PROGETTO DI RICERCA OPERATIVA

Anno Accademico 2014/15

PROGETTO 16 - CAR POOLING

Consegna Progetto

Sono note le origini degli spostamenti casa-lavoro degli studenti del polo.
Supponendo che tutti abbiano gli stessi orari (vogliano arrivare al polo alla stessa ora a inizio giornata), si organizzino gli itinerari così da **minimizzare la somma delle durate** dei percorsi delle macchine.

Tutte le macchine hanno capienza 5 e ciascun utente ha un massimo tempo di viaggio consentito, delle preferenze sui compagni di viaggio, e può essere o autista o passeggero.

NUOVE VISIONI E NECESSITÀ

Problematiche e Soluzioni attuali

Il decreto interministeriale **Mobilità Sostenibile nelle Aree Urbane** del 27/03/1998, ha introdotto la figura professionale del **Mobility Manager** di azienda.

Egli ha l'obiettivo di ridurre l'uso dell'auto privata adottando strumenti che favoriscono soluzioni di trasporto alternativo a **ridotto impatto ambientale**.

Attualmente sono presenti soltanto soluzioni proprietarie sviluppate da enti e aziende di grosse dimensioni (e.g. Software DREAMS del Politecnico di Milano), per tale motivo si propone un'implementazione **innovativa e open source**.


RIGHT WAY
Alternative transport solutions

USER INTERFACE

Multiplatform with Db Client-side

USER INTERFACE

Features Principali

DB PEOPLE

PLACE & TIME

HTTP REQUEST

MAPS VIEWS

PYTHON WEB SERVICE

Safety, Reliable and Concurrent

PYTHON WEB SERVICE

Step Principali

MAPS API

SOME MAGIC

BENCHMARK

JSON RESPONSE

DEFINIZIONE PROBLEMA

Obiettivo, Dati e Vincoli

Si vuole soddisfare le richieste di un insieme di utenti, ognuno dei quali desidera spostarsi fra due punti di una rete, arrivando tutti ad un'ora prestabilita. Si tratta quindi di determinare un insieme ottimo di percorsi su grafo, nel rispetto dei vincoli dati, in modo da minimizzare la durata totale delle singole auto. (**Problema di Ottimizzazione**)

Ogni utente ha un insieme di **Dati** personali: numero identificativo, nome, indirizzo, massimo tempo di viaggio consentito e preferenze di compagni di viaggio.

L'ammissibilità delle possibili soluzioni è data da un insieme di **Vincoli**: la capacità massima di ciascuna macchina, la possibilità di inserire un tempo limite di viaggio, e la preferenza negli abbinamenti dei compagni di viaggio.

FORMALIZZAZIONE PROBLEMA

Scelta di Dati Reali

```
Via Otello Putinati 122
Corso Porta Po 30
Via Giuseppe Saragat 1
Via Giuseppe Fabbri 11
Via Bologna 11
Via Palestro 22
Via Pomposa 48
Via Porta Reno 24
Via Corso Giovecca 1
```

```
Node {
    int id: 1
    String name: "Tommaso Berlose"
    String address: "Via Otello Putinati 122"
    String img: "" //default, img url not set
    long maxDur: 0 //default, not set
    String notWith: "5,6" //other people ids
}
// seguono setter e getter
```

Per ogni utente selezionato viene creato un **Nodo** che contiene tutte le informazioni necessarie. Queste formano i punti di ancoraggio sulla mappa.

FORMALIZZAZIONE PROBLEMA

Google Maps API Distance Matrix

```
https://maps.googleapis.com/maps/api/distancematrix/json?
origins=Vancouver+BC|Seattle&destinations=San+Francisco|Victoria+BC
```

```
Arc {
    int id_i: 1
    int id_f: 2
    long dur: 2120 //sec
    long dist: 14505 // metri
}
```

```
Arc {
    int id_i: 2
    int id_f: 1
    long dur: 1560 //sec
    long dist: 13905 // metri
}
```

Per ogni coppia di nodi viene creato un **Arco** che contiente tutte le informazioni necessarie. Ogni arco è bidirezionale con costo differente in base al verso di percorrenza.

FORMALIZZAZIONE PROBLEMA

Grafo e Soluzioni

Il problema è quindi formalizzato a partire da un grafo **completo**, **connesso**, **non orientato**, con costi differenti su ogni arco in base al verso di percorrenza.

Dai dati inseriti, tramite determinati metodi costruttivi, è possibile ottenere una **Soluzione** del problema che rispetti i vincoli di ammissibilità.
Si considera soluzione la disposizione di tutti gli utenti in un insieme di vettori ordinati, detti macchine, dove il primo elemento è l'utente Autista e l'ultimo è il nodo Destinazione.

MODELLO MATEMATICO

Classi di Complessità

Il problema studiato si dice di **classe NP** poichè il problema di certificato associato (anche detto test di ammissibilità) è risolvibile in tempo polinomiale.

