Vehicle Routing Problem with Time Windows (VRPTW)

Descrizione del problema e del suo contesto

Il problema del Vehicle Routing Problem with Time Windows è un problema noto nella letteratura della ricerca operativa. Si basa sul problema del commesso viaggiatore, ammettendo più veicoli per effettuare le consegne ma aggiungendo anche vincoli di finestre temporali che rappresentano gli unici momenti durante i quali i clienti possono accettare le consegne. Di questo problema esistono numerose varianti con diversi vincoli che si adattano a diversi contesti a cui è applicato: ad esempio esistono versioni con fattorini non uguali tra di loro (diverse velocità e/o capacità), con finestre temporali singole o multiple, con penalità se non si rispettano le finestre stesse, etc. etc. In questo progetto lo scenario preso in considerazione è quello di una pizzeria che deve ottimizzare le consegne dei fattorini ai propri clienti, per cui nelle prossime sezioni si parlerà di come si è scelto di modellare il problema rispetto agli specifici vincoli

Vincoli relativi al contesto

La pizzeria presa in considerazione fornisce ai suoi clienti finestre temporali di 30 minuti per la consegna al momento della prenotazione. I suoi dipendenti (fattorini compresi) iniziano a lavorare alle 18:00 e finiscono alle 22:00. Il picco massimo del numero di consegne avviene alle ore 20:00 circa. Tutte le consegne sono effettuate nella città di Carpi, che ha un diametro di circa 15 minuti e la pizzeria è circa al centro della città. Per cui in media un punto di consegna dista circa 7.5 minuti dalla pizzeria. Il numero massimo di ordini trasportabili da un singolo fattorino è 3 (per non far raffreddare troppo le pizze) e tutti fattorini sono considerati equivalenti fra di loro. Una volta che un fattorino è arrivato a destinazione si considerano 5 minuti per effettuare la consegna e il pagamento da parte del cliente.

Applicazione dei vincoli al problema

- Il problema si configura con un unico deposito (depot) dal quale tutti i fattorini devono uscire e rientrare per effettuare le consegne.
- La capacità di ogni fattorino è di 3 unità (ordini) e ad ogni consegna questa cala di 1. Quando un fattorino raggiunge il depot, la sua capacità torna automaticamente a 3. Il numero di fattorini è un dato del problema (nel caso della pizzeria sono 4) ed è comunque impostabile dal file di configurazione corrispondente al dataset che si sta testando.
- Attendere fermi al depot è consentito e non costa nulla mentre attendere al di fuori del depot è proibito.
- In sede di test degli algoritmi che risolvono il problema è necessario creare grafi i quali archi hanno proprietà simili a quelle della città di Carpi. Forzando la velocità dei veicoli a un valore tale da rendere la media degli archi uscenti dal depot di 7.5 minuti, è possibile ricavare il costo di tutti gli altri archi sarà calcolato di conseguenza. Questo accorgimento rende possibile testare gli algoritmi praticamente su qualsiasi grafo. Per semplicità si considerano questi tempi invarianti durante tutta l'esecuzione. La distanza (spaziale) presa in considerazione per calcolare la distanza finale (che è temporale) è quella euclidea. Inoltre, i grafi testati si considerano completi.
- Gli archi che collegano due clienti tra loro o uscenti dal depot hanno il loro costo temporale aumentato di 5 minuti per rappresentare il vincolo del tempo di consegna, mentre gli archi entranti nel depot non sono soggetti a questo vincolo.
- Dato il tempo di operatività della pizzeria e del suo picco, si può approssimare la distribuzione delle finestre temporali a una normale con media centrata alle 20:00. Una varianza adatta a rappresentare il problema è stata trovata sperimentalmente attraverso il pacchetto python numpy.
- Si considera possibile consegnare pizze in anticipo rispetto alla finestra temporale (pagando una penalità) ma non in ritardo. Questo vincolo è ragionevole dal momento in cui non si vuole

scontentare un cliente consegnando in ritardo mentre la consegna in anticipo (ma non di troppo, altrimenti sarebbe come avere solamente una finestra temporale sul tempo massimo di consegna) potrebbe essere accolta anche positivamente dal cliente.

