lgvRouting

Luca Bartoli
Massimiliano bosi
24/09/2021

Premessa

Questo documento vuole racchiudere gli algoritmi sviluppati per risolvere un problema di routing degli lgv all'interno di uno stabilimento, effettuando operazione di carico e scarico. Verranno presentati le varie tecniche implementate e valutate secondo il punto di vista dei costi e delle tempistiche.

Problema

Il problema che vogliamo andare a risolvere può essere descritto con il seguente modello:

$K = insieme\ delle\ navette$
$C = insieme\ dei\ punti\ di\ carico$
$S = insieme\ dei\ punti\ di\ scarico$
$m_{cs} = se\ esiste\ una\ missione\ tra\ c \in C\ e\ s \in S$
$w_{sc} = costo\ della\ missione\ da\ c \in C\ e\ s \in S$

$t_k = tempo\ di\ completamento\ della\ navetta\ k$
$x_{csk} = se\ la\ navetta\ k \in K\ completa\ la\ missione\ tra\ c \in C\ e\ s \in S$

$$min\ \max(t_k) \qquad \forall k \in K$$

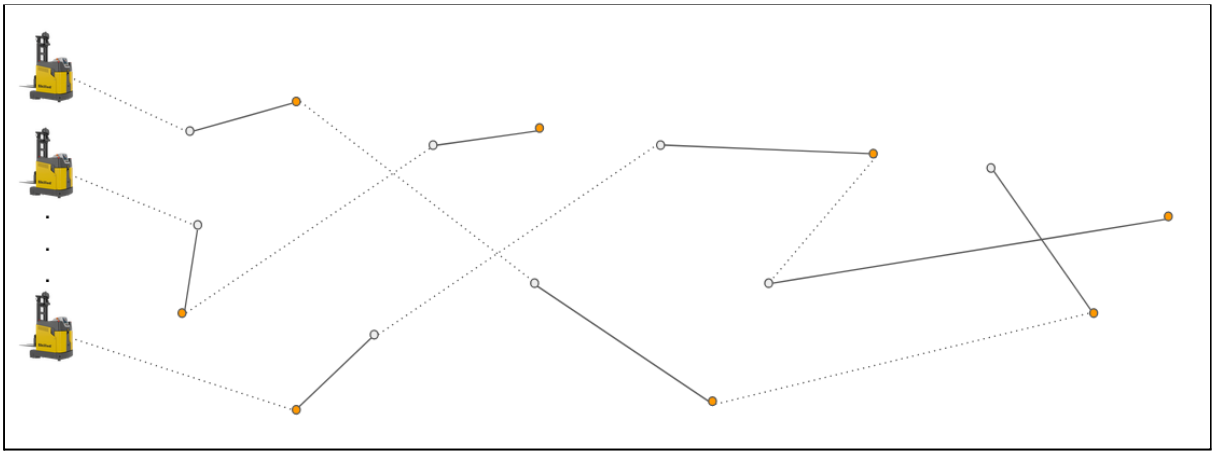
$$\sum_{k \in K} x_{csk} = m_{cs} \qquad \forall c \in C,\ s \in S$$

$$\sum_{c \in C} \sum_{s \in S} x_{csk} w_{sc} = t_k \qquad \forall k \in K$$

$$t_k \in Z^+, \qquad \forall k \in K$$

$$x_{csk} \in \{0, 1\}, \quad \forall c \in C, s \in S, k \in K$$

Il nostro obiettivo è andare a completare il routing delle navette nel minor tempo possibile, tenendo in considerazione il tempo della navetta che ci mette più tempo a completare tutte le missioni. Di seguito riportiamo il problema sotto il punto di vista grafico.



