The dynamic vehicle routing problem with stochastic customer requests and multiple delivery routes

Lorenzo Sciandra, Stefano Vittorio Porta,
Jean-François Côté

A.A. 2021-2022

Università degli Studi di Torino

Indice Argomenti

Indice argomenti

Indice argomenti

Definizione del Problema

Problema Dinamico

Matematicamente il problema è definibile con un grafo diretto simmetrico $G = (L \cup \{0\}, A)$ dove L è l'insieme dei nodi o posizioni possibili dei **clienti**, 0 è il **deposito** e A è l'insieme degli archi con associati **tempo di viaggio** t_{ij} e **costo** c_{ij} per passare dal nodo i al nodo j.
Viene definito con T l'**orizzonte temporale** ossia le ore lavorative del deposito e quindi l'orario limite dal quale può partire un **veicolo**.
R è invece l'insieme delle **richieste dei clienti** per i quali dobbiamo caricare dal deposito gli oggetti ordinati per essere consegnati. Di queste generalmente solo poche sono conosciute fin dall'inizio dell'orizzonte temporale, la maggior parte infatti diventeranno note col procedere del tempo, questo è l'aspetto **dinamico** del problema: non tutto è conosciuto a priori.

Problema Stocastico

Ogni richiesta k è caratterizzata da un **release time** r_k che è il tempo nella quale questa diventa nota e da una finestra temporale per la consegna. La finestra è un intervallo $[e_k, l_k]$ i cui estremi rappresentano rispettivamente il primo e l'ultimo possibile tempo di arrivo per servire il cliente k.
Quando eventi futuri risultano conosciuti solo dalla loro distribuzione di probabilità, il problema viene definito **stocastico**. In questo problema distribuzioni di probabilità riguardanti il numero di clienti che chiederanno il servizio, il momento in cui lo faranno, la loro posizione e le finestre temporali sono generate usando dati di scenari passati. Ad esempio è previsto che il tasso di arrivo delle richieste future sia un processo di Poisson e inter-arrivo esponenziale.

Route o Percorso

Per quanto riguarda la consegna abbiamo un numero finito e fisso M di veicoli considerati, per semplicità, con capacità illimitata. Il percorso di un veicolo inizia dal deposito, serve un certo numero di clienti e ritorna al deposito. Formalmente una **route** è una sequenza $[0, k_1, ..., k_n, 0]$ dove $1 \leq k_i \leq |R|$ sono le richieste tutte distinte e 0 rappresenta il depot. Il **costo di un percorso** è $c(p) = c_{0,k_1} + c_{r_1,r_2} + \cdots + c_{r_n,0}$, discorso analogo per la distanza del percorso.

Piano di instradamento e funzione obiettivo

Un **piano di instradamento** è un insieme di routes $\{p_1, ..., p_M\}$, una per ogni veicolo, che serve ogni cliente esattamente una volta. Il piano assegna ad ogni cliente un solo predecessore ed un solo successore all'interno del percorso. Il **costo di viaggio di un piano** è la somma dei costi delle sue routes e quindi $\sum_{i=1}^{M} c(p_i)$. Le decisioni prese durante l'esecuzione sono basate su un **piano prescelto** che evolve nel tempo. Il piano prescelto viene selezionato da una **funzione consenso** e tutti gli altri mantenuti devono essere consistenti con esso.

La **funzione obiettivo** è **gerarchica**: prima si massimizza sul numero di richieste servite e poi si minimizza sulla distanza di viaggio percorsa.

Modellazione del Problema

Dynamic Pickup and Delivery Problem with Time Windows and Release Dates

Il problema è stato modellato come un **Dynamic Pickup and Delivery Problem with Time Windows and Release Dates**. Ogni richiesta k ha quindi associata una coppia di nodi (i, j) in cui i è il deposito con il pickup del materiale da consegnare e j è il punto in cui risiede il cliente e dove dovrà avvenire quindi la consegna. Sia la raccolta che la consegna hanno associata una finestra temporale all'interno della quale l'azione dovrà avvenire e, rispettivamente si parla di $[r_k, l_k]$ per la raccolta al deposito e $[e_k, l_k]$ per la consegna al cliente.

Ottimizzazione Event-Driven

Come si fa generalmente per problemi dinamici l'**ottimizzazione è event-driven**, ossia viene performata ogni volta che si hanno nuove informazioni veicolate da un nuovo evento che si è verificato nel sistema. Quando si presenta un nuovo evento si colleziona tutto ciò che si conosce e si esegue un passo di ottimizzazione che pianifica le routes da eseguire.

Eventi

Gli eventi che vengono considerati sono:

1. si ha una nuova richiesta nota e c'è un veicolo al deposito pronto per partire: si ottimizza per cercare di servire anche la nuova richiesta
2. un veicolo ha finito una route e arriva al deposito: si ottimizza in modo da assegnargli una nuova route
3. un veicolo ha terminato il tempo di attesa al deposito: se le richieste note sono poche il veicolo ha infatti la possibilità di aspettare che ce ne siano un certo numero minimo per avere soluzione migliore e più robusta riguardo a ciò che dovrà accadere
4. una consegna è stata completata: quando un veicolo ha consegnato un pacco ad un cliente può riottimizzare il percorso rimanente

Reoptimization

Ogni volta che si verifica un evento abbiamo 3 algoritmi a disposizione:

- **Reoptimization**: classico metodo usato per risolvere problemi dinamici:

Algorithm 1: Reoptimization

Create a plan s that contains the requests known at `time` 0;
while *there is an event* $\vee$ $time = 0$ **do**
    Lock all performed `actions` in plan s;
    Add the `new_requests` to s;
    Optimize s and implement the `new_actions`;
end

Two lookahead algorithms

- **Two lookahead algorithms**: che cercano di anticipare le informazioni future per fare migliori pianificazioni. Questi usano scenari probabilistici con l'informazione stocastica campionata precedentemente descritta. Ottimizzando gli scenari si ottengono insiemi di routes, uno per ogni scenario, contenenti la gestione sia di richieste note che di probabili richieste future. Ovviamente vorremmo che un veicolo prima di partire aspetti al deposito che le richieste future previste si verifichino o meno per sapere se caricare il materiale da consegnare. I due approcci usati e uniti nel codice di Jean-François Côté sono quelli proposti in Bent e Van Hentenryck[1] e Hvattum, Løkketangen e Laporte [2] che andremo nel seguito a descrivere.