Un' importante conseguenza di queste modifiche sui grafi presi in considerazione è il fatto che si perdono i riferimenti con i creatori dei grafi stessi, che hanno impostato un determinato numero di veicoli, determinati costi sugli archi e determinata funzione obiettivo. Di conseguenza, a meno di implementare un metodo esatto per risolvere i problemi dati, non si conosce la soluzione ottimale dei problemi.

Soluzione proposta: funzione obiettivo

Ogni algoritmo proposto mira a minimizzare la seguente funzione obiettivo:

$$\min \sum_{v \in V} P_v + \sum_{a \in T_v} c(a)$$

dove $v \in V$ è un singolo veicolo (fattorino), T_v è il tour del veicolo v (formato dai suoi archi), $c(a)$ è il costo dell'arco a sul grafo preso in considerazione e P_v sono le penalità associate al veicolo v. Se consideriamo la time window di un nodo n come $W_n = [s_n, f_n]$, il tempo in cui il nodo n è stato servito come T_n ($T_n, s_n, f_n \in [0, 240]$), allora le penalità sono assegnate nel seguente modo:

$$P_v = \sum_{n \in N_v} c(n), con\ c(n) = \begin{cases} \left(\frac{s_n - T_n}{2}\right)^2 se\ s_n > T_n \\ 0\ altrimenti \end{cases}$$

dove N_v è l'insieme dei nodi serviti dal veicolo v. Questa scelta è dovuta al fatto che dividendo per 2 la differenza tra il tempo di arrivo e il tempo iniziale della time window si pagano pochissimo anticipi piccoli (da 1 a 3 minuti), mentre si pagano tantissimo gli arrivi troppo anticipati (già 10 minuti prima costa 25 in più). Parametrizzando il coefficiente per cui è divisa la differenza tra il tempo di servizio e l'inizio della time window si può modificare il concetto di "arrivi troppo anticipati" adattandolo al contesto del problema.

"Vanilla" Local Search

La soluzione proposta per risolvere il problema è un algoritmo local search. L'algoritmo proposto è composto da due parti: un algoritmo di costruzione della soluzione iniziale basato sul concetto di "regret" e un algoritmo di miglioramento della soluzione iniziale basato sullo scambio di nodi.

Algoritmo di creazione della soluzione regret-based

L'algoritmo greedy di creazione della soluzione si basa sul concetto di regret. L'idea di questa tecnica è di creare una soluzione inserendo un nodo alla volta, scegliendo il nodo che ha il maggior punteggio. Il punteggio di ogni nodo si ottiene prima calcolando i primi due modi di inserire il nodo considerato nella soluzione attuale poi calcolando la differenza tra il secondo miglior modo di inserire il nodo considerato con il miglior modo di inserire quel nodo (che, volendo minimizzare il tempo, sarà minore). Ripetendo questo processo per ogni nodo ancora da inserire è possibile ottenere un punteggio per ciascuno e alla fine dell'iterazione si sceglie di inserire il nodo con punteggio maggiore con il corrispettivo miglior modo. Questa tecnica fa sì che se c'è grande differenza tra il primo e il secondo miglior modo di inserire un nodo, è molto probabile che quel nodo sia scelto e questo è un bene. Infatti, se il costo di regret è alto significa che si ha un'occasione molto vantaggiosa come prima possibilità e una molto poco vantaggiosa come seconda. Quindi, se non si sceglie immediatamente la prima si rischia di aumentare di molto l'inserimento di quel particolare nodo in quanto il veicolo che realizza la prima possibilità potrebbe decidere di muoversi su un altro nodo e lasciare come miglior inserimento del nodo la seconda possibilità, che sappiamo essere molto costosa. In base a quante possibilità si prendono in considerazione per ciascun nodo si creano vari tipi di regret, motivo per cui in genere si parla di k-regret, dove k rappresenta il numero di possibilità prese in considerazione. In questo progetto si è scelto il 2-regret. Nel caso generale si ha come costo di regret:

$$K - regret = \sum_{k=1,K} (f(k,i) - f(1,i))$$

dove con *K* si indicano tutti i *K*-migliori modi di inserire il nodo i nella soluzione e $f(k,i)$ rappresenta il costo di inserimento del nodo i con il metodo k.
L'algoritmo è quindi il seguente: all'inizio tutti i veicoli sono fermi al depot al tempo 0 (ovvero le 18:00). Dopodiché, vengono presi in considerazione tutti i nodi che ciascun veicolo può visitare arrivando nella time window del nodo destinazione (che all'inizio saranno gli stessi ma poi cambieranno) e per ciascuno di essi viene calcolato il corrispettivo regret. Se un nodo ha un solo modo di essere raggiunto allora il suo regret è il costo di inserimento in quel singolo modo. Una volta stilata la classifica dei nodi con più regret, ne viene selezionato uno e viene aggiunto alla soluzione. In vista di una possibile implementazione multi-start, si è deciso che la scelta di quale nodo inserire nella soluzione abbia un elemento casuale. In particolare, si sceglie quale nodo inserire nella soluzione dai 3 nodi con regret più alto. Ripetendo questo processo finché ci sono nodi ancora da servire si arriverà a trovare una soluzione iniziale.
Il motivo per cui si è scelto di utilizzare il regret per costruire la soluzione iniziale è legato al fatto che la letteratura lo indica come uno dei metodi più efficaci per trovare una soluzione iniziale ammissibile, elemento determinante per una local search su questo problema dato che anche solo trovare una soluzione iniziale ammissibile può essere molto difficile ([1]). Il suo design, infatti, mira a creare una soluzione iniziale ammissibile piuttosto che a minimizzarne il costo e in questo framework (che ha un algoritmo dedicato al solo miglioramento della soluzione) si è pensato fosse una scelta più adatta.
Se non ci sono nodi servibili dai veicoli in un dato instante, l'algoritmo mette i fattorini ad aspettare al depot, in attesa che uno di loro riesca a servire un qualche nodo. Se ciò non avviene, allora l'algoritmo concluderà che non è possibile creare una soluzione iniziale. Ad ogni modo, se non viene trovata immediatamente una soluzione iniziale, vengono effettuati altri 2 tentativi onde evitare di lasciare l'algoritmo senza una soluzione su cui lavorare. La possibilità che un secondo o un terzo tentativo dell'algoritmo trovi una soluzione ammissibile è legata al fatto che essendo presente nell'algoritmo un elemento casuale, magari nei tentativi precedenti sono state fatte scelte casuali sfortunate che hanno portato al fallimento della costruzione di una soluzione iniziale. Se anche dopo questi tentativi non si riesce a trovare una soluzione iniziale è consigliabile alzare il numero di veicoli per quella istanza del problema.

Algoritmo di miglioramento swap-based

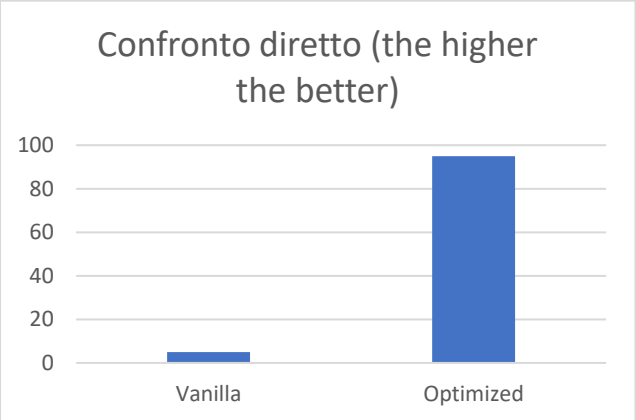
Il vicinato scelto per l'algoritmo di miglioramento della soluzione iniziale è quello derivato dallo scambio di ogni possibile nodo visitato da un veicolo v con un nodo visitato da un qualsiasi altro veicolo. In ottica di realizzare un algoritmo prima di tutto funzionante non si è pensato a particolari ottimizzazioni, come inserire i due nodi scambiati nel posto migliore di ciascun tour o aggiustare le attese al depot per far sì di ammortizzare (o addirittura rimuovere) eventuali penalità legate alle time window. Per cui se due veicoli scambiano due nodi, il nodo proveniente dal primo veicolo sarà posizionato al posto del nodo del secondo e viceversa. Dopodiché si cercherà di capire se la soluzione così creata è ammissibile oppure no e nel caso lo sia, quanto vale l'incremento o il decremento della funzione obiettivo. Una volta raccolti i dati di tutti i possibili scambi vantaggiosi, si ordinano per guadagno decrescente e si effettuano tutti gli scambi possibili (in maniera greedy) facendo sì che non vengano effettuati due scambi sullo stesso veicolo. Questo perché se si eseguisse solo lo scambio più conveniente, tutti gli scambi provenienti dagli altri veicoli sarebbero ancora presenti l'iterazione successiva, venendo quindi ricalcolati. Per cui per accelerare il processo di ottimizzazione, più di uno scambio è accettato ad ogni iterazione, a patto che riguardi veicoli sempre diversi tra loro. Una volta che non ci sono più scambi vantaggiosi disponibili, l'algoritmo si arresta e restituisce la soluzione attualmente disponibile.