Dataset

Si è scelto di utilizzare il dataset di VRPPD P2 proposto da Breedam. Il dataset contiene 60 istanze, ognuna di esse formata da 100 stop.

```
  0   50   50    0 9999 9999 9999    0    0   0  (depot)
  1    5    5    0 9999 9999 9999   10    0   0  (stop 1)
      ....                                  ...
100   95   95    0 9999 9999 9999   10    0   0  (stop 100)
  |    |    |    |    |    |    |     |    |   |
  |    |    |    |    |    |    |     |    |   delivery=0 / pickup=1
  |    |    |    |    |    |    |     |    service time
  |    |    |    |    |    |    |     demand
  |    |    |    |    |    |    closing time TW2
  |    |    |    |    |    opening time TW2
  |    |    |    |    closing time TW1
  |    |    |    opening time TW1
  |    |    Y coordinate
  |    X coordinate
  stop nr.
```

Ogni riga rappresenta uno stop ed è composta, in ordine, dalle colonne:

- numero identificativo dello stop
- coordinata X
- coordinata Y
- tempo di apertura della finestra 1
- tempo di chiusura della finestra 1
- tempo di apertura della finestra 2
- tempo di chiusura della finestra 2
- quantitativo di merce da fornire o richiesta
- tempo per effettuare il servizio di carico o scarico
- booleano per identificare se lo stop è di carico o scarico

Negli algoritmi sviluppati non si utilizzano i dati relativi alle capacità e ai tempi poiché si ipotizza di spostare un pallet alla volta e il costo per spostarsi da uno stop all'altro è calcolato come distanza euclidea.

I tempi riportati sono calcolati in media utilizzando tutte le 60 istanze del problema.

Swap

Il concetto di swap risulta molto importante perché verrà utilizzato in tutti gli algoritmi implementati. Con swap intendiamo l'assegnazione casuale di una missione ad un lgv diverso da quello deciso dalla soluzione costruttiva o randomica. Questa metodologia ci permette di muovere all'interno del vicinato di una soluzione, o di allontanarci impostando un numero di swap elevato. Nel local search viene impostato ad uno in quanto risulta molto sensato muoverci all'interno del vicinato stretto.

Constructive

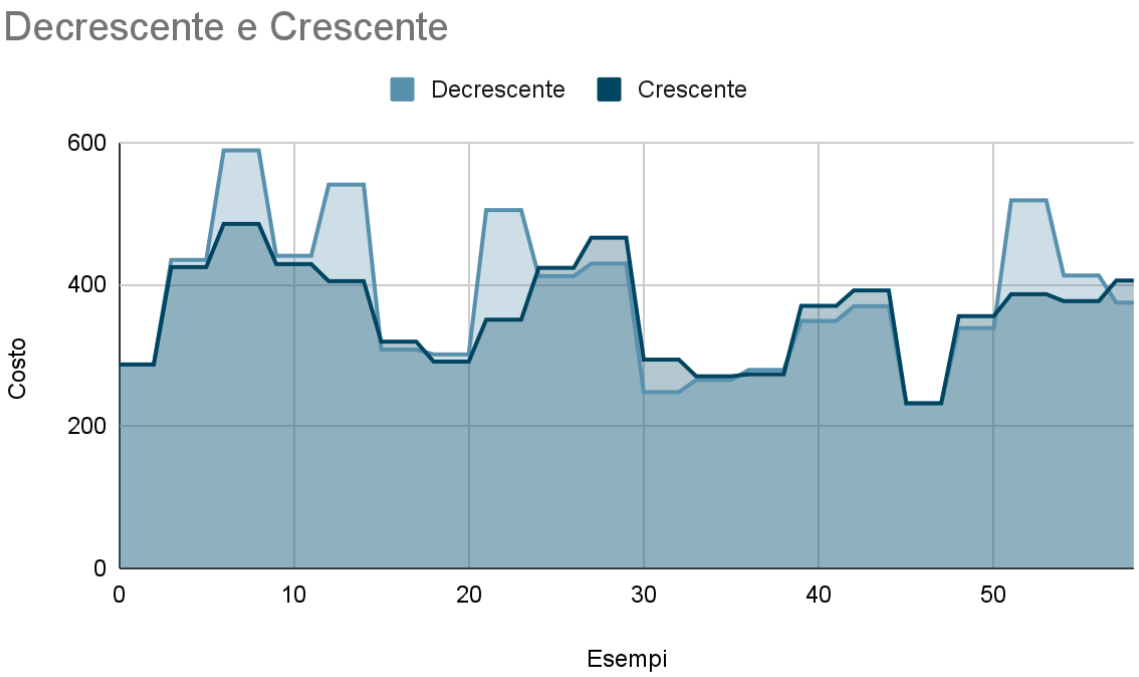
L'algoritmo constructive è stato sviluppato sfruttando un euristica costruttiva. Date N navette e M missioni da compiere, l'algoritmo ordina le missioni in base al peso e le assegna alla navetta con il minor carico. Questo algoritmo viene utilizzato per inizializzare gli algoritmi descritti in seguito.