Esempio

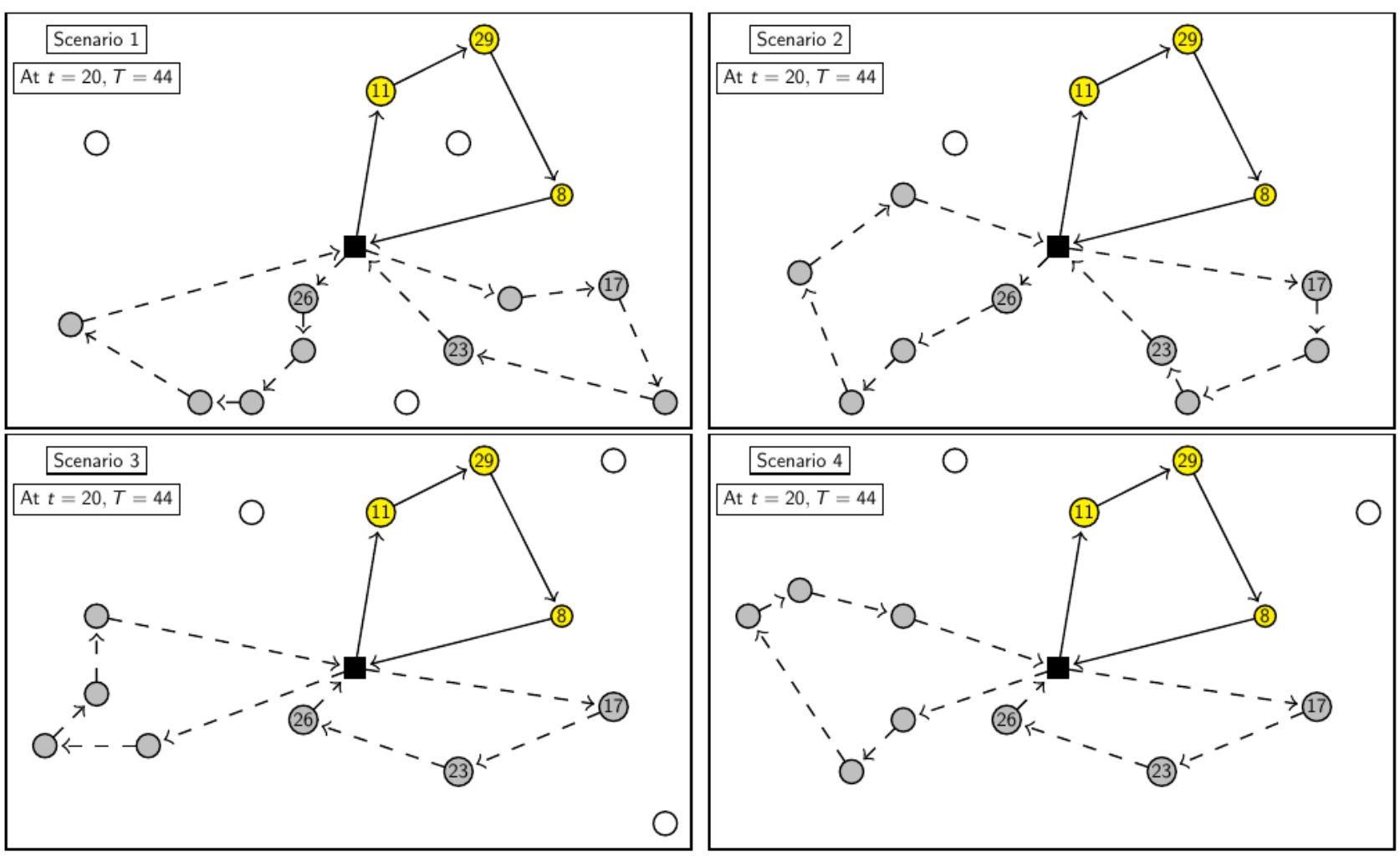


Figure 1: Scenari Ottimizzati

Funzione Consenso

Ovviamente, nel processo di ottimizzazione, il numero di scenari generati è variabile con la seguente considerazione: più questo numero è grande più la soluzione sarà costosa, ma più sarà affidabile in quanto mi permetterà di esplorare meglio la randomicità del problema e avere piani prescelti robusti, che affrontano meglio il futuro. Nello specifico una volta ottimizzati tutti gli scenari bisognerà scegliere quale usare, o meglio, quali azioni andare ad effettuare nel mio piano reale. Per far questo si usano delle funzione consenso all'interno dei due algoritmi prima menzionati che *"guardano nel futuro"* e che ora andremo a spiegare meglio.

SBPA Algorithm

Nel paper di Bent e Van Hentenryck[1] si propone di scegliere come scenario finale quello che presenta il migliore score ottenuto con una certa **funzione consenso**, in questo caso mediante una **Route Similarity**. Entrando nei dettagli ad ogni piano viene associato come score la somma delle volte in cui una sua route compare uguale anche nei piani ottenuti per altri scenari. In questo caso con *"compare uguale"* intendiamo proprio che devono risultare le stesse richieste e nello stesso ordine. Non vengono considerate nel conteggio le route che contengono almeno una richiesta futura e per le quali dobbiamo attendere.

Scenario-Based Planning Approach Algorithm

Algorithm 2: Scenario-based planning approach

Create a plan s that contains the requests known at `time` 0;
while *there is an event* $\vee$ $time = 0$ **do**
    Lock all performed `actions` in plan s;
    Add the `new_requests` to s;
    Create a scenario set Ω of `fictive_requests`;
    for *each scenario* $w \in \Omega$ **do**
        $s_w = s \cup w$;
        Optimize s_w;
    end
    Implement the plan s_w having the highest score $f(s_w)$;
end

Route Similarity

Figure 2: Route Simalirity

Approccio Miope - Nuove Funzioni Consenso

Questo criterio, mostrato in fig 2, risulta però **miope** e non riesce a comprendere il trend reale che si sta verificando: sceglie sempre la strategia che richiede impegno minimo. Nello specifico quando i piani per i vari scenari sono molto diversi sarà scelto quello con meno routes e tutte molto corte, dato che avranno più probabilità (dal momento che sono composte da poche mosse) di presentarsi in altri piani. Per questo Jean-François e i suoi collaboratori hanno proposto due diverse **funzioni consenso**: **Assigment Similarity** e **Edit Distance**.

Assigment Similarity

Lo score di ogni piano è il numero di volte in cui la coppia (richiesta, numero di route) nel piano si presenta allo stesso modo in altri piani:

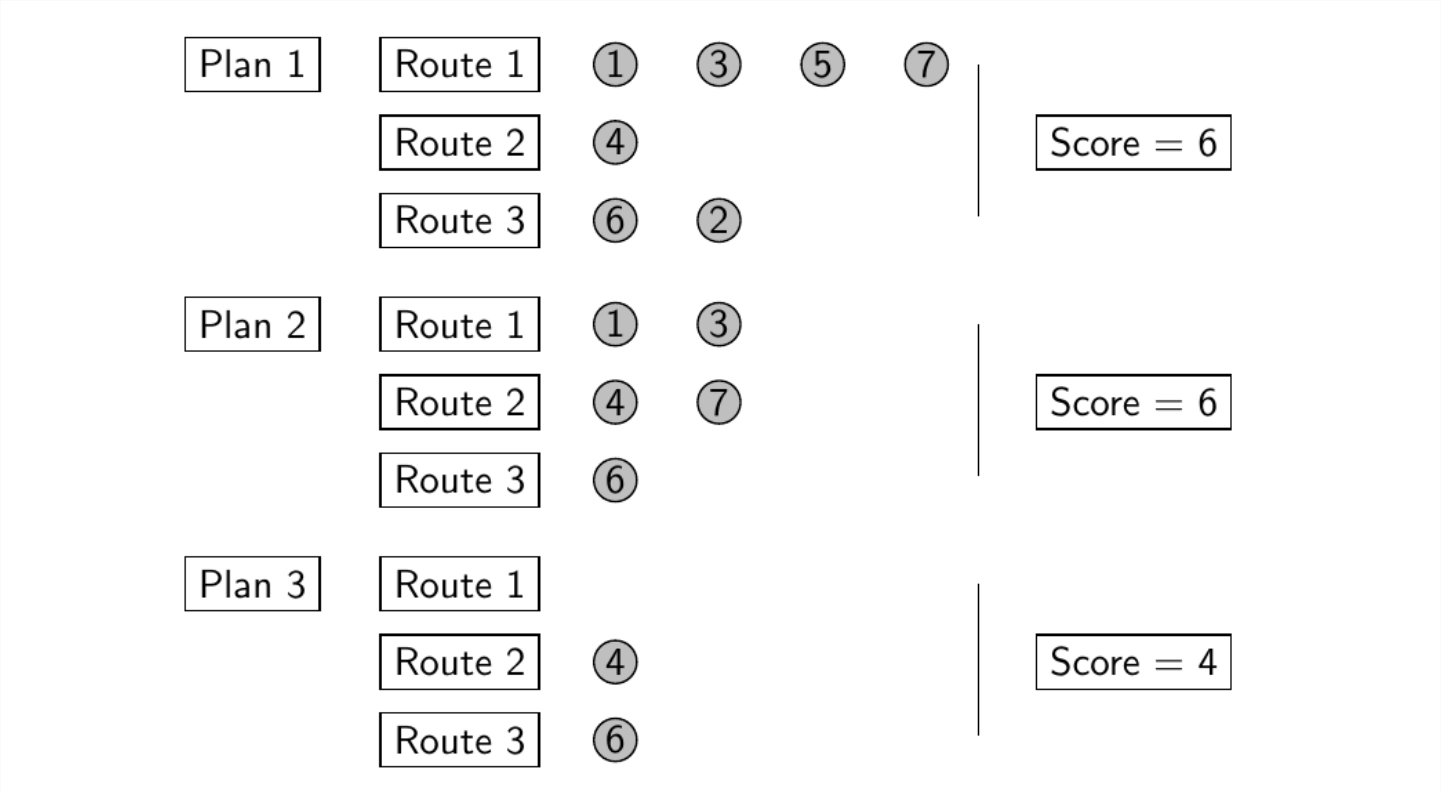


Figure 3: Assignment Similarity

Edit Distance

Lo score di ogni piano è la somma dei numeri di cambiamenti che necessita per essere uguale agli altri piani. In questo caso, a differenza dei precedenti, più lo score è basso meglio è:

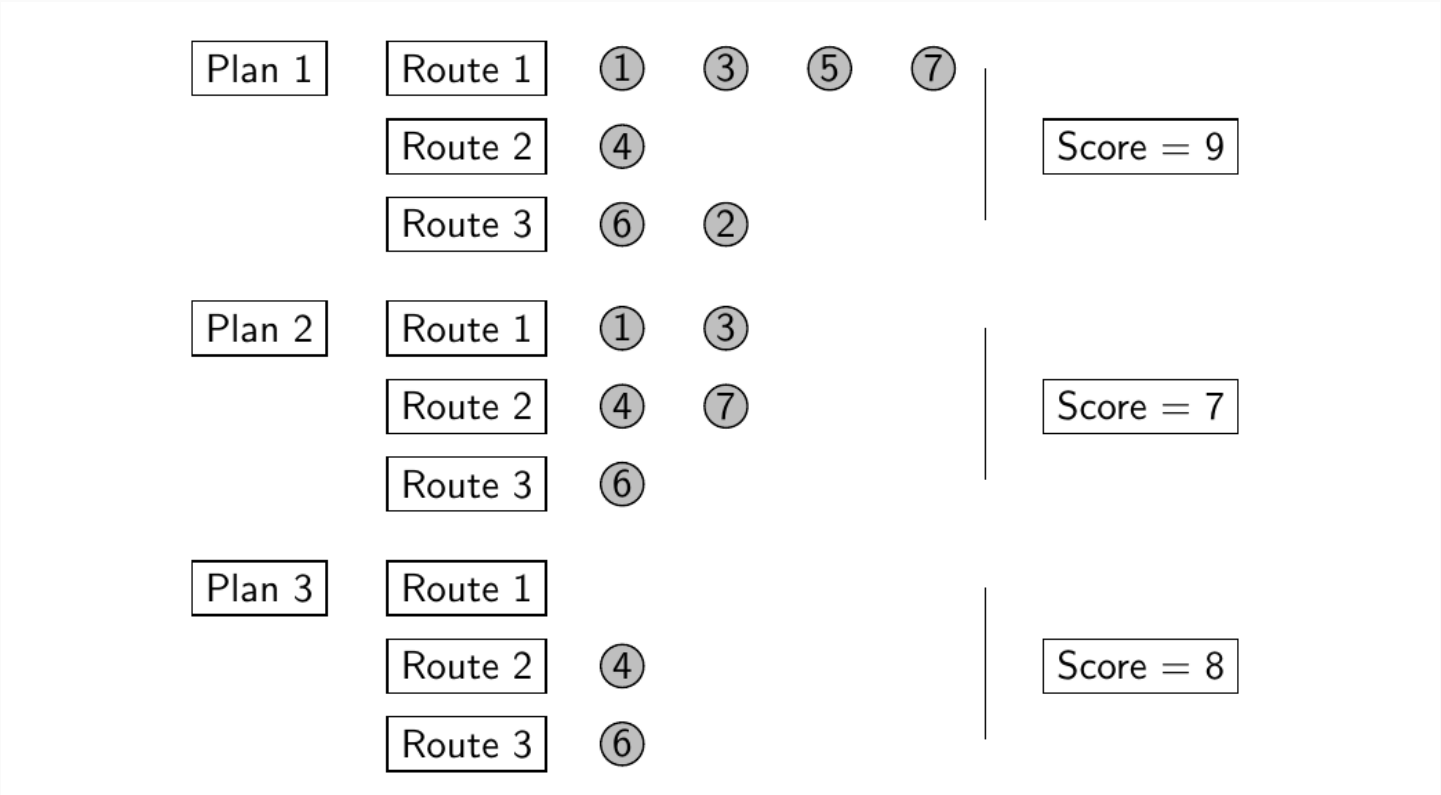


Figure 4: Edit Distance

Problemi Funzioni Consenso

Queste 3 diverse funzioni consenso possono non essere concordi sulla scelta del piano da eseguire, come le combiniamo?
Una possibile soluzione consiste nel scegliere una delle funzioni consenso ed usare le altre in caso di pareggio di score tra piani diversi. Nelle immagini mostrate verrebbe scelto il piano 2 in quanto ha Assignment Similarity, ma minore Edit Distance.

Un altro problema che vorremmo risolvere che interessa il SBPA è che ogni piano risulta troppo specializzato per lo scenario previsto, il che ovviamente può non presentarsi.Si vorrebbe definire un criterio che non risulti dipendente da un solo scenario considerato, ma che sia mediamente buono. Per far questo abbiamo bisogno del Branch & Regret.

Branch & Regret

L'approccio proposto in Hvattum, Løkketangen, and Laporte [2] è simile al precedente SBPA, ma prevede alcuni passi addizionali a livello algoritmico. Nello specifico una volta ottimizzati e risolti tutti gli scenari ci si assicura, forzandole, che ogni richiesta conosciuta e reale sia servita nello stesso intervallo di tempo in tutti i piani e che in ogni piano ogni veicolo visiti per prima la stessa richiesta. Entrando nei dettagli il B&R prende le richieste non fissate reali e va a vedere quale sarebbe la media dei costi in tutti gli scenari se ognuna di questa (una per volta) venisse gestita nella prossima route o in una delle successive. Si sceglie l'alternativa migliore e si fissa in tutti gli scenari. Una volta fatto ciò per tutte le richieste l'algoritmo viene rieseguito per fissare la prossima visita di ogni veicolo.

Branch & Regret Algorithm

Algorithm 3: Branch-and-Regret

$L = \{$ set of `unfixed_requests` $\}$;

while *!empty(L)* **do**

  Select a request $k \in L$;

  Visit k inside the next time interval and solve all scenarios;

  Visit k after the next time interval and solve all scenarios;

  Fix k to the least cost alternative;

  $L = L \setminus \{k\}$;

end

Repeat for fixing the next visit of each `vehicle`;

Diverso Schema di Branch-1

Nel B&R la fase di **Branch** consiste nel scegliere se una certa richiesta vada gestita ora o in futuro, mentre la fase di **Regret** consiste nel fissare l'alternativa con il costo più conveniente. Per adattarlo all'esigenze del problema in questione Jean-François e i suoi colleghi hanno proposto un diversa fase di braching con schema:

- **go_now**: se la richiesta k farà parte di una route che parte ora
- **wait**: se la richiesta k sarà assegnata ad una route che partirà dopo
- **reject**: se la richiesta k non sarà servita

Un possibile albero generato dal B&R può essere il seguente:

Diverso Schema di Branch-2

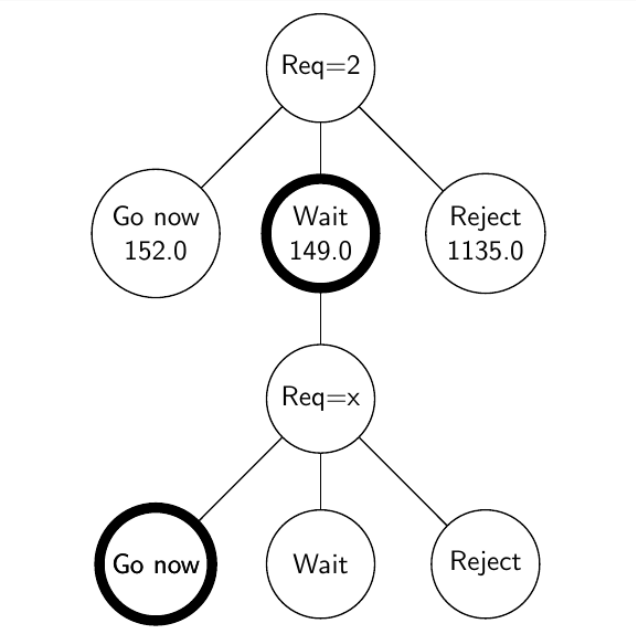


Figure 5: Branch & Regret View

Tra tutte le richieste non servite viene scelta la $k = 2$ che sarà gestita con l'azione che presenta costo minimo tra tutti i piani, in questo caso **wait**. La richiesta immediatamente successiva x può invece portare ad un costo complessivo minimo se gestita immediatamente. Il processo di ottimizzazione procede in questo e viene stoppato quando tutti i piani implementano le stesse alternative di gestione di ogni richiesta reale. A questo punto viene selezionato un piano con una delle 3 funzioni consenso prima definite.

Branch And Bound

Scopo tesina

Lo scopo della nostra tesina è stato quello di verificare l'idea proposta da Jean-François e dai suoi colleghi. Nello specifico abbiamo implementato un algoritmo esatto, un Branch & Bound, in grado di trovare il miglior percorso all'interno dell'albero delle richieste ad ogni evento. Così facendo è possibile verificare quando, in che modo e in quali contesti la soluzione individuata dalla loro implementazione si discosti dalla migliore. L'implementazione è avvenuta in C++ estendendo il progetto fornitoci da Jean-François.

Basi Implementative

Ovviamente per poter sviluppare il nostro algoritmo si è resa necessaria l'implementazione di una nuova classe dedicata all'ottimizzazione mediante Branch & Bound, basandoci anche su parte delle API già presenti nel progetto, senza le quali sarebbe stato necessario un lavoro molto più a basso livello. In questo caso invece, ci siamo potuti concentrare su strutture dati e algoritmi ad alto livello, permettendo uno sviluppo, testing e debugging più rapido e semplice, anche grazie all'adozione di linee guida e pattern suggeriti anche da Jean-François.

Ottimizzatore

L'algoritmo viene richiamato all'interno di una funzione wrapper `Optimize()` spesso implementata da Jean-François nella sua libreria per guidare tutto il processo ottimizzatorio e di simulazione a partire dall'istanziazione iniziale delle variabili necessarie per proseguire con la gestione di ogni evento lungo tutto l'orizzonte temporale e la stampa finale dei risultati. Esattamente come prima indicato a livello teorico, al verificarsi di ogni evento saranno generati n scenari e ottimizzati. L'ottimizzatore sfrutta un'implementazione dell'ALNS [3] sviluppata da Jean-François che effettua un numero di iterazioni di distruzione e ricostruzione parziale dei percorsi generati in modo da cercare di ottimizzarli. Il numero di iterazioni è specificato da un parametro impostato prima dell'inizio della ricerca di soluzioni.