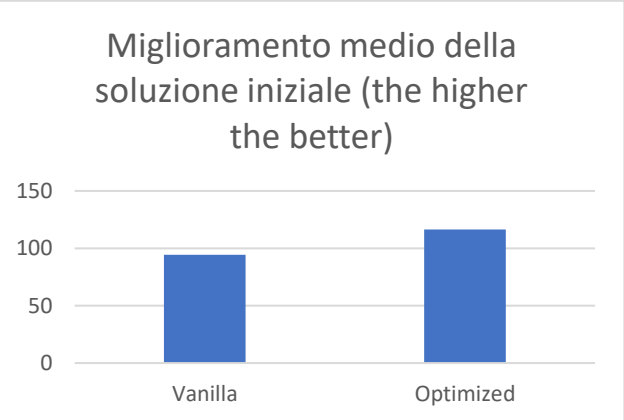
Dataset presi in esame, implementazione python e preprocessing dei dati

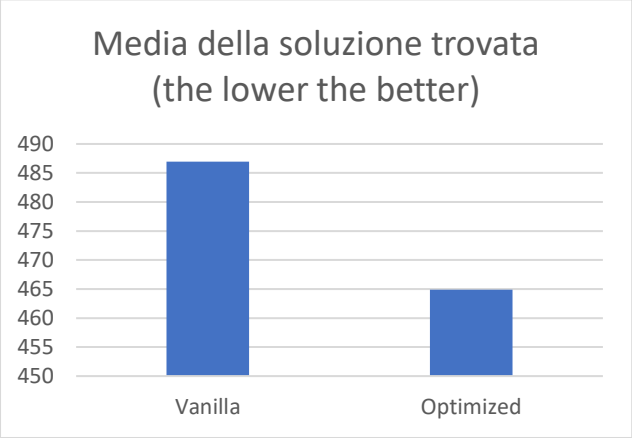
I dataset presi in considerazione sono principalmente due. Il primo è quello fornito dal docente per svolgere il primo homework del corso e il secondo è [2]. La scelta di questi dataset è legata per lo più alle dimensioni del problema, che devono essere per lo meno realistiche al contesto di questo progetto: il dataset fornito dal docente contiene istanze di grafi con dimensioni molto variabili (dai 16 ai 480 nodi) mentre del [2] sono state prese in considerazione solo istanze da 200 nodi a causa dell'eccessivo costo computazionale legato a problemi di dimensione maggiore. L'implementazione dell'algoritmo vanilla è contenuta nel file "Vanilla_Local_Search.py". Ogni implementazione di ogni algoritmo proposto contiene un preprocessing al grafo del dataset preso in considerazione. In primis viene ricalcolato il peso di ogni singolo arco prendendo in considerazione i vincoli del problema (come descritto sopra). In seguito, vengono fissate delle nuove time windows ad ogni nodo in maniera tale da rispettare la distribuzione considerata nel nostro contesto. Queste modifiche vengono salvate in un nuovo file con lo stesso nome del dataset iniziale ma con estensione .graph. Le informazioni ausiliarie come il numero di veicoli considerati, quale nodo è il depot e la capacità dei veicoli sono contenuti in un secondo file con lo stesso nome del dataset considerato ma con estensione .info. Questi file sono quelli che verranno presi in considerazione dagli algoritmi dalla seconda esecuzione in poi. Modificando questi file si modifica il problema su cui l'algoritmo lavora. Nel caso in cui non si trovi facilmente una soluzione iniziale al problema è possibile modificare il file .info corrispondente per aumentare il numero di veicoli a disposizione.
Ogni volta che un algoritmo è eseguito controlla che la soluzione che ha appena trovato sia la migliore tra tutte quelle che abbia mai creato. Per fare ciò, la miglior soluzione trovata fino a quel punto di ciascun dataset è memorizzata nella cartella "best_solutions". Quando l'algoritmo termina controlla il valore della soluzione appena trovata con quello presente in questa cartella e se è migliore sostituisce i corrispondenti file, salvando valore della funzione obiettivo, la lista dei tour dei veicoli e un'immagine che rappresenta la soluzione.