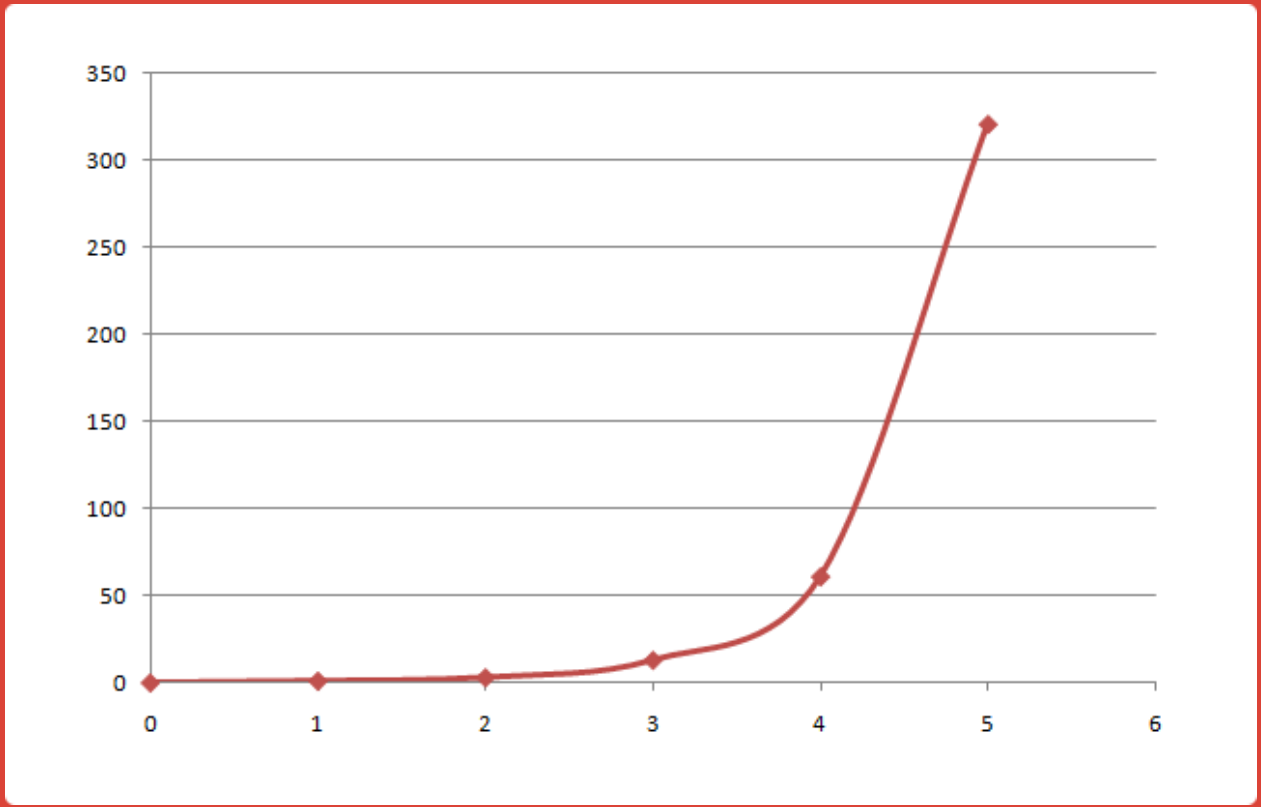
Inoltre, poichè appartiene alla famiglia dei cosidetti problemi di routing e scheduling, si può ricondurre al TSP che è della sottoclasse **NP-Hard**.

Per poter affrontare le istanze reali di un problema NP-Hard si deve necessariamente ricorrere ad **approcci di tipo euristico** accontentandosi di trovare delle 'buone' soluzioni amissibili. Questo perchè metodi esaustivi divengono inutilizzabili non appena il numero dei dati in ingresso aumenta oltre un certo limite.

COMPLESSITÀ

Numero delle Soluzioni da valutare

Nodi	V(n)
0	0
1	1
2	3
3	13
4	61
5	321



APPROCCIO EURISTICO

Considerazioni Iniziali: Come procedere?

Come primo approccio si sceglie un'euristica di tipo **Greedy** poichè di facile implementazione e notevole efficienza computazionale.
Gli algoritmi Greedy determinano la soluzione attraverso una sequenza di decisioni parziali (localmente ottime), senza mai modificarle.

Si vuole poi procedere valutando i risultati delle singole euristiche, ottenuti tramite un **Benchmark** di circa 1000 ripetizioni su 46 elementi, così da valutare i miglioramenti da ricercare ad ogni step.

GREEDY

Costruzione e Funzionamento

Si inizia scegliendo il nodo più lontano dal nodo destinazione: questo verrà inserito come primo elemento di un nuovo vettore macchina (diventando un **nodo autista**).

La procedura **Best** esamina i nodi non ancora visitati e va a selezionare il nodo più vicino a quello in cui mi trovo tramite l'algoritmo "Nearest Neighbor".
Non vengono considerati i nodi che sono "oltre" il nodo destinazione poichè aggiungerli porterebbe un peggioramento.

La procedura **Ind** verifica che siano rispettati i vincoli di cardinalità dell'auto, i massimi tempi di viaggio consentiti e le preferenze in fatto di compagni di viaggio.

Se sono rispettati i vincoli si aggiunge il nodo al vettore macchina e si va a selezionare il nodo successivo. Altrimenti viene generata una nuova macchina al ciclo successivo.