Branch & Bound

Abbiamo realizzato il Branch & Bound sia in versione iterativa che ricorsiva e a loro volta entrambe sia in versione Best-First che Depth-First. Tutte le versioni sono void e la migliore soluzione intera si troverà, alla fine dell'esecuzione, nello spazio di memoria puntato dal puntatore passato in input. Per spiegare correttamente il Branch & Bound dividiamolo nelle sue parti fondamentali che compongono l'algoritmo all'interno del file `BranchAndBoundSimulation.cpp`.

Branching-1

Preso un certo nodo dell'albero delle decisioni i suoi figli vengono ottenuti, all'inizio della procedura, con la chiamata a `GetNextActionDecisions()`. Quest'ultima funzione andrà ad assegnare al vettore `current_decisions` passato come parametro una triplice versione della decisione selezionata una per ogni tipo di azione che possiamo performare su di essa: *go_now*, *wait*, *don't deliver*. Se questo vettore è vuoto vuol dire che abbiamo già analizzato tutte le possibilità per tutte le richieste reali per ora note e possiamo terminare, aggiornando eventualmente la miglior soluzione trovata con quella corrente che stiamo considerando.

Branching-2

Nel caso invece in cui ci sia ancora almeno una richiesta reale da analizzare andiamo ad ottimizzare ogni scenario forzando in ognuno di essi tutte le decisioni precedentemente prese insieme a quella nuova corrente da considerare. E' importante quindi mantenere un vettore di `working_decisions` che vada aggiornato mano a mano che una richiesta è gestita. Si andranno quindi a generare i nuovi nodi, figli della `current_decision` che saranno immediatamente esplorati nel caso Depth-First ed inseriti nella lista dei nodi da esplorare nel caso Best-First.

Bounding

La fase di bounding permette di evitare parte dell'esplorazione dell'albero. Nello specifico, prima di fare la chiamata ricorsiva su un nodo, o equivalentemente prima di espanderlo ed inserire i figli nella lista nella versione iterativa, ci si accerta che abbia un costo medio su tutti gli scenari minore dell'ottimo finora trovato e possa quindi migliorare la nostra soluzione corrente.

Best-First vs Depth-First

L'albero generato dal B&B può essere esplorato in due modi diversi, il cui controllo avviene tramite un parametro globale dell'ottimizzazione che permette di scegliere una **Depth-First** oppure una **Best-First**. Entrambe sono realizzate con una lista di nodi con la sola differenza che il primo approccio prevede di usarla in maniera LIFO, mentre il secondo inserisce ed estrae i nodi da esplorare in accordo con il loro valore di costo medio. C'è da precisare il fatto che nel caso nella versione ricorsiva questo non porta ad un approccio Best-First globale, complessivo di tutte le richieste nell'albero, come invece accade per la versione iterativa. Quello che si origina è invece una Best-First a livello in cui ogni padre decide prima di esplorare i figli più promettenti. Questo verrà chiarito con un paio di esempi di esecuzione.

Risultati

Alberi Graphviz

Per accorgerci dell'eventuale differenza di piano ottimo restituito dal Branch & Regret rispetto a quello reale trovato con il Branch & Bound abbiamo deciso di stampare, durante la generazione dell'albero, comandi Graphviz che ci permettessero di visualizzare graficamente la situazione.

Per fare questo abbiamo definito una classe di supporto `BBNode` (Branch&Bound Node) che contiene tutte le informazioni rilevanti riguardo alla posizione di un nodo nell'albero.

BBNode

Precisamente ogni BBNode contiene:

- ID univoco del nodo
- ID della richiesta considerata
- ID del nodo genitore
- Tipo di decisione che il nodo rappresenta
- Costo raggiunto per poter svolgere la decisione correlata al nodo
- Ordine di visita del nodo
- Puntatori ai BBNode figli

Una volta generato completamente tutto l'albero, che sarà un vettore di `BBNode`, questo viene post-processato aggiungendo ad ogni nodo informazioni booleane riguardo l'appartenenza al percorso ottimo e/o a quello definito dal B&R. A questo punto ciclando sul vettore si stampa per ogni nodo la dichiarazione del nodo stesso e l'arco che lo collega univocamente al padre.

Path con sole azioni di wait

Prima di mostrare e confrontare i risultati del B&B da noi implementato con il B&R di Jean-François è necessario accennare al fatto che nel codice l'algoritmo B&B, esattamente come il B&R, non viene chiamato nel caso in cui il piano preveda sole wait. Questa situazione che a primo avviso potrebbe sembrare rara capita invece circa il 90% delle volte per semplici istanze. In questi casi il costo del B&R coincide con il B&B e le differenze tra i due algoritmi è pressocchè nulla. Vediamo ora invece le differenze tra i due algoritmi nel caso in cui la sequenza di azioni non sia composta da sole wait. Mostriamo ora delle immagini di esempio per evidenziare diversi tipi possibili di esecuzione:

BB e BR con stessa path

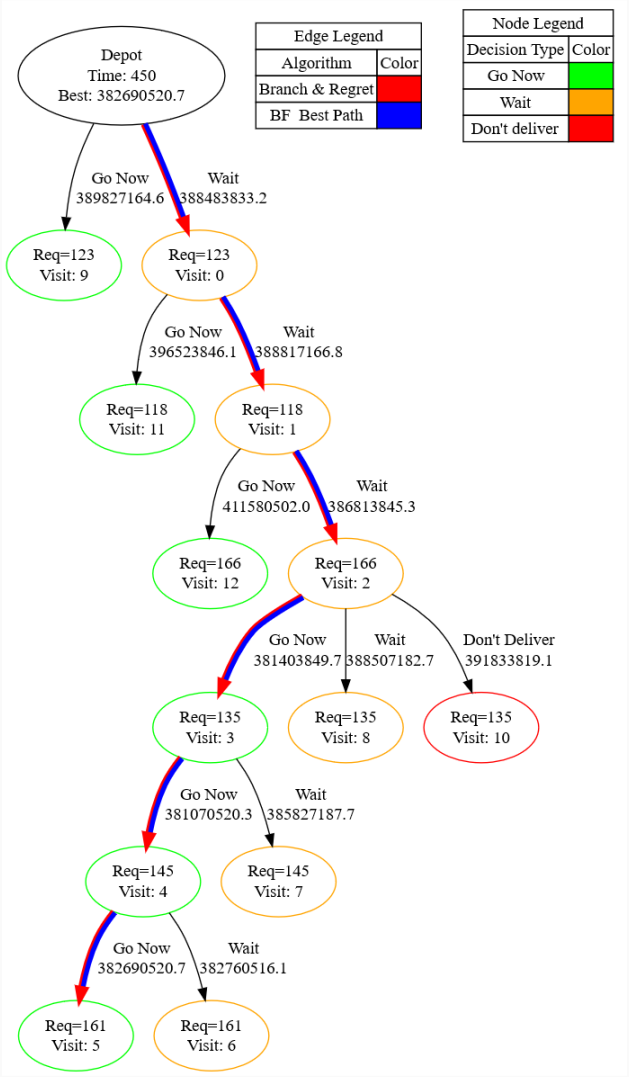


Figure 6: B&B Best-First e B&R scelgono lo stesso percorso

BB Best-First vs BR

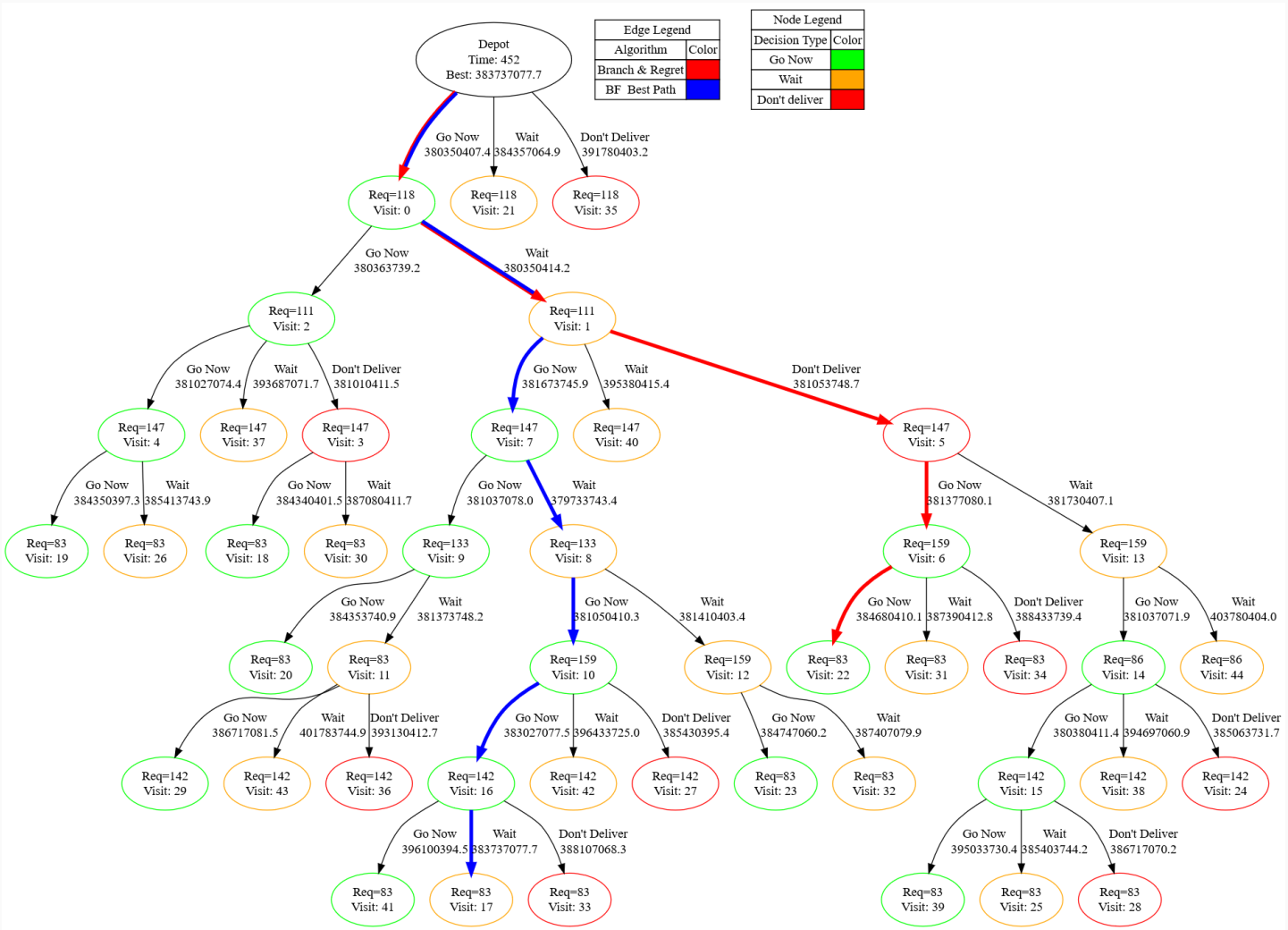


Figure 7: B&B Best-First trova un percorso migliore del B&R

BB Best-First doppio percorso ottimo

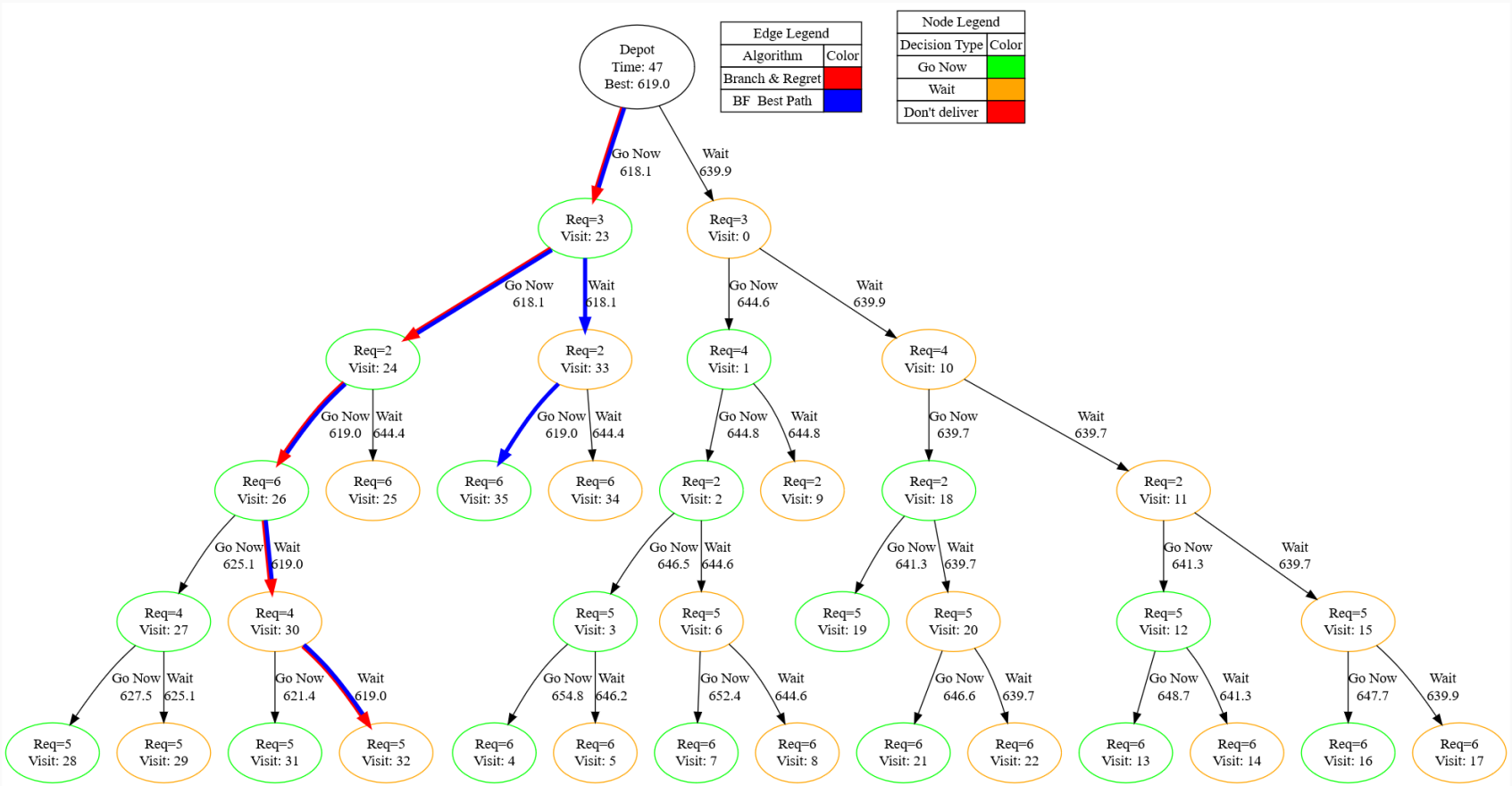


Figure 8: B&B trova due percorsi ottimi di uguale costo

BB Depth-First vs BR

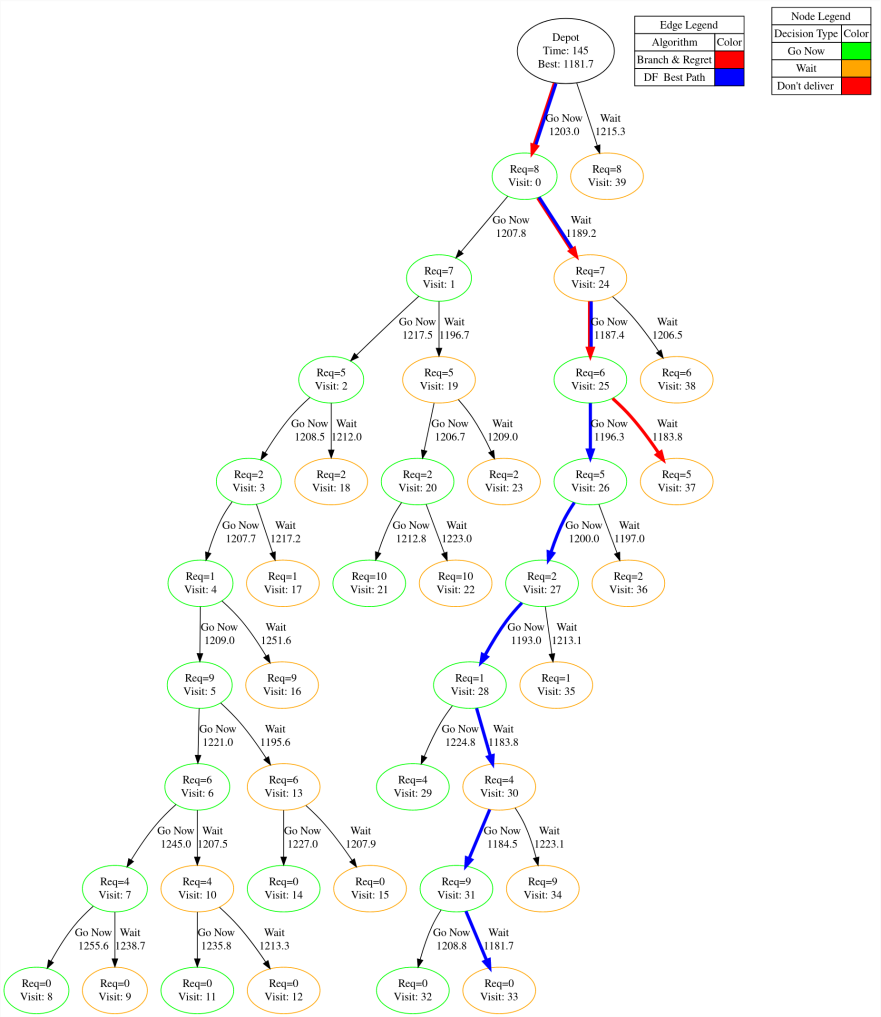


Figure 9: B&B Dept-First trova un percorso migliore del B&R

Dataset

Le istanze che abbiamo usato per fare i test fanno parte del benchmark disponibile sul sito di Jean-François. Le istanze del problema sono classificate in base a molte delle loro caratteristiche che vengono messe in successione nel nome del file stesso, esattamente come descritte in Voccia et al.[4].