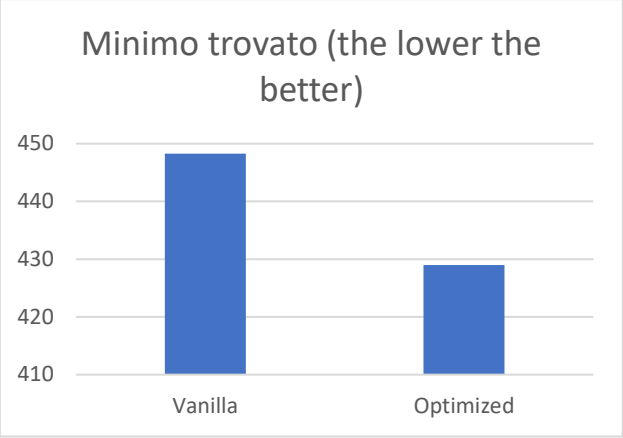
Primo miglioramento: ottimizzazione tempi di attesa

Durante l'implementazione dell'algoritmo vanilla ci si è resi conto che tantissimi scambi erano possibili ma producevano grandi perdite nella funzione obiettivo. Il motivo era legato alle penalità: se uno scambio è possibile ma uno dei due nodi ha una time window troppo anticipata rispetto all'altro, l'algoritmo finiva per assegnargli un tempo di consegna molto vicino all'apertura della time window che però fa arrivare il veicolo troppo in anticipo al nodo successivo, causando una grande penalità che rende lo scambio non conveniente anche se magari la mossa risulterebbe efficace. Per questo motivo, prima di calcolare il guadagno o la perdita della mossa è stata introdotta una funzione che se rileva delle penalità nella soluzione passata, fa in modo di far aspettare il veicolo al depot (dal momento che aspettare al depot è gratis) per ritardare al massimo possibile la consegna al nodo successivo e minimizzare quindi le penalità. In questo modo emerge la bontà della mossa e quindi emergono nuove mosse vantaggiose.
Per testare la bontà del cambiamento è stato implementato nel file "Comparison_optimized_and_vanilla" un algoritmo che sviluppa una soluzione iniziale su un dato dataset e cerca di migliorarla prima con l'algoritmo vanilla e poi con l'algoritmo ottimizzato. I risultati sono riportati nel grafico sottostante (riferiti al dataset 2l_cvrp_0/E036-11h.dat, ovvero un problema di normali dimensioni per il nostro contesto):