GREEDY

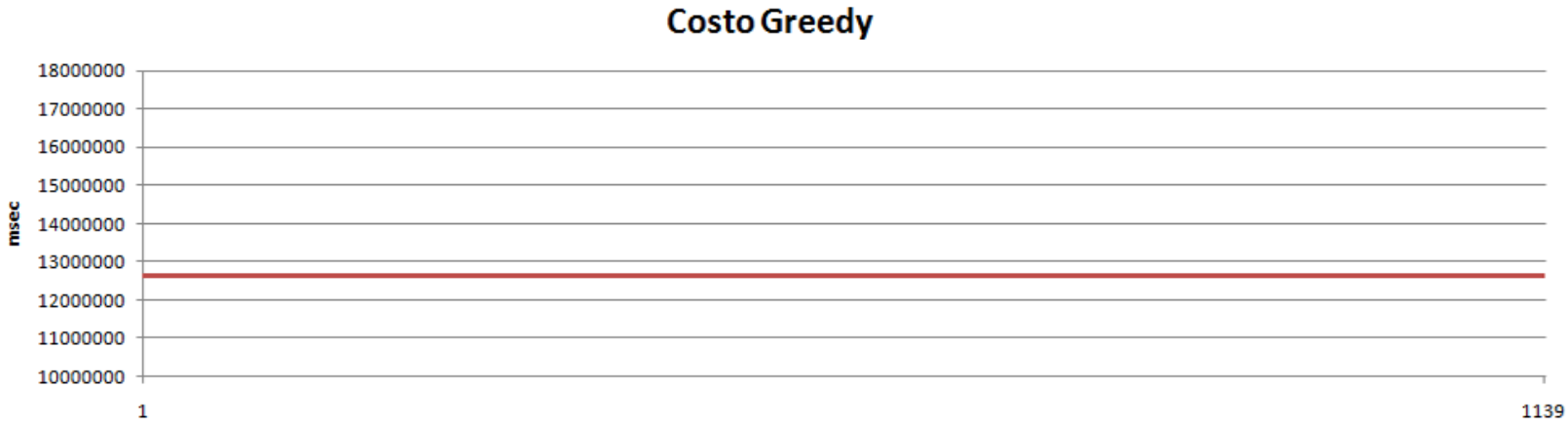
Algoritmo

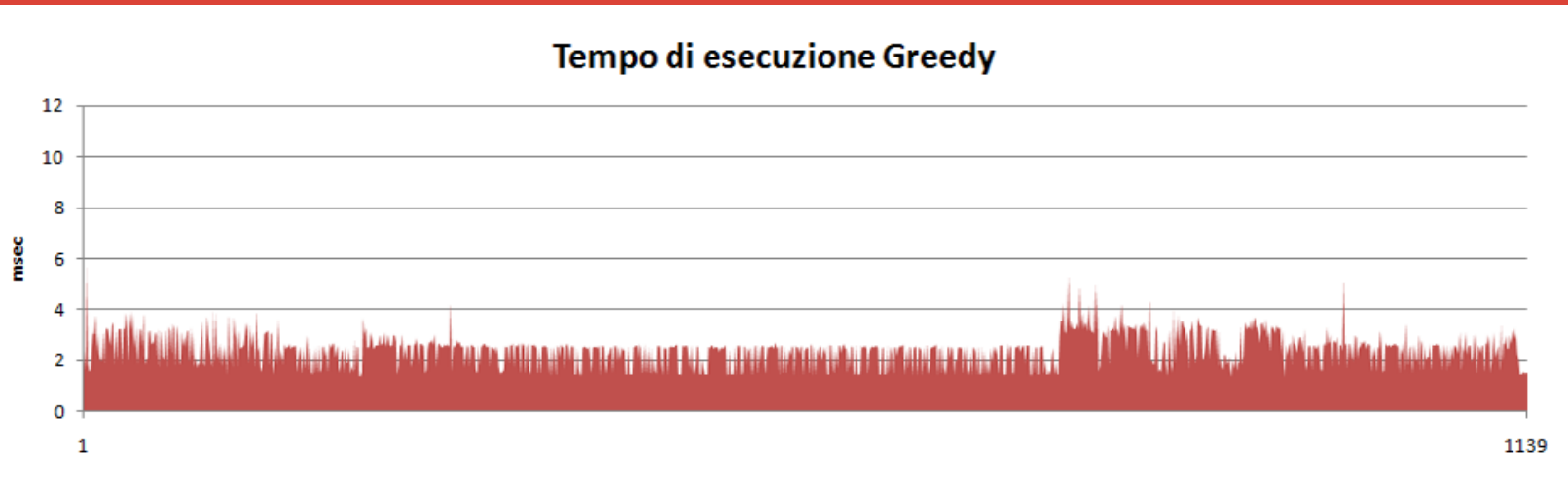
```
S = {}
N = node_to_visit
  while N ≠ {∅}
    Y = Best(x ∈ N)
    .
    .
    if Ind(S ∪ {Y})
          .
          .
          .
          then (Add Y to S)
          .
          and  (Remove Y from N)
return S
```

```
def greedy(self):
  arcToDest_dict[key] = self.arc_dict[ke
  while len(arcToDest_dict) > 0:
    car = sorted(arcToDest_dict.values()
            .getId_i()
    for arc in arc_start.values():
      if (self.NotVisited([...]) and
              self.CheckDuration([...])
              self.checkNotWith([...]) a
              self.minDurationGreedy([..
        car = arc.getId_f()
        cars_list[nauto].append(car)
        arcToDest_dict.pop(car)
  return (cars_list)
```

GREEDY

Benchmark





GREEDY

Come Migliorare

L’euristica Greedy **non garantisce l’ottimalità della soluzione** perchè compie decisioni parziali "localmente ottime" senza mai modificarle.
Costruisce infatti la soluzione attraverso iterazioni successive, effettuando ad ogni iterazione la scelta più favorevole compatibile con i vincoli del problema.
Le ultime scelte però sono prese solamente in base alla ricerca di ammissibilità della soluzione e non in base al suo costo.

Combinando un algoritmo di ricerca locale a una Greedy randomizzata si ottiene l’euristica **GRASP** (Greedy Randomized Adaptive Search Procedure) che è un’evoluzione della Multi Start Local Search.

GRASP

Costruzione e Funzionamento

Si utilizza lo stesso algoritmo creato per la Greedy, ma ad ogni passo **viene scelto un nodo random** tra un sottoinsieme di k nodi "migliori" (nel nostro caso k=3, con k=1 si avrebbe la Greedy standard).

Si utilizza una funzione random anche per selezionare l'autista iniziale in modo che non sia sempre lo stesso ma ne venga selezionato uno a caso tra i k più lontani dalla destinazione. Le procedure Best ed Ind e il metodo costruittivo delle soluzioni rimangono invariate rispetto all'algoritmo della Greedy.