Parametri:

- **Decreasing:** vero per ordinare le missioni in ordine decrescente, falso altrimenti.

```
sort(missions, Decreasing)
lgv[n_lgv] = 0
sol
for m in missions do:
  idx = findMinCostIndex(lgv)
  lgv[idx] += m.cost
  m.id = idx
  addMissionToSolution(sol, m)
end for
return sol
```

Viene effettuato un confronto a livello di costi per ogni singola istanza del dataset variando l'ordinamento delle missioni.



Dal grafico precedente possiamo notare come il valore del costo in media è migliore nel caso di ordinamento crescente delle missioni.

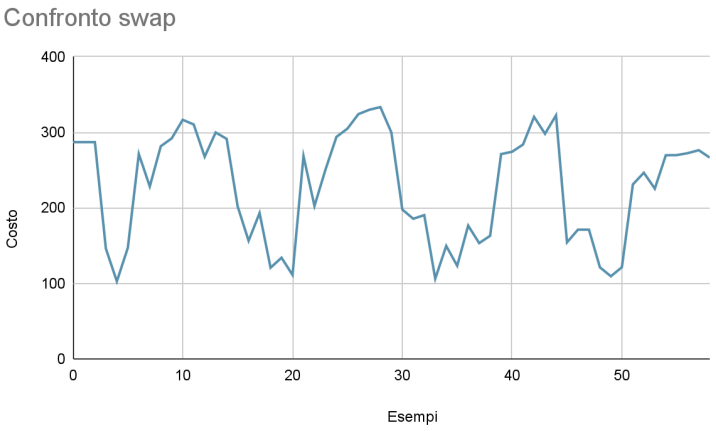
Local Search

L'algoritmo local search è stato costruito partendo dall euristica costruttiva precedente. Per migliorare la soluzione di partenza l'algoritmo sceglie il miglior risultato della soluzione attuale tra 1000 vicinati calcolati attraverso l'applicazione di uno swap. L'esecuzione termina con il raggiungimento del timeout o il raggiungimento di una soluzione peggiore rispetto a quella attuale.

Parametri:

- **timeout:** timeout del programma

```
start = findInitialConstructiveSolution()
while True do:
  localBest = ∞
  for i in 1000 do
    sol = makeSwap(1, start)
    if sol < localBest do
      localBest = start
    end if
  end for
  if start > localBest then:
    start = localBest
  else
    break
  end if
  checkTimeout(timeout)
end for
return start
```



Viene effettuato un confronto a livello di costi per ogni singola istanza del dataset come presente nel grafico precedente. Il risultato ottenuto è nettamente proporzionale al numero di vicini che andiamo ad esplorare per ottenere una soluzione migliore rispetto a quella attuale. Il parametro come specificato nell'introduzione è settato a 1000 per questo dataset.