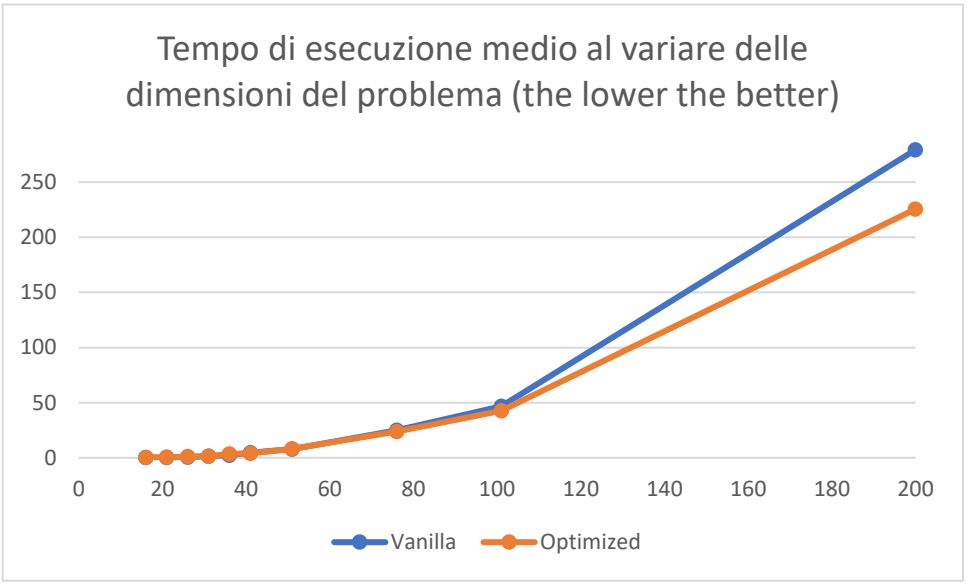




Come si evince dai grafici a parità di soluzione iniziale il metodo ottimizzato trova quasi sempre una soluzione migliore. Non stupisce il fatto che raramente sia battuto dal metodo vanilla in quanto, essendo che anche il miglioramento della soluzione è eseguito da un algoritmo greedy, la scelta di una mossa ottenuta dal metodo ottimizzato può rivelarsi sfortunata solo alla fine dell'ottimizzazione in quanto potrebbe aver indirizzato la local search su un minimo locale solo apparentemente migliore a quello intrapreso dal metodo vanilla. Ad ogni modo, è evidente come il metodo ottimizzato sia in grado di trovare soluzioni migliori al metodo standard, sia in media sia in assoluto. L'algoritmo ottimizzato è implementato nel file "Optimized_local_search.py".

Tempi di esecuzione

In seguito, vengono riportati i tempi di esecuzione dell'algoritmo vanilla e ottimizzato al variare delle dimensioni del problema (Intel Core i7 7700, 16Gb ram):



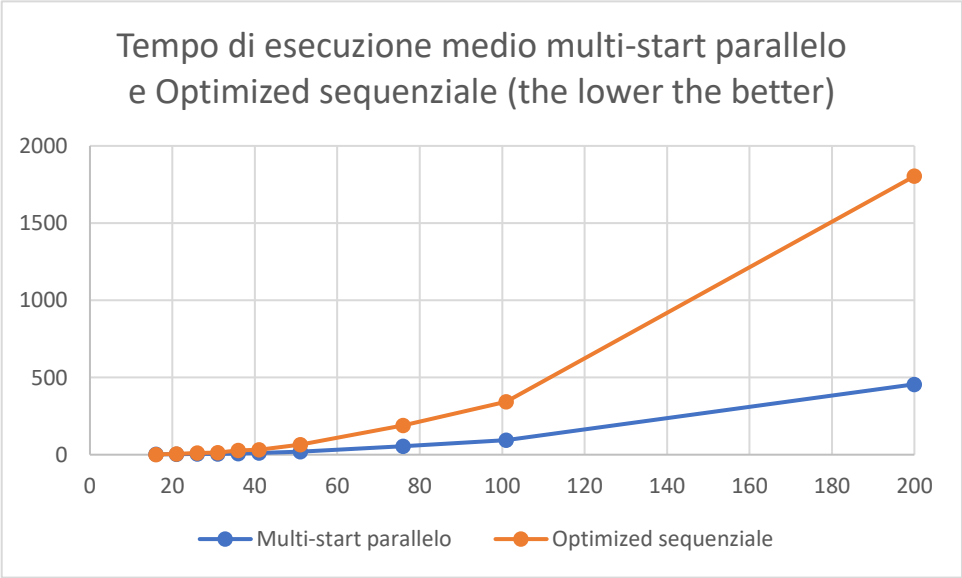
Come si può notare, l'algoritmo vanilla è più veloce su dataset più piccoli a causa dell'overhead aggiuntivo che l'algoritmo di ottimizzazione ha introdotto. Tuttavia, per dimensioni maggiori del problema questa tendenza si inverte: pagare poco di più un'iterazione si rivela vantaggioso dato il miglioramento più rapido della soluzione, che si assesta più velocemente su un minimo locale che è anche di maggior qualità rispetto all'algoritmo vanilla. L'aumento computazionale dovuto all'ottimizzazione è di un fattore di $O(t)$ dove t è la lunghezza media di un tour. Questo significa che l'overhead introdotto è un termine di grado inferiore rispetto a quello dominante (che è $O(n^2)$, dove n è il numero di nodi del grafo introdotto dalla local search).

Controllo della correttezza della soluzione

Il controllo della correttezza della soluzione è realizzato nel file “step_by_step_debug.py”. Il programma crea un report completo dell’esecuzione dell’algoritmo ottimizzato (sia per quanto riguarda la costruzione della soluzione sia per quanto riguarda l’ottimizzazione) nella cartella “runs”. La sottocartella corrispondente ha lo stesso nome del dataset scelto per il test. Si consiglia di ridirigere lo standard output su file per poterlo studiare attentamente in un secondo momento.