GRASP

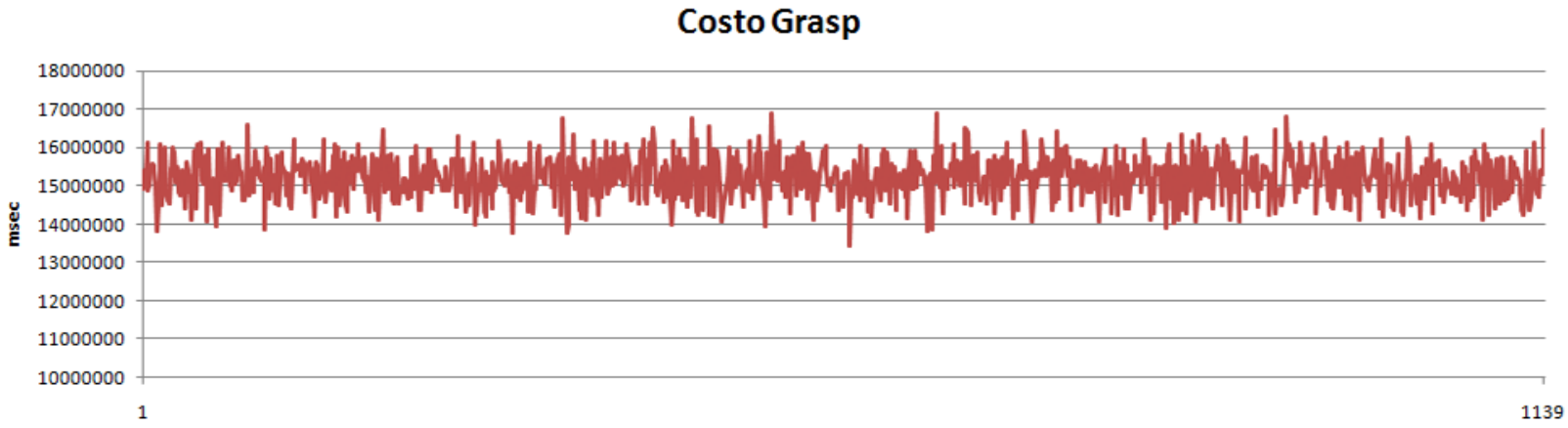
Algoritmo

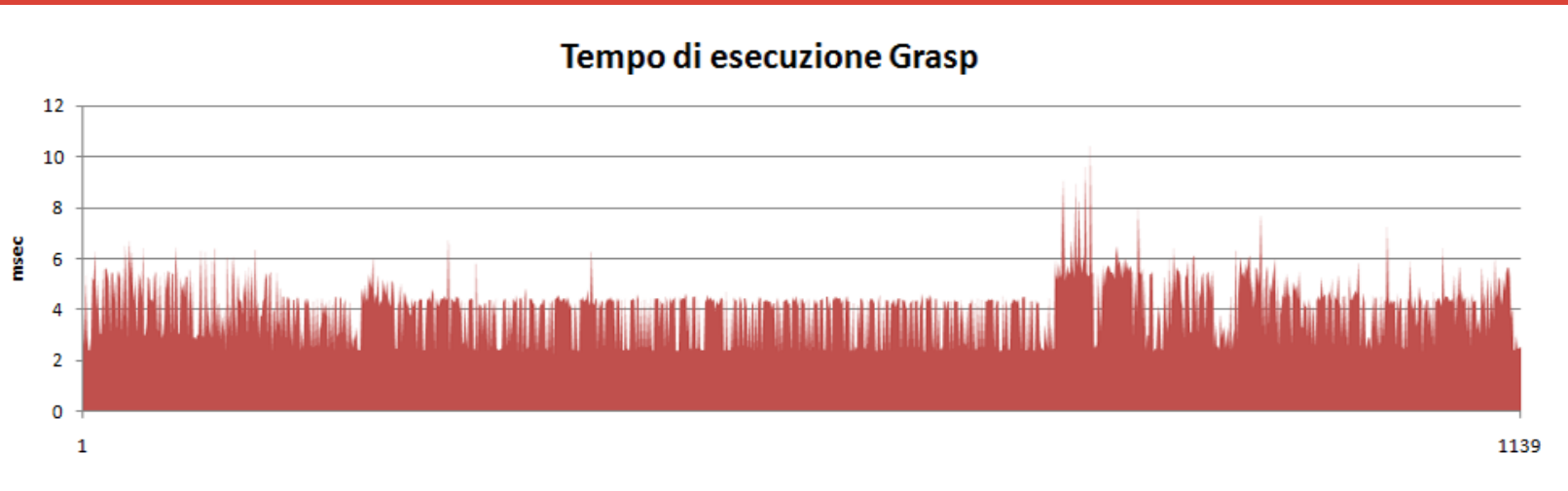
```
S = {}
N = node_to_visit
  while N ≠ {∅}
    .
    Y = RandomBest(x ∈ N)
    .
    .
    if Ind(S ∪ {Y})
          .
          .
          .
          then (Add Y to S)
          .
          and  (Remove Y from N)
return S
```

```
def grasp(self):
  arcToDest_dict[key] = self.arc_dict[ke
  while len(arcToDest_dict) > 0:
    k = 5
    bestKE = sorted(arcToDest_dict.value
    car = Random(bestKE, 1).getId_i()
    for arc in arc_start.values():
      if (self.NotVisited([...]) and
              self.CheckDuration([...])
              self.checkNotWith([...]) a
              self.minDurationGreedy([..
        car = arc.getId_f()
        cars_list[nauto].append(car)
        arcToDest_dict.pop(car)
  return (cars_list)
```

GRASP

Benchmark





GRASP

Come Migliorare

I limiti della GRASP sono rappresentati principalmente dal fatto che **non possiede memoria** (ogni restart è quindi indipendente dai precedenti) e che non dà nessuna garanzia sulla qualità della soluzione iniziale.
La procedura inoltre non accetta peggioramenti, **rimanendo intrappolata negli ottimi locali** di qualità spesso peggiore rispetto all'ottimo globale (potenzialmente si converge più volte allo stesso ottimo locale).

Si può trovare una soluzione a questi problemi utilizzando l'euristica **Path Relinking**, che esplora le soluzioni ottenute operando delle mosse a partire da una popolazione di soluzioni elite $\{ s_i \}$ avvicinandosi ad una soluzione target s^*.

PATH RELINKING

Costruzione e Funzionamento

Si costruisce una **popolazione di soluzioni elite** $\{ s_j \}$ ripetendo la GRASP più volte e aggiungendo la Greedy trovata precedentemente.
Si operano quindi in successione delle mosse di scambio di componenti per avvicinarsi alla soluzione target s*.

Ad ogni passo si valuta ogni **mossa** potenziale, cioè la modifica di una componente di s_j con una della target s*, per ciascuna delle componenti per cui s_j e s* differiscono (e.g. lo scambio di due passeggeri in due macchine differenti).

Tra tutte le soluzioni trovate, una per ogni mossa possibile, si seleziona quella più conveniente e si itera l'algoritmo. Conclusa l'esecuzione si sceglie la **migliore soluzione trovata** lungo il percorso.

