_ammissibile, int& L_inammissibile,
int& C_ammissibile, int& C_inammissibile, const int & h) {
   if (C_ammissibile == h) {
      a = a / 2;
      C_ammissibile = 0;
   }
   if (C_inammissibile == h) {
      a = a * 2;
      C_inammissibile = 0;
   }
   if (L_ammissibile == h) {
      b = b / 2;
      C_ammissibile = 0;
   }
   if (L_inammissibile == h) {
      b = b * 2;
      C_inammissibile = 0;
   }

}
```

2.3.3 TABUROUTE

L'algoritmo TABUROUTE può essere descritto in maniera più snella rispetto a quant ofatto per la SEARCH, in quanto essendo TABUROUTE composto da molte chiamate alla SEARCH, possiamo tralasciare l'analisi approfondita a queste chiamate. TABUROUTE viene effettivamente richiamata tramite la chiamata a taburoute, che ha la seguente implementazione:

```
void taburoute(int lambda) {
   // Inizializzazione
   preleva_dati("a.txt");

   inizializza_variabili(100, FLOAT_INF,5);
   inizializza_soluzione(S_migliore,n_veicoli);
   inizializza_soluzione(S_migliore_fattibile,n_veicoli);
   inizializza_soluzione(S_temporanea, n_veicoli);

   // Configurazione parametri d'ingresso per SEARCH(P1)
   vector<Nodo> W = V; W.erase(W.begin());
   int q = 5 * n_veicoli;
   int p_1 = V.size() / n_veicoli;
   int p_2 = 0;
   int theta_min = 5;
   int theta_max = 10;
   float g = 0.01;
   int h = 10;
   int nmax = 100;

   // Diversificazione
   for (int i = 0; i < lambda; i++)
   {
      assegna_soluzione(S_temporanea,genera_soluzione(n_veicoli));

      aggiorna_parametri(S_temporanea);
      tabu_search(S_temporanea, W, q, p_1, p_2, theta_min,
      theta_max, g, h, nmax); // SEARCH(P1)
      aggiorna_parametri(S_temporanea);
      print_soluzione(S_temporanea);
   }
```

```
if (S_migliore_fattibile.F1 - FLOAT_INF < 0)
   tabu_search(S_migliore_fattibile, W, q, p_1, p_2, theta_min,
   theta_max, g, h, nmax);
else tabu_search(S_migliore, W, q, p_1, p_2, theta_min,
theta_max, g, h, nmax);

// Intensificazione

// Configurazione parametri d'ingresso per SEARCH(P3)
vector<Nodo> V_temp = W;
// Insertion sort, ordinamento crescente
for (int j = 1; j < V_temp.size(); j++) {
   Nodo key = V_temp[j];
   int i = j - 1;
   while (i > 0 && V_temp[i].spostamenti > key.spostamenti) {
      V_temp[i + 1] = V_temp[i];
      i = i - 1;
   }
   V_temp[i + 1] = key;
}
W.clear();
// Vengono sfruttati due indici, a causa dell'ordinamento crescente
for (int j = V_temp.size()-1; j >= V_temp.size() / 2; j--)
   W.push_back(V_temp[j]);

cout << W.size() << endl;
q = W.size();
nmax = 50 * dim;

if (S_migliore_fattibile.F1 - FLOAT_INF < 0) tabu_search(
S_migliore_fattibile, W, q, p_1, p_2, theta_min, theta_max, g,
h, nmax);
else tabu_search(S_migliore, W, q, p_1, p_2, theta_min,
theta_max, g, h, nmax);

if (S_migliore_fattibile.F1 - FLOAT_INF < 0) {
   cout << "SOLUZIONE OTTIMA TROVATA:\n";
   print_soluzione(S_migliore_fattibile);
}
else cout << "IL PROBLEMA NON AMMETTE SOLUZIONI\n";
```