Multistart

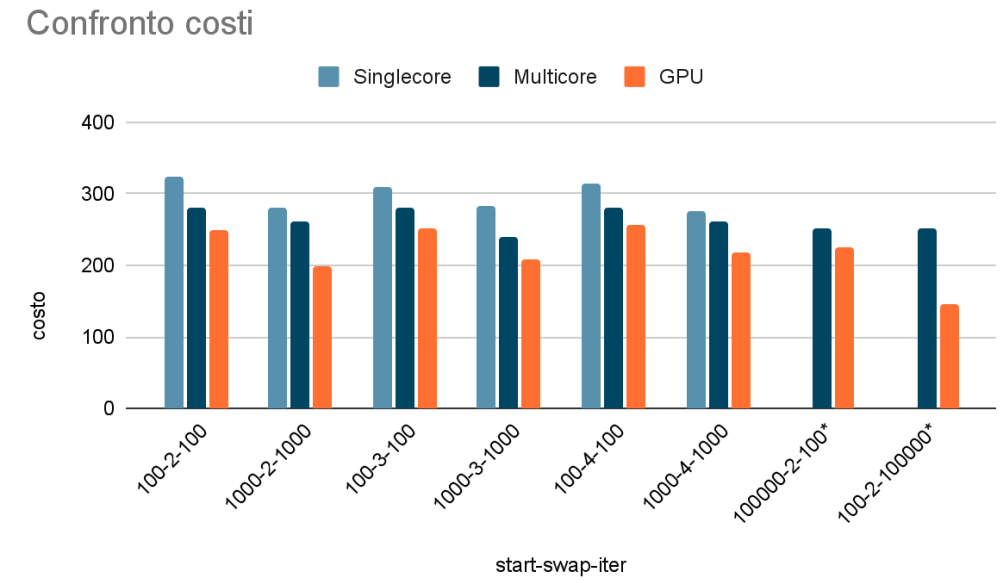
L'algoritmo multistart e stato sviluppato partendo da n soluzioni casuali e da queste viene cercata una soluzione ottimale effettuando degli swap per un certo numero di iterazioni

Parametri:

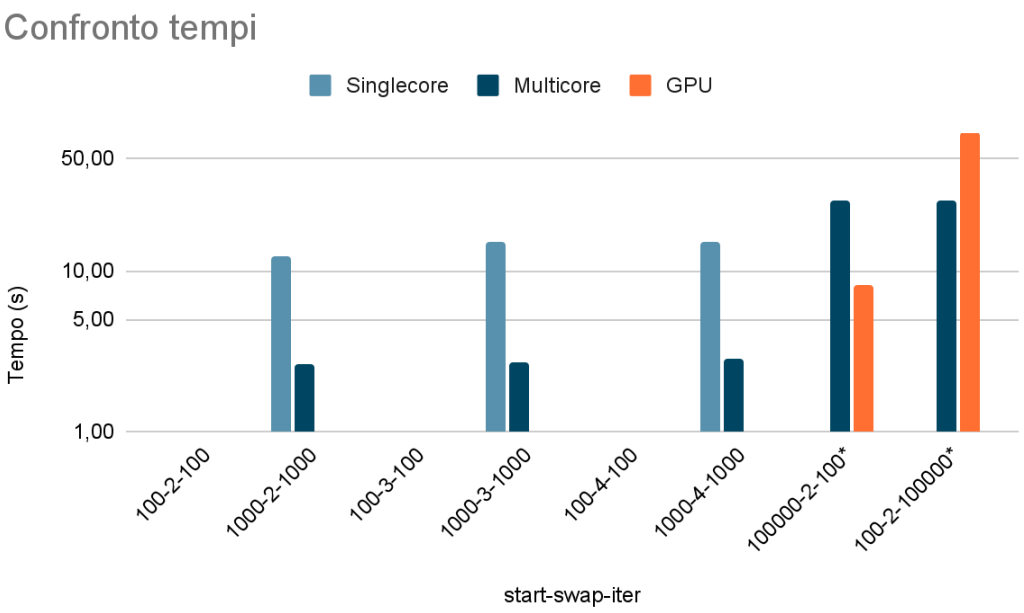
- **Nswap**: numero degli swap da applicare alla soluzione iniziale casuale;
- **timeout**: timeout del programma;
- **Nstart**: numero soluzioni iniziali da cui partire;
- **Niter**: iterazioni dell'algoritmo.

```
best = ∞
for start in Nstart do
      iterBest = findInitialRandomSolution()
      for iter in Niter do
            localBest = ∞
            start = iterBest
            for i in 1000 do
              sol = makeSwap(Nswap, iterBest)
              if localBest > sol do
                localBest = sol
              end if
            end for
            if localBest < iterBest then
                  iterBest = localBest
            end if
            checkTimeout(timeout)
      end for
      if best > iterBest
            best = iterBest
      end if
end for
return best
```