Prendiamo come esempio di problema `TWdf_C_1_hom_1_actual.txt` e proviamo a descrivere le sue caratteristiche.

- **TWdf**: questo attributo caratterizza le finestre temporali (*TimeWindow*) della richiesta. Nel nostro caso '**d**' significa che la finestra ha una *Deadline* di un certo valore temporale, mentre '**f**' significa che a partire dal release time r_k della richiesta, la finestra inizia ad un istante fissato (*Fixed*) nel futuro. Questi due ultimi attributi possono essere sostituiti con **h** se l'inizio della finestra temporale è campionato da una distribuzione uniforme sugli istanti rimanenti nell'orizzonte temporale T. Si può usare invece **r**, per rilassare il valore precedente **h**, e avere sempre l'inizio della finestra temporale campionato, ma senza il vincolo che questo debba coincidere con l'inizio di una certa ora

- **C_1**: questo attributo caratterizza le aree geografiche di localizzazione dei clienti. Il valore può essere **C** se sono raggruppate (*Clustered*), **R** se sono disperse in modo casuale (*Random*) o **RC** se sono sia disperse in modo casuale che raggruppate. Per ogni area geografica, sono stati creati nove datasets con 100 posizioni dei clienti, indicizzati da 1 a 9, le cui informazioni sono state campionate da altri datasets noti in letteratura
- **hom_1**: rappresenta il tipo di tasso di arrivo delle richieste, in questo caso omogeneo (*Homogeneous*). Questo può essere sostituito con **het** se il tasso di arrivo è eterogeneo (*Heterogeneous*). Il numero x che segue il tipo di tasso rappresenta il tasso di richieste che arrivano, nello specifico si avranno $x \cdot 0.1$ richieste per minuto

- **actual**: il tipo di realizzazione delle richieste che può essere effettivo, come in questo caso, oppure campionato (**sampled**)

Sono state da noi considerate 810 istanze eseguite sia con B&B che con B&R in modo da tenere in considerazione una buona varietà di caratteristiche. Abbiamo preso tutti e i tre tipi principali **TWdf**, **TWh**, **TWr** con varianti **C**_X, **R**_X e **RC**_X con $1 \leq X \leq 9$. Di ognuno di questi tipi di istanze è stato fissato l'attributo `hom_1`, strettamente correlato con la complessità di calcolo necessaria per la risoluzione, e l'attributo `actual`.

Tempo di Calcolo - `runner.py`

La computazione è stata eseguita tramite uno script Python 3.10 (`runner.py`) sia sul Branch&Bound iterativo con esplorazione Best-First, sia sul programma di Jean-François, per un totale di 1620 esecuzioni. L'esecuzione è avvenuta su un calcolatore con le seguenti caratteristiche:

- Processore AMD Ryzen 7 5800X (4.6GHz)
- 32GB di RAM DDR4 (3600MHz)

Dal momento che i programmi sviluppati non sono stati progettati per sfruttare le potenzialità di un processore multi-core, l'unico fattore limitante è stata la frequenza del processore. Il tempo di calcolo complessivo per il B&B è di `3443.94` secondi (circa 57 minuti), mentre il B&R ha richiesto circa il doppio: `6571` secondi (circa `110` minuti). Dopo ogni computazione i risultati prodotti dai programmi vengono salvati in file `.txt`

Creazione dei risultati - `extractor.py`

Una volta ottenuti i file di output un altro script (`extractor.py`) utilizza espressioni regolari per leggere i files e comprimere le uniche informazioni per noi utili presenti nelle stampe di output in un nuovo file `.csv`. Il risultato di questa fase ci permette di passare dai 1620 files di output, uno per ogni run, ad una sola tabella `csv` per il B&B e una sola per il B&R, con le seguenti colonne:

- **Problem**: Il nome del problema risolto
- **Instance**: Quale delle 30 istanze del problema è stata analizzata
- **Cost**: Il costo della migliore soluzione identificata
- **Dist**: La distanza totale percorsa da tutti i mezzi
- **Time**: Il tempo (in secondi) richiesto per completare la computazione
- **Events**: Il numero di eventi che si sono considerati
- **Skipped**: Il numero di eventi/piani saltati perché composti solo da attese (wait)

Plot - `analyzer.py`

Mostreremo ora alcuni grafici ottenuti da un terzo e ultimo script (`analyzer.py`) che attraverso le librerie **pandas**, **numpy** e **matplotlib**, ci ha permesso facilmente di confrontare i risultati compressi dallo script precedente. Nei grafi che seguono ogni punto rappresenta la media calcolata su tutte le 90 istanze con le prime due caratteristiche del nome coincidenti. Ogni punto sarà plottato mostrando esplicitamente il valore dell'ordinata corrispondente con associata deviazione standard.

Tutti i risultati che saranno mostrati prevedono l'esecuzione di 100 iterazioni di ALNS, 30 scenari generati e 10 veicoli.