```
   //Distruzione

   distruggi_soluzione(S_migliore);
   distruggi_soluzione(S_migliore_fattibile);
   distruggi_soluzione(S_temporanea);
};
```

L'algoritmo fa uso di un singolo parametro in ingresso, ossia λ, che serve a specificare quante volte generare delle soluzioni iniziali a partire dalle quali cercare di migliorare la soluzione migliore fattibile; a tal proposito, è opportuno specificare ancora la gestione delle soluzioni, che in questo caso è anche qua piuttosto ambiguo. L'algoritmo mantiene due cose importanti: `S_migliore_fattibile` e `S_migliore`. Queste due soluzioni sono l'equivalente di quanto viene gestito dalla `SEARCH`, con la differenza che la `SEARCH` gestisce questa coppia localmente, non avendo visione di quelle globali gestite da `TABUROUTE`, mentre `TABUROUTE` tiene traccia della sua coppia globale e le modifica sulla base dei risultati ottenuti dall'ottenimento delle prime soluzioni iniziali, ma anche e soprattutto dagli esiti delle chiamate successive alla `SEARCH`.

- Dopo le varie inizializzazioni che specificano tutti i parametri di ingresso per le prime chiamate a `SEARCH` da effettuare, avviene il primo approccio di *diversificazione*. La diversificazione consta, essenzialmente, di un approccio multi-start dove a ogni tentativo viene generata una soluzione di partenza mediante `genera_soluzione`, e successivamente, dopo aver aggiornato i parametri così come già accennato nella funzione `aggiorna_parametri`, viene effettuata la chiamata di `tabu_search`, a cui segue ovviamente un altro aggiornamento dei parametri.

- Al termine della diversificazione avviene il raffinamento della migliore soluzione (o migliore soluzione fattibile) trovata dall'approccio multi-start. Secondo la logica adottata, la `SEARCH` viene chiamata sulla migliore soluzione (`S_migliore`) se non è stata trovata alcuna `S_migliore_fattibile`, altrimenti viene effettuata la chiamata con `S_migliore_fattibile`.

- L'ultimo passo consiste nella *intensificazione*. In questo passo abbiamo innanzitutto la rimodulazione dei parametri; viene innanzitutto richiesto che l'insieme W venga popolato con $\frac{|V|}{2}$ nodi campionati dall'insieme V, presi in ordine decrescente per numero di spostamenti svolti nelle

precedenti chiamate a `SEARCH`. Successivamente viene ancora fatta la chiamata condizionale alla `SEARCH`, e a valle di questa chiamata l'algoritmo terminerà, mostrando la migliore (se trovata) soluzione fattibile.

genera_soluzione La sottofunzione `genera_soluzione` consente di generare, per λ volte, una soluzione di partenza differente, configurando il nostro approccio multi-start. Essa lavora considerando l'insieme dei nodi V: campionando in maniera casuale questi nodi, e dunque mischiandoli, procede successivamente a creare i cluster, ragionando secondo un approccio RFCS: l'ordinamento casuale dei nodi configura un ciclo hamiltoniano, mentre le successive operazioni procedono alla clusterizzazione dei nodi, partendo dal primo e arrivando fino all'ultimo, configurando tre possibilità:

1. Il cluster viene creato soddisfacendo i vincoli di lunghezza e capacità;
2. Se rimane l'ultimo veicolo, e quindi bisogna creare l'ultimo cluster, vengono inseriti tutti i nodi rimanenti, lasciando la possibilità che la soluzione risulti essere inammissibile;
3. Se i veicoli per creare le rotte sono terminati, i successivi cluster vengono creati vuoti.

Il codice è il seguente:

```
Soluzione genera_soluzione(const int& m) {
   Soluzione S;
   inizializza_soluzione(S, m);
   vector <Nodo> V_Copy = V;
   V_Copy.erase(V_Copy.begin());
   vector <Nodo> Copy = V_Copy;
   vector <Nodo> V_casuale;
   V_casuale.clear();
   srand(time(NULL));
   for (int i = 0; i < dim - 1; i++) {
      int j = (int)(rand() % (dim - i - 1));
      V_casuale.push_back(Copy[j]);
      Copy.erase(Copy.begin() + j);
   }
   float domanda = 0;
```

```
float lunghezza_percorso = 0;
int indice_prec = 0;
int n_nodi = dim - 2;
int c = 0;