L'algoritmo è stato sviluppato in 3 varianti: singlecore, multicore e GPU.
Viene effettuato un confronto a livello di tempi e costi delle tre tipologie sulla base della terna (**Nstart, Nswap, Niter**)



Dal grafico precedente possiamo notare che il valore del costo non è legato dal numero degli swap effettuati. Sembra invece contare molto il numero delle iterazione rispetto a quello degli start points. La versione su GPU ha un costo migliore perché i valori random sono generati secondo una funzione di NVIDIA all'init (cuRAND) che genera tutti i valori in modo lineare secondo l'n da generare. Ora soffermiamoci sui tempi:



I tempi risultano migliori su GPU quando abbiamo molti punti di start e poche iterazioni, questo perchè sulla singola iterazione la GPU è più lenta, invece la parallelizzazione a livello di thread viene in modo automatico essendo un architettura SIMD.

*I valori su singlecore non sono riportati per il tempo necessario al computamento

Simulated Annealing

L'algoritmo Simulated Annealing è ispirato a ciò che fanno i fisici per ottenere la configurazione di un materiale allo stato solido di minima energia. Sia E_x l'energia di uno stato del materiale x e i,j due stati consecutivi, allora:

- se $E_j <= E_i$ allora i è sempre accettato
- se $E_j > E_i$ allora j è accettato con una probabilità:

$$P = e^{\left(\frac{E_i - E_j}{k_B^T}\right)}$$

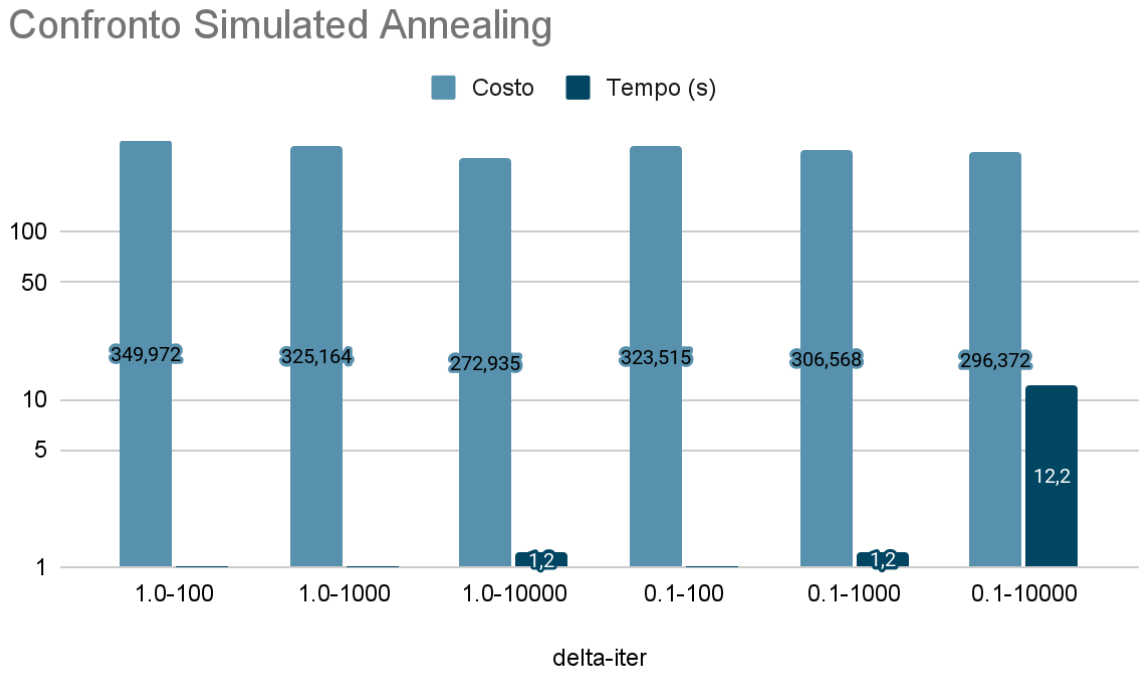
Dove T è la temperatura iniziale e k_B la costante di Boltzman.