PATH RELINKING

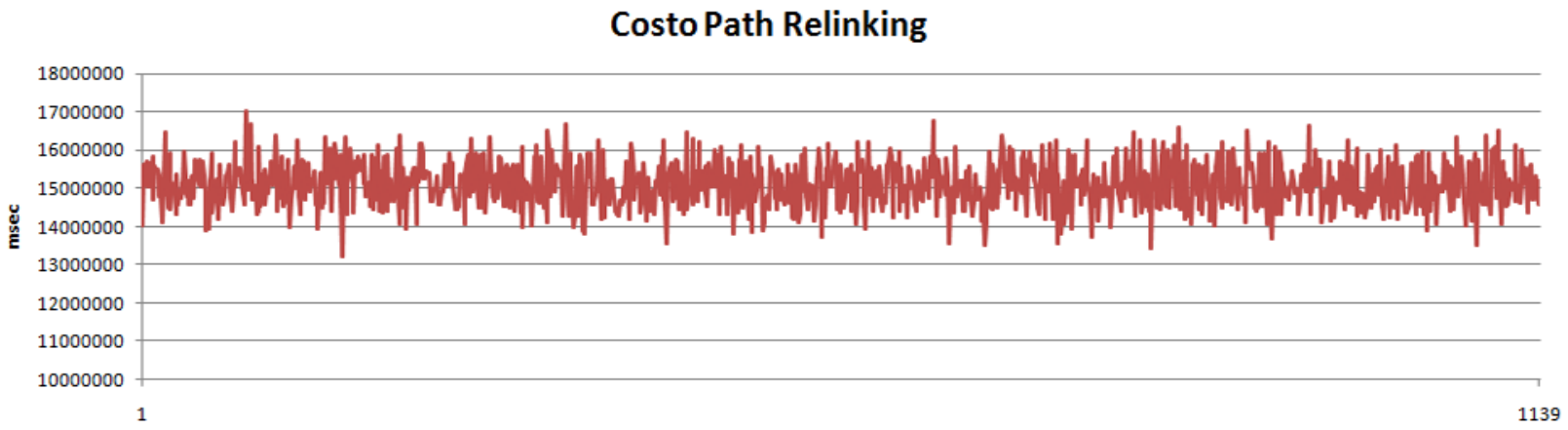
Algoritmo

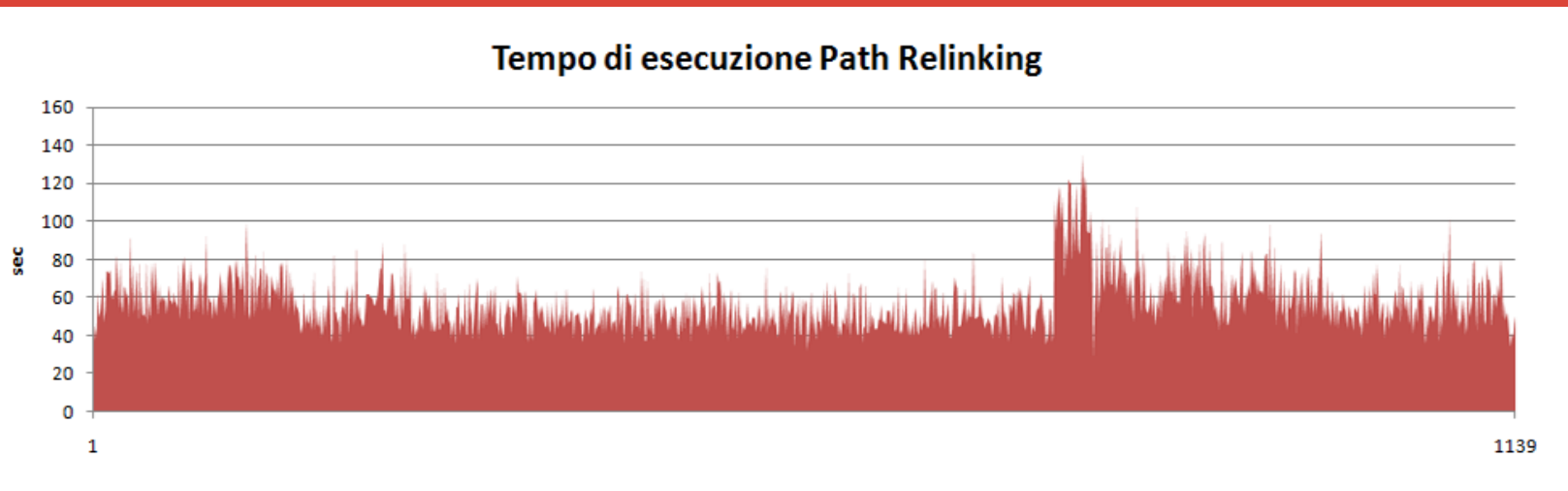
```
def path(self, target, elite, iteration):
  while True:
   for x in Cars:
     for y in Cars.Utenti:
       if target_tmp.cars[x][y] != elite_tmp_pope[-1].cars[x][y]:
         elite_tmp_pope[-1].cars[x][y] = target_tmp.cars[x][y]

  if self.controllo_stop(target, elite_ok, iteration):
    return elite_ok
  else:
    iteration += 1
    deep_sol = self.path(target, elite_ok, iteration)
    if depp_sol.getDur() > elite_ok.getDur():
      return elite_ok
    else:
      return deep_sol
```

PATH RELINKING

Benchmark





REVERSE PATH RELINKING

Costruzione e Funzionamento

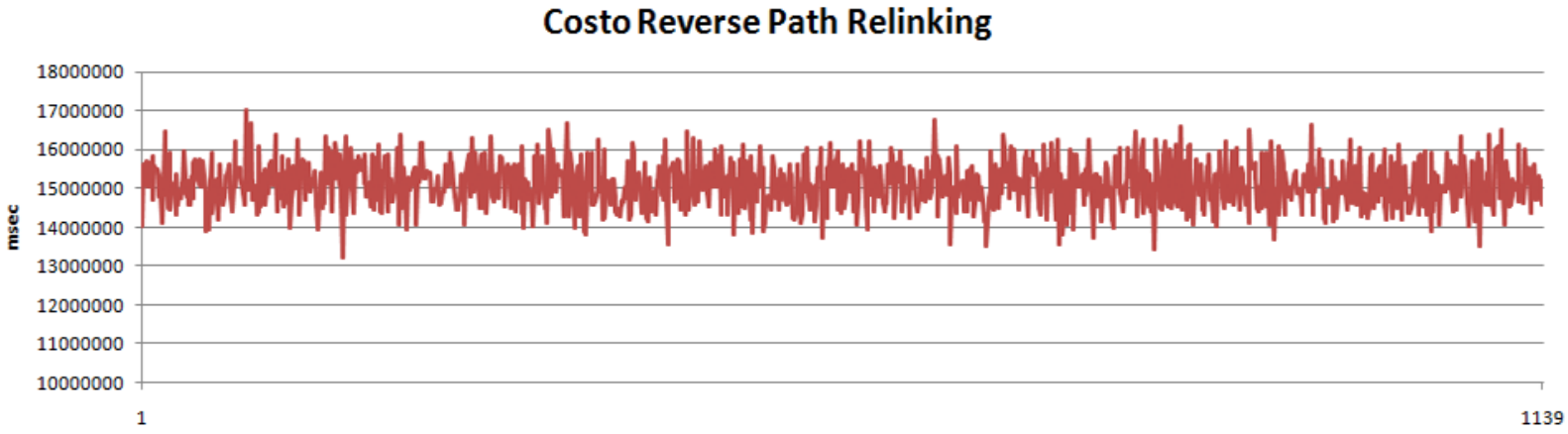
Secondo alcuni studi è più conveniente procedere in **senso inverso**, quindi dalla soluzione target s*, ad ogni mossa, si cerca di raggiungere la soluzione elite s_i.