Strategia di multi-start parallela

Alla luce dei tempi di esecuzione, si è provato a ottimizzare l’esecuzione dell’algoritmo tramite la parallelizzazione di alcune sezioni. Tuttavia, questo non è stato possibile a causa dell’interprete python utilizzato per lo sviluppo (ovvero l’interprete base di python v3.6.0). Siccome il lock sull’interprete base può essere acquisito solo da un thread alla volta (questo è gestito dal Global Interpreter Lock, GIL), il supporto multithread non è utilizzabile e bisogna quindi ricorrere a una diversa strategia di parallelizzazione. In particolare, bisogna creare una soluzione process-based e le alternative sono due: o realizzare una parallelizzazione delle sezioni parallelizzabili tramite processi (quindi fare scale-out con i processi e con tutto ciò che comporta: di solito in questo caso i thread sarebbero molto migliori e creerebbero meno overhead ma non sono disponibili) o implementare una politica di scale-up per far lanciare più local search contemporaneamente. Si opta per questa seconda opzione dato che teoricamente è la più promettente. Nel file “vrptw.py” è quindi implementato l’algoritmo ottimizzato lanciato da 8 diversi processi. Questi elaborano una soluzione ciascuno e una volta che hanno terminato, le soluzioni finali vengono ordinate. La soluzione migliore viene tenuta in considerazione per valutare se è la migliore mai creata per il dataset considerato ed essere quindi salvata nel caso lo sia.
Il vantaggio (teorico) è che nel tempo in cui si effettuerebbe un tentativo solo vengono elaborate 8 soluzioni invece che una sola. Tuttavia, prendendo in considerazione l’overhead dovuto ai processi, agli accessi in memoria condivisa e al fatto che la randomicità introdotta fa variare il tempo di esecuzione questo non è del tutto vero. Ad ogni modo, sperimentalmente si è trovato comunque vantaggioso creare più processi in quanto il tempo impiegato da questa tecnica è comunque molto minore rispetto al creare 8 soluzioni distinte col programma sequenziale, come illustrato dal grafico:



Considerazioni

- Durante il debug step by step, nella fase di local search si vedono numerosi swap che non cambiano la soluzione obiettivo (di guadagno 0). Queste mosse possono essere sfruttate in una tabu-search o comunque in altre metaeuristiche.
- Date le dimensioni tipiche del problema in questo contesto (tra le 50 e le 65 consegne a serata) potrebbe essere considerabile un metodo di risoluzione esatto visto che stando a [1] soluzioni esatte sono calcolabili per istanze fino ai 150/200 nodi.
- Sfruttando interpreti diversi senza GIL (come IPython), potrebbe essere interessante valutare una parallelizzazione del tipo scale-out.
- Col senno di poi, il vicinato scelto potrebbe non essere stato il migliore: la soluzione iniziale crea i tour dei veicoli con delle visite al depot in determinati punti. Questi ritorni al depot non possono essere modificati in alcun modo dal vicinato scelto (non ha senso scambiare un depot per un nodo di consegna o con un altro depot). Probabilmente, se si utilizzasse un vicinato in grado di spostare/rimuovere visite ai depot, la soluzione riuscirebbe a diminuire di molto.
- Un modo interessante di estendere il problema per renderlo ancora più attinente alla realtà potrebbe essere quello di inserire un altro vicinato in grado di togliere veicoli alla soluzione in maniera tale da diminuire il numero totale di veicoli utilizzati, per dare un'indicazione anche di quanti veicoli si ha bisogno durante il turno di consegna. Il numero di veicoli entrerebbe poi nella funzione obiettivo.
- Al contrario di quello riferito da [3], spesso nell'implementazione multistart parallela anche soluzioni iniziali molto alte si sono rivelate essere tra le migliori del lotto una volta ottimizzate (talvolta creando anche minimi assoluti).

Bibliografia

[1] P. Kilby, «Multi-Vehicle Routing with Time Windows,» [Online]. Available: https://www.youtube.com/watch?v=BZA_UaX8rs8.

[2] H. A. H. J. Gehring, «A parallel hybrid evolutionary metaheuristic for the vehicle routing problem with time windows,» 1999.

[3] M. G. Olli Bräysy, «Vehicle Routing Problem with Time Windows, Part I: Route Construction and Local,» 2005.