Parametri:

- **Nswap**: numero degli swap da applicare alla soluzione iniziale casuale;
- **initialT**: temperatura iniziale;
- **finalT**: temperatura minima;
- **deltaT**: delta di decremento delle temperatura;
- **Niter**: iterazioni di discesa della temperatura;
- **timeout**: timeout del programma.

```
best = findInitialRandomSolution()
temp = initalT
while temp > finalT do
      for iter in Niter do
            sol = makeSwap(Nswap, best)
            if sol < best then
                  best = sol
            else
                  p = e^((sol-best)/temp)
                  if p > rand(0,1) then
                        best = sol
                  end if
            end if
            checkTimeout(timeout)
      end for
      temp = temp - deltaT
end while
return best
```

A questo algoritmo è legato un teorema di convergenza: se la temperatura decresce abbastanza lentamente, allora la probabilità di ottenere un ottimo globale tende a 1 quando il tempo di calcolo è abbastanza grande.
Come possiamo notare nel grafico successivo, più aumentiamo le iterazioni dell'algoritmo e più il valore dell'ottimo trovato sarà migliore.
Viene fissata la temperatura iniziale a 10 e finale a 0.1. Il numero degli swap a 2.



Ovviamente aumentando il numero di iterazioni l'algoritmo necessita di più tempo di calcolo per terminare. Dal teorema precedente sembra possibile raggiungere un ottimo globale con questo algoritmo, ma se andiamo a calcolarci i tempi scopriamo che ha un ordine di $o(n^{n^2}(n-1)) >> o(n!)$, I tempi necessari al completamento risultano pressoché infiniti, di conseguenza dobbiamo andare ad aggiungere un euristica sul tempo oppure sulla velocità della discesa della temperatura.