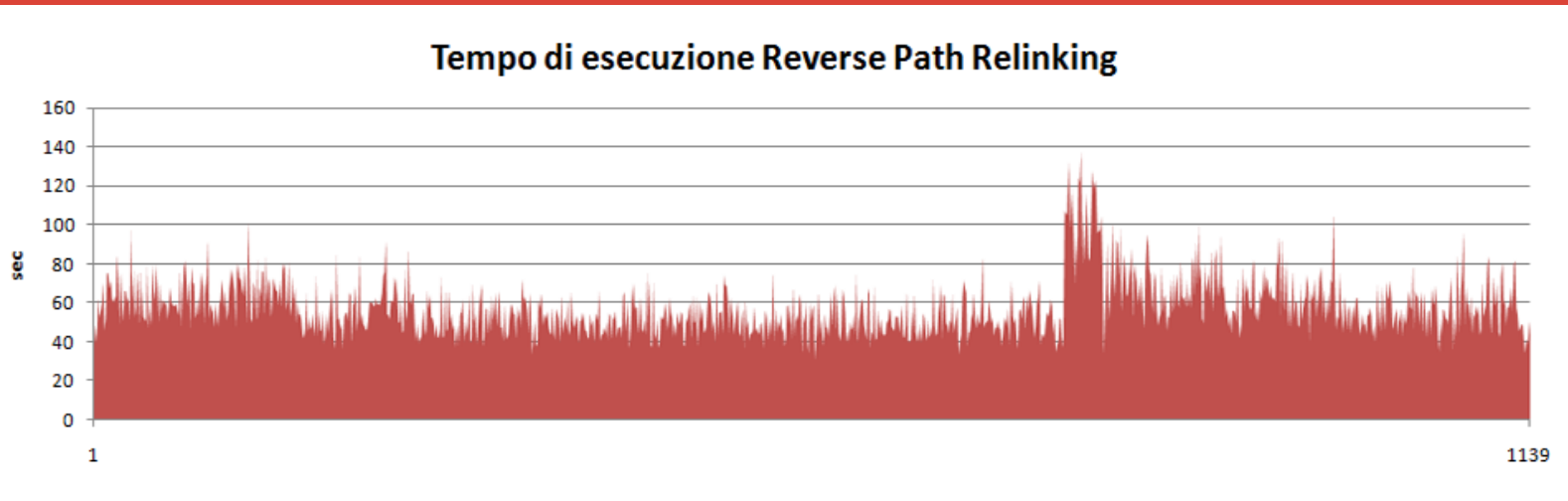
Si può infatti verificare che ad ogni mossa effettuata il grado di ramificazione delle possibili soluzioni venga ridotto ma secondo il **Proximate Optimality Principle** spesso si osserva che gli ottimi locali sono concentrati in una regione limitata.
Per questo è preferibile iniziare la ricerca in senso inverso così da controllare maggiori soluzioni vicino alla soluzione target s* che è di fatto un ottimo locale.

Questo approccio, come per la Path Relinking normale, è possibile grazie alla proprietà di **Raggiungibilità** di tutte le soluzioni del problema.