Means Events

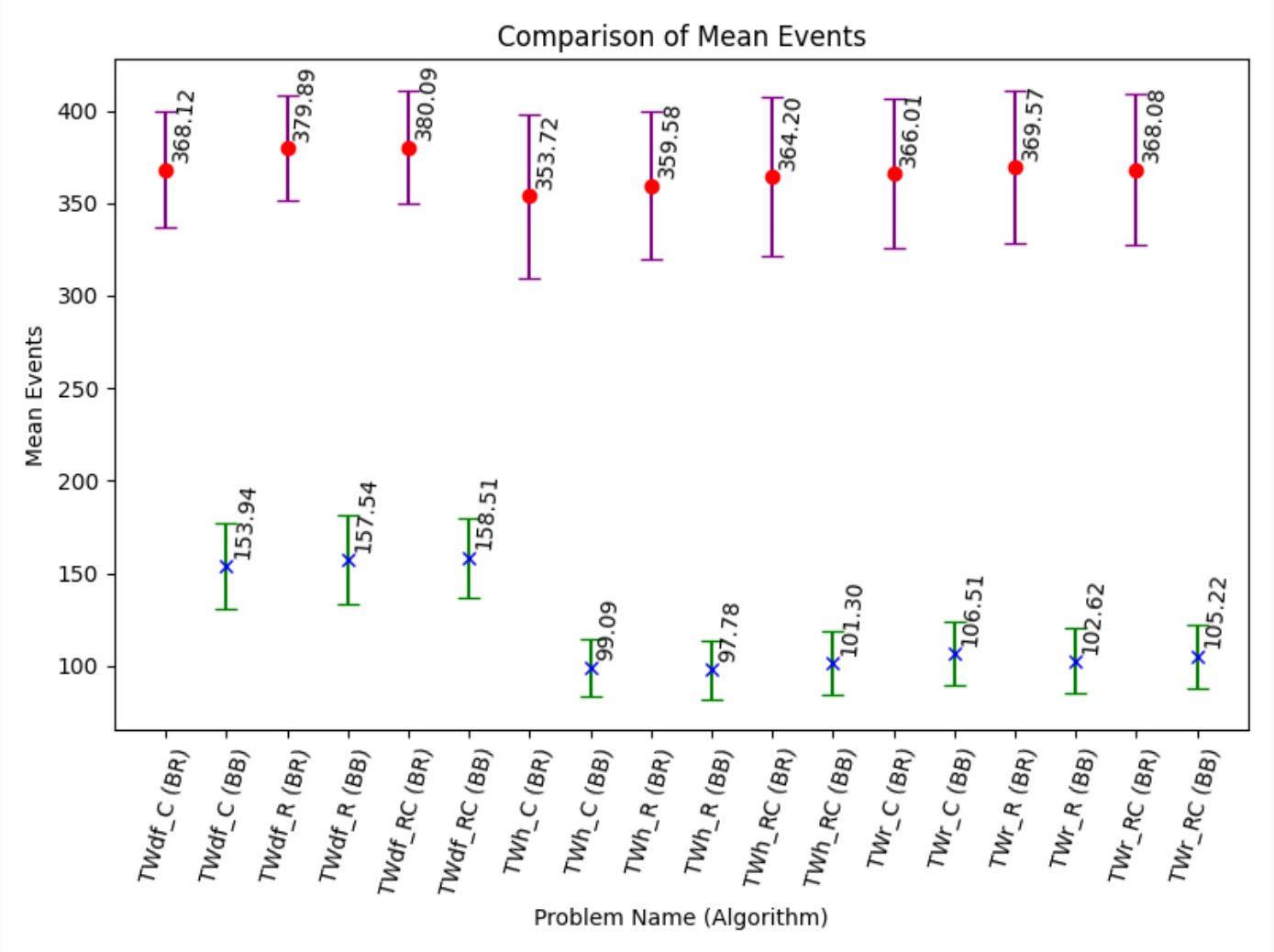


Figure 10: B&B vs B&R: numero medio di eventi

Mean Distance

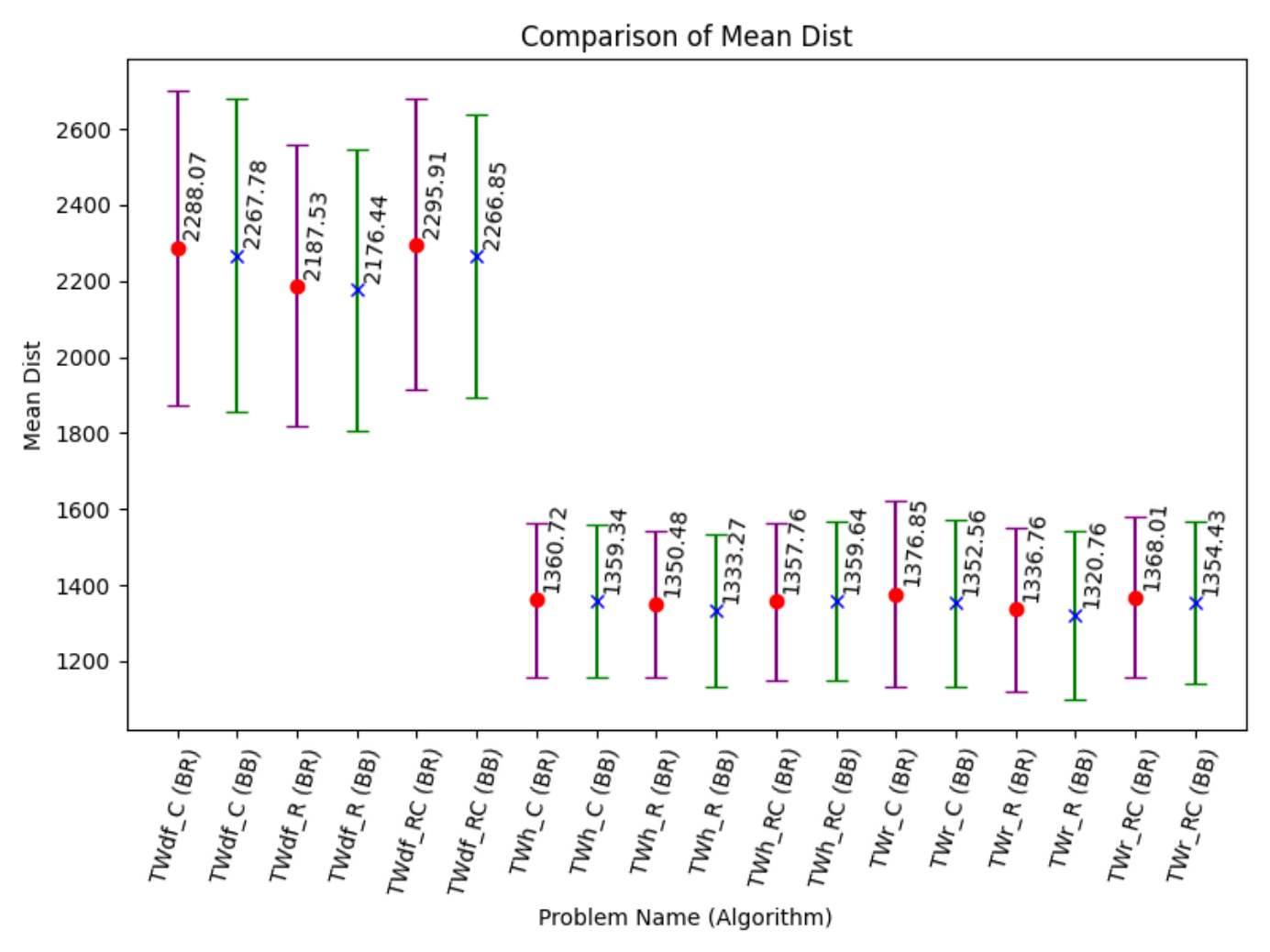


Figure 11: B&B vs B&R: distanza media percorsa

Mean Skipped

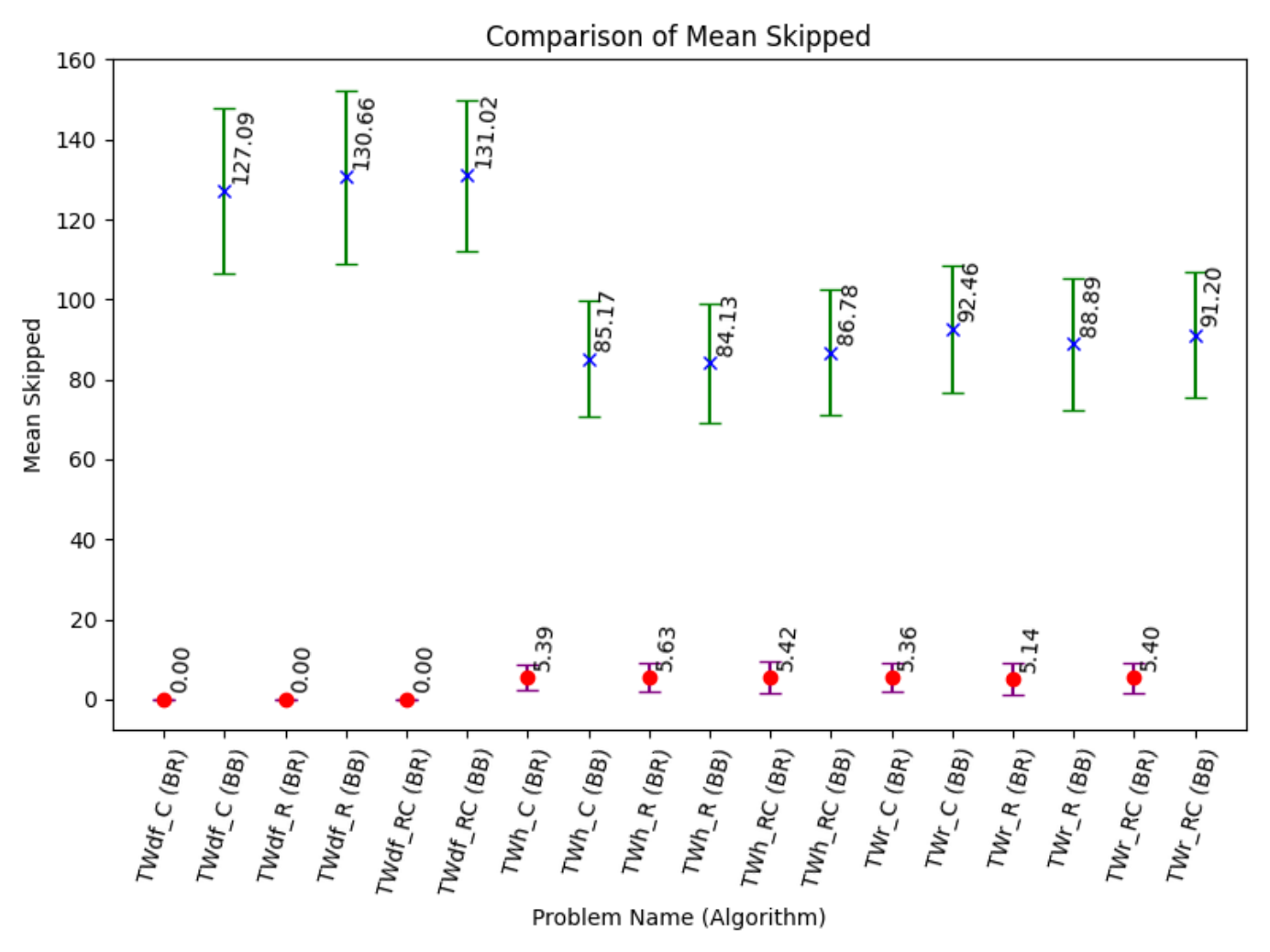


Figure 12: B&B vs B&R: numero medio di eventi/piani saltati

Mean Cost

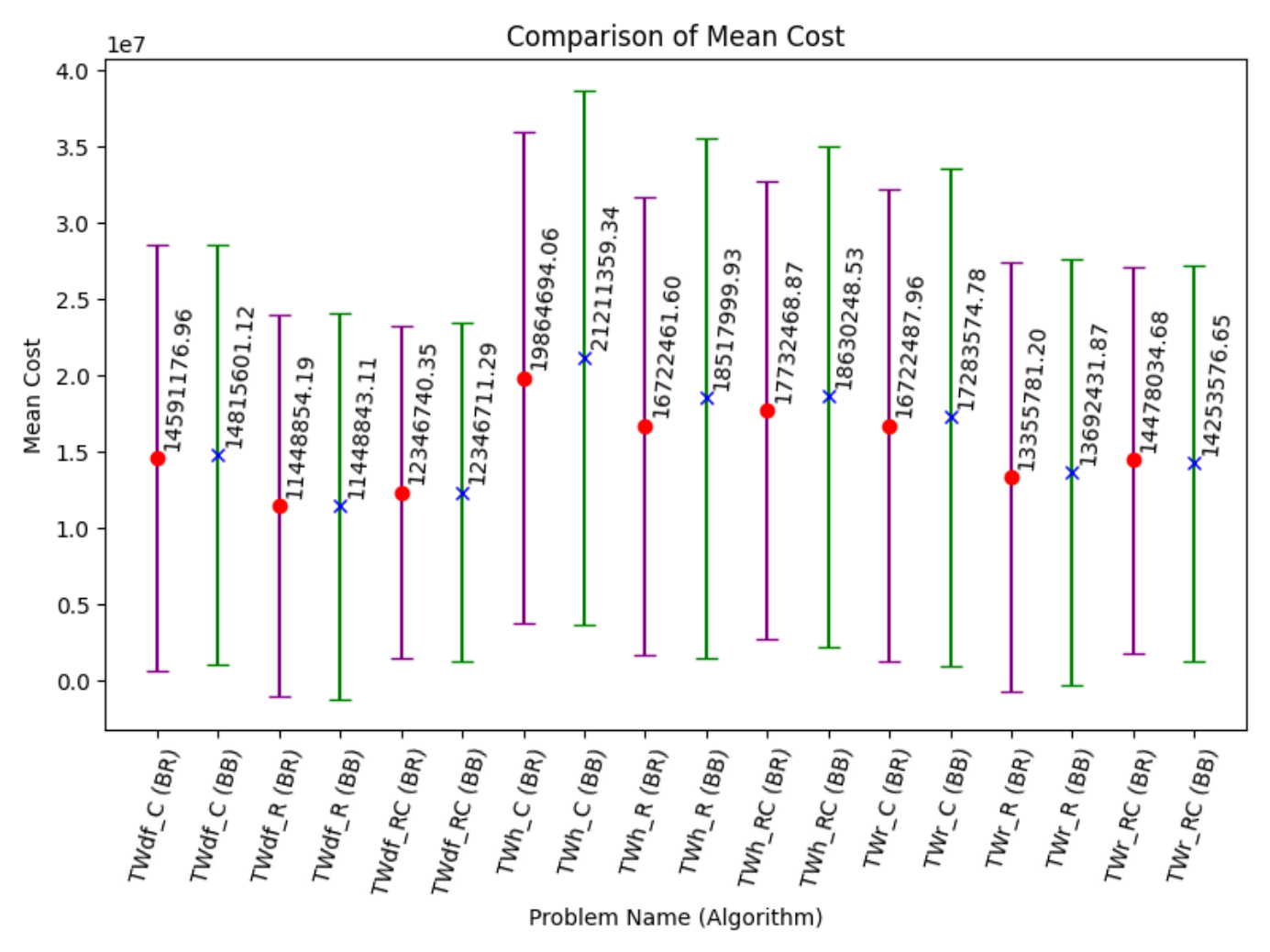
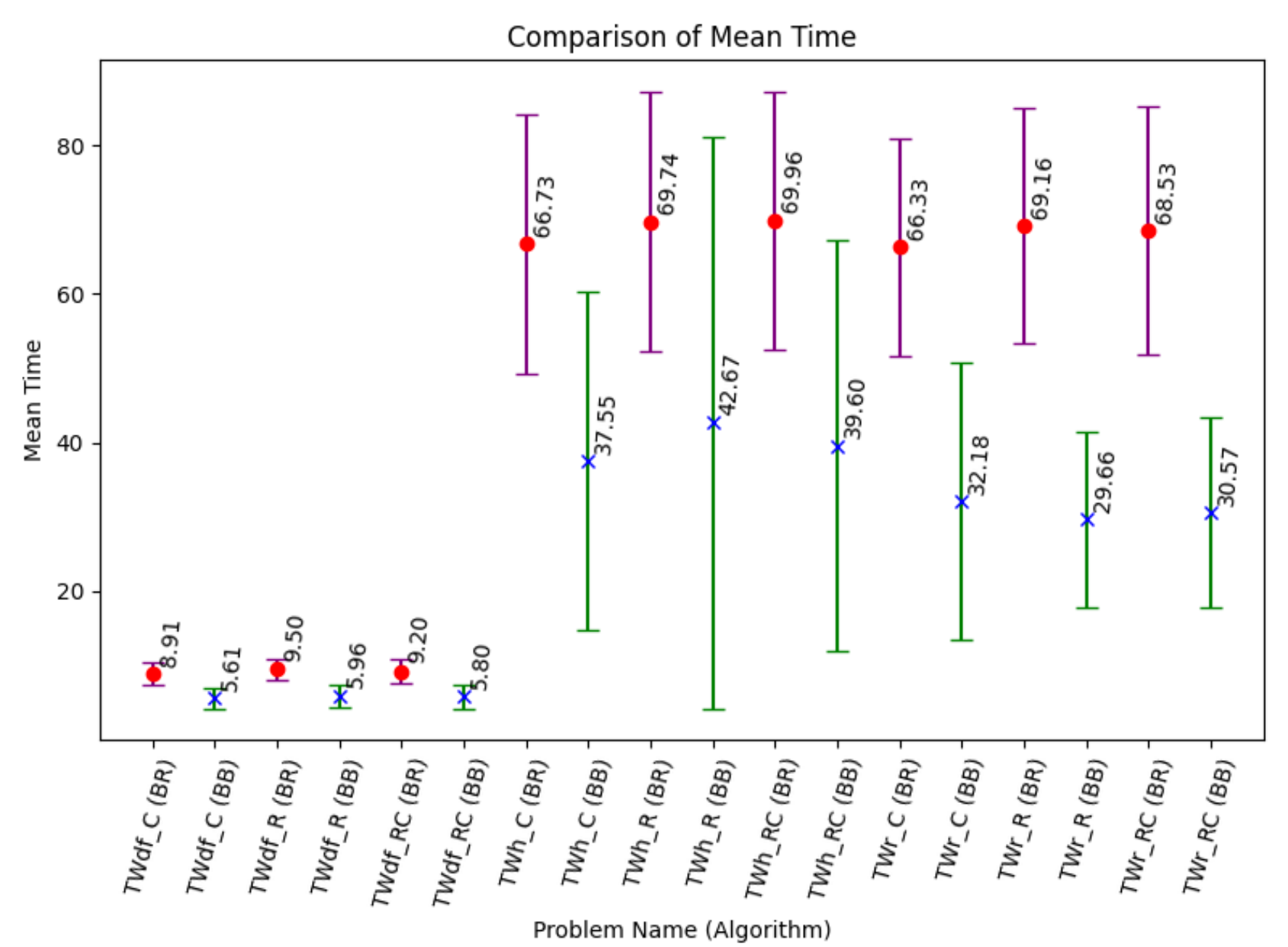


Figure 13: B&B vs B&R: costo medio delle soluzioni

Mean Time


Comparison of Mean Time
Mean Time
Problem Name (Algorithm)
20
40
60
80
8.91
5.61
9.50
5.96
9.20
5.80
66.73
37.55
69.74
42.67
69.96
39.60
66.33
32.18
69.16
29.66
68.53
30.57
TWdf_C (BR)
TWdf_C (BB)
TWdf_R (BR)
TWdf_R (BB)
TWdf_RC (BR)
TWdf_RC (BB)
TWh_C (BR)
TWh_C (BB)
TWh_R (BR)
TWh_R (BB)
TWh_RC (BR)
TWh_RC (BB)
TWr_C (BR)
TWr_C (BB)
TWr_R (BR)
TWr_R (BB)
TWr_RC (BR)
TWr_RC (BB)

Bibliografia

Bibliografia i

[1] Russell Bent and Pascal Van Hentenryck. "Scenario-Based Planning for Partially Dynamic Vehicle Routing with Stochastic Customers". In: *Operations Research* 52 (Dec. 2004), pp. 977–987. DOI: 10.1287/opre.1040.0124.

[2] Lars Magnus Hvattum, Arne Løkketangen, and Gilbert Laporte. "A branch-and-regret heuristic for stochastic and dynamic vehicle routing problems". In: *Networks* 49.4 (2007), pp. 330–340. DOI: https://doi.org/10.1002/net.20182. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/net.20182. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/net.20182.

Bibliografia ii

[3] David Pisinger Stefan Ropke. "An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows.". In: *Trans-portation science* 40.4 (2006), pp. 455–472. DOI: 10.1287/trsc.1050.0135.

[4] Stacy A. Voccia, Ann Melissa Campbell, and Barrett W. Thomas. "The Same-Day Delivery Problem for Online Purchases". In: *Transportation Science* 53.1 (2019), pp. 167–184. DOI: 10.1287/trsc.2016.0732. eprint: https://doi.org/10.1287/trsc.2016.0732. URL: https://doi.org/10.1287/trsc.2016.0732.