Tabu Search

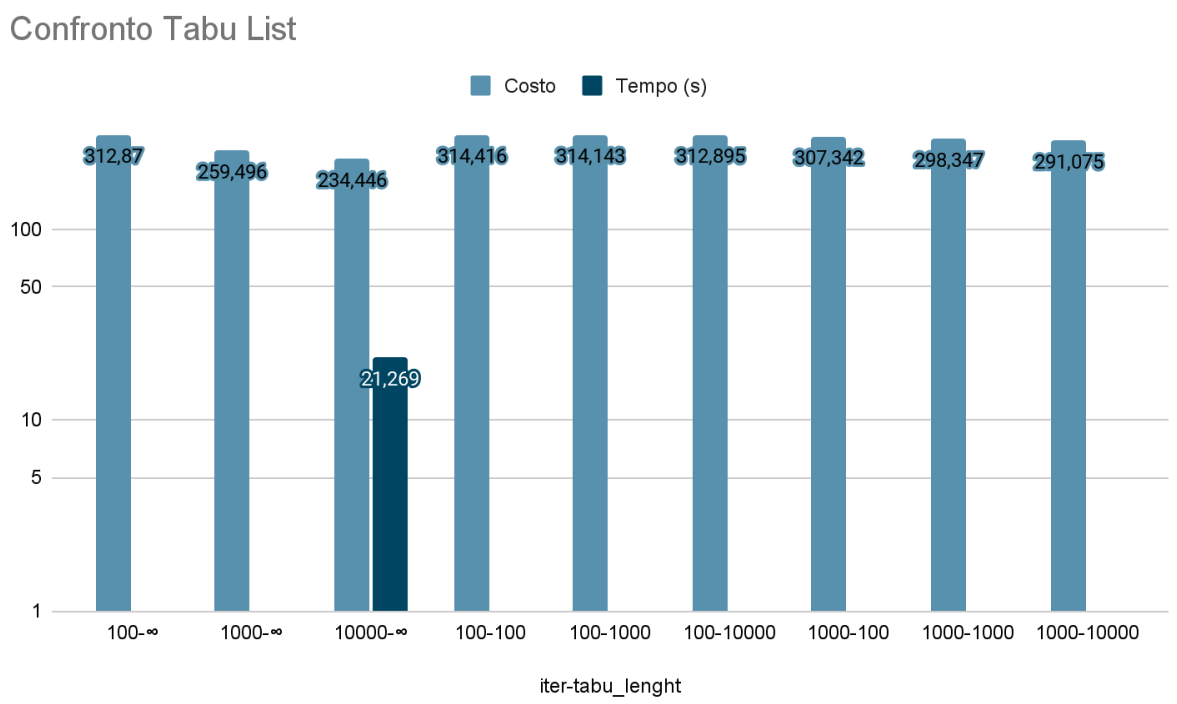
Partendo da una soluzione iniziale tramite euristica costruttiva, l'algoritmo effettua Nswap per Nit iterazioni prendendo anche soluzioni che si discostano di un certo threshold per scappare da minimi locali. Soluzioni già esplorate vengono evitate tramite l'inserimento in una tabu list con lunghezza configurabile. Più la lunghezza della tabu list sarà lunga, più andremo ad effettuare diversificazione tra le soluzioni trovate.

Parametri:

- **Nswap**: numero degli swap da applicare alla soluzione iniziale costruttiva;
- **Niter**: iterazioni dell'algoritmo;
- **MaxDelta**: massima differenza di costo per accettare una nuova soluzione;
- **Ndata**: lunghezza della lista;
- **timeout**: timeout del programma;

```
best = finalBest = findInitialConstructiveSolution()
tabu = createQueueWithLenght(Ndata)
for iter in Niter do
      sol = makeSwap(Nswap, best)
      if sol not in tabu then
            push(tabu, sol)
            if sol - best < MaxDelta then
                  best = sol
            end if
      end if
      if finalBest > best then
            finalBest = best
      end if
      checkTimeout(timeout)
end for
return finalBest
```

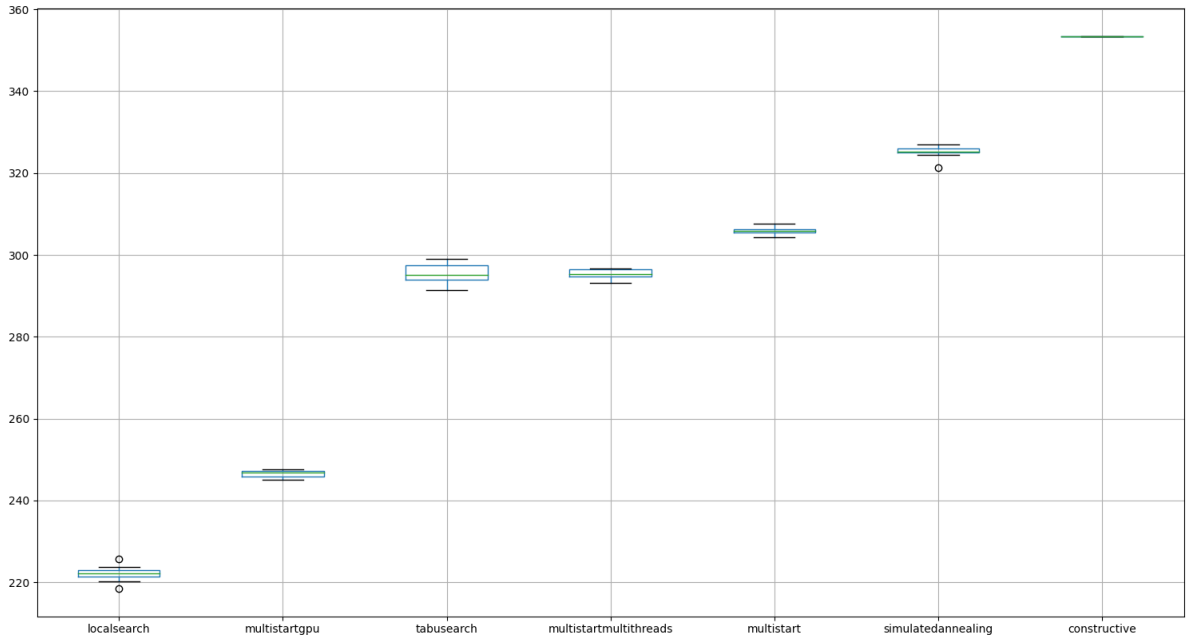
Il confronto è stato fatto sulla base delle iterazione e la lunghezza della tabu list. Come possiamo osservare dal grafico sottostante aumentando la lunghezza della tabu list rendendola infinita, otteniamo un buon risultato a scapito dell'aumento dei tempi di computazione in maniera esponenziale, rispetto a tenerla di una lunghezza massima. Il numero delle iterazioni influisce sul risultato, ma non come la lunghezza della coda. La coda è implementata mediante una lista standard del c++.


Confronto Tabu List
Costo
Tempo (s)
312,87
259,496
234,446
21,269
314,416
314,143
312,895
307,342
298,347
291,075
100
50
10
5
1
100-∞
1000-∞
10000-∞
100-100
100-1000
100-10000
1000-100
1000-1000
1000-10000
iter-tabu_lenght

Risultati

Come comparazione abbiamo utilizzato le configurazioni di default presente alla repo github e riportate in questo documento a pedice. I risultati sono stati calcolati eseguendo ogni singolo algoritmo sul dataset P2 composto da 60 istanze per 33 volte, in modo da ottenere una distribuzione dell'errore su ogni metodo.

Risultati Costi



Eseguendo tutti gli algoritmi impostando come timeout 1 secondo otteniamo che il costo più basso è dato dal locasearch con 1000 iterazioni sul vicinato. Subito dopo troviamo il multistartgpu, che risulta molto più performante rispetto a quello su cpu.

Appendice

Configurazione di default:

```
Constructive:
  Decreasing: false

LocalSearch:
  timeout: 1000

DepthLocalSearch:
  swap: 2
  iteration: 10000
  timeout: 1

MultiStart:
  swap: 4
  iteration: 1000
  start: 1000
  timeout: 1

MultistartGpu:
  swap: 4
  iteration: 1000
  start: 1
  timeout: 1

MultiStartMultithread:
  swap: 2
  iteration: 1000
  start: 1000
  threads: 16
  timeout: 1

SimulatedAnnealing:
  initalTemperature: 10
  minTemperature: 0.1
  coolingRate: 1.0
  iterTempDec: 1000
  timeout: 1

TabuSearch:
  swap: 1
  iteration: 1000
  diffCost: 1
  listLenght: 0
  timeout: 1
```

Bibliografia

Dataset VRP (https://neo.lcc.uma.es/vrp/)
Github lgvRouting (https://github.com/lucabart97/lgvRouting)