REVERSE PATH RELINKING

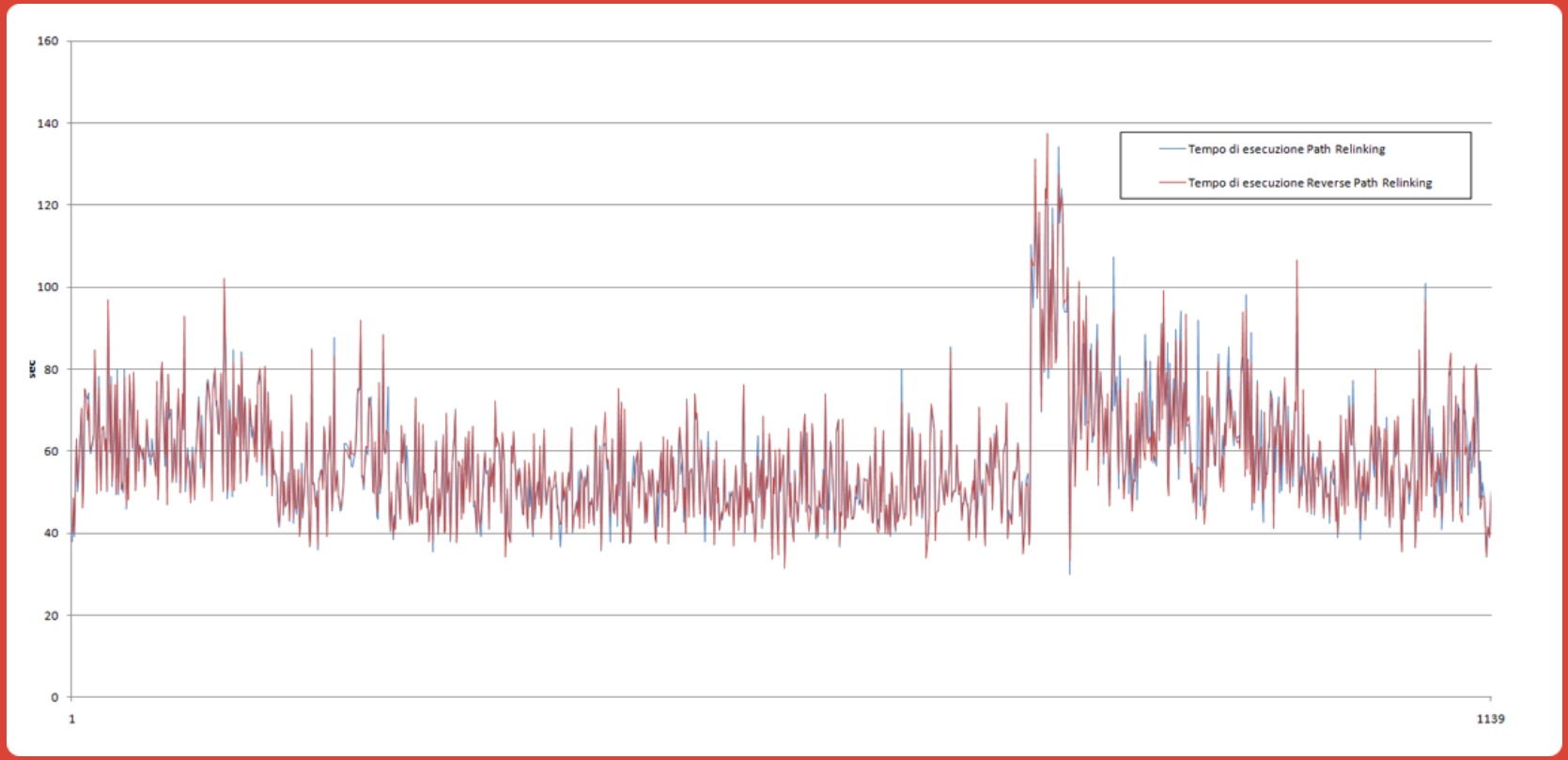
Benchmark





CONFRONTO PATH RELINKING

Tempi di Esecuzione



Confrontando le due euristiche si è notato che, sebbene il costo delle soluzioni sia identico, il tempo di esecuzione della Path Relinking è leggermente migliore nel 54,53% dei casi.

CONFRONTO PATH RELINKING

Come Migliorare

Una limitazione che rimane anche nella Path Relinking riguarda il fatto di non avere memoria, per questo si rischia di ritornare in soluzioni già visitate (e quindi andare in loop).

Per risolvere questo problema, e uscire quindi dal bacino di attrazione degli ottimi locali, si può usare la tecnica **Tabu Search**.
Questa euristica ammette anche soluzioni peggiorative e non ammissibili durante la propria esecuzione (penalizzando in caso di violazione di vincoli) partendo da una soluzione random e valutando ogni possibile mossa di spostamento.

TABU SEARCH

Costruzione e Funzionamento

Si parte da una soluzione generica e si valutano tutte le possibili soluzioni a distanza di una mossa: **Swap**, **Stack** o **Unstack**. Si seleziona poi la soluzione migliore dell'intorno valutato (si accettano anche peggioramenti in modo da sfuggire agli ottimi locali) e si mantiene memoria della soluzione migliore visitata. Nel caso in cui la soluzione non sia ammissibile allora verrà aggiunta una **penalty** al costo.

Poichè una scelta peggiorativa porterebbe a dei loop, occorre che l'inversa della mossa appena effettuata venga memorizzata in una **Tabu List** di lunghezza k (e.g. k = 7), e che le mosse presenti nella lista siano proibite per le prossime k iterazioni (senza considerare il **Criterio di Aspirazione**).

Iterando questa ricerca, l'algoritmo finisce nel caso siano eseguite 30 iterazioni senza trovare una soluzione migliore di quella già in memoria o dopo un numero di iterazioni globali massimo.