S.cluster[c]->indice = c;
S.cluster[c]->percorso.push_back(V[0]);
while (n_nodi >= 0 && c < m - 1) {
   if ((domanda + V_casuale[n_nodi].domanda <= Q) &&
   (lunghezza_percorso + V_casuale[n_nodi].tempo +
   adjMatrix[indice_prec][V_casuale[n_nodi].indice] +
   adjMatrix[0][V_casuale[n_nodi].indice] <= L))
   {
      domanda += V_casuale[n_nodi].domanda;
      lunghezza_percorso += V_casuale[n_nodi].tempo +
      adjMatrix[indice_prec][V_casuale[n_nodi].indice];
      S.cluster[c]->percorso.push_back(V_casuale[n_nodi]);
      indice_prec = V_casuale[n_nodi].indice;
      V_casuale.pop_back();
      n_nodi--;
   }
   else {
      calcola_lunghezza_percorso(*S.cluster[c]);
      calcola_domanda_percorso(*S.cluster[c]);
      calcola_tempo_percorso(*S.cluster[c]);
      indice_prec = 0;
      domanda = 0;
      lunghezza_percorso = 0;
      c++;
      S.cluster[c]->indice = c;
      S.cluster[c]->percorso.push_back(V[0]);
   }
}
if (n_nodi >= 0) {
   while (n_nodi >= 0) {
      S.cluster[c]->percorso.push_back(V_casuale[n_nodi]);
      V_casuale.pop_back();
      n_nodi--;
   }
   calcola_lunghezza_percorso(*S.cluster[c]);
```

```
      calcola_domanda_percorso(*S.cluster[c]);
      calcola_tempo_percorso(*S.cluster[c]);
      c++;
   }
   if (c < m - 1) {
      calcola_lunghezza_percorso(*S.cluster[c]);
      calcola_domanda_percorso(*S.cluster[c]);
      calcola_tempo_percorso(*S.cluster[c]);
      c++;
      while (c < m) {
         S.cluster[c]->indice = c;
         S.cluster[c]->percorso.push_back(V[0]);
         S.cluster[c]->domanda_totale = 0;
         S.cluster[c]->lunghezza = 0;
         S.cluster[c]->tempo_servizio = 0;
         c++;
      }
   }
   calcola_F(S);
   return S;
}
```

aggiorna_parametri La funzione è piuttosto semplice; consente di aggiornare la migliore soluzione e la migliore soluzione fattibile mantenute da TABUROUTE:

```
void aggiorna_parametri(const Soluzione& Soluzione_in) {

   if (Soluzione_in.ammissibile == 0)
   {
      if (Soluzione_in.F2 < S_migliore.F2)
         assegna_soluzione(S_migliore,Soluzione_in);
   }
   else if (Soluzione_in.ammissibile == 1)
   {
      if (Soluzione_in.F1 < S_migliore_fattibile.F1)
      {
         assegna_soluzione(S_migliore,Soluzione_in);
         assegna_soluzione(S_migliore_fattibile,Soluzione_in);
      }
   }
}
```

2.4 Organizzazione fisica

L'algoritmo viene sviluppato usufruendo di tre librerie, nominate `HGL.h`, `Search.h`, `Taburoute.h`. Ognuna di queste librerie racchiude un insieme coeso di informazioni; in particolare:

1. `HGL.h`
   È una sorta di libreria 'di base'; essa incapsula tutte le librerie standard adoperate per il progetto, insieme alle strutture dati comuni a tutti i sorgenti, e funzioni di utilità (quali funzioni ausiliarie sfruttate nel corso dell'implementazione e funzioni di debug).

```
#pragma once

#include <iostream>
#include <cmath>
#include <fstream>
#include <vector>
#include <list>
#include <ctime>
#include <cstdlib>
#include <fstream>
//#include "pricer_vrp.h"

#define FLOAT_INF 3.40282e+038

using namespace std;

struct Nodo;

struct Rotta {
   list<Nodo> percorso;
   float lunghezza;
   float tempo_servizio;
   float domanda_totale;
   int indice;
};

struct Nodo {
   float x;
```

```
   float y;

   int indice;
   int domanda;
   int tempo;
   int spostamenti;

};

struct Mossa {
   int nodo;
   int rotta_origine;
   int rotta_destinazione;
   int posizione_in_origine;
   int posizione_in_destinazione;
   float costo;
};

struct Soluzione {
   float F1;
   float F2;
   float F_temp;
   float penalita_capacita;
   float penalita_tempo;
   bool ammissibile;
   vector<Rotta*> cluster;
   Mossa m;
};


struct Tabu {
   Mossa m;
   int TTL;
};

struct Tabulist {
   vector<Tabu> mossa_tabu;
};
```

```
//FUNZIONI

void preleva_dati(const string & nome);
void inizializza_variabili(const float& q, const float& l);
void distruggi_variabili(Soluzione & S);
void inizializza_soluzione(Soluzione& S,const int &);
void distruggi_soluzione(Soluzione& S);
void assegna_soluzione(Soluzione& S1, const Soluzione& S2);
void calcola_lunghezza_percorso(Rotta&);
void calcola_domanda_percorso(Rotta&);
void calcola_tempo_percorso(Rotta&);
void calcola_F(Soluzione & S);
int posizione_nodo(Rotta&, const Nodo&);
int rotta_del_nodo(const Soluzione&, const Nodo&);


//FUNZIONI DI DEBUG TEMPORANEE

void random_node(Nodo&);
void random_set();
void print_node(const Nodo& v);
void print_set(const vector<Nodo>& V);
void print_adjMatrix();
void debug_build_S(Soluzione & S);
void print_soluzione(const Soluzione& S);
```

2. Search.h
   L'implementazione di SEARCH sfrutta quasi solamente le interfacce di questa libreria. Essa contiene tutte le firme delle funzioni e sottofunzioni che sono state necessarie per implementare l'algoritmo, e caratterizza sicuramente la parte più corposa del progetto.

```
#pragma once
#include "HGL.h"
#define STOP 10000



void tabu_search(Soluzione& S,vector<Nodo>& W, int q, int p1,
int p2, int theta_min, int theta_max, int g, int h, int n_max);
```

```
//FUNZIONI COMPONENTI SEARCH

void seleziona_vertici(vector<Nodo>& PSI, int q,
vector<Nodo> W, int dim);

Soluzione valutazione_mossa(const vector<Nodo>& PSI, int q, int p1,
vector<Soluzione> & Soluzioni_temporanee, Tabulist&, Soluzione & S,
Soluzione& Soluzione_migliore_fattibile,
Soluzione& Soluzione_migliore, const float &, const int &);

void determina_prossima_soluzione(Soluzione& Soluzione_candidata,
Soluzione& S, int& L_ammissibile, int& L_inammissibile,
int& C_ammissibile, int& C_inammissibile);

void aggiorna_lista_parametri(const Soluzione& Soluzione_candidata,
Tabulist& lista_tabu, Soluzione& Soluzione_migliore_fattibile,
Soluzione& Soluzione_migliore, float& delta_max,
float& f_v, int& t, int& n_miglioramenti, const int& theta_min,
const int& theta_max, const float& F2_prec);

void correzione_penalita(int& L_ammissibile, int& L_inammissibile,
int& C_ammissibile, int& C_inammissibile, const int & h);


//SOTTOFUNZIONI VALUTAZIONE_MOSSA

void costruzione_vicinato(const Nodo& v, vector<int> & N,
const int &p1);

void identifica_soluzioni_temporanee(const vector<int> & N,
const int & indice_nodo, const Tabulist, Soluzione& S,
vector<Soluzione>& Soluzioni_temporanee, const int& indice_q);

void controlla_soluzioni_tabu(
vector<Soluzione>& Soluzioni_temporanee, Tabulist& listaTabu,
Soluzione& S, Soluzione& Soluzione_migliore_fattibile,
Soluzione& Soluzione_migliore,const float& delta_max,
const int& f_v);
```

```
Soluzione determina_soluzione_candidata(
vector<Soluzione>& Soluzioni_temporanee);

//SOTTOFUNZIONI DI COSTRUZIONE_VICINATO

int minimo_indice(vector<float>& V, const int& indice_nodo);


//SOTTOFUNZIONI DI IDENTIFICA_SOLUZIONI_TEMPORANEE

bool intersezione(const vector<int>& N, Rotta* cluster,
int& indice_intersezione);

Mossa calcola_mossa(Rotta Rr, Rotta Rs, const int& indice_origine,
const int& indice_destinazione);

void costruisci_soluzione(vector<Soluzione>& Soluzioni_temporanee,
const Mossa& mossa, const Soluzione& S);


//SOTTOFUNZIONI DI CONTROLLA_SOLUZIONI_TABU

bool cerca_tabu(Tabulist& listaTabu, const Mossa& mossa);
```

3. `Taburoute.h`
   Come è facile aspettarsi, dato che `TABUROUTE` fa uso quasi esclusivamente di chiamate a funzioni già sviluppate, la sua intestazione è molto ridotta, ma viene comunque qui presentata:

```
#include "Search.h"

void taburoute(int lambda);

void aggiorna_parametri(const Soluzione& Soluzione);

Soluzione genera_soluzione(const int& m);
```