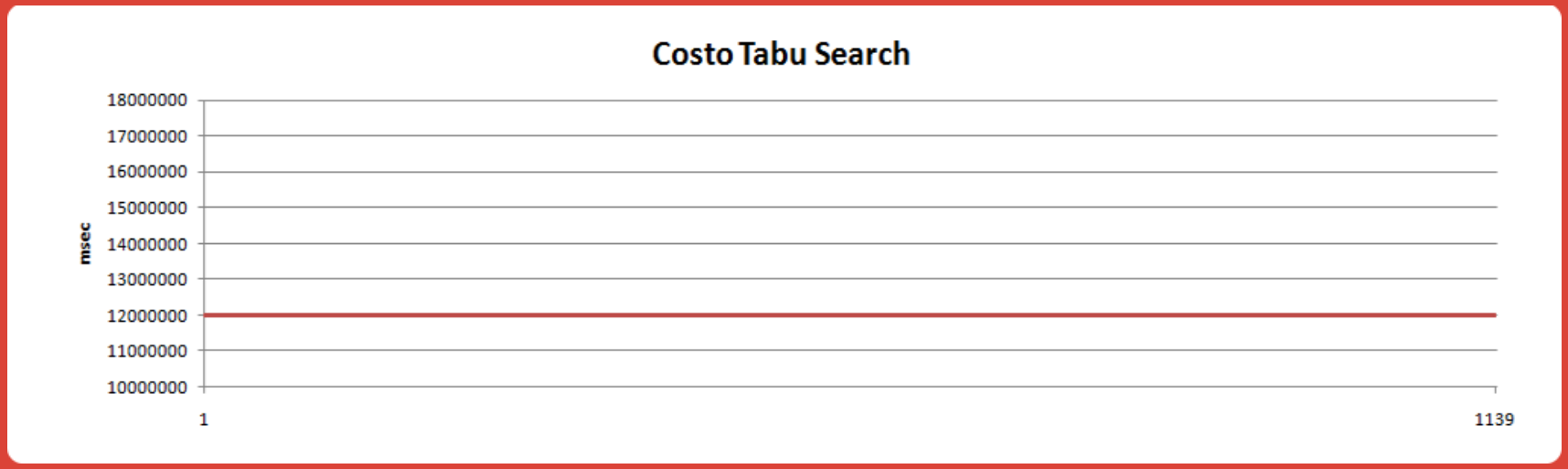
TABU SEARCH

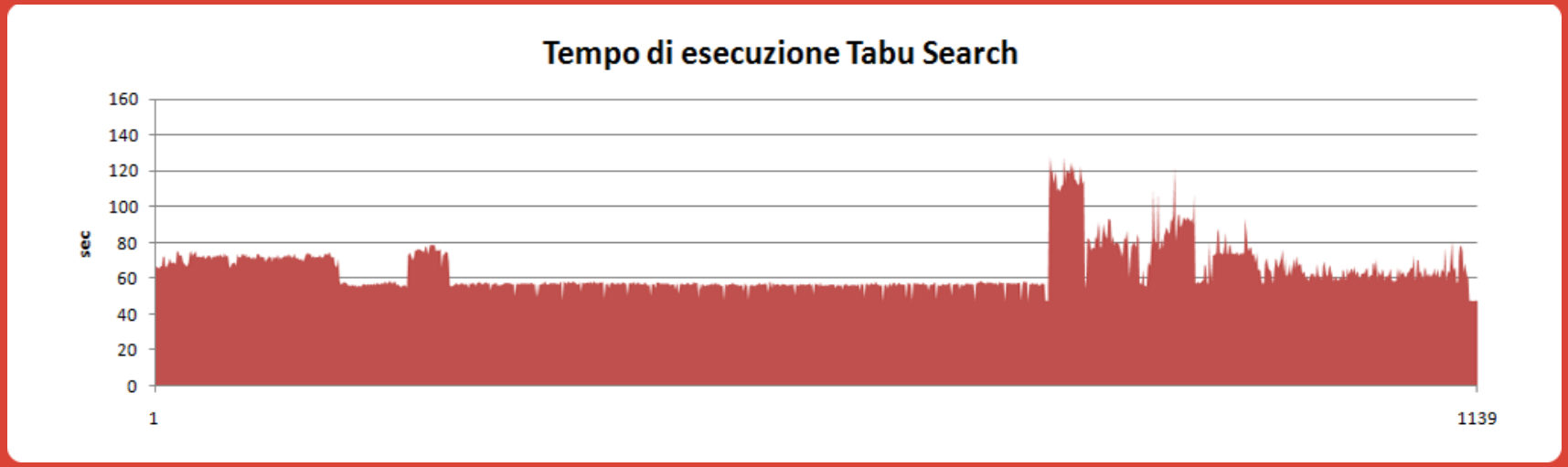
Algoritmo

```
def tabu(self, best_solution, actual_solution, [...]):
    global_iteration += 1
    [...]
    # SwapCar
    for x in range(len(cars)):
        for y in range(len(cars[x])):
            if x != len(cars):
                for x1 in range(x+1, len(cars)):
                    for y1 in range(len(cars[x1])):
                        [CheckOldPenalty]
                    eur = Euristiche(self.node_dict, self.arc_dict)
                    eur = self.reorder(eur)
                    if not self.ammissibileNotWith(eur):
                        totDur += penality
                    if not self.ammissibileMinDur(eur):
                        totDur += penality
                    totDur += self.ammissibileCard(eur)*penality
```

TABU SEARCH

Benchmark





OSSERVAZIONI CONCLUSIVE

Tempi di Esecuzione in funzione dell'Input

E' possibile confrontare i tempi di esecuzione dell'algoritmo utilizzato in funzione di valori di input (numero di nodi da valutare) differenti.
All'aumento del input corrisponde, infatti, una **minore linearità** nei tempi di esecuzione ed un loro **aumento esponenziale**.

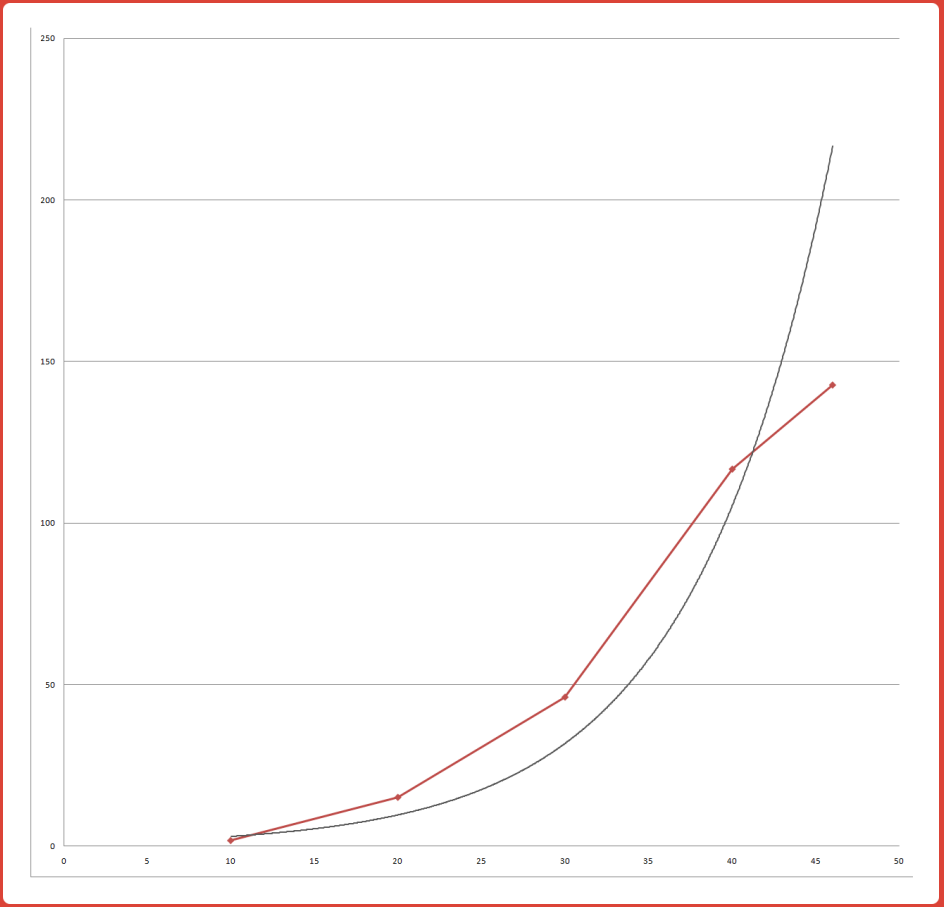
Sono presenti alcune limitazioni a questo studio a causa dell'accesso free alle API di Google Maps; non si è potuto superare il valore di 46 nodi e si è dovuto tenere conto del tempo necessario tra una richiesta e l'altra.

OSSERVAZIONI CONCLUSIVE

Crescita Esponenziale

Nodi	Tmp Exec(s)
10	1,7271
20	15,0993
30	46,0919
40	116,6371
46	142,6476

Visualizza tutti i Dati.



OSSERVAZIONI CONCLUSIVE

Confronto Costi

A conclusione dello studio effetuato si può osservare che tramite la **Tabu Search** si riesce ad ottenere un buon risultato al problema NP-Hard del **Car Pooling** con un input di dimensione non ridotta. La soluzione finale risulta inoltre non solo molto buona ma anche applicabile a contesti reali.

Dai dati ottenuti tramite benchmark risulta però che la **Greedy** sia non troppo peggiorativa rispetto alla Tabu in confronto ad un tempo di esecuzione nettamente inferiore: la Greedy in media restituisce il risultato in **2,445 millisecondi** mentre ad esempio la Tabu impiega ben **64,928 secondi**; il tutto per ottenere un miglioramento, in termini di costo, di solo **11 minuti** sulla somma delle durate delle singole macchine.

Per concludere è stata sviluppata una piattaforma che, in tempi limitati e con sforzo computazionale relativamente basso, riesce a rivolvere un problema di grosse dimensioni il cui utilizzo è di interesse reale e attuale.

FINE

Grazie per l'attenzione

Federica Pelli, Giulio Riberto, Tommaso